A remote fine grained scheduler, with a case study on an Nvidia BlueField DPU device

This thesis,

by Cyrus James Grahame Legg,

is presented for the degree

of Doctor of Philosophy,

at University College London (UCL).

Declaration

I, Cyrus James Grahame Legg, confirm that the work presented in my thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Signature: ____________________________

REDACTIONS:

This publication version of the thesis has been redacted compared to the version accepted by the examiners (with corrections), which was also filed with the University authorities. The redactions made were:

- **Removal of the signature on this page, for privacy.**
- **In Table 7, which quotes 3rd party code, the extent of the quotation has been further trimmed out of an abundance of caution with regard to fair use of copyright works.**
- **As explained on page 264, the code annex has been removed as licensed under the LGPL3 licence, which is not compatible with the Creative Commons licence under which this document will be published by the University.**

Abstract

The QuickSched fine-grained task-based scheduler (used in the Swift astrophysical smooth particle hydrodynamics code) was modified so that the scheduler is located in a separate process from the computational threads, with the computational threads calling the scheduler functions via an RPC message loop. Thus, the time-divided scheduler of QuickSched on the computation threads was replaced by a dedicated ‘remote’ process. The efficiency of the scheduler was analysed in view of the likely detriment caused by the messaging. The investigation was of an existing example of the tiled QR factorisation, and various locations of the remote scheduler were tested: on the same and remote hosts on the same LAN as the computational host, and on the general-purpose Arm processor of BlueField cards located on these hosts.

Under optimisation of the tile size, a region of high performance was found and was the same region for the original and new remote schedulers. Within that region the new remote scheduler performed as well as the original to within a few percent, and so the new scheduler location is viable despite the additional messaging latency. The possibilities for extra functions for the scheduler opened up by the extra resources of made available to the scheduler being in its own process are discussed. The mechanisms affecting the performance change between the original and new schedulers are complex. In the optimised region, message latency can be insignificant in some cases and in others a decrease in the time spent in kernels on changing to the new scheduler can partially but significantly compensate for the latency introduced. QuickSched’s scheduler rule of keeping unoccupied threads fed with ready tasks was seen to dilute the effectiveness of the rule to allocate tasks to cores having input data for the task in its cache.

Impact Statement

Undertaking this PhD has had significant impact for me personally, providing retraining after a different career and a successful move into a new one of academic high-performance computing (HPC) at UCL, where I have started a job working on supporting cluster users, and in future helping to develop new clusters and research the usefulness of novel accelerator cards. In this job we are currently working on deployment of a new package manager, Spack, at UCL, which may not only simplify the building of the hundreds of applications needed by researchers but should improve the reproducibility of those builds and hence of the results those applications produce.

I hope that my work will stimulate further work in my study area of fine-grained task scheduling in HPC codes. I was privileged to give outline talks to two highly prestigious conferences: GTC in November 2021 and the European ISC High Performance, also in 2021, as well as at three other conferences and meetings: HPC AI Japan 2021, HPC-AI Advisory Council Cloud-Native Supercomputing Workshop (China) in November 2021 and Singapore SCAsia22 in March 2022.

Quoting from his comments on my impact, Richard Graham, form the sponsor and a founder of the OpenMPI messaging library project, said, *"This has also been useful to understand some of the performance challenges future BlueField devices should address, for this class of problems. The placement work focused on creating a performance analysis tool for the MPI_Alltoall() collective operation in native application use. This work [the placement] has been incorporated into the open source tool, and is available for general use."*

So, the PhD work has provided the sponsor with a case to take into account for future BlueField cards, and skills learnt are having impact elsewhere.

As set out in this thesis, I believe the work contributes practical ideas to, and furthers understanding of, controlling HPC codes "from the network" and that it has generated many ideas for that that could be developed further. I also believe that contacts developed with Nvidia and other vendors during this work will be of benefit in my future work in the team developing future HPC systems at UCL.

Acknowledgements

I should like to acknowledge the help and advice of all those who assisted me along the way. They have been many, but I can only mention a few here:

At UCL:

Rev Dr Jeremy Yates
Dr Owain Kenway
Prof Serena Viti
Mr Denis Timm
Mr Tristan Clark

At Nvidia (formerly Mellanox)

Dr Richard Graham
Dr Geoffroy Vallee

Other

Joanna Clark
Dr Martyn Clark

Computing Resources

The work was primarily carried out on the facilities of the HPC AI Advisory Council (https://www.hpcadvisorycouncil.com/). However, I would also like to acknowledge the use of the HPC systems of Advanced Research Computing at UCL and of the DINE system at The Institute for Computational Cosmology at Durham University (https://www.dur.ac.uk/icc/cosma/facilities/dine/).

Contents

List of Figures

Page:

List of Tables

Page:

List of Equations

Page:

1. Scope of Investigation

1.1. Project Sponsors

My place to study at UCL was sponsored by Mellanox. Mellanox were taken over by Nvidia in 2019/2020 [1], and became the Nvidia Networks division. Nvidia Networks are manufacturers of networking equipment for local area networks (“LAN”), both Ethernet and Infiniband, and include both host channel adapters (“HCA”), often called “network cards” or “network adapters”, and network switches. Nvidia Networks’ Infiniband equipment is often selected for high performance computing (“HPC”) clusters, as can be seen from the Top 500 listing of such systems [2], where the name Mellanox is still quoted.

While the company has continued to improve the basic performance of these HCAs, networking technology is still challenged by the demands of the compute and data aspects of these computing systems. As a result, Nvidia have been developing additional technologies to improve performance in specific cases. One new technology of theirs in the area of LAN equipment is called “BlueField”. This is described in more detail in section 1.3; however, briefly stated a BlueField card is an HCA which has an on-board Arm cored general purpose computer – so general purpose that you can run Linux on it – which can communicate with its host compute node and, in one configuration, with other hosts and with BlueField cards hosted on those. This card became a central feature of the project.

1.2. Project Brief

The initial brief for the investigation was to control a cluster from the network during computation. The scope of that is extremely wide and does not immediately suggest any particular problem to be solved nor any particular direction toward a solution. It was also implicit that any solution would be software based – no facilities for designing any kind of circuitry were part of the context. A first vague idea was to build a network switch from a PC with multiple network cards so that it could be at the centre of the network and have processing power there that could provide organisational effort. That obviously did not gain any traction because it was not clear what that organisation would be and because the performance of such a switch would not anywhere near match that of a standard network switch. Some more specific problem had to be found.

1.3. Sources of the problem to be solved

Two separate studies resulted in the basic idea that formed the germ of this project.

The first was that after a while Mellanox/Nvidia were, generously, able to provide me with access to an experimental cluster fitted with BlueField cards.

Nvidia promote the use of BlueField cards for networking, accessing network storage, and security [3]. This is of course no surprise: BlueField is a “smart NIC” and so its uses would be to do with interaction with the network, and this continues in the line of their previous developments such as remote direct memory access (“RDMA”) [4] and Sharp [5], both of which offload networking work from the host CPU to the network card. However, after some familiarisation with BlueField, it was clear that its Arm cored processor subsystem was indeed general purpose: Linux can be run on it and a user can log into it and compile and run whatever code they wish. It was also possible to send messages between programs on the BlueField’s Arm processor and the programs on the host processor using standard general-purpose messaging libraries, in particular OpenMPI [6]. So, it was realised that cooperation between the BlueField’s processor and the host processor was not necessarily limited to operations involving the network and that therefore other operations helpful to the host processor could be offloaded to the BlueField. (Having said that, however, later in this thesis – section 14.6 – matters will come full circle with the network connection of the BlueField becoming relevant again, and in the direction of controlling the computation *“from the network”*.)

The other study was into the workings of the Swift code. Swift is a new *“hydrodynamics and gravity code for astrophysics and cosmology”* [7], with big ambitions: *“we want to be able to simulate a whole universe!”*. I took a deep look into how the 70,000 lines of this code organised the calculation. As they mention, it is, *inter alia*, task-based and uses the QuickSched library for scheduling the tasks. Task-based scheduling in general, and QuickSched in particular, are explained in the Chapters 2, 3 and 5, but, briefly stated, the computation is divided into many small blocks of computation and these blocks, or tasks, are dynamically allocated to the available processing units (typically the cores of the processor) by a scheduler. In Swift, this scheduling forms the backbone of the code, directing operations. Also, it was notable that this direction only takes place from time to time: when a core has completed a task, the scheduler is woken up and asked to find a new task for the core to execute.

1.4. Problem to be solved

Thus, the idea formed that this scheduling might be something that could be taken away from the main processor of a compute node and be offloaded to the processor of the BlueField card. That was of course just the germ of the idea; the practicalities, compromises and benefits of this arrangement are explored in this work.

2. Task Based Scheduling

The QuickSched library for task-based scheduling is analysed in Chapter 5 with a view to determining which parts of it might usefully be offloaded to the processor of a BlueField card, or any other “smart NIC” having a similar facility. The library is however quite detailed in its data structures and operation and so an introduction to task-based scheduling is first given here at a general abstract level.

Task-based parallelism is one of many kinds of parallelism employed in computers and computing. This particular approach involves dividing some overall data processing task into a set of separate smaller tasks and then allotting those to a pool of available processors in an appropriate order. Data dependencies between the tasks are taken into account so that, simply stated, a task may not start until all its inputs are ready. (QuickSched includes richer concepts for data dependencies than this basic one – sections 3.4 and 5.10.)

Figure 1 is a block diagram showing how the approach works at a high level; the language of the labels in brackets in the diagram relate to the particular implementation of QuickSched. The Figure is adapted from a slide used in various of my conference presentations explaining my work.

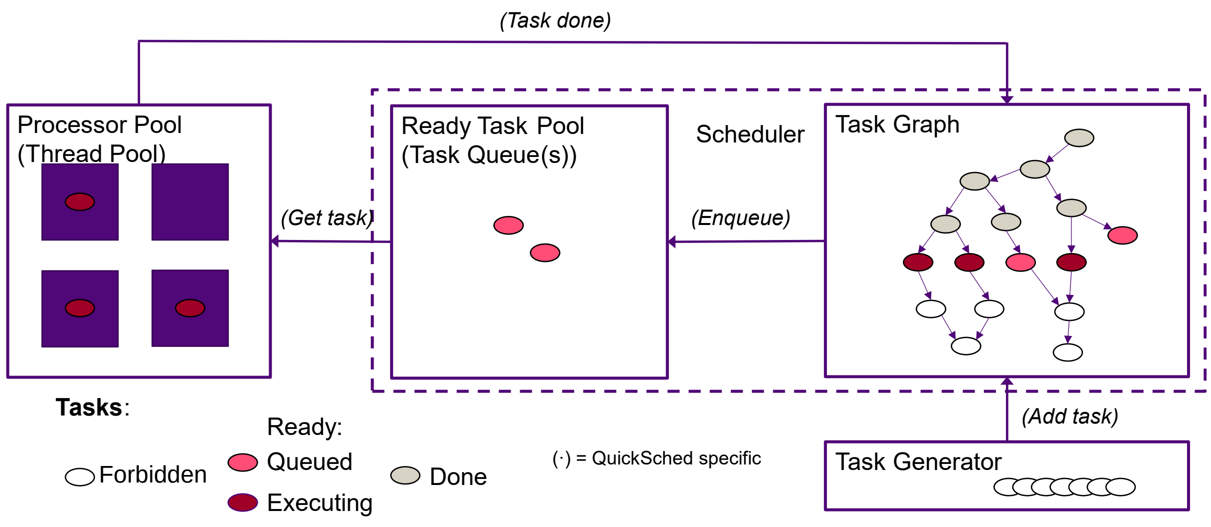


Figure 1 – Task-based parallelism

On the right is the task graph, shown in graphical form but standing also for an actual data structure representing it and that therefore having physical location on some computer. In the graph the nodes represent the tasks, and the directed edges represent the dependencies between the tasks, pointing from preceding to subsequent tasks. Acyclic task graphs are considered (a task graph with a cycle has tasks whose dependencies can never be fulfilled). On the left are a set of individual processors, labelled a *processor pool* to which the tasks are allocated, one task at a time to any particular one of the processors. The *thread pool* designation in brackets is in accordance with the implementations used in this project, as is explained later, but in general *task-based parallelism* may be applied to sets of other kinds of processing elements; so, the term *processor pool* is apt for this more general definition. (For a different example, at a much larger grained scale than in this project, each single processor element to which a task may be assigned could even be a different (and dynamically formed) set of hosts in a HPC cluster, with the *task graph* comprising task nodes that are jobs that are (i) defined in the cluster job scheduler and (ii) that are linked by job dependencies. While most users usually use jobs without dependencies, these facilities do exist [8].)

The *task-based scheduler* of the Figure, which is the dotted box, is an object[1] that operates to carry out the *task-based scheduling*, allocating further tasks of the *task graph* to the *processor pool*, making use of a *task pool,* which is a collection of tasks that are ready to be executed but which have not yet been allocated to a processor. In the diagram this *task pool* is labelled task *queue (s)* in brackets in

[1] The term object is used in this thesis in a very general sense and is not intended to imply an instance of a class in some particular object-oriented programming language, for example C++. The "objects" in this diagram merely have some state, some associated operations to update that, and some messages that pass between them. Extra object-oriented programming features such as encapsulation, inheritance, polymorphism and so on by, for example, C++, are not implied. The usage here therefore is consistent with a particular one of the several definitions given in [94], namely that in the second paragraph, first sentence, first half sentence, which is *"In the object-oriented programming paradigm, <u>object</u> can be a combination of variables, functions, and data structures;"*.

accordance with the implementations used in this project, but again that level of specificity is not needed in this general discussion.

The *task generator* is an object or process responsible for constructing the representation of the *task graph*. In general, this is executed first to construct the *task graph* representation and then the task graph is run by the *task scheduler*. A task graph that is completed before it is executed, and not changed thereafter, is known as a “static task graph; a task graph that is added to whilst it is being executed is a “dynamic task graph”. The *task generator* is shown as separate from the *processor pool* but of course it does have a physical location in any particular implementation, and in many examples given here it exists in the same overall process as the *processor pool*, and that process there both first generates the *task graph* and then, second, executes its tasks.

The form of the *task graph* of course depends primarily on the algorithm that it implements, and it is the responsibility of the programmer to provide code that generates the tasks that will carry out the algorithm. Algorithms sometimes have internal parameterized choices and, if so, those may well also be reflected in the structure of the resultant *task graph*.

In use, there are cases where the *task graph* is generated and used just once and others in which the problem requires many task graphs applied in sequence, one after another. A reason for breaking the processing up into alternate periods of task generation and task execution is that at some point it may be sensible to change either which tasks to have in the next phase, or their arguments, in response to values calculated by the last phase.

2.1. Converting a program to task-based parallelism

While a manual of requirements and strategies for a programmer to analyse an algorithm and to implement *task-based parallelism* would be a highly useful document, it is not the focus of this thesis, which concentrates on the efficiency of the operation of the *scheduler* machinery of Figure 1 to both build and execute the *task graphs* and this for fine-grained tasks within a high-performance computational code.

Nonetheless, it is instructive, in order to understand *fine-grained* tasks, to consider briefly how a programmer may go about the analysis of an existing serial computational program into such tasks.

The first step is to identify computational *kernels* in the serial code. A kernel has sets of data that are input and output; while that may be said of any section of code, the programmer will want to find sets of data that are easily recognisable or defined, and that preferably are already related to a defined interface in the code, e.g., a function call. These kernels will cover all the operations of the computation with nothing left out. The optimal size of these kernels is discussed in section 2.2 and in the results in section 13.10.

Those kernels will become the tasks of the task graph. If a kernel is called more than once, each call has a respective node in the task graph, although with different arguments for the kernel assigned to it. An edge between two tasks in the graph represents the dependency between the output dataset of one of the tasks and the input datasets of the other.

A useful tip is that the programmer has to keep clear in their mind the distinction between (a) the abstract datasets passed between tasks and (b) the physical areas of storage (working memory or backing store files) used to represent those datasets. In the practicalities of programming, physical areas of storage can store different versions of the same dataset because, for efficient programming, datasets often get updated, rather than a complete fresh new dataset being written to fresh memory to provide the output. Mistakes will happen if the two are confused. The input to a task is a particular version of a dataset.

The serial computational code to be converted will have control structures surrounding the kernels. Loops having a fixed number of passes are easiest to deal with. Each kernel inside the loop has a node in the task graph for every pass through the loop. The edges between them will then connect with nodes outside the loop or with the correct ones of the nodes in the same or other passes of the loop, which can be worked out by considering the arguments of the kernel functions and the relationships between those arguments from different passes through the loop, particularly with respect to which data items are referenced as a result of the values of those arguments.

How conditional control structures may easily be dealt with depends on the length of their branches. A single task may of course contain an entire conditional control structure within it, although handling predictions of how long it will take (for the purpose of scheduling tasks) will become more complicated. Decisions about large sets of tasks which may or may not be executed may have to be left to defining new task graphs in a cycle of defining task graphs and executing them. For an intermediate sized conditional control structure, one could define each branch as a task and include all those in the task graph, but each such task as a first step in its operation checks the condition for the branch to be executed but if it is not met the task then does no actual computation but returns immediately reporting that the task is complete. This of course incurs the overhead of the scheduler calls and so should not be done too often.

2.2. Effect of task size

The size of the tasks, so the size of the chunks of processing and of the chunks of data that are processed, are to be determined by the programmer. If the chunks are too small, there will be many tasks creating a large overhead of the scheduler’s processing of the tasks. If the chunks are too large, this will result in inefficiencies. First, there might not be enough ready tasks to keep all of the processors of the *processor pool* supplied. Second, the data may be too large to fit in the processor cache. The latter is a particular concern of QuickSched, the tasked-based scheduler used in the implementations of this project.

2.3. Executing a task graph

Once the *task graph* is generated, the task scheduler proceeds to execute it. As the processing proceeds, the scheduler keeps track of progress by identifying *ready tasks*, those that have their dependencies fulfilled. This process has to be started with those tasks that have no dependencies, since starting elsewhere would of course violate the stated dependencies. These initial tasks can be identified by the scheduler searching for them in the *task graph*, or perhaps by nomination by the *task generator*.

These initial *ready tasks* are recorded somehow as the *task pool*, the set of such tasks. *Ready tasks* found subsequently are included at that point in this same *task pool*. Again, the *task pool* will, of course, have a representation with a physical location.

The next function of the scheduler is to allocate such tasks from the *task pool* to the *processor pool* to be executed. Only one task at a time is assigned to an individual processor. Therefore, not only does the scheduler need a means of establishing a new task on a processor so that it may be executed there, but also a means of determining that it has finished processing the task. The arrows labelled *get task* and *task done* respectively represent those two, but again the specificity of their wording belongs to *QuickSched*.

When a task is allocated to a processor, that fact has to be recorded so that the scheduler does not allocate it again, which for example may be by marking its representation with that status, or as shown here by removing it from the *task pool*. (The Figure illustrates the task pool as containing just queued, but not executing, tasks.)

Once the execution of a task is complete, the scheduler uses that information to determine, which, if any at that stage, of the *forbidden* tasks, i.e. those that do not yet have all their dependencies fulfilled, now do, and so will now become *ready tasks* and be added to the *task pool*. This adding to the *task pool* is marked *enqueue* in the Figure; again, this name is in accordance with the QuickSched scheduler implementation.

The process of tasks completing execution, adding to the task pool, and allocating ready tasks to processors continues until all the tasks in the graph have been processed.

3. Background and other work in task based parallelism

3.1. What gets accelerated, and where?

Where to process data and whether to provide more specialised hardware than a general purpose processor is a long-standing theme in computer design.

One long-standing aspect of this has been what to put in the central processing unit (CPU). So, for example at one time microprocessors did not have floating point hardware and later they did, with an intermediate stage of putting the floating point hardware in a separate coprocessor integrated circuit (IC) mounted on the motherboard close by to the main CPU. The process has continued and general purpose microprocessors have since gained further arithmetic hardware, for example vector instructions.

Having multiple processors in the same machine is another long-standing theme, but of course technical goals, economics and other motivations affecting it have changed. In recent years multi core microprocessors are the norm and standard servers have two microprocessor packages having access to a common RAM. In that arrangement, these cores have been identical but now these have become cores of different strengths incorporated in the same integrated circuit.

Having different processor ICs of disparate kinds in the same machine to perform different kinds of computing task is also done. Graphics processing units (GPUs) were originally designed to carry out rendering for video displays but are now an established part of the arsenal in scientific computing, being used, for example, for machine learning calculations since the GPU hardware is particularly suited to that. The delivery of data between these processors then becomes an issue; should they have their own RAM and should each processor have some, or equal, access to the RAM of the other processor.

Other special processing hardware of a mathematical kind includes hardware for encryption, data compression, or searching with regular expressions. Computers of course also need to interface with the outside world, with storage devices and with other computers in a local area network or further afield. Those may be provided with their own separate integrated circuits but for some applications are included with the CPU, e.g. a “system on a chip” (SOC).

All these components, and their vendors, form an ecosystem, and as it evolves and the applications they support change, the combinations and locations of all these elements changes, to solve the problems in and of the current context. The BlueField card, and other smart NICs, is one such development. These are designed with particular goals, but where there are new arrangements of hardware there will be new opportunities.

3.2. Offloading to the network

As mentioned, the present work made use of BlueField cards. These combine, in a single card, Nvidia Network’s Connect-X network adapter with an Arm architecture processor, together with various specialised acceleration hardware circuits.

Nvidia Networks, and Mellanox before them, stress the idea of “offloading” of operations from the main CPU to the network. There is a hierarchy of the kinds of operation offloaded.

At the lowest level are simple operations needed for the transmission of data. So, for example in MPI messaging there is the process of tag matching. A tag is a user defined subject matter, or purpose, marker attached to a message, and the receiver can choose to receive messages with particular tags; so this is for a particular section of a code to receive messages that it is dealing with. While this sorting out of messages was originally conducted by the host CPU, the tag matching function is now done in special hardware in the network adapter [9]; there is therefore no need to interrupt the host CPU, which may be left to its other processing.

RDMA [10] allows a network card, on being issued with an instruction to transmit data in the host RAM to the network, to do that using a Direct Memory Access (“DMA”) engine on the network card to access the RAM.

At the next level, the network card carries out more computation operations. Encryption is another feature that may be offloaded [11]. Clearly it is directed at data transmission, the security of it, so is conceptually part of the business of a network card, but it also offloads from the CPU a more computational operation, the encryption.

Nvidia's SHARP™ technology [12] offloads the organisation/calculation of OpenMPI collectives to the network cards in a cluster ("MPI collectives" are operations distributing and/or processing data from or to multiple points in the network [13]). If left to the host CPU, that would be interrupted many times to send the various messages involved and, for certain collectives, to calculate the data reduction. So, this offers primitive (as in building block) **cooperative** data processing **between the nodes** of a cluster.

Above that level, Nvidia have now provided, in their BlueField cards, a general purpose Arm architecture CPU subsystem. Being general purpose, it could, of course, in principle, offload any kind of work; thus, there is wide variety of opportunity to consider what that might be. At an extreme, an early proposed application of BlueField [14] for a storage server did away with the host CPU entirely and connected the flash storage of the storage server to the BlueField, which served the data stored to the network over its integrated network adapter.

This work, however, aims to provide a cooperation between the BlueField card and the host. This is not primarily to do with low level networking, nor with offloading computation *per se*, but to fulfil the brief as noted at sections 1.2 to 1.4 – to control a computation "from the network".

3.3. Background relating to task-based scheduling

A general outline of what task-based scheduling is has been given in Chapter 2 and the importance of QuickSched in the context of the SPH code Swift to its adoption in this project was noted at 1.3. Wider aspects of scheduling with QuickSched, and of Swift, and other current efforts in task-based scheduling are now discussed.

3.4. Prior uses of Quick Sched – Chalk and Swift

QuickSched is a *fine-grained* task-based scheduler developed by the contributors Pedro Gonnet, Matthieu Schaller and Aidan Chalk and is available at their Source

Forge repository [15]. Aidan Chalk in his PhD thesis of 2017 [16] makes use of QuickSched. It has also been adopted in the Swift smoothed particle hydrodynamics (SPH) code [7].

Chalk's thesis begins with a survey of existing task-based libraries such as Cilk, Intel Thread Building Blocks, SMP Superscalar, StarPU, Quark and DAGuE. The author states that the advantages of QuickSched include: efficient use of the multiple cores of a processor because, as long as there are enough queued tasks, an idle core may be assigned one; the use of a *static task graph* means that the critical path through the task graph is known and so can be used in scheduling decisions; it pays attention to "data locality", i.e., aims to use data in the processor cache. He also notes the full task graph declaration as a disadvantage for the programmer, particularly for conversion of an existing code, and notes that for linear algebra a full task graph can be large, *"$O(n^3)$"*. He sets out a hope for QuickSched, of using it in shared and distributed memory systems and in accelerators, such as GPUs and aims to establish its use on GPUs, with perhaps a departure from GPU normal programming in that the GPU performs, *"many [QuickSched] tasks in parallel inside a single GPU kernel."*

For DAGuE he notes that it provides cooperation between compute nodes to transfer data needed between them, which is administered by a separate thread, and that there are control messages between nodes about the completion of tasks.

He also highlights for QuickSched that it supports "conflicts" – the data dependency of *"the concurrent updating of [a] resource"*. Another noteworthy point he makes is that task queues in QuickSched are cheap to maintain, being heaps (i.e. the tree data structure – see, for example, [17]), or priority queues, but that these do not have an efficient way for traversing them when looking beyond the head task of the queue, although he asserts that it is good enough for the purpose.

Amongst other uses, Chalk [16] reports use of QuickSched in Swift and a use on a GPU of a tiled QR matrix decomposition. The algorithms are analysed for the tasks needed to carry them out. In his Chapter 6, in a molecular dynamics application, task queues are implemented on the GPU, data movement to and from the GPU is modelled as tasks, and load balancing for distributed memory systems, i.e. the distribution of tasks between nodes, is discussed, [16] at 6.3. In particular, a graph (not a task graph) is made, with nodes representing the cells that contain the particles of the simulation and edges between cells for which there is interaction in the simulation, weights are assigned to represent the work to be done and the METIS graph partitioning program [18] is used partition the graph, thereby also partitioning the data, i.e., the cells, between compute nodes of a cluster. Chapter 7 of his thesis deals in more detail with task-based parallelism on distributed memory systems. Interestingly for one of the suggestions in Chapter 14 of this thesis, he asserts, *"as techniques such as work stealing is not efficient between nodes, as nodes would need to pass multiple messages back and forth to perform work stealing"*. In that Chapter 7, he also introduces send and receive tasks to transfer data between nodes at appropriate times. He adopts an organisation where all nodes know all of the tasks and the location of all of the data, but each node only has part of the data. Tasks are created on each node, but these are then consolidated for the complete task graph to be partitioned between the nodes. That leads to a partitioning of the data. A rule for efficiency is that tasks and data are kept together to the extent that a task must have one of its data resources on its node. But there are of course tasks that have different parts of their data on different nodes. So finally send and receive tasks are created on the nodes to service such tasks; however, an efficiency is that the versions of the data resources held by the nodes involved are considered and data transfer tasks are only created where the version held is not up to date.

This arrangement of tasks for data movement between nodes' tasks was kept in mind and forms an element of a suggested arrangement for data movement under the control BlueField cards made in Chapter 14 of this thesis.

In an early paper, 2013, on their smooth particle hydrodynamics code [19], Gonnet, Schaller, Theuns and Chalk introduce *"task-based shared memory parallelism"* to improve *"parallel scaling and efficiency on multi-core architectures"*. The paper begins by discussing aspects of the algorithm such as organisation of the particles of the simulation into trees and cells, finding near neighbour particles in the simulated space as a precursor to calculating the physical interactions. The limitations of OpenMP, then *“arguably the most well-known paradigm for shared-memory, or multi-threaded parallelism”*, are discussed. The points made include the “branch and bound” problem of OpenMP parallel *for loops* – each parallel thread generated to process the *for loop* in parallel will terminate at different times but then have to wait for a synchronisation point at the end of the loop and then there will be a serial section wastefully using only one thread before the next parallel section of the code begins. They propose a switch to task based processing, since it is *“inherently parallelizable”*. It is a feature of task based processing, however, that tasks on different threads may overlap and not have to begin and terminate at the same time, and further, a new task can be allocated to a thread immediately once the last has terminated. They mention the allowance for the *“conflicts”* type concurrency in their task graph model. Also here, the feature of a *“ghost task”* is mentioned, which is a single task between the phase in the computation of calculating the density of material in the simulation and the phase of calculating the forces on each particle, which of course depend on the density; the ghost task provides a point of synchronisation of all the threads between these phases. (In his thesis Chalk mentioned that this was for reasons of reducing the number of task dependencies, but of course it is also a synchronisation point in the calculation.) The application of tasks to the algorithm was then discussed and the results of using the resultant code on well-known simulation test cases reported, including a high parallel efficiency of 75% and being 8 times faster for a cosmological volume simulation than the well-known Gadget-2 code. Finally, they foreshadow the developments of the Swift code to shared/distributed memory parallelism and communication between compute nodes.

In a 2015 paper [20], Gonnet covers similar material to Chalk’s thesis in respect of smoothed particle hydrodynamics and the task-based approach across multiple compute nodes. I found motivation in Chalk’s Figure 7.7 and Gonnet’s Figure 12, which both show the substantial discrepancy between nodes for the time at which they finish their set of tasks for one time-step of a cosmological simulation. Clearly there needs to be some further cooperation between compute nodes to even that out, so that compute time is not wasted before the next time step can start. Gonnet's Figure 12 has white space, which is *"time spent in the MPI_Test function."* This is part of the communication processing of data transfer tasks. It appears in that Figure that it is the compute nodes that have that feature that take longest to complete their timestep. In the context of my studies, it appeared to me that that part of the code should be offloaded "to the network".

In another paper [21] from 2015, Theuns, Chalk, Schaller and Gonnet discuss the types of tasks used in Swift to perform the density and gravity calculations.

In a first paper of 2016, Gonnet, Chalk and Schaller [22], cover similar material as included in Chalk's thesis, although it does not include the multi node computations, nor hence the data transfer tasks. One notable, perhaps additional comment there is, on page 4, *"Finally, the task granularity is an important issue: if the task decomposition is too coarse, then good parallelism and load balancing will be difficult to achieve. Conversely if the tasks are too small, the costs of selecting and scheduling tasks, which is usually constant per task, will quickly destroy any gains from parallelism."* They continue as follows:

> *"In the examples presented herein, we have chosen our task decomposition and granularity such that*
> *● Each task maximizes the ratio of computation to the data required,*
> *● The resources required for each task fit comfortably in the lowest-level caches of the underlying system.*
> *The first criteria is biased towards bigger tasks, while the second limits their size. The parameters controlling the size of the tasks in the examples, i.e. the tile size in the QR decomposition and the limits nmax and ntask were were determined empirically and only optimised to the closest power of two or rough power of ten respectively. Further tuning of these parameters could very likely lead to further performance gains, but such an effort would go beyond the scope and point of this paper."*

During this work, the need to keep the task data in the cache was ever present in my mind, particularly from studying the workings of the QuickSched scheduler, where it is a key part of the task allocation rules. The significance of the rule is studied in this work. The optimisation of the task size is not stressed in this reference and is not discussed there in any detail; it is only said that there are performance gains for optimising in respect of it. An optimisation on task size is performed in this work and is studied in detail, particularly in relation to messaging, which is not mentioned *per se* here. The phenomenon noted in the brief remark in this reference of the costs of scheduling being *“usually constant per task”* does emerge in the study in this work of the effects of messaging.

They also show the progression of tasks from the scheduler into the queues and onto the threads and note that both the scheduler and the queues have roles in contributing to efficiency of execution and reiterate the importance of allocating tasks to threads that have memory resources in their cache. They also mention the need to follow the longest critical path through the task graph, ascribing the better performance of QuickSched over OmpSs to it.

In a second paper [23] of 2016, for the Proceedings of the Platform for Advanced Scientific Computing Conference, Schaller, Gonnet, Chalk and Draper presented their work on the Swift simulation using the task-based approach based on QuickSched and, in particular, presented comparative runs on various clusters to demonstrate the independence of the approach from the particular hardware used.

By 2018 development of Swift was still concerned with efficiency but had turned to matters other than task scheduling *per se*. In [24], Willis, Schaller, Gonnet, Bower and Draper used hand coded vector processor operations inside the tasks that sort the list to find the neighbours that interact in the simulation. In [25], Borrow, Bower, Draper, Gonnet and Schaller examined implementing multiple time step sizes in Swift.

3.5. Other relevant work in task based scheduling

In [26] (from 2020 so now contemporaneous with this work), Samfass, Weinzierl, Charrier and Bader, discuss their system for offloading tasks to another compute node. In the computation method of “Arbitrary high-order DERivative Discontinuous Galerkin”, they identify a class of tasks that may be sent to another compute node for processing, owing to the length of time before their results are needed. In this scheme the tasks are processed on the remote node with high priority and the results sent back to the originating node as soon as they are completed; there is no long-term transfer of the task or its data. Nodes that are causing delays to other nodes are identified by MPI waiting times, and these are designated to be in need of help and are the source of the tasks offloaded. The tasks are offloaded to nodes that are most able to take up more work without decreasing their performance, again as indicated by MPI wait times.

In [27], also from 2020, Samfass, Weinzierl, Hazelwood and Bader introduce their TeaMPI code for task-based codes. This addresses the problem of failure of hardware elements of large clusters by replicating task data; each node in the cluster having a small number of replica nodes. The replicas are in general slated to run the same tasks, but the order is shuffled between them, and results of a task (or particularly, for preference, of tasks with high amounts of computation but compact results) are distributed to the node’s replicas. If a node fails, the computation may continue. On the other hand, to avoid duplication of execution of a task, if a node has received the results of a task that it now wants to schedule for execution, it recognises that and cancels the task and moves on using the received results. This paper ends, *“…, we plan to investigate whether emerging technologies such as SmartNICs can be exploited to offload the task sharing fully to the network hardware and to guarantee sufficient MPI progress.”*

The Excalibur research program [28] funds projects needed for computing on exa-scale clusters; in their words, *“redesigning high priority computer codes and algorithms to meet the demands of both advancing technology and UK research.”* According to their website one project relates to task-based codes: “Exposing Parallelism: Task Parallelism”, [29] is a response to the difficulty, or at least labour, noted by Chalk and at section 2.1 above, of converting codes to task-based ones. A tool, called Otter [30], has been developed which allows a programmer to annotate their code at instances of task like structures and when the code is run the annotations are logged as they are passed. An analysis part of the tool provides a graph of the events and provides suggestions of how to "taskify" their code, at the level of OpenMP pragmas to add. Clearly such a tool will be useful both practically and psychologically to overcome the perceived difficulties and as a teaching aid. This work uses the QR factorisation, discussed later, as its example task-based code; this has a clear task structure, but such a tool will help with codes for where the task nature of the algorithm is less clear.

4. Hierarchy of software components in this project

The diagram of Figure 2 shows, in the left column, the dependencies of the software components used in this project, while the right column provides references to the main chapter or section of this thesis where the component is discussed. The libraries highlighted in blue are either heavily adapted from an existing library or are completely new.

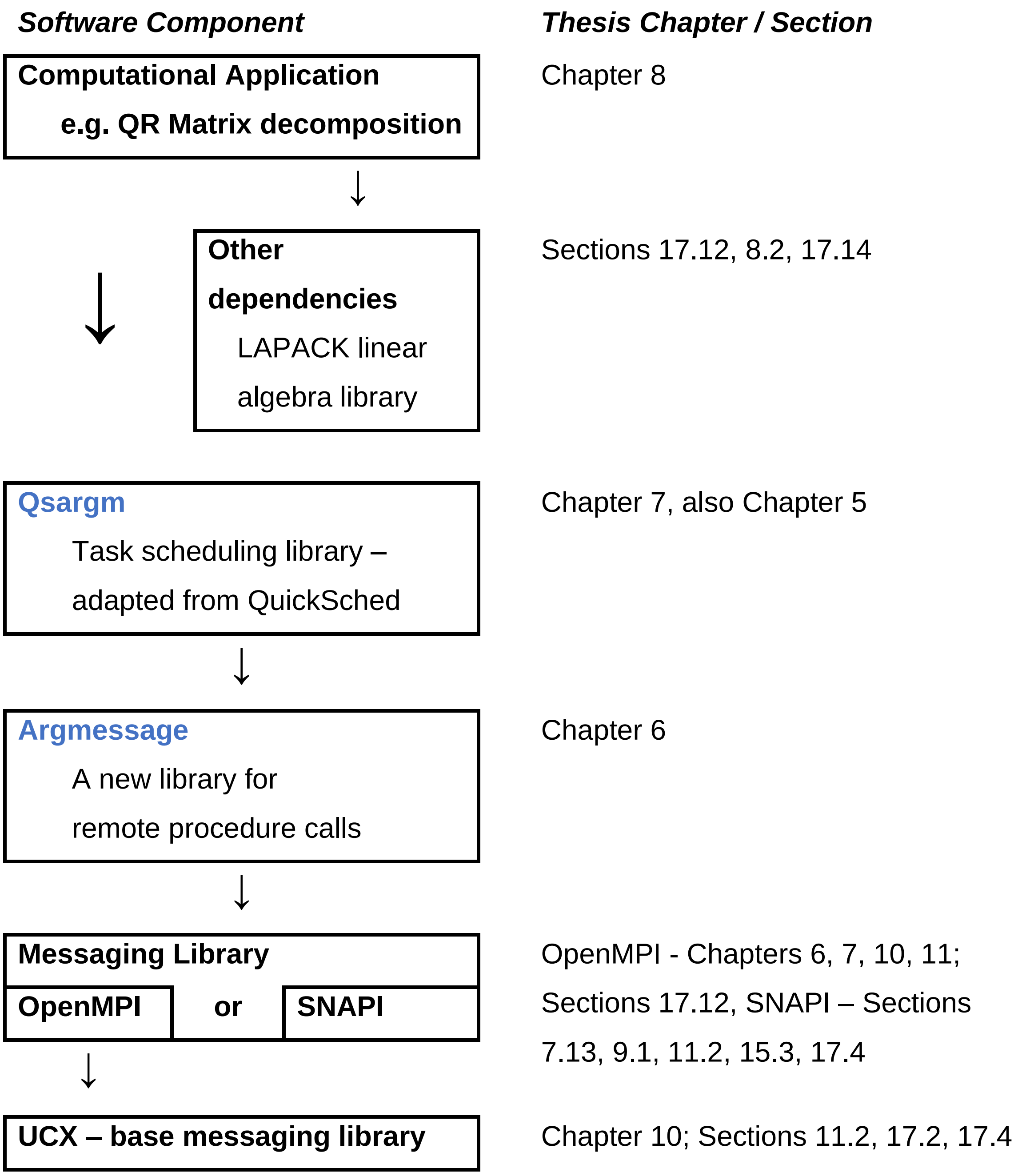

Software Component
Thesis Chapter / Section
Computational Application e.g. QR Matrix decomposition
Chapter 8
Other dependencies LAPACK linear algebra library
Sections 17.12, 8.2, 17.14
Qsargm Task scheduling library – adapted from QuickSched
Chapter 7, also Chapter 5
Argmessage A new library for remote procedure calls
Chapter 6
Messaging Library
OpenMPI
or
SNAPI
OpenMPI - Chapters 6, 7, 10, 11; Sections 17.12, SNAPI – Sections 7.13, 9.1, 11.2, 15.3, 17.4
UCX – base messaging library
Chapter 10; Sections 11.2, 17.2, 17.4

Figure 2 – Software dependencies in this project

Thus, a computational application uses the Qsargm library to perform its task-based scheduling and may of course have other dependencies. Qsargm uses the Argmessage remote procedure call (RPC) library for communications between its client and server processes. (The terms “client” and “server” are used herein with their normal meaning as used with RPC and similar arrangements – the “client” process makes a request, or “calls”, to another process, the “server”, which is then responsible for carrying out the request, and which then returns an answer to the client.) Argmessage sends messages between the client and server with either the OpenMPI or SNAPI messaging libraries. OpenMPI has options for a base messaging library; in this project the UCX [31] library was used.

Omitted from the software diagram are the scripts used to launch the application program (which includes the Qsargm/Argmessage client) and the Qsargm/Argmessage server process. These scripts are not trivial since multiple programs are to be launched on heterogeneous processor architectures. The scripts are explained at sections 17.2, 17.3 and 17.14 in Appendix A.

4.1. Plan for discussion in subsequent chapters

The discussion of the relevant ones of these components does not, however, proceed in that order, but rather, as follows.

First, the original QuickSched task-based scheduling library is analysed to find how it might be split into client and server processes. Next comes the features and development of the new Argmessage library, which allows communications between such a client and server, but which both enables the library using it, to hide that from the application using it, and promotes reuse of functions of a library being adapted to use Argmessage. That then allows an explanation of how QuickSched was adapted to use Argmessage. Experiments were carried out on a QR factorisation example (originating from the QuickSched library archive) and so that example is explained before, finally, the results of the experiments are presented.

5. Analysis of QuickSched

Chapter 2 described some general features of task-based scheduling. As foreshadowed, this chapter now analyses in detail the particular task-based scheduler, QuickSched, of which use is made in this work, to identify how and to what extent it might, in principle, be offloaded to a BlueField card. The modifications made in this project to the QuickSched library based on this analysis and the practical results of those are discussed in the later chapters.

5.1. Origin and uses of QuickSched

As discussed in Chapters 1 to 3, QuickSched is a *fine-grained* task-based scheduler developed by Pedro Gonnet, Matthieu Schaller and Aidan Chalk. My code developed for this project uses their code from their Source Forge repository, in particular version 1.1.0 [15]. A version of the original QuickSched has also been incorporated into the Swift smoothed particle hydrodynamics astrophysical code [7], but with some other modifications. This analysis is based on QuickSched 1.1.0 as available on Source Forge, and on the corresponding description in Chalk's doctoral thesis [16], as well as on my own analysis of those.

5.2. Timescale of tasks

QuickSched is a scheduler library to be used in an application program to schedule units of work, "tasks". At a general level it has the organisation set out in the earlier chapter on task-based scheduling. The tasks of QuickSched are quite general but in its target use area, of scientific computing, the most common type of task is some part of the computation. QuickSched is designed for *fine-grained* tasking, meaning that the duration of the tasks is on a timescale suitable for the efficient scheduling of a large computation. In one example in the experiments described later the mean task execution time was around 1×10^{-4}s, and there the task size was designed around the processor cache size, which is a common requirement in computational applications. Table 1 is of some typical timescales for scheduling (not just *task-based*) for context.

Example of scheduling	**Timescale**	
A scripted data processing workflow having tasks that are each the run of an application program	Minutes to days	
Operating System process scheduling	10 ms	[32]
QuickSched *fine grained task-based* scheduling	0.1 ms	
Processor instruction hardware scheduling	1 ns	

Table 1 – scheduling timescales

So, this gives an indication of the timescale in which QuickSched expects to operate. On such a timescale it is clearly a challenge for QuickSched to take its decisions and establish new ready tasks on a thread as fast as possible. Its design certainly reflects that.

5.3. Build and execute cycle

A first thing to note about QuickSched is, obviously, that everything takes place in the same process (as defined, for example, in a Linux system).

The *processor pool* of Figure 1 that executes the tasks is, in QuickSched, a pool of *threads* in that single process. The threads are executed on the cores of one or more CPU integrated circuits in a shared memory fashion. The QuickSched scheduler operates to assign tasks to these threads. Interaction between the main program of the process and the QuickSched library is via C function calls.

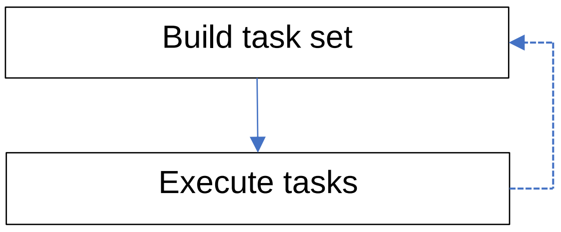


Figure 3 – Main phases of operation of example programs of QuickSched

The example programs using the QuickSched library that are included in its archive follow the pattern shown in the flow chart of Figure 3, in which the program has separate phases of generating tasks and of executing those tasks. An application using QuickSched may use the two phases just once or repeat them. (In fact, QuickSched does allow dynamic task creation during the execution phase, but that did not feature in anything considered here.) Thus, the *task generator* of Figure 1 is also to be identified as belonging to the same process as the process that executes the tasks themselves.

5.4. Time slicing of the scheduler object

The library maintains a scheduler object, to keep the state of the *task graph* and the *task pool*, etc., and when a call is made by a thread that thread becomes the scheduler for the period needed to process the call. Thus, in Figure 1 of Chapter 2, the *processor pool* operates both the execution of the tasks and as the scheduler. This is at the heart of QuickSched's design, and because during the execution phase operating the scheduler takes processing time away from executing the computational kernels the scheduler is designed to be as efficient and lightweight as possible.

This time slicing of the processors of the *processor pool* between executing the tasks and executing the scheduler was a feature I found striking when reviewing QuickSched to see what might be offloaded to a BlueField, and this task execution phase has emphasis in this project.

5.5. Location of task generator

That the *task generator* was part of the same process as the task execution and scheduling was more implicit and not initially questioned. The original QuickSched has, of course, to be that way since (i) it is a library that operates by function calls and (ii) the task building calls to the library exist to create the representations of the task graph and task pool within the single process so that they are then ready there to be used when the function calls to the scheduler are made during the task execution phase.

Figure 1, while showing the general arrangement of task scheduling, was of course drawn, after the modifications of this project had been made and the experiments run, in a manner emphasising that the scheduler and task generator are actually separate concerns from executing the tasks. The modifications made in this project concentrated on the task execution, but the task generation phase is also important as the experimental results were to show.

5.6. What to offload to the BlueField card

Having seen that the scheduling decisions and executing of tasks are separate concerns in QuickSched, the operations of the scheduler were studied in more detail to see which parts of the scheduler might be relocated to the BlueField. The objective being to relieve the host of the work of executing the scheduler, it was decided to leave the execution of the tasks on the host x86 processor, (There were also some important motivations for having the scheduling data on the BlueField, which are about what else could be enabled by that; those are discussed in Chapter 14.)

Now, offloading the operation of the scheduler to the BlueField card would require some messaging between the host and the scheduler. Such messaging of course introduces delays and so should be kept to a minimum. Therefore, this study considered (a) the data structures employed by the scheduler, since a decision about where to keep them, on the host or the BlueField, would be needed, and (b) the flow of information or instructions about them, which might be turned into messages.

5.7. QuickSched data structures: task DAG and ready task queues

As in the general chapter on *task-based* scheduling, Chapter 2, QuickSched uses a *directed acyclic graph* (“DAG”) as the data structure for the task set created in the first phase. The nodes of the graph represent the tasks to be done and the edges represent the dependencies between them, i.e., a task must not be executed until its parent task(s) are finished.. With the caveat of the conflicts mechanism noted in sections 3.4, 5.10, 5.16 and 9.4, this *task graph* provides, in general, full information as to the relationships between the tasks of the application and how they could be executed serially and in parallel whilst ensuring that the resultant computation is **correct**.

It is also the job of the scheduler, in the second phase of executing the tasks, to work through the tasks in an **efficient** manner. With the same caveat, the data structure that QuickSched employs for this efficiency aspect is a set of queues, one per thread.

The number of threads is intended in QuickSched to be equal to the number of cores allotted to run them, with each thread being pinned to a respective core. In this way QuickSched takes responsibility for the scheduling, while neutralising any, potentially competing, attempt by the operating system to schedule threads on and off and to different cores.

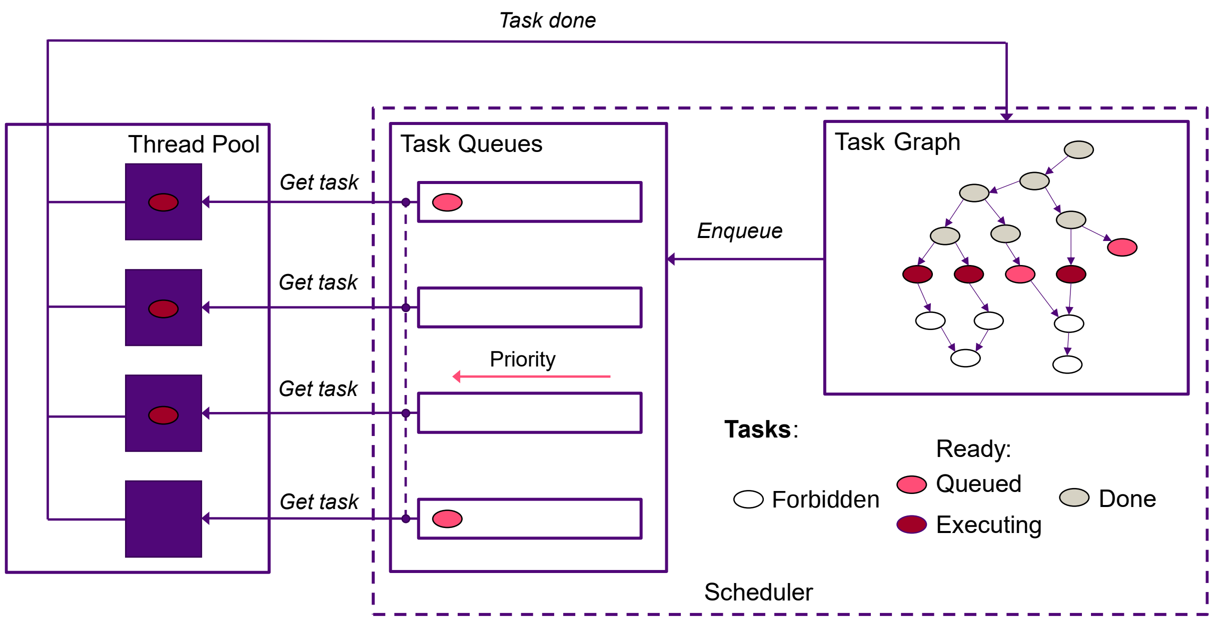


Figure 4 – QuickSched scheduler – data structures at task exectution time

Figure 4 shows the task graph and task queues of QuickSched, so in more detail than given in Figure 1. Although they have not been identified yet the messages between the data structures are shown in Figure 4; the details of this follow in sections 5.8 to 5.14, which discuss how they operate.

5.8. QuickSched: library functions and call graph

While extremely useful to a programmer wanting to maintain or improve code, a static call graph for a C code is not simple to construct [33] (see section 1.2 thereof). Commercial call graph analysers were not available to me, and in any event, I was able to compile the static call graph, Figure 5, for the library by inspection of the code and from the set of partial call graphs in QuickSched's documentation of each of its code functions. I have also added a colour coding for the functions, and some annotations.

Each box in Figure 5, apart from the orange ones, is a function in the library. The orange boxes are example programs that come with the library, that make use of it. As earlier, with reference to Figure 1, the example programs use the QuickSched library to first build a task graph and then execute it.

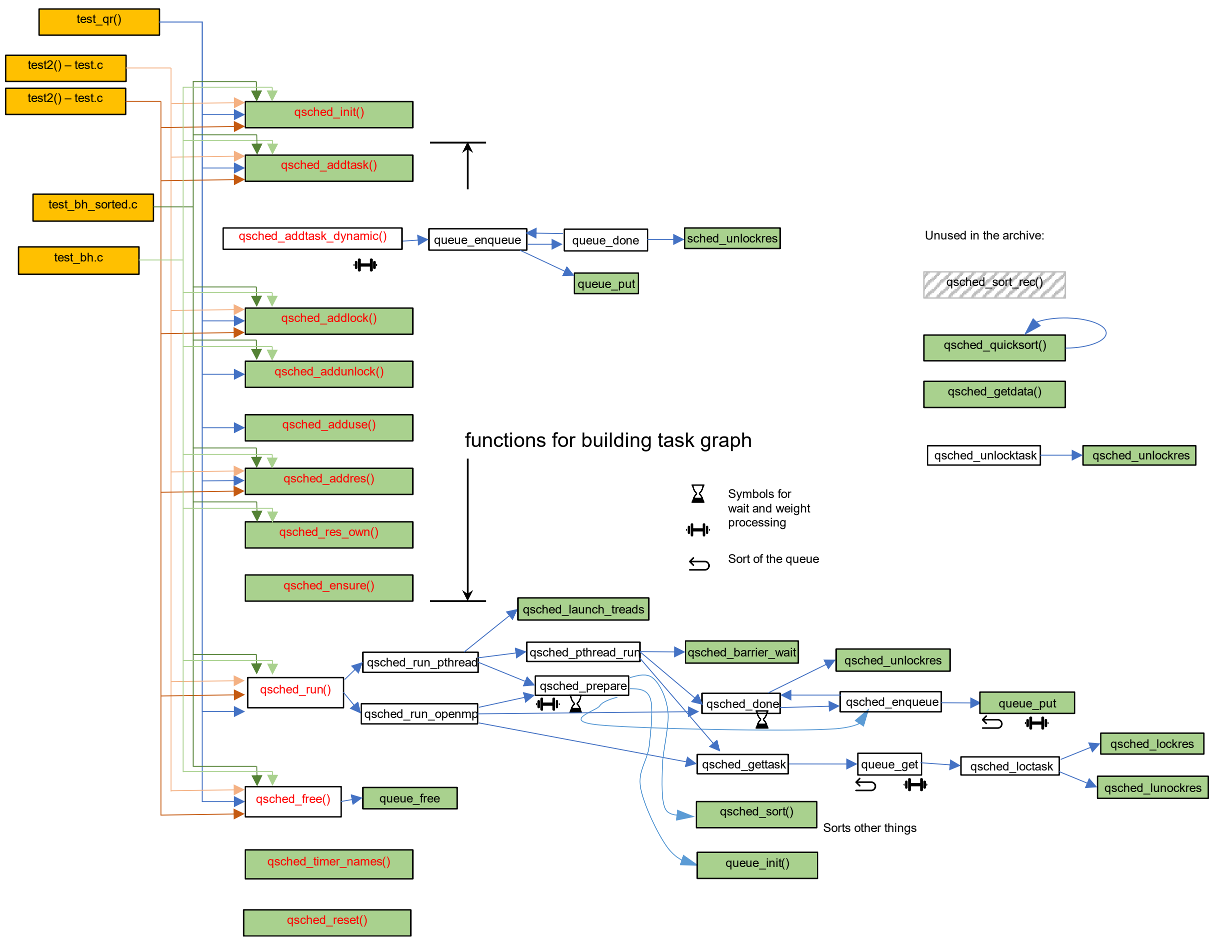


Figure 5 – Call graph of QuickSched

The items in the second column, with the function names in red, are the top-level functions, which form the public interface of the library. These fall into four general categories:

a) library initialisation and teardown:	*qsched_init(),*
	qsched_reset(),
	qsched_free()
b) functions for building the *task graph*:	*qsched_addtask(),*
	qsched_addres(),
	qsched_addlock(),
	qsched_addunlock(),
	qsched_adduse,
	qsched_addres(),
	qsched_res_own(),
c) the main function for executing the tasks:	*qsched_run(),* and
d) auxiliary functions:	*qsched_ timer_names(),*
	qsched_getdata().

Table 2 – public interface functions of QuickSched

It is notable that the interface has many functions (b) for building the task graph but only one (c) for executing it. The code for these functions may obviously be reviewed in the official archive. The corresponding functions used in this project are discussed in the chapter on Qsargm, the version of library produced for the project.

5.9. QuickSched functions for initialisation and finalisation

First the initialisation functions (a), are straightforward:

qsched_init() allocates memory for the scheduler's tables, which are:

- Resources
- Tasks
- Unlocks
- Locks
- Uses

It is these tables that provide the representation of the *task graph*. The role of the many functions of category (b) is to populate these tables with various aspects of the *task graph*. (Note that, as will become apparent from the descriptions of the category (b) functions in section 5.10, while the names “unlocks” and “lock” appear, just from the names, to be opposites, that is not the case, and they relate to quite different matters.)

qsched_reset() zeros the counts of the items on those tables, so that the allocated memory blocks for them may be reused.

qsched_free() deallocates the memory of the tables.

5.10. QuickSched functions for building the task-graph

So, after initialisation, the application program builds the *task graph* using the functions (b):

qsched_addtask() allows a new task record to be recorded. The data supplied and recorded is:

- A *type* identifying the type of work to be done by the task. In particular in QuickSched, this identifies a particular one of the kernel functions mentioned earlier that is responsible for that type of work.
- Some *data* allowing further definition of the task. In QuickSched, these are the arguments to be passed to the kernel function.
- A *flag* that may be *task_flag_none, task_flag_skip,* or *task_flag_virtual*.
- A *cost*, which is an estimate of the time it will take for the task to run. The units are arbitrary and relative to the other tasks. The scale is intended to be additive. This forms the basis of the scheduler’s selection of which task to allocate next to a thread.

qsched_addunlock() is called to register a dependency between two tasks. So, it is this function that links the tasks into the *task graph*. These are recorded as a table of dependency-depending task pairs. The “unlock” in the function name connotes that when the dependency task is completed it unlocks or frees up the depending task to become a ready task. Of course, a depending task may have plural dependencies, and all of these have to be completed before a depending task is “unlocked”. Chalk [16] briefly mentions (at section 3.1.1) the terms *Read after Write* and *Write after Read* for this task relationship. So, for where the relationship is about data, this is the usual type of relationship: if task B is dependent on task A, then task A must finish writing to a data item before task B consumes it and/or task B must not write to a data item before A reads it. *qsched_addunlock*() is used to declare that relationship. The book “Computer Architecture” by Hennessy and Patterson [34] describes the various kinds of data dependencies, in section 3.1 and appendix C; although that is in the context of instruction pipelining in processors, the concepts are relevant here.

qsched_addres() allows a resource to be declared. To the scheduler each resource is an abstract item; it is up to the user program to know what they represent. Typically, a resource is a set of data items or block of memory. The index to the resource that is returned by this function is used in other functions in this section to register relationships of the resource with the scheduler.

qsched_addlock() is for specifying that a task needs to obtain exclusive access to a particular resource. The scheduler operates, when allocating tasks for execution, to obtain a lock on this resource for a candidate task and does not allocate the task if a lock is not available. (The lock is obtained on the abstract resource and not on the data it represents, so the application programmer of course has to respect this.)

Now, the data dependencies between dependency and depending tasks are already represented with the *qsched_addunlock()* calls. However, according to QuickSched, those relationships can be too restrictive. If two tasks require access to the same resource and that access needs to be exclusive but the order does not matter, then this function *qsched_addlock()* may be used in respect of both those tasks and the resource in question to declare the relationship. If the resource is a data item, then that means that both tasks modify the data item, but the order does not matter. Chalk [16] terms this a “conflict” (at section 2.3) and gives an example, which is an accumulator data item into which both tasks add respective results. (Thus this declaration is needed for correctness of the algorithm, since a task dependency is not being used.) He notes that this more efficient for scheduling than declaring a dependency, i.e., with *qsched_addunlock()*, which would fix an arbitrary order between the two tasks, potentially reducing efficiency of processing.

The *qsched_addlock()* declaration also serves a purpose in the efficient ordering of tasks during the execution phase, beyond the freedom of order mentioned in the previous paragraph: *qsched_enqueue()* takes note of the resources so linked to the tasks in the allocation of tasks to a thread having tasks operating with the same data. This is in the hope that the newly allocated task will find at least some of its data in the processor cache.

qsched_adduse() this also records a *resource* against a task, this time as a *use*. This is also used in the *qsched_enqueue()* function to support the rules for the efficient allocation of tasks to threads. Here this is only for that purpose rather than as with *unlock* to declare a dependency needed for correctness. So, a data *resource* used by a kernel should be declared as a *use* if it is not declared as *lock* if the task allocation rules are to operate.

These functions therefore build the task graph representation. As noted earlier, initially the location of the task generator was not recognised as an issue. However, if task generation is to be processed in another process from that of the scheduler, then the information for the tasks will need to be transmitted between those processes as messages. As ideas about this were prompted by the experimental results, those are not discussed here but after those, in Chapter 14.

5.11. QuickSched functions for executing the task graph

So, the data collected in the build phase will serve three purposes during execution of the task graph: (i) working through the task graph identifying ready tasks, (ii) allocating tasks to queues having tasks using similar *resources*, and (iii) deciding which of the ready tasks in the queues is next to be executed.

5.12. Establishing the threads

The single function of set (c), *qsched_run(),* for executing the tasks of a task graph is shown in detail in Figure 5. At the most general level, this function executes the tasks of the task graph in a correct order, each on just one of a pool of available threads. So, its first job is to set up some threads to be available. QuickSched is capable of using threads provided by either Linux pthreads or OpenMP; hence the two functions *qsched_run_pthread()* and *qsched_run_openmp(),* which are next down the call graph from *qsched_run()*, in Figure 5*.* These are therefore alternatives, only one of which is used during any run. In the call graph setting up pthreads takes a set of functions: *qsched_run_pthread(), qsched_launch_threads(), qsched_pthread_run()* and *qsched_barrier_wait(),* while the OpenMP threads alternative just uses *qsched_run_openmp()*.

In QuickSched, the threads are pinned to respective cores, which discourages the operating system from engaging in its own, quite possibly competing, scheduling of the threads. The operating system's scheduling of threads may well include wastefully moving them around between cores and this is contrary to and aspect of QuickSched's operation, which is to allocate tasks to threads where the respective processor core already has data that is needed in its cache. (The pinning does not appear in the QuickSched code *per se*; it is left to the user to arrange that when launching an application that uses the library.)

5.13. Preparing to run

Both alternatives then have a common path. As its name suggests, *qsched_prepare()* makes some preparations before actually executing any tasks, which are:

- One preparation is to sort the various tables of items created by the various task declaration functions. These sorts are carried out in parallel on different threads.
- Another preparation is to initialise the queues of Figure 4, with *queue_init().*
- A *wait* value for each task is calculated, which is the number of dependency tasks it has, which is used in *qsched_done()*, described at sections 5.14, 5.15, and 5.16, for determining whether a depending task has become a *ready task*.

- A *weight* for the task is derived from the costs, which is used for the priority for allocation of *ready tasks* from the queues. In particular, the *weight* is calculated, as Chalk [16] describes (at section 3.1.1), as the *cost* of the task plus the cost of all descendant tasks in the task graph, which are therefore a hinterland of tasks that are being held up from processing by the present task.
- The final preparation is to enqueue the initial task(s), which are those that have no dependencies, which are the root(s) of the DAG. (This may be a single task, but the task DAG could have more than one root.) Enqueuing a task, with *qsched_enqueue()*, is described in more detail in section 5.16.

5.14. Running through the tasks

With the preparations completed, each thread repeats the cycle of (i) allocating tasks to itself of the *thread pool* and (ii) determining that the thread has become free for a further allocation.

The reason in QuickSched for using a pool of threads in the same process is so that all tasks may have rapid, direct access to all the data items of the computation in RAM, i.e. “shared memory” operation. Thus, the thread pool is an essential feature of QuickSched, and was preserved in the modifications made.

So, at this point each thread is running either *qsched_run_openmp()* or *qsched_pthread_run()* and this first calls *qsched_gettask()*. This call is made on each of the threads of the pool in parallel. If the call is successful, in that the scheduler is able to return to the thread in question, a task from the queues, that thread executes the task. *qsched_gettask()* returns a pointer to the task record, and the thread, in function *qsched_run_openmp()* or *qsched_pthread_run(),* uses this to look up the task *type* and task *data* (section 5.10) and then the function identified by *type* is called using the arguments *data*. In fact, in QuickSched the decoding of the *type* and *data* into a kernel and call and its parameters is delegated to a function that the programmer of the application program using QuickSched must supply.

Once a thread has completed the task, so now back in function *qsched_run_openmp()* or *qsched_pthread_run(),* it notifies the scheduler that the task has been carried out and is now complete by calling *qsched_done()*, allowing the scheduler, as described at section 5.16, to find more *ready tasks*. The thread is now free to carry out more work, so it calls *qsched_gettask()* again. This cycle is repeated until there are no more tasks to be executed. That that has occurred is determined simply by initially counting the tasks in the graph and keeping track of how many have been completed. When all are completed the active one of the two "run" functions terminates.

5.15. Proposal for splitting the scheduler

From this analysis it appeared that a suitable place to divide QuickSched, for the run phase, would be with *qsched_gettask()* and *qsched_done()* providing the interface to the remote scheduler, so that the functions *qsched_run_openmp()* and *qsched_pthread_run()* and those above them in the call graph would remain on the host, while those within would be on the BlueField.

There remain further details of QuickSched to look at to understand the implications of the proposed split (and to check on to see whether some other split would be useful).

5.16. QuickSched functions for getting tasks and reporting their completion

When *qsched_gettask()* is called by a thread, the scheduler removes a task from a selected one of the queues as follows. For preference, the thread takes a task from its own respective queue. Its own queue contains tasks that are likely to have their data in the cache of the processor core to which the thread is pinned. If a task is not available in that queue, the other queues that are not empty are identified and those queues are checked through once in a random order. This is termed *work-stealing*, and is well known; for example, it was used in [35], [36] and [37].

To query any particular queue *qsched_gettask()* calls *queue_get()*. That works through the queue, ideally in its priority order, checking if each task is locked, which will fail if any of its associated *resources* are locked, (thus implementing the conflict mechanism – sections 3.4 and 5.10). If there is a task that is not locked then that is the one assigned to the thread, after first locking its resources so that no other task may concurrently use its resources. That task is also removed from the queue and the queue is resorted according to its priority order, which prioritises tasks which have the longest path to completion of the *task graph*, which information is embodied in the *weight* value for the task, which was calculated by *qsched_prepare()*.

In Figure 4, the detail of each thread querying its own respective queue first is shown by having a respective arrow from the thread to its queue. The subsequent search of the other queues when necessary is indicated by the dotted line tying those arrows together.

When *qsched_done()* is called by a thread, the scheduler marks the task as complete and unlocks its resources so that that does not prevent other tasks from running. It also reduces each counter belonging to its depending tasks of its remaining dependencies. It then determines therefore which of the tasks in the task graph are now therefore *ready tasks*, i.e., have no more dependencies to be fulfilled. *qsched_done()* then adds the tasks identified as *ready tasks* to the queues by calling *qsched_enqueue()*.

qsched_enqueue() assigns the task to the queue that has the highest number of *resources* related to it by the queue's current tasks, (related by both the *locks* and the *uses* – sections 5.9 and 5.10*)*. A resource related to plural tasks in the queue is counted that many times. In turn *qsched_enqueue()* calls *queue_put()* to actually assign the task to the queue selected. That first waits to get a lock on the queue, because there may be concurrent requests from other threads to modify that particular queue. It does this by adding to it to the end of the queue and *"Bubbl[ing it] up"* to its position in the queue as determined according to the *weights* of the task in the queue. Therefore, the weightiest get processed for preference, since they are holding up a greater amount of work.

The queue is a priority queue, in the form of a heap (i.e. the tree data structure – see, for example, [17]). This only guarantees to put the highest priority item at the front of the queue. This is fine if that can be removed, but if it is unavailable through locking, then *qsched_gettask(*), will not in general allocate the most *weighty* task that is not locked, because in that case it works through the tasks in their physical order in the memory of the queue until it finds one that may be allocated. Chalk [16] notes this (at section 3.1.2), but states it is *“sufficient for efficient task-based computations”*. To try to remove a task from a queue *qsched_gettask()* calls *queue_get()*. When that finds a task that may be removed it does that and then sorts the queue as a heap by *weight*, similarly to *queue_put()*.

It is also notable that the scheduler does operate in a multithreaded manner, so calls to it from different threads can be processed concurrently. However, certain functions require a lock to be obtained. For example, updates to a queue require a lock on that queue. Potential concurrent updates to some counters are handled with an atomic decrement function, for example, the global count of tasks not yet completed and the count each task has of it the dependencies that remain to be fulfilled.

5.17. Further discussion of the proposed split

The proposed split leaves to the BlueField all of the processing carried out by the scheduler to determine the next task. This includes both that for finding the *ready tasks* from the *task graph* and making the choice from the *queues* of which task to process next, so that that will result in efficient processing of the tasks as a whole. The objective was to offload processing of the scheduler, so leaving all that to the BlueField is in accordance with that. Further, the split only requires two small messages between the host and the BlueField per task, those for *qsched_gettask()* and *qsched_done()*. The number here is important as replacing a function call with a message will introduce significant extra latency. The smallness is less important in that extra information could be added to a message without much extra overhead, but it also suggests that compiling the messages is unlikely to require much additional work to derive their content.

The proposed split also keeps the scheduler data, the five tables identified earlier in this chapter, in the same place as each other, so their processing will not be negatively impacted by messaging. Further, the data returned by the *qsched_gettask()*, the *type* and the *data* values are abstract to the scheduler in that it does not process them *per se* and just hands them to the threads which are then responsible for interpreting them, so there is no need for extra exchange of data with the scheduler during the execution of a task on the host. All this therefore was encouraging to pursue this split for the offload.

It was mentioned earlier that the task generator, in both the examples of QuickSched and in the experiments was located on the host. This means of course that all of the information in the five tables set up by *qsched_init()*, which support both the task graph and queue processing, has to be supplied from the host to the BlueField, with messages for each.

There is another seeming division in the data used by the scheduler, which is between the *task graph* on the one hand, and the *queues* on the other. These are connected by the *qsched_enqueue()* function. So, one could perhaps envisage putting just one or other of the task graph and the queues on the BlueField. Each task is enqueued once, so the number of messages is, at a high level, not changed. The main indication against is, however, that the objective is to offload work, so doing less of it with the same number of messages in a more complicated way did not appear attractive.

5.18. Need for an RPC library

The other aspect that came out of the analysis of QuickSched was the scheduler has a public interface of function calls and that these were going to be called from the host but executed on data, the task graph and the queues, that would be located on the BlueField and then return data as the return type of the function. This pattern was recognised as generally similar to a remote procedure call (RPC). Consequently, it was decided to implement the interaction between the host and the BlueField using such a pattern. As explained in the next chapter it was further decided to implement that aspect as a separate and new library.

6. Argmessage

The Argmessage library is the lowest level of component that was written for this project. Apart from the lower-level messaging libraries on which it depends, it is a new project, completely written from scratch.

The library provides facilities for a client application on a host to execute functions on a remote server device. In the main use made of this library in this thesis, the host is a compute node of a cluster (so most likely having an Intel or AMD x86-64 processor), while the “remote” server is the Arm processor that is provided as part of a BlueField smart network card, and this card is attached to the PCIe bus of the host.

That example already indicates one requirement for Argmessage, which is that client and server processes exist in different address spaces and so inter process communication between them will generally be of the messaging kind, in distinction from communication via shared memory. (So, while the BlueField card of the example mentioned here is physically located not that far away, i.e., in the same enclosure as the host, it is “remote” in that sense.)

In its main version, the Argmessage library uses the OpenMPI messaging library for sending the messages *per se* between the client and server processes, for several reasons. OpenMPI is quite straightforward to use, it is very mature, and will work on different processor architectures, as is also required here. It is also designed for codes that are initiated at the same time to run simultaneously on multiple nodes in a cluster, which are target requirements here since the goal is to support large computational programs running on multiple nodes.

(OpenMPI has data type declaration facilities [38] that solve the problem of different endianness, in cases where it exists, of data items in messages transmitted between machines of different architectures. Note, however, that did not occur in the present work and the facilities were not used in the Argmessage code, which takes a manual approach to packing and unpacking the messages, for which see sections 6.5 and 6.7.)

6.1. Remote procedure calls

This overall function of the Argmessage library is, in general, a remote procedure call (RPC) function. *Ex post facto*, one might ask why RPC libraries were not systematically surveyed at the outset and an appropriate one selected, rather than writing a special one for this application. In fact, in order to be able to support QuickSched, Argmessage developed a particular set of features. These are explained in this section and contrasted with the generalities of RPC frameworks.

RPC libraries are many and diverse, built to suit various computing environments. For example, once the RPC pattern was recognised, I first recalled an RPC library that I had used before, Simple Object Access Protocol (SOAP), but obviously that was not going to be at all suitable for this project, since it uses slow transports such as Hypertext Transfer Protocol (HTTP) or even email. It is important to note that the timescales needed for this project are, contrastingly, sub millisecond. Equally inappropriately it uses an XML markup for the values sent, but again, in contrast, in this project the message format would have to be small and simple to meet the timescale requirement. I was also aware of the highly complex CORBA. So, of course, these were not going to be useful.

I did find, however, at a later date, a more low-level library than those, an RPC library project called ONC+ [39], which was more appropriate, and which is also free. This was not pursued as Argmessage had advanced, but, notably, in ONC+, each call to the server from the client waits for the operations on the server to complete and return a result before continuing [40]. That was not always necessary in this project and so would have degraded performance in some aspects, and indeed allowing not to have a return type is now a feature of Argmessage.

A recent brief survey of RPC frameworks (in a Rutgers University course note) [41], analyses RPC frameworks into first and second generations. A difference between those that is relevant here is that the first generation (there including ONC) provide *stateless* calls to the server, i.e. each call provides all information needed from the client needed to answer it, so not taking into account any history of the calls made. The second generation frameworks are *stateful*. This generation is also object oriented and a client makes RPC calls that run methods on objects that may persist between calls and modify their state. An important aspect of Argmessage is that it is *stateful*; as it is used for Quicksched, it maintains the state of the scheduler on the server side.

A further aspect that came out in the project was that the cooperation between client and server in using QuickSched via Argmessage could involve parallel processes (in fact parallel threads) on both the client and server sides, giving rise to interactions between the effects of messages. This is explored in Chapter 7. (It should be noted that, as Argmessage and Qsargm were developed together and for that purpose only, it is possible that Argmessage is not a full treatment of the problems that could arise for RPC communication between multithreaded processes.)

It was also important that OpenMPI was successful, without too much coding difficulty, in sending messages between the heterogeneous host and BlueField card installed therein. Added to that, OpenMPI is well suited to low latency communications in the cluster, and it was hoped by extension that it would also be so across the connection between those two items. This and the RPC approach mean that Argmessage could be a useful tool for rapidly adapting other libraries to make use of remote communication in the HPC context.

6.2. Argmessage Messages

A key component of any messaging library, or an application or library that runs over one, is, of course, the structure of the messages sent.

As noted at section 5.18, Argmessage is an RPC library and so has a client-side interface that sends messages, on behalf of some client application, to an Argmessage server component, which responds to that to execute a procedure on the server as specified in the message. The result of that procedure is returned in a reply message from the server to the client, which passes the result to the client application. Argmessage relies on a base messaging system, OpenMPI, to transport these messages for it. The Argmessage client and server construct their messages for each other in a buffer in a format discussed section

6.3. This buffer is then the payload of a message constructed in the underlying message system, by calling the appropriate functions of the underlying message library. These calls will of course wrap the payload buffer with metadata that concern that base message library, for example the network addresses of the client and server; the format of that outer message is generally not important here, but of course the Argmessage library does supply the necessary data items, such as the network addresses, using the message construction functions of the underlying messaging library. These values are generally fixed, and they are established by the client and server of the Argmessage library in an initialisation phase.

If the underlying messaging library were to be changed, Argmessage's calls to it would have to be changed correspondingly. A new underlying messaging library might have different concepts or related hardware, and if so, that may well make the changes needed more complex. However, because Argmessage uses just the simple buffer as its actual message between its client and server, it is expected that little, if anything, would, in fact, have to be changed in that. (Although note the uses of MPI tags described later in this thesis, which are an exception to all the application's data being in the buffer.)

6.3. Message structure

The Argmessage message buffer is designed, in the time-honoured fashion, with a header of fixed control data, and a more free-form second part. This second part provides flexibility to accommodate different messages for different RPC calls. In particular, however, the second part is always the packed arguments of the remote procedure call to be called on the server; in particular they are the arguments of the C function on the server that implements the call. For simplicity, both Argmessage's internal housekeeping functions, and those for RPC calls made to achieve the client application's actual purpose, employ the same message format and indeed the same RPC mechanism.

An example of an Argmessage message buffer is shown in Table 3, as printed out by the print statement trace included in the library for debugging purposes (function *argmessage_printbuffer()* from file *argmessageinternalhelpers.c*).

```
ARGMESSAGE SERVER: raw message received:
Mess size:    Proxy ID:     Prox seq no:  Adapter ID:   Func ID:
Argbuff size:
1C 00 00 00 | 00 00 00 00 | 04 00 00 00 | 00 00 00 00 | 66 00 00 00 | 04
00 00 00 |
00 00 00 00 |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~
~  ~  ~ |
 ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~
~  ~  ~ |
Text Equivalent :-
            |             |             |             |  f          |
|
            |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~
~  ~  ~ |
 ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~  ~  ~  ~ |  ~
~  ~  ~ |
selected struct members:
  adapterid=0, args buffer address=0x2983a78, message start
address=0x2983a60
print message buffer: returing from printbuffer
```

Table 3 – Example Argmessage message buffer

The printout, which is of course reminiscent of the *hexdump* utility, shows all the bytes of the message in hexadecimal byte values, and below that the text equivalents. The first 24 bytes, comprising six 4-byte integers, are the fixed data items, with the remainder being the flexible argument buffer part. The fixed items are named in the line above. The tilde ("~") symbol is for padding the print display with items beyond the end of the message (the messages are of variable length, but the print is not), and if the bytes have no text equivalent they are shown as blank spaces. On receipt the message buffer is cast into a C struct, and the items at the end of the printout are printed from that. This deals with the items of the fixed header part – the second flexible part is decoded subsequently.

More details of the data items in the message buffer are given in Table 4.

Message size	the total number of bytes in the header and the argument buffer
Proxy ID* and *Adapter ID	as explained later, the library can handle having multiple instances of the proxy and the adapter objects at the client and server ends, respectively.
Proxy sequence number	a serial number for the messages issued by a proxy object. This is useful for debugging and maintaining the correctness of the program executed on the server as intended by the client. The functions executed on the server must be executed in a correct order. The server checks that messages arrive in the order they are sent and raises a fatal error if they are not. Such a constraint is a good starting point for debugging and understanding behaviour of the library, but it is envisaged that it could be relaxed if need be.
Function ID	identifies which one of the user-supplied remote functions is to be called on the server side. Argmessage's housekeeping messages also have these, but of course with different values from those for the user's functions.
Argbuffsize	gives the size in bytes of the remainder of the message, which is the set of arguments to be passed to the function to be called.
Argbuff	A variable length field which is the packed arguments of the function to be called, (not labelled in Table 3 *per se* but visible as the second and subsequent data lines).

Table 4 – Argmessage buffer fields

6.4. Modules of Argmessage and Operation

Figure 6 is a block diagram showing the code modules and data structures of the Argmessage library. An application using the library runs as two processes: a client process (the large grey box on the left) and a server process (the large blue box on the right). In the main use of this library in this thesis, the user application, e.g., some scientific computation, runs on a host compute node, while the server typically runs on a BlueField card installed on that host.

The overall functionality provided by Argmessage is for the user application main code (top left in the Figure) to be able to call the user function remote code (bottom right) on the server, preferably just using a simple function call, transparently giving the same appearance as if the call were local.

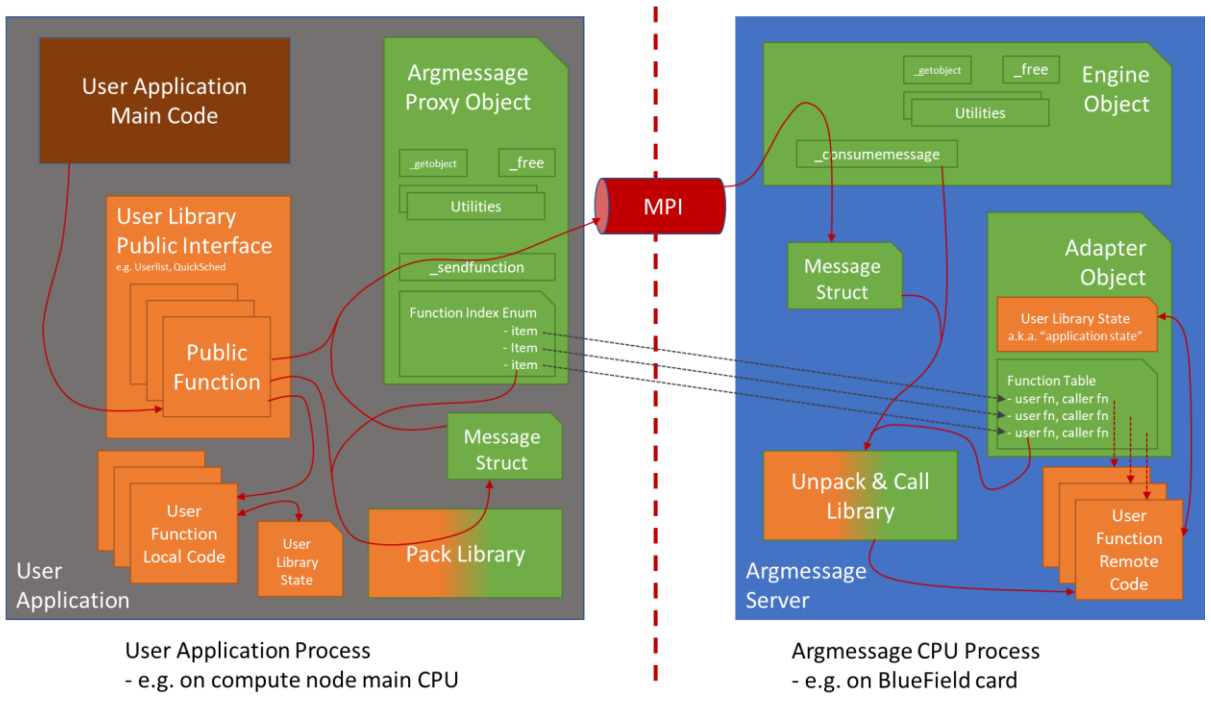


Figure 6 – Modules of the Argmessage library

The user function remote code on the server is provided by the user, rather than by Argmessage *per se*, and provides a set of functions needed by the user application. The diagram shows the core code of Argmessage in green and some functions provided by the user in orange. On the client side the user application code could use the core facilities of the Argmessage library directly; it is however logically much clearer for the user to provide the user library public interface comprising a set of function calls to operate the Argmessage library, thus providing the transparent interface. If the user is converting a library of functions that previously executed in the client process, then, in general, the functions of that library can be used to provide the remote user functions on the server side and the interface of the existing can be reproduced as the user library public interface on the client side. This pattern is used in the main application of Argmessage in this thesis.

The terms *proxy object* and *adapter object* are used in the diagram. These names are explained as follows (following the terminology of [42]). The *proxy object* is so named because, at least wrapped in the *user library public interface*, it stands on the client side for the remote library of functions, providing the transparency property. The *adapter object* is named that because it adapts the incoming messages from the client to the function library on the server side.

So now, in detail, the operation of a call from the user application to one of the public interface functions is described, in sections 6.5 to 6.10. This is of course for the case where the programmer providing the public interface functions has used the Argmessage library as it is expected to be used. The first half of this operation follows the red arrows in the diagram.

6.5. Packing functions

In response to the main application code calling one of the public interface functions, that function first packs the arguments passed to it into the contiguous memory region *Argbuff* of an instance of the message buffer (section 6.3). Argmessage provides functions to do that packing for common function signatures. For example, pack functions for signatures comprising several different small counts of integer parameters are provided. If Argmessage does not include a *packing function* matching the public interface function’s signature, then the user must provide one, preferably using the helper functions provided in Argmessage, and following the pattern of the *pack functions* that it does provide. It is not necessary for the arguments to be arranged in the buffer in the order they appear in the function signature but that would be a helpful convention for a programmer to adopt, and indeed that convention is followed by the *pack functions* provided in Argmessage. In the uses to which Argmessage is put in this thesis, the arguments are simple atomic values. In principle there would be no problem with further arguments being added to the *Argbuff* buffer that were not provided in the public interface function call, so which would be retrieved or calculated from values somewhere else on the client side.

The *pack function* also sets some of the fixed fields in the header of the message buffer, namely the *proxy* and *adapter IDs* and the sizes. The *function ID* is set to whichever of the *remote user functions* it is desired to call and, in most cases, will be a constant for a particular one of the public interface functions. To aid the programmer to use the right *function ID*, a function index *enum* is declared, and is used both on the client and server sides.

6.6. Sending the RPC request message

Next, the public interface function called calls the *argmessage_send()* function associated with the previously initialised proxy object. This adds the sequence number to the message buffer and then sends it to the server process using OpenMPI. The network address of the server (the “rank” in OpenMPI terminology) is needed for that and has been established in an earlier initialisation phase.

6.7. Receiving RPC requests

On receipt at the server, the message is picked up by an *argmessage_consumemessage()* function and the server then uses it to call the correct *remote user function* on the server, as follows. A previously prepared *function table* on the server comprises an entry for each *remote user function*, comprising a pointer to the *remote user function* and a pointer to an *unpack and call function* appropriate to the signature of the *remote user function*. The *argmessage_consumemessage()* function uses the *function ID* in the received message to retrieve those, and calls the *unpack and call function*, which, in turn, unpacks the arguments from the *Argbuff* section of the incoming message and calls the function pointed to with those arguments. As with the *pack* functions on the client side, Argmessage provides *unpack and call functions* for common function signatures, and again, if a user remote function needs a signature that has not been provided for, then the user will need to provide one. Again, *the unpack and call functions* provided with Argmessage follow the convention that arguments are applied to the *user remote function* in the order in which they are arranged in *Argbuff*.

Before calling the *remote function*, *argmessage_consumemessage()* checks the sequence number of the message to ensure that it is indeed the next in the sequence. The user program does not supply Argmessage with any information as to which order the remote functions should be executed for correctness. It is therefore assumed that the calls to the user library interface will be made in a correct order and Argmessage then executes the corresponding remote functions in that order. MPI guarantees the order of messages [43], see section 3.5, and so no issues in this respect were encountered, but the check could well prove useful if the messaging is converted to some other library.

6.8. Remote user functions

The *remote user function* may well need to access to some state to interact with. This state can be attached to the *adapter object* on the server (see also Appendix A). This is certainly done in the Qsargm application described in later sections, but it would also be possible for the *adapter* to interact with other data either in other processes on the server or even on other machines further afield.

In Argmessage it is allowed for the *remote user functions* to have a return type or not.

If there is no return type, processing ends there on the server, and on the client the public function does not wait for a reply and continues its processing, by returning to the main user application. Thus, for such calls the processing time for the call on the server and possibly some of the overhead of the messaging introduced by Argmessage is not experienced by the user application program. This is because the processing occurs in parallel with the application program on another processor. Nonetheless, there will be some overhead of the messaging because the *proxy* does have to call the functions of the underlying messaging library.

6.9. Reply messages

If there is a return type, a message containing it is sent back using a similar process as before (there are no arrows in the Figure for this stage). The return value is packed into a message, which is sent back to the client using OpenMPI. There the public interface function that started the whole call is waiting to receive it and once it does, it unpacks the value and returns it as the return type of that public interface function.

If a function would naturally return a pointer to some data on the server, then the programmer would have some more analysis to do, since, at the client the data would not be directly available. The data could, for example, be packaged into the return message rather than the reference, but then of course the data might then only be a snapshot of the authentic current data on the server. Further functions to access the data with the pointer may have to be added to the user library.

6.10. Relationship between public interface and remote functions

Typically, the relationship of *user library public interface functions* to *remote user functions* is 1:1, but it may be convenient in some cases for this to be 1:many. This will depend on the particular user library being implemented and on the programmer’s design choices.

Also, an analysis of the problem to be solved by the *user library* could, depending on the case, suggest that some of the processing of the *public interface functions* (in part or the whole) would be better carried out on the client. In that case *local function code* (bottom left in the diagram) can be provided on the client. This may be called before sending the message to the server and/or after receiving the response message as needed. Such code may need to refer to some state data. This state may be internal and so could be attached to the *proxy object,* but is not shown as so attached because another possibility is that the state could be some values in the main application to be referred to or updated. The latter would break a pure function call pattern of the user library public interface, but on the other hand it is not uncommon for functions in C to interact with global variables.

6.11. Appendix material on Argmessage

Further information useful to a programmer using Argmessage is to be found in Appendix A, which covers initialisation and finalisation of the various components and considerations for launching programs using it. Appendix B includes remarks on compiling Argmessage.

6.12. Potential uses of multiple adapters

One instance of the Argmessage server will support multiple *adapters*. The Argmessage server will service multiple clients, one per *rank*, and initialises a fixed number of *adapters* for each *proxy*. As it stands all of these are initialised with the same *user library functions*, just in case a proxy needs more than one instance of the library – for example if the *user library* has some main object that needs to be instantiated to operate and each of instance of this can then be kept by a respective *adapter* object. The *proxy* specifies the index of the one of its *adapters* that it wants to use in each call to the server. The number of adapters per proxy is defined by a C macro, so this may be tuned for a particular application, or indeed it could be replaced by a runtime parameter supplied to the Argmessage initialisation functions.

If the server is required to support more than one user library, then the *remote user functions* from all of those should be compiled with the server, but the server initialisation should be changed to map the functions of each user library to different ones of the adapters. It would also be possible in this way to map different user libraries to different ones of the clients if desired, so if the different clients had different roles.

6.13. Potential uses of multiple clients

Allowing for multiple ranks to share the Argmessage server was provided with two particular purposes in mind. A first one is that the server machine may, depending on the application, be powerful enough to service more than one client. So, for the example of a BlueField card, if these were provided in a cluster at a density of less than one per host machine, then several host machines could share the server as a common resource.

The other purpose would be a way to support cooperative processing of information at the server from various ranks. The particular application of that in mind is an extension for the QuickSched application – *adapters* for several *ranks* would be keeping their progress information in the same place making that readily available to make quick cooperative decisions about cooperation on tasks, for example, for arranging work stealing between nodes.

6.14. Providing for multiply threaded client applications

Applications based on the QuickSched library are natively multithreaded. Argmessage will support multiply threaded applications, but this needs some additions to the code described so far. A first problem is that Argmessage uses an underlying messaging library, so if an application has multiple threads that are going to make use of that by sending RPC messages to the Argmessage server then the underlying messaging library must be able to support that. The underlying messaging library used here, OpenMPI, fortunately supports it. Accordingly, the multithreaded version of OpenMPI was used. In addition to selecting the correct build options for OpenMPI, this only requires changing the *MPI_Init()* call to *MPI_Init_thread()*. This has a parameter to select between three modes of multithreaded operation for the MPI library [44]; MPI_THREAD_MULTIPLE was selected because this allows all threads to issue calls to the library with unrestricted timing, so on the face of it seems to promise not holding up any of the user threads, which is what is desired for this project. On the other hand, this option must be the most challenging for the OpenMPI code to handle so it could result in more overhead or blockages than the others. It was not however tested whether the other options in fact performed better in actual uses of Argmessage, but rather it was trusted that the issues of competing amongst threads for messaging resources had been better solved by the OpenMPI library authors than my trying to do so.

With that in place and with the general architecture of Argmessage described in this Chapter, Argmessage can easily support a multithreaded client library, e.g., QuickSched, since each call to the client library from whatever thread simply translates the call into an OpenMPI send, which can cope with concurrent threads. Any concurrency issues in the values of the parameters packed into those OpenMPI messages will generally have existed in the original client library and will have been solved already to that extent.

6.15. Multithreaded server operation

In the earlier experiments performed, Argmessage and Qsargm, which is a version of QuickSched using Argmessage developed in this project, were operated with a single threaded server. Multithreaded operation is discussed in section 7.10.

6.16. Main use of Argmessage in this work

The main *user library* to which Argmessage was applied in this work was, of course, the QuickSched library. The next chapter describes how Argmessage was applied to it; this also discusses some modifications that were found to be needed to Argmessage in order to support QuickSched.

7. Qsargm: application of Argmessage to QuickSched

Chapter 5 was an analysis of how the QuickSched library might be split into client and server portions and Chapter 6 described a new RPC library, Argmessage, designed to handle the communications between those two portions. This chapter explains how Argmessage was applied to QuickSched. The resultant library is called Qsargm. This process also revealed that some further changes were needed to Argmessage to support QuickSched and these are also discussed.

It was proposed that the split of the QuickSched library for the task run phase of its operation should be to have on the server both the principal data structures of the task graph and the queue data and their attendant processing, while the client only operates in the function *qsched_run()* to set up / destroy the threads and to execute allocated tasks, while sending between them messages representing the *qsched_gettask()* and *qsched_done()* function calls to the scheduler on the server. Those latter two function calls were therefore implemented as RPC calls using Argmessage. This arrangement is illustrated in Figure 7. This is similar to Figure 6 of the last chapter but redrawn to show this split of the QuickSched user library functions and their principal data structures (but with some of the Argmessage machinery simplified).

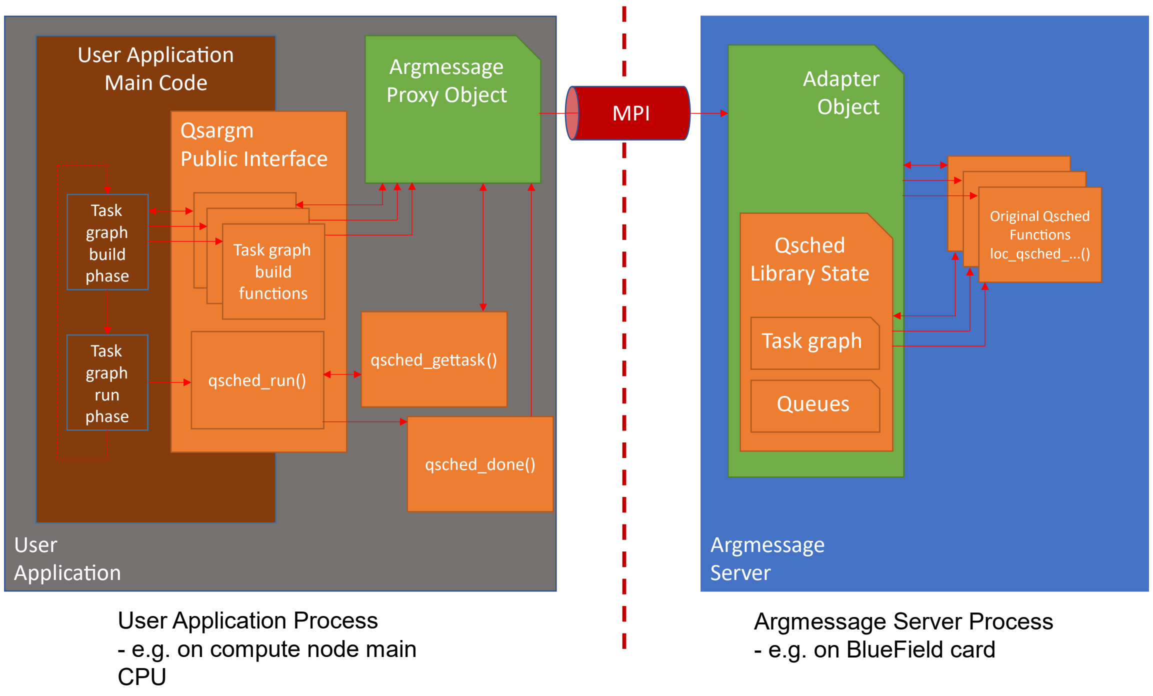


Figure 7 – Qsargm: Argmessage applied to QuickSched

As may be seen, in the task run phase, the user application calls the public interface function *qsched_run()* and this then operates to keep the host cores/threads occupied with tasks, which it obtains from the scheduler on the server by calling *qsched_gettask()*, with it notifying the scheduler of completed tasks by calling *qsched_done()*.

Each of the public interface functions of QuickSched was implemented as an RPC call to the server. These included all the functions used to build the task graph. This choice, as discussed in sections 13.2 and 14.3, results in many messages being necessary to build the task graph representation on the server, and hence poor performance. Nonetheless, converting the library in that way was straightforward and mechanistic, so increasing the chances that the converted library would operate correctly, which was achieved.

In the archive, the code files for Qsargm are to be found mainly in the directory *qs/src*. The original QuickSched library is in the directory *qsoriginal/src* and the code there is utilised by the converted library.

This original QuickSched library is unmodified, except for the renaming of many of its functions by adding the prefix "loc_" (see these on the server side in Figure

7) and some of the files with the suffix "_local". These names are to suggest that the function will operate in the local process (on the server), rather than making an RPC call. The transformation into Qsargm therefore, as a general rule, just uses Argmessage to change where in the system they are located, with *qsched_run()* being the main exception to that. Both of these are described in detail in the sections 7.1 to 7.4.

7.1. Public interface functions

In Qsargm, the file *qsched.h* contains prototypes for the public interface functions with the same names as in the original QuickSched. In this way any user application program using QuickSched can use Qsargm without changing the function names.

However, since the experiments on the library will compare the performance of Qsargm to the original library, each implementation of those functions in *qsched.c* does not immediately carry out the operations of QuickSched via RPC but first provides a switch between: (a) calls to the Qsargm versions of the functions, which make use of RPC calls to the server as appropriate, and (b) calls to the original functions, whose names are now prefixed with "loc_". The switch switches between them at run time in accordance with a global variable *activestrategy*, which is constant for any particular run of the application code, so that any particular run provides results for either the original, QuickSched, or the new, Qsargm, version of the library. To avoid confusion this second use (b) is not shown in Figure 7.

In use (a), the new library, each public interface function calls a similarly named function, but prefixed with "qsargmproxy0" rather than "qsched", and these are in the file *qsargm_proxy0.c*. The code of each such function has the form of an RPC call, and so packing the arguments of the function into an Argmessage buffer in the manner described in the previous Chapter, which is then sent to the Argmessage server with the function *argmessage_proxysendfunction()*. (Refer back to Figure 6.)

For functions that have a return type, that is obtained by immediately calling *argmessage_proxyreceivefunctionresult()* and unpacking the result from the reply message buffer, once it is received, and using that as the return value of the function. Although the call is made immediately the reply message will, of course, not come back immediately and so the client waits here for the server to respond.

On the server side, counterpart functions were provided for each of the public interface functions, with similar names but prefixed “qsargmadapter0”, and having the same signature except an additional parameter of the adapter id, and these are in the file *qsargm_adapter0.c*. The adapter id is used to retrieve the state of the QuickSched library, and the counterpart then calls the original QuickSched library function with a reference to the retrieved state. Conveniently for achieving that, the original QuickSched library defined a main singleton object, of type *struct qsched*, to contain its state and required a reference this object to be passed to each of its functions, so that object was attached to the adapter object. In that way, it was quite straightforward to reproduce exactly the same operation of the scheduler functions on the server, because they use the same code and the same data.

The remote counterparts to *qsched_init()* and *qsched free()*, and *qsched_addtask()* and *qsched_gettask()* were also modified, but the changes were not particularly significant.

qsargmadapter0_addtask() was changed only to initialise a new task with additional fields of the size of the task data and a record of its own id. The point of this is described in the section 7.4 on *qsargmproxy0_run_openmp()*, in particular the part relating to *qsargmadapter0_gettask()*.

7.2. Pack and call functions

It was mentioned in Chapter 6 that Argmessage requires a client library that is using it to define pack functions and call functions for any function signatures for which it did not provide those itself. In converting QuickSched it was needed to provide those for the *qsched_addtask()* function, because it uses a pointer to a small array of data (the definition of the task), whose individual items are packed into the Argmessage message (section 6.5) and at the server side are unpacked into a new array at the client side.

7.3. Client and server programs and initialisation and finalisation of qsargm

A client program developed for the Qsargm library was *qs/examples/qsargm_test_qr*, and was the one that was used in the performance measurements. This was developed from the QR factorisation test supplied with the original QuickSched.

Its initialisation and finalisation were adapted to use the Qsargm library, instead of the original QuickSched library, simply by

- replacing the QuickSched library with the Qsargm library, which, as described has the same interface, but which makes RPC calls using the Argmessage library to a remote instance of the QuickSched library.
- providing the Argmessage initialisation and finalisation statements for an Argmessage client, as described in Appendix A at section 17.1.

The counterpart server program *qs/src/qsargm_server.c*

- was provided with the Argmessage general initialisation and finalisation statements for an Argmessage server,
- included the original QuickSched library and include statement,
- and was provided with a function *qsargmadapter0_serverregisterfunctions()*, which initialises the table of functions used in the Argmessage server when unpacking the RPC messages and calling the library function requested, with of course entries pointing at the original QuickSched functions.

7.4. Converting *qsched_run()*

As noted at section 5.15, this function has its operations split between the client and server. The version of the client-side function *qsargmproxy0_run_openmp()* part of that is discussed, since the *pthreads* option was not investigated. That function like its original, *qsched_run_openmp()*, calls the function to *prepare* the scheduler and then starts a set of parallel OpenMP threads equal in number to the cores assigned to the client process and to the number of queues. Each thread has its own respective core to run tasks on the client and its own respective task queue on the server. Each thread on the client executes the same loop of getting a task from the scheduler, executing it, and then reporting to the scheduler that the task has been completed.

The *prepare* operation is in relation to the task graph tables and queues and was described in the earlier chapter on the analysis of QuickSched. Since it was decided to put these on the server then this is implemented in *qsargmproxy0_run_openmp()* by an Argmessage RPC call to call on the server *qsargmadapter0_prepare()*, which, in turn, simply calls an unchanged *loc_qsched_prepare()*.

The loop of getting a task from the scheduler, executing it, and then reporting to the scheduler that the task was implemented as follows. First, an Argmessage RPC call to *qsargmadapter0_gettask()*, which operates, as the original generally does to return a task, which is then communicated to the client in the return message. Second, the task is executed in the client thread just as in the original. Third, an RPC call is made to report that execution of the task is complete, to *qsargmadapter0_done()*. This call generates no reply.

The introduction of messaging between client and server at these points does, however, amount to significant difference in the mechanics of the interaction of the threads and the scheduler compared to the original QuickSched, significant enough to require further modifications to Argmessage and some details of QuickSched itself. In the original QuickSched each thread in effect becomes the scheduler when it is not executing a task. In contrast, in Qsargm the scheduler is in a separate process elsewhere and needs messages between the threads and the scheduler. A number of issues were recognised, and modifications made.

The original *qsched_gettask()* returned a pointer to a task, and the original *qsched_run_openmp()* ran that task using a pointer to the task parameter data looked up in the task table entry for the task. In Qsargm, this would have required lookup from the client to the server, and so more messages between the client and server; so, to obviate and pre-empt that in Qsargm, in *qsargmadapter0_gettask()* on the server, the task data is looked up there and packed, with the task id also, into that function’s reply message, thereby limiting this operation to a single RPC message and reply. *qsargmproxy0_run_openmp()*

receives the task id but has no use for it now to look up the data, but the task id is used to encode in that message whether the task graph has issued all its tasks, with a special code of for the task id (-1 to avoid the positive values used for the task ids of actual tasks).

In the discussion of Figure 4 in Chapter 5 on the analysis of QuickSched, it can be seen that the threads interrogate the task queues for the next task to execute, and in the call graph of Figure 5 it can be seen that that is done by the function *queue_get()*. That suggests that an alternative place to split the *qsched_run_openmp()* function between client and server might be at the *queue_get()* calls. This, however, would be more inefficient, which is explained as follows. Interrogating a queue can fail in a number of ways: the queue(s) can be locked because other threads are modifying them, because another thread has locked a task in the queue (see inside function *queue_get()*), and because there are no tasks in the queues at present, which could be the case if the scheduler has not finished moving to the queues tasks that have recently become ready tasks. *qsched_gettask()* responds to these failures by trying again until a task is eventually obtained. There could be several such attempts, and each would require a message and reply between the client and server if the split were made at *queue_get()*. However, the thread cannot be put to any useful work until *gettask* is successful in obtaining a task, so it is better for that period of waiting to involve just the single message and reply that is needed by the *gettask* split.

7.5. Single threaded operation

QuickSched is of course intended to operate in a multithreaded manner: the essence of QuickSched is to supply tasks to multiple parallel threads, and each thread can concurrently become the scheduler as needed (with the exception that some operations within the scheduler are protected from concurrent updates with locks). Nonetheless, the operation of Qsargm was first debugged and verified using single threaded operation, on both the client and the server because it is much easier to follow what was happening in that case.

7.6. Supporting multithreaded operation

Moving to multithreaded operation presented both some design choices to make and problems to solve. In the original QuickSched there is only one pool of threads, those that execute the tasks and that each become the scheduler when a new task is needed. In Qsargm there are now two pools: i.e., the pool on the host that executes the tasks, as before, but now there is potentially a second independent pool on the server to execute the scheduler.

7.7. Reply tagging

A modification that was needed here was to ensure that each thread on the host receives the correct reply to each of its RPC call (for those that have a reply). The potential problem is that two threads may both be waiting to receive a reply, so could, if the matter were not attended to, receive the reply intended for the other.

OpenMPI was used for the messaging and was used with the kind of multithreading provided by initialising with the constant *MPI_THREAD_MULTIPLE*. This was chosen from the options available so that Qsargm could pass on its RPC calls, made with *MPI_Send()*, to the underlying OpenMPI library in the simplest and quickest way, the assumption being that the OpenMPI library would sort out the basic concurrent messaging issues more efficiently than any effort that might be made in this project.

MPI allows a receive operation to select a message on the basis of a tag attached to the message on the send side. This mechanism was used in Argmessage to ensure that each thread on the server receives only its own RPC replies. The mechanism should be efficient as it is hardware based. Specifically, Argmessage RPC requests are marked with the unique *omp_thread_num* of the thread and that identifier is applied to the reply as the MPI tag and the thread listens for that tag in its receive operation for the reply.

An extra field was added to the fixed header of the Argmessage message buffer, namely the *omp_thread_num* identifier, of the outgoing RPC request to provide a way for the scheduler to know which tag to apply to the reply. The scheduler on the server is set to receive RPC messages from any thread – as noted in section 7.8, a one-to-one relationship between client and server threads was rejected.

This tag field of MPI is a separate parameter of MPI function calls, i.e., it is outside of the message buffer. This is contrary to the goal advanced in Chapter 6, of keeping as much information as possible in the Argmessage message buffer so that the messaging library can be changed as easily as possible. This tag mechanism for directing the RPC replies to the correct thread is therefore noted as something that will require specific implementation if the underlying messaging library is changed, whether that is by analogous interaction with the messaging library *per se* if it has suitable features, or by handling the issue somehow in Argmessage itself. However, not only is the tag feature of MPI longstanding and so should be efficient, but hardware tag matching is now available.

7.8. Number of server threads

It was recognised that operating the scheduler in an analogous way to the original QuickSched with the scheduler having a fixed respective thread for each queue / task executing client thread was not going to be appropriate.

A first point is that a thread only operates as the scheduler for a small fraction of the time, compared to task execution, so the scheduler would only use each thread of the server pool for that small fraction of the time, which is very inefficient. A second point is that in the typical use, an x86 compute host and a BlueField card, the number of cores on the host will typically be much larger than on the BlueField, so this one-to-one mapping would not even be possible.

The goal should be to use as few threads on the BlueField as possible without it becoming “overloaded”, so that the other BlueField cores could be put to some other use. (“Overloading” would manifest as a growing queue of RPC messages to the server that need servicing by the scheduler.)

7.9. Single server thread and doevents mechanism

Nonetheless, although it might cause some extra delay to replies to simultaneous *gettask* requests, and hence reduction of efficiency of task execution on the compute host, the first multithreaded configuration tried was multiple threads on the host serviced by only a single thread on the BlueField. Again, this was for simplicity of debugging and following what was going on.

This, however, led to a condition in which the operation of the scheduler came to a halt. Gregory Andrews, in his book, “Concurrent programming – Principles and Practice” [45], defines *deadlock* thus, *“A process is in a deadlock state if it is blocked waiting for a condition that will never come true”* and *livelock* thus *“A process is livelocked if it is spinning while waiting for a condition that will never come true.”* This was a *livelock*. The problem found occurred when the scheduler queues became empty and there was a *gettask* message at the head of the received queue of messages, when the *task done* message that would allow creation of more *ready tasks* was behind it. In the server, the processing of the *gettask* request is to repeatedly interrogate the *ready task* queues, which therefore remain empty.

The solution found to this problem was for the scheduler on the server, when processing a *gettask* request, and on finding that there are no *ready tasks* available, to listen again for the next message from the client and process it, in the hope that the next message is a *task done* message which may well then free up at least one new ready task. This was achieved by the *argmessage_consumemessage()* function calling itself. If the next message received by the next level of *argmessage_consumemessage()* is a *task done* message, the new *ready task(s)* are then found by the server, and the *gettask* request is then responded to in the lower level of *argmessage_consumemessage()*.

If it happens that the next message received by the recursively called *argmessage_consumemessage()* is also a *gettask* then the recursion is repeated. This recursive process does not however go on forever. Each client thread can only send one *gettask* at a time; it must wait for a reply before it sends any more messages. (This may leave several *gettask* requests outstanding at end of the task graph. At this point the server takes its usual action of replying with the special task id code indicating that the task graph is finished, and the client thread reacts by ending its processing of tasks.)

(The term used in the code in relation to this mechanism is “doevents”, after the similarly named function in the Visual Basic.)

7.10. Multithreaded Argmessage and Qsargm server operation

Multithreaded server operation was also experimented with, but this needed further improvements over the “doevents” mechanism described at 7.9.

The general approach for this was to set each server thread to independently listen for messages from the client, with a blocking MPI_Recv, with each listener then processing whatever message the underlying OpenMPI messaging library choose to present to it from its queue of received messages, with the server thread thereafter returning immediately to listening. So, this again utilised the MPI_THREAD_MULTIPLE version of OpenMPI multithreading.

The aim (achieved) was also for the threads to operate in a symmetrical manner, i.e., all using that policy, rather than, e.g., some scheme in which one or more threads listened and then farmed out the processing of the messages to other threads. Such a symmetrical arrangement was imagined to be simpler to program and would require less thinking about how to scale to different numbers of threads.

However, there was again a problem, when *ready tasks* were scarce, of the scheduler becoming locked, this time *deadlocked*. It was found that, when a *task done* message was received by a listening thread on the server, server threads processing *get task* messages were finding none available. The cause was that scheduler data was being locked by a thread processing the *task done* message, causing threads processing *gettask* messages to revert immediately to listening. This absorbed any further *get task* requests, with the same return to listening happening again. The resultant state was that all the client threads were left waiting for a reply to *get task*, while all the server threads were paused listening for new messages. The server thread processing the *task done* message did, of course, supply new *ready tasks* to the *task queues* but by that stage all the other server threads were listening again; the *task done* server thread also returned to listening but the client threads would not then issue further *get tasks* as they were waiting for a reply to one they had just issued.

This problem was overcome with a new strategy, which was for *get task* processing by each server thread not to immediately listen again when no *ready* *task* could be retrieved, but to inspect, or peek into, the incoming OpenMPI message queue to see if a *task done* message had arrived and if so, processing that. If not, the server thread polls for ready tasks. This cycle of peeking and polling is repeated until a *ready task* is retrieved from the scheduler queues, at which point the retrieved task is returned to the client and the server thread listens again for another message.

(OpenMPI provides non-blocking *probe* functions to inspect the received incoming messages and such messages may be filtered by the MPI tag; Argmessage messages from the client are tagged by their type, and that was used to find the *task done* type messages. The MPI_Improbe version of the *probe* function was used; ‘I’ in the name is the non-blocking version, and ‘m’ in the name means that the function both returns a handle to the message it finds and reserves the found message found to be received by a subsequent OpenMPI receive statement quoting the handle. This solves the problem that an attempt to receive the message would fail if another thread had received it in the meantime.)

7.11. Closing open listeners after the end of the task graph.

There will of course come a time when all tasks have been exhausted and that often occurred while some server threads were in the listening state. These were satisfied by the expedient of sending *no operation* messages from the client to the server. The number of messages needed was not accurately determined but rather enough were sent to satisfy the maximum conceivable number of listening threads, with that being the number of server threads minus one (one thread will be needed to process Argmessage termination messages). This number of messages was communicated by adding a reply, containing that number, to the *kill adapter* message, which previously did not have any reply.

7.12. A Qsargm problem – taskgraph build is single threaded

Another problem with multithreading the server was that during the *build task* phase the order of the function calls does matter, in that some of those function calls require arguments that are handles for items already created in the scheduler. The example of QR factorisation (discussed in Chapter 8) used a single threaded program section to build the task graph to ensure that the order of the task build messages was preserved.

In order to mirror this behaviour on the server a modification was made, which was to divide the server operation between a first single threaded phase to handle the *task graph build phase* and a multithreaded phase to handle the task graph run phase.

7.13. An observation on Argmessage messages having no reply

It was observed that a feature of errors arising when the server was first run in a multithreaded manner was that they were related to Argmessage messages, e.g., *task done*, that did not, unlike the prototypical RPC, have a return message. This is understandable in that it causes more uncertainly about the order of the processing of the messages on the server. For example, a client thread, once it has sent a *task done,* will immediately send a *get task*, which therefore on the server will both be immediately received, but on different server threads; so those will be processed on the server concurrently. Whereas, if the *task done* had a reply the *get task* would not be issued until the client thread has received the reply, or if the server was single threaded it would process the *task done* and *get task* in the order they were issued (since MPI guarantees, in general, to preserve the message order).

8. Test example of an application: Tiled QR factorisation

8.1. Selecting an application for performance testing

A task-based computation was needed to test the performance of the of the Qsargm library. The original QuickSched library came with several examples. The tiled QR factorisation was selected and has the following useful properties.

- The result computation is always the same, making it a good regression test. As noted earlier the example came with a test that uses another computational method to verify the result, hence providing strong evidence that the calculation proceeded as intended. This test is quick to repeat when the code or the libraries have been updated.
- The kernels always perform the same computational operations. This would mean that the variation in the execution times of the kernels would be small, so less likely to swamp the difference in performance between Qsargm and the original QuickSched. (The processors used do, in general, adjust their clock speeds dynamically, in response to the load, but not necessarily in a repeatable manner, and so variation in execution time, as measured in real world time (or *wall time*), could result, since, of course, instructions execute at a rate determined by the processor clock.)
- It has an internal parameter, the tile size, that can be optimised with respect to the performance of the scheduling algorithm, both for the original QuickSched and the new Qsargm library. The tile size parameter also affects the size of the tasks, and relatedly the number of tasks in the task graph.

8.2. What is tiled QR factorisation?

A QR factorisation of a matrix is its factorisation into the product of an orthogonal matrix Q and an upper triangular matrix R. The factorisation is widely used and useful enough to appear in linear algebra computation libraries such as LAPACK and Mathematica [46] and Eigen [47]. According to the LAPACK manual [48] this usefulness includes, *"Orthogonal factorizations are used in the solution of linear least squares problems. They may also be used to perform preliminary steps in* *the solution of eigenvalue or singular value problems",* with the Mathematica manual giving similar applications.

"Tiled" refers to splitting the matrix to be factorised into a grid of blocks, or tiles, and organising the computation around operations on those tiles. Fortunately, such an algorithm exists, but for the present purposes it is not necessary to understand why exactly the operations of the algorithm achieve a QR factorisation. Each of these operations on the tiles provides, however, a task that may be scheduled in a task-based manner. It is the scheduling of these tasks which is studied in this project. How the QR factorisation is actually achieved is, however, explained in a paper by Buttari *et al* [49], which is also cited by [16] and [50], the latter being a paper very similar to one to be found in the documentation in the original QuickSched repository [15].

The pattern of operations on the tiles by this algorithm, and hence its task graph, is now explained with reference to these documents. This gives a concrete example of a task graph. (Section 9.3 explains that there was a difficulty with the tasks generated by the QuickSched code in its QR factorisation example; the task graph pattern shown here provides a basis for understanding that.)

Figure 4.10 from [16] or Figure 7 from [50], which are the same, and which it will be helpful to have in view, show a 4 by 4 grid of tiles of matrix elements of the matrix to be factorised. These are operated on by the kernels of the tasks. [16] and [50] explain the algorithm via a number of "levels", each of which is a high-level sweep across all the tiles, and which are also the index of the outermost for loop of the task building code. In each level the processing of the top row and left column of the active tiles of the level reach their final form and are omitted from the active tiles of the next level.

Pseudocode for the algorithm is given in Table 5:

```
for each level k in 1..n, where the matrix has n x n tiles, with parallelism at *:
    (i) perform the function task S/DGEQRF (red)
                        on data of tile(tile_row = k, tile_column = k),
    then (ii) {
        (a)   { for each column c in k+1..n, in parallel:
              perform S/DLARFT (green) on data of tile(k,c),
                        using also the upper right potion of tile(k,k) }
        and (b), in parallel with (a),
              { for each row r in k+1..n, in sequence:
              perform S/DTSQRF (blue) on tile(r,k) and on lower left of tile(k,k),
              and then  { for each column c in k + 1, in parallel:
                        perform S/DSSRFT(yellow) on tile(r,c) and
                              on tile(k,c) and using tile(r,k) (but only
                              once S/DLARFT or S/DSSRFT operating
                              in level k on tile(r-1,c) has completed)
                        }
              }
          }
but for k = n: only perform S/DGEQRF (red) on data of tile(n, n)

* parallelism for level k: wherein if the function tasks on a preceding level k-1
have completed writing to the particular input data of function tasks on the next
level k, those particular functions may begin before all function tasks on level k-1
have completed, once also the dependencies of their own level k have been met.
```

Table 5 – Pseudocode for tiled QR factorisation

where the function names follow Chalk's and hence LAPACK's naming scheme, with 'S' and 'D' refer to single and double value types. Figure 4.10/Figure 7 of those cited references also shows, with arrows, the dependencies in the task graph between these function tasks. The colouring of the functions in Table 5 matches that of the tasks in those Figures.

Figure 8 represents the same tasks as a task graph, but which I derived, in order to check, from the task creation code in the original QuickSched QR factorisation example (*archive:expt0078/original_qs_for_ref/quicksched-1.1.0/examples/test_qr.c*), but with the correction to be explained in section 9.3. In the Figure the nodes (ovals) again represent the tasks and the edges represent their dependencies in the task graph; again the tasks' colours represent the particular kernel function used (using the same scheme as Figure 4.10/Figure 7), but each task is now marked with the identity of the first tile mentioned after each function in Table 5 (again according to the Figure 4.10/Figure 7 scheme). With the correction the dependencies in this Figure 8 match to those in Figure 4.10/Figure 7. Finally, the nodes have been grouped into respective boxes for each level to aid comparison with that, but otherwise the tasks were laid out by Graphviz to accommodate the dependencies. As with Figure 4.10/Figure 7, there are 16 tiles in a 4 by 4 grid.

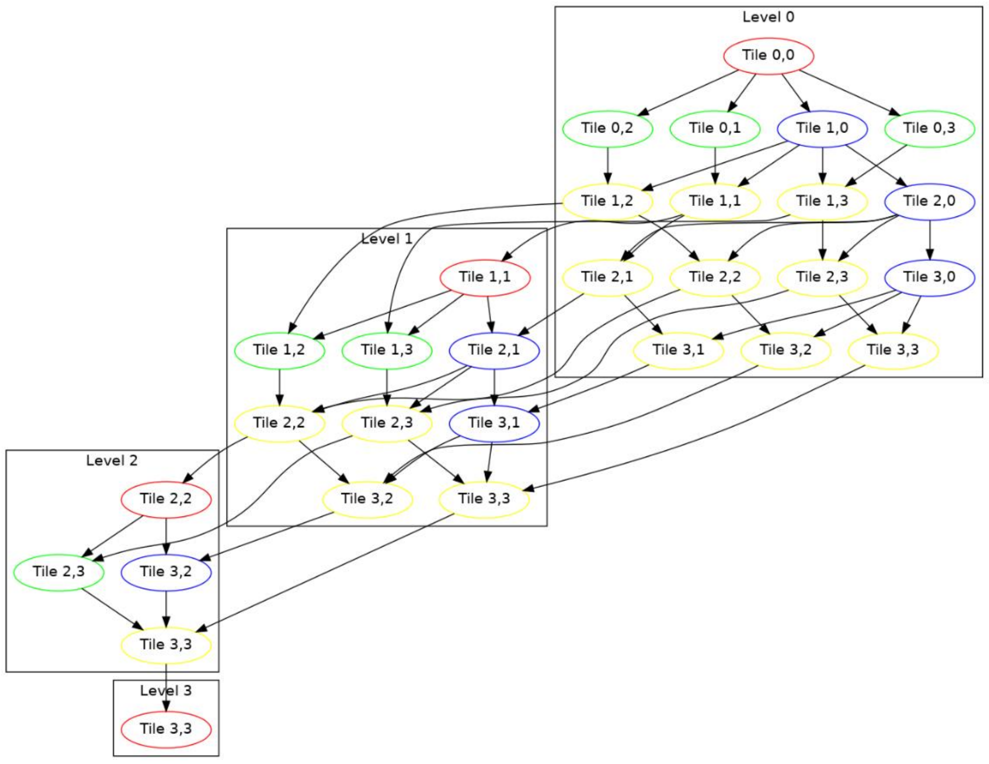


Figure 8 – Correct task graph of QR factorisation example (n=4, so 4 x 4 tiles)

A different layout of Figure 8 is given in Figure 9, namely with the grouping of nodes into levels removed.

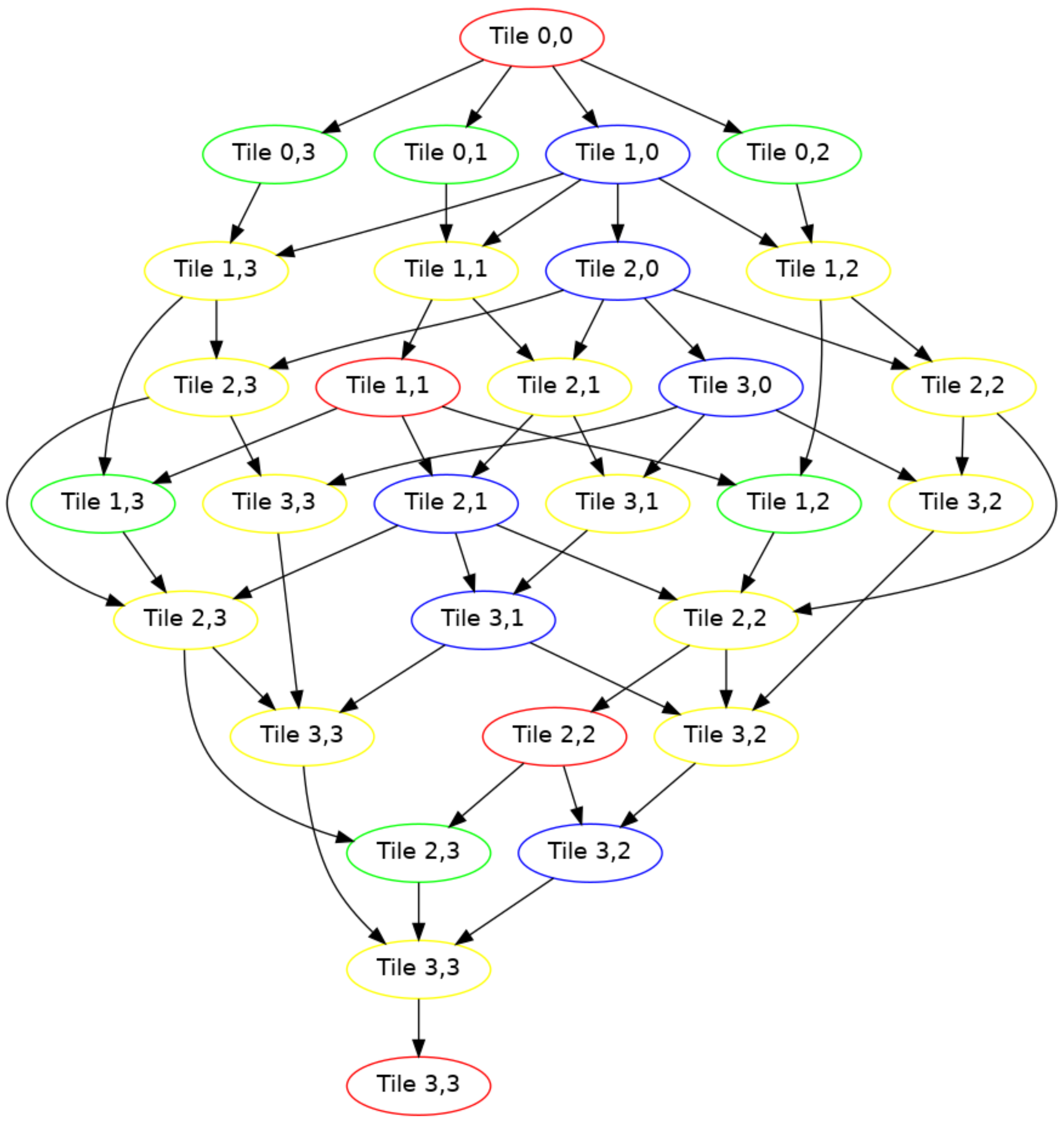


Figure 9 – Task graph of Figure 8 with no level grouping

This perhaps even better illustrates that tasks from the different levels can be executed that the same time.

9. Verification and Validation

9.1. Summary

The operation of the various code modules of this project were verified as follows.

The function of the basic Argmessage code (Chapter 6) was verified primarily with toy examples, described in section 9.2. These were mainly overall functional tests (although some intermediate values in the logs were also tested for). Explicit unit tests should be added if Argmessage is developed further with a view to supporting other distributed applications. However, Argmessage has a client and server architecture, and the overall function depends on the interaction between the two. Therefore, mocks [51] supplying incoming messages will be required for unit tests of the individual functions within the client and server.

The Qsargm library is at a higher level than Argmessage and so for this functional testing was adopted. The main example used in the experiments to test performance of the code, the QR matrix factorisation, from the original QuickSched library and discussed in Chapter 8, came with a functional correctness test comparing the outputs of the QR factorisation using QuickSched to that of the same factorisation using a standard matrix library for equality. It was therefore convenient to use the same correctness test of the QR factorisation for the Qsargm library of this project. Therefore, this is a functional correctness test and is also a regression test in that Qsargm is expected to arrive at the same result as Quicksched for the same QR factorisation. The test is powerful in that it is a completely independent calculation of the result and can be rerun for many different random matrix inputs. Further since the output is two $n \times n$ matrices, so a large number of separate values, the test is likely to be sensitive to a wide range of potential errors in the code.

That test and also the toy example tests of Argmessage were automated as *pytest* tests, to be found in the project archive at *archive:expt0069/argmessageprivate/test_endtoend*. Unfortunately, the original QR QuickSched library did not pass this overall functional test, so it was important to fix this issue, which was done as described in section 9.3 onwards.

The performance experiments conducted and discussed in Chapter 13 are measurements that rely on hardware timers, and on library functions to read them. These functions and timers were validated as described in section 11.7.

9.2. Argmessage Toy examples

Some toy applications of Argmessage were developed as both as tests of the Argmessage code, and as a demonstration to an application programmer of how to use the library and how it operates.

A first toy example (archive:expt0069/argmessageprivate/examples/ex2) demonstrates just a server program with just the built-in Argmessage RPC functions and has a corresponding client program to call them. So, that is with no *user library* functions supplied.

argmessageserverbare.c has the Argmessage library as an include and simply then makes calls to the library to initialise it, set it running and then finalises it. Accordingly, the server program therefore consists of just these calls:

```
argmessage_serverenginegetobject()
argmessage_serverengineechoenginestatus()
argmessage_serverenginerun()
argmessage_serverengineechoenginestatus()
argmessage_serverenginefree()
```

So, this is the basic server; just three function calls – the calls to *argmessage_serverengineechoenginestatus()* are not essential and just echo some status of the server for debugging purposes.

The corresponding client program, *argmessageclientsendstdout.c*, is again basic and consists primarily of just these calls to the Argmessage library:

```
argmessage_proxygetobject()
argmessage_proxyechoproxystatusonproxy()
argmessage_proxyrun()
argmessage_proxysendstdoutmessage()
argmessage_proxykillserverrequest()
argmessage_proxykilladapter()
argmessage_proxyechoproxystatusonproxy()
argmessage_proxyfree()
```

So again, these are calls to initialise the Argmessage library, to set it running and then to finalise it. (The calls to *argmessage_proxyechoproxystatusonproxy()* are also not essential; again, they echo some status for debugging and demonstration purposes, but this time on the client.) It is the client's responsibility to terminate the run phase of the server by issuing the calls to *argmessage_proxykillserverrequest()* and *argmessage_proxykilladapter()*. As an example, a single call is made to the server during the run phase, which is to *argmessage_proxysendstdoutmessage(),* which sends a message to the server causing it to call a corresponding built-in server function to print in its own *stdout* a text provided as the argument to the client function. That text is of course transmitted as the *Argbuff* of the Argmessage payload message.

As the next example demonstrates, the client and server applications for cases where an actual *user library* based on Argmessage is implemented have the same basic structure as these examples. The purpose of this *user library* in this second toy example (archive:expt0069/argmessageprivate/examples/ex3) is to maintain a list of strings on the server and thus provides a programmer with an example to follow if they wish to create a library of RPC callable functions using Argmessage. The example comprises both library functions of the *user library* and example client and server programs using that.

The library header file, *argmuserlist.h*, sets out many of the key components of the *user library*. These functions, declared therein, form the client-side *public interface* of the *user library*:

```
struct argmuserlist_publiclist* argmuserlist_createlist()
void argmuserlist_additem(struct argmuserlist_publiclist* list,
                              char value[])
void argmuserlist_printatserver(struct argmuserlist_publiclist* list,
                              int item_idx)
char* argmuserlist_returnlistitem(struct argmuserlist_publiclist* list,
                              int item_idx)
```

Table 6 – Client-side public interface of the "user list" user library

The toy purpose of the *user library* is to store and access a list of strings on the server, so the functions in Table 6 respectively operate: to begin a new list, to add an item to the list, to print a particular item at the server, and to return a particular item from the list to the client.

Each of those functions has to be implemented by the *user library* programmer to send an appropriate message to the server to cause it to call a corresponding *remote function* on the server using the facilities provided by Argmessage.

Further, before they can be used the *remote library functions* need to be registered with the server *adapters*. For this the programmer of the *user library* has provided a function *argmuserlist_serverregisterfunctions()*, which registers them in the adapter function table in the same way as is done for the built-in functions, by calling *argmessage_serverengineregisterfunction()* for each function. *argmuserlist_serverregisterfunctions()* is called in the server program for this library, *argmessageserverregisterfunctionslist.c* described later in this section, just after the call to *argmessage_serverenginegetobject()*.

The user library requires some initialisation on the server side, which is provided in the function *argmuserlist_serversinit()*, which attaches to each *adapter* a state object, of type *struct argmuserlist_serverstate*, which comprises a count of lists created by the user application and for each potential list up to a pre-set maximum, a state for that list comprising a pointer to the first item of the list and a count of the items in that list.

Taking the first function of Table 6 as a first example of the *client-side public interface functions*, the code is:

```
struct argmuserlist_publiclist* argmuserlist_createlist(){
    struct argmuserlist_privatelist* newitem =
                     (struct argmuserlist_privatelist*)
                      malloc(sizeof(struct argmuserlist_privatelist));
    newitem->itemcount = 0;
    newitem->instance_id = argmlist_client_listcount;
    /* call to proxy of argmessage proxy */
    struct argmessage_message * message = packunpack_dummypayload();
    message->functionid = createlist;
    message->adapterid = proxy->myrank * ARGMESSAGE_ADAPTERS_PER_PROXY;
      // in this example we are only using the zeroth adapter of this
proxy
    argmessage_proxysendfunction(message);
    argmlist_client_listcount++;
    return (struct argmuserlist_publiclist*) newitem;
}
```

In the centre of this is the call to function *packunpack_dummypayload()*. Its job is both to create the Argmessage message buffer and to fill in some of its data items. In particular, reflecting its name, it packs the arguments to be passed to the *remote server function* into the *Argbuff* section of the message buffer, and sets the *Message size* and *Argbuffsize* fields, as the values of those are now determined, and sets other fields to -1, as a *not set* value. In this case the function has no arguments, so the pack function provides a dummy.

Next, further items of the message buffer are set. The *functionid* is an index identifying the server function to be called and the *adapterid* identifies the *adapter* that is to be used on the server to process the call. For clarity, and to avoid mistakes, the *user library* programmer should declare an *enum*, in the *user library* header file mentioned earlier in this section, for the different *functionid* values. The file *argmuserlist.h* includes one: *enum argmuserlist_remotefunctions*. For the programmer's convenience these can start at 0, which is the default value for an *enum*, since the built-in server functions have index numbers starting at a much higher value, which is defined by a constant set with as a C macro in the file *argmessagesizes.h*. As the comment in the code states, only a single *adapter*, the zeroth for the single proxy is used in this example. Finally, the Argmessage function *argmessage_proxysendfunction()* is called to send the message buffer to the server.

This example illustrates a use of client-side state by a *user library*. Instead of waiting for a reply message to receive a handle or reference to the new toy list as it exists on the server, the function *argmuserlist_createlist()* creates one, *newitem*, immediately and passes this object back to the application program for it to keep. *newitem* contains, in fact, not only the handle *per se*, *instance_id*, but also a count of how many items there are in the list, *itemcount*. This makes these data items readily available to the application code, but at the cost of duplicating and keeping synchronised information on the client and the server (the server will want to keep track of these values as well). In this case the cost is not that great because the *instance_id* is simply one more than the last one issued (kept in *argmlist_client_listcount*), which may not be much of a cost compared to not having to wait for a reply from the list creation function and also the application program not having to ask the server how many items are on the list if it needs to enquire.

The application programmer will have to note however that the *instance_id* and *itemcount* values are read-only to it because they are maintained by the library.

The toy list object is passed back to user as a pointer. However, for this step, it is first cast from *argmuserlist_privatelist* struct type to the *argmuserlist_publiclist* struct type, which may not have been wise. The public and private versions of this type differ by hiding the list's *instance_id* to avoid trying to present the application program with, in effect, duplicate information – a pointer and an *id* – and hiding a data item, the *id*, that the application program should not write. However, this is only partial encapsulation it does not make the field *itemcount* read only. It would have been clearer to not make the cast and just have a rule that the application programmer using the *user library* should respect what is read-only.

On the server side the message sent by *argmuserlist_createlist()* is received by the server engine which calls a corresponding function, *argmuserlist_servercreatelist()*, which has also been provided by the *user library* programmer. This retrieves the library's state and updates one of its items, *listcount*, by incrementing it, which also keeps this value in line with the corresponding value on the client. There is no other initialisation required here because the rest was already done in the general initialisation carried out by *argmuserlist_serversinit().*

The next function *argmuserlist_additem()* has the code:

```
void argmuserlist_additem(struct argmuserlist_publiclist* list, char
value[]){
    struct argmuserlist_privatelist* list_private = (struct
argmuserlist_privatelist*) list;
    /* call to proxy of argmessage proxy */
    struct argmessage_message* message =
argmuserlist_packoneintonestring(list_private->instance_id, value);
    message->functionid = additem;
    message->adapterid = proxy->myrank * ARGMESSAGE_ADAPTERS_PER_PROXY;
// in this example we are only using the zeroth adapter of this proxy
    argmessage_proxysendfunction(message);
    list_private->itemcount++;
    return;
}
```

This allows a user to add an item, which is a string, to a one of the lists identified by a parameter. The basic Argmessage library does not include a *pack function* for a signature comprising an int and a string, so the programmer of this toy *user library* has provided one, *argmuserlist_packoneintonestring()*, which follows the standard pattern for these functions. Once the Argmessage message buffer has been sent to the server, that calls the corresponding *remote server function*, *argmuserlist_serveradditem()*, which creates a new object, of type *argmuserlist_serverlistitem*, to store the new list item and links that into the linked list structure used to store the particular list that begins with the pointer to the first item of that as given in the state object attached to the *adapter*. The client and server functions both increment their own counts of how many items there are in the particular list in order to have that information available locally.

The final two functions of Table 6, *argmuserlist_printatserver()* and *argmuserlist_returnlistitem(),* perform the functions their names suggest but do not illustrate any further points. The user library also includes a function *argmuserlist_serversfree()* to free the data structures used to store the lists on the server in the finalisation stage.

This example also has, of course, exemplary client and server application programs. Compared to the previous example of an Argmessage program this example includes a *user library*. The server application program, *argmessageserverregisterfunctionslist.c*, is similar to that of the previous example, *argmessageserverbare.c*; it therefore includes the library and has calls to *argmuserlist_serverregisterfunctions()*, to register the *remote user library* functions in the function table for each *adapter*, and a call to *argmuserlist_serversinit()*, to initialise the server-side state of the *user library* (as described in Appendix A at section 17.8). The client application program, compared the that of the previous example, includes the *user library*, and then when in Argmessage’s run mode it has example statements to create lists, add items to them, print them at the server and retrieve items; otherwise, the backbone of calls to initialise, run and finalise the Argmessage *proxy* remain the same.

9.3. Correcting the QR factorisation example from QuickSched

A correction of the tiled QR example from the original QuickSched, as obtained from its Source Forge repository [15], was briefly mentioned in sections 8.2 and 9.1. The difficulty with the example was that it did not pass its own functional verification test. The error was traced to the section of the task building section of the example code; the malfunction did not touch on the QuickSched library itself. The task building code from the QR factorisation example from the original QuickSched repository is reproduced in Table 7, with key lines in the building of the task graph in bold and with colours added to match the colouring scheme in Figure 4.10/Figure 7 of the references by [16] and [50], and hence that of the task graphs of Figure 8 and Figure 9. An ellipsis character is used to mark sections omitted for brevity. In the code, indices *k*, *i* and *j* are respectively the level, the row of the current tile in the tile grid, and the column of the current tile.

```
void test_qr(int m, int n, int K, int nr_threads, int runs, double*
matrix) {


…

  enum task_types {

    task_DGEQRF,
    task_DLARFT,
    task_DTSQRF,
    task_DSSRFT

  };
…

  /* Build the tasks. */

  for (k = 0; k < m && k < n; k++) {


    /* Add kth corner task. */

    data[0] = k;
    data[1] = k;
    data[2] = k;

    tid_new = qsched_addtask(&s, task_DGEQRF, task_flag_none, data,
                             sizeof(int) * 3, 2);

…

    /* Add column tasks on kth row. */

    for (j = k + 1; j < n; j++) {
      data[0] = k;
      data[1] = j;
      data[2] = k;
```

```
   tid_new = qsched_addtask(&s, task_DLARFT, task_flag_none, data,
                                sizeof(int) * 3, 3);

…

  /* For each following row... */
  for (i = k + 1; i < m; i++) {

   /* Add the row taks for the kth column. */

   data[0] = i;
   data[1] = k;
   data[2] = k;

   tid_new = qsched_addtask(&s, task_DTSQRF, task_flag_none, data,
                                sizeof(int) * 3, 3);

…
   /* Add the inner tasks. */
   for (j = k + 1; j < n; j++) {
    data[0] = i;
    data[1] = j;
    data[2] = k;

    tid_new = qsched_addtask(&s, task_DSSRFT, task_flag_none, data,
                                 sizeof(int) * 3, 5);

    qsched_addlock(&s, tid_new, rid[j * m + i]);
    qsched_adduse(&s, tid_new, rid[k * m + i]);
    qsched_adduse(&s, tid_new, rid[j * m + k]);

    // qsched_addunlock(&s, tid[k * m + i], tid_new);
    qsched_addunlock(&s, tid[j * m + i - 1], tid_new);

    if (tid[j * m + i] != -1) qsched_addunlock(&s, tid[j * m + i],
tid_new);

    tid[j * m + i] = tid_new;
   }
  }

 } /* build the tasks. */
 …

 /* Loop over the number of runs. */
 for (k = 0; k < runs; k++) {


  /* Execute the the tasks. */

  …
 }
```

Table 7 – Annotated code from file test_qr.c of original QuickSched library

The yellow arrow in Table 7 marks a code line for adding the task dependencies from the DTSQRF task on each row to each of the DSSRFT tasks in the same row, which in the repository version was unfortunately commented out.

With the line uncommented, the example QR factorisation does pass its verification test. Having found the error it is also reassuring that the line is uncommented in the version of the task creation code of Figure 14 of Appendix B of [50].

9.4. Conclusions of QR factorisation task graph generation code analysis

Having corrected an error in the original QR test of the QuickSched code, there was now a version that would make a realistic test of the QuickSched in its new form, Qsargm. It also sheds some light, by example, on how *locks* and *uses* should be assigned. Finally, it allows reasonable speculation, as noted in this section, on how an incorrect attempt at correcting the incorrect QR test might affect the performance of QuickSched.

So, with respect to *locks* and *uses*, the key difference between them is that while both inform the scheduler about which data are being accessed and so therefore are likely to be in a processor core’s cache, and so provide hints to the scheduler as to which core to use for a task, a *lock* is also used to prevent concurrent updates, where that is not prevented by the task dependencies.

Unfortunately, while working to correct the error, further, unnecessary changes were made to the task graph creation code. Awkwardly, these incorrect corrections did not affect the correctness of the QR factorisation calculation and, as a consequence, were not immediately detected and so were in the code used in this project to generate earlier ones of the experimental results, but of course they did provide a contrast to the main results.

The errors were that for each blue task, instead of specifying a use of the top-left tile, a *use* of the tile in the row above in the same column was specified, and for a yellow task, instead of specifying a *use* of the tile at the top of the same column, a use of the tile in the row above in the same column was specified. As these are only *uses*, this explains why they did not affect the correctness of the QR factorisation. They might, however, cause a task to be assigned to a less efficient queue because the task has thereby been associated with data that it is not in fact using, and in turn this might cause additional cache misses when the task is assigned to a core. That will only of course be a significant degradation in performance if this mechanism to avoid cache misses was in fact already providing significant improvements to performance.

10. MPI Message Latency

A key quantity in the performance of the experimental QuickSched library, using Argmessage, is of course the *message latency*, i.e., the time taken for a message sent from one of the client or server processes to arrive at the other. This definition is taken to mean the time for the whole message to arrive and be available for use at the destination. It therefore will, in general, be dependent on the size of the message, since the data of messages are transmitted serially.

This definition is consistent with the “latency” quantity as measured by the OSU Microbenchmarks [52] and with, for example the definition given at item 1 of section 1.3 of “Interconnection Networks, An Engineering Approach” [53]. (Note the contrast with other definitions of latency in which the time for a signal to propagate, or a response begins to be received, is measured, which is not affected by the subsequent length of the rest of the signal, for example the length of the block of data being transmitted. For example, this paper [54] about "low latency data transmission” uses latency to mean the signal propagation delay and measures the quantity in µs/km.)

For most of the experiments with Qsargm, OpenMPI was used as the message layer for Argmessage. The *message latency* of OpenMPI was therefore measured for each of the experimental *geometries* (which are defined in detail in section 11.2), in particular using the OSU Microbenchmarks [52].

To allow close comparison to the clocks used in the experiments of the current project, the OSU benchmarks were compiled and run using essentially the same build and run scripts as were used for the experiments. Therefore, they were compiled and run using the same OpenMPI and UCX libraries. The benchmarks were also compiled separately for the different processor architectures used.

For completeness, the derivation of the clock used by the benchmark was checked in the code of its sources and indeed is valued in microseconds, as follows. The benchmark *pt2pt/osu_latency* benchmark from the OSU Benchmarks obtains clock values from the OpenMPI *MPI_Wtime()* [55] function. In turn this obtains its clock reading from the OpenMPI function opal_clock_gettime() [56]. In turn that obtains its clock value from either *clock_gettime()* of *time.h* [57] or *gettimeofday()* of *sys/time.h* [58]. Both of those return their clock value as two-part structs of seconds and nanoseconds or microseconds and this value is passed up the chain with *MPI_Wtime()* presenting it as a combined double precision value in seconds. The benchmark scales that to microseconds.

From inspection of the benchmark code, the latency time it measures is half the average time taken to do a round trip: an MPI_Send and MPI_Recv on MPI rank 0 and a corresponding MPI_Recv and MPI_Send on MPI rank 1. (MPI ranks 0 and 1 are respective Linux processes in their respective locations that are started by MPI's *mpirun* command to run respective copies of the benchmark program in those processes.) The benchmark measures the message latency for a range of message sizes (in bytes).

The results were taken for various locations of the two MPI ranks (matching the experimental *geometries* defined in section 11.2) in the *jupiter* and *thor* cluster systems at the HPCAC; details of these systems are discussed in more detail at section 11.4.

The *message latency* measurements from the OSU benchmark showed that there is a minimum message size below which the latency does not reduce further and that the value of this minimum is dependent on the *geometry*. That and other key values for the message latency are given in Table 8. The minimum message latency is for the rank 1 process of the benchmark being on the other socket of the same host as the rank 0 process, with that message latency being 0.3 μs for *jupiter* and 0.4 μs for *thor*. Using a BlueField card for the rank 1 process increases the message latency to 1.4 μs for a local BlueField and to 1.6 μs for a remote one on another host. Interestingly, on one of the machine types, *thor*, for the server on the remote host (thor005 to thor006), the messaging is quicker than to a local BlueField (thor005 to thor-bf05).

Cluster and geometry	Minimum	32-byte messages	64-byte messages
thor005 to thor005	0.38	0.5	0.51
jupiter029 to jupiter029	0.30	0.48	0.49
thor005 to thor-bf05	1.39	1.48	1.56
jupiter029 to jupiter-bf29	1.35	1.45	1.54
thor005 to thor006	1.20	1.23	1.31
jupiter029 to jupiter030	1.66	1.73	1.81
thor005 to thor-bf06	1.64	1.73	1.81
jupiter029 to jupiter-bf30	1.60	1.72	1.81

Table 8 – MPI message latency (µs) from the benchmark

Values for the 32- and 64-byte message sizes of the benchmark are also noted and are relevant as they are similar in size to the Argmessage message used in Qsargm as discussed at Chapter 13, so will be better values than the minimum for comparison to the Qsargm RPC latency in the experiments of this project.

10.1. Relationship to Argmessage / Qsargm RPC calls

In Argmessage, to receive the answer to a function call two messages are needed, a first to call the function and a second to return the answer. So, for such calls the messaging overhead will be **twice** the values in Table 8, or at least that as the use of OpenMPI may be slightly more complicated there than that of the benchmark, and further there will be the extra overhead of packing and unpacking the arguments sent.

10.2. Comparison to QuickSched function calls

The equivalent operation in the original QuickSched scheduler is to call a function. That itself may have no overhead if the function is in-lined by the compiler. If it is not the overhead may be O(10ns), so O(100) times faster than the messaging introduced by this project.

10.3. Speculations on payoffs for message latency penalty

These patterns of message latencies are considered later in Chapter 13 when discussing the experimental results of the modified QuickSched scheduler, Qsargm, in actual measured examples. However, the following remarks and speculations can be made at this stage.

If the messaging delay is indeed significant in the scheduler performance, there will need to be some pay-off to make having the scheduler in a separate process worthwhile. This may be of two general kinds:

- Processing of the scheduler in parallel with computation of kernels on the thread(s) calling the scheduler
- Extra scheduler functionality

The latter is considered in Chapter 14.

Based on the message latency results, it is a reasonable conjectureTable 8 that the best place to locate the separate scheduler process, to avoid the latency penalty, is to put it on dedicated cores of the same host that is running the client computational tasks. Doing this however would reduce the number of cores available to the client, which in general would reduce performance. However, not all codes will have enough tasks in their task graph to keep a large number of cores occupied. For such codes allotting host cores to the server process is free.

Locating the scheduler process on the BlueField card does increase the messaging time still further but these cores are not, without extra arrangements being made, available to process the computational kernels. Thus, cores on the BlueField are always free in that sense, even for codes that are able to use all of the cores of the host processor.

However, in the basic arrangements described so far and studied in the experiments, while they have the computation and scheduling performed in separate processes on separate cores or processors, these do not particularly seek to overlap the scheduling of the processes with the computation in time; when a computational core asks for a new task it still waits while the scheduler identifies the new task. Suggestions about how overlap might be achieved are however discussed in Chapter 14.

While there are small differences in the messaging latency values for the rank 1 process of the benchmark on the local BlueField, on the remote BlueField, and on the remote host, these values are generally similar to each other. This suggests that it may be possible to run the scheduler for several client hosts on a common BlueField card or on a common dedicated host without much degradation in performance. The multiple schedulers will each receive messages from their respective client so the number of messages to the common BlueField card or host will be multiplied up, but the messages are small and infrequent in smaller cases and so the messaging may well not be a bottleneck – there will be messaging limit but it is not speculated whether that will take effect before the capacity of the scheduler processor is exhausted. A common location to process multiple schedulers might however offer a significant advantage when it comes to organising work stealing between computational hosts: this advantage is that all, quite up to date, information on the state of progress on the tasks is available in one place, the common location of the schedulers, and so quite complex processing of that could be considered. That possibility has, however, not been explored in any detail in this work.

Another use for Qsargm, in particular in the local host *geometry* for the scheduler, might be for use on host processors where the cores are not symmetrical. Examples of such processors include Intel's latest[2] 12th Generation Core processors (codename Alder Lake – first released at the end of 2021). These are provided with two classes of cores, called by Intel, "Performance-cores" and "Efficient-cores" [59]. In the Qsargm model, the processing of the scheduler on the one hand and the processing of the kernels on the other are not symmetric. So scheduling, being a less demanding operation, should perhaps therefore be allocated to one of the Efficiency cores. However, if that significantly affects the latency of the scheduling calls then perhaps, counterintuitively, it might turn out that one of the Performance cores should be allocated. Experiments would be needed.

[2] "latest" at the time of writing –13th generation parts are due to be released in the final quarter of 2022 and the first of 2023 and will continue the performance and efficiency distinction.

In any event, such new processor designs could pose significant challenges to the scheduling model of both of the original QuickSched and Qsargm, which assumes equal performance of the cores in applying the scheduling rules – the tasks have *costs* and *weights,* but these are not adjusted for likely performance of whichever core might in future be allocated to them, which in turn would not be straightforward to predict. The allocation of the scheduler to one or other of the core classes does not alleviate this problem because the two classes are provided by the manufacturer in similar numbers. Although the Performance ones allow hyperthreading whereas the Efficient ones do not, meaning the majority of threads are on Performance cores, there are still significant numbers of Efficiency cores, so scheduling of tasks will have to be on to a mix of Performance and Efficiency cores. The problem is in fact worse than that in that Alder Lake processors also have what Intel call “Turbo Boost Max Technology 3.0” [59], which recognises the difference in clock speeds that each individual Performance core is capable of and schedules threads to them accordingly; this is not just a static determination but dynamically takes into account factors of many kinds [60]: local physical environment factors of power consumption and temperature; “type of workload” so presumably based on the instructions of the software being executed; and the number of cores active, which in task-based scheduling is a result of the task schedule itself, making the problem recursive.

Having asymmetric cores is not confined to Intel processors. Many Arm processors have different cores of different types [61], and there is increasing interest in building HPC systems using Arm processors.

So, in future, more complex models may well be needed for efficient task scheduling in HPC applications.

11. Experiments - general arrangements

11.1. Test Program

The experiments were conducted with the tiled QR matrix factorisation example using both the original QuickSched scheduler and the modified Qsargm scheduler of this project, which, of course, places the scheduler in a separate process remote from the process executing the computational kernels, with calls to the scheduler being made via the Argmessage remote procedure library of this project. The original version of QuickSched employed was version 1.1.0 from Source Forge.

11.2. Test Geometries

Figure 10 is a block diagram showing the *geometries* used in the experiments for the location of the scheduler relative to the computational kernels used. The host computers used were “dual socket” meaning that the mother board had two processor integrated circuits. The processors had x86-64 architectures. Both of those are the norm in HPC clusters. Further, also as usual for HPC, the *dual socket* host processors were identical processors and shared the same memory space for the main RAM attached to them on the motherboard. In addition, each host computer, or ‘node’, was equipped with a BlueField™ card. These are fairly new to the market and are only becoming available in a few experimental HPC systems. These BlueField cards combine a network adapter (Ethernet and/or Infiniband), in particular a Connect-X™ adapter, with an Arm architecture processor on the card with its own RAM in its own address space and with its own Linux operating system, on which user programs may be run. Precise specifications of the machines and interconnect used are given in section 11.4.

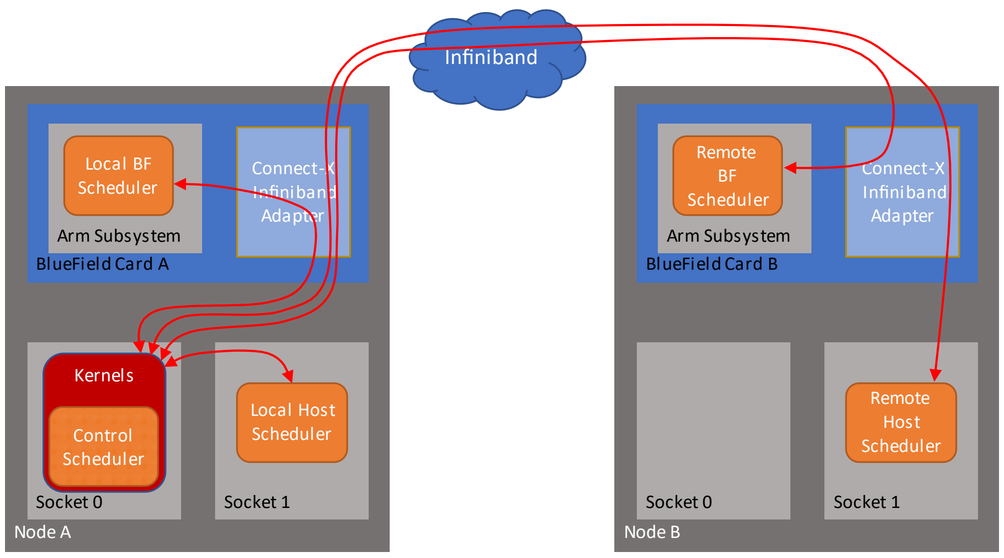


Figure 10 – Experimental geometries, i.e., relative process locations

In the experiments, the threads of the process running the computational kernels were pinned to cores of one host processor integrated circuits (socket 0 in Figure 10) as intended by both the original and modified QuickSched. Four possible locations were used for the *experimental* scheduler process. Those were:

- A process having its threads pinned to the cores of the other processor integrated circuit of the host (socket 1 in Figure 10) in the same node (Node A) as the computational kernels process. This geometry is designated *local host scheduler* in this thesis.
- A process running on a BlueField card (BlueField Card A) mounted to the same mother board as the processor running the computational kernels, designated *local BlueField scheduler*.
- A process running on a BlueField card (BlueField Card B) mounted to the mother board of another compute node (Node B) of the cluster, designated *remote BlueField scheduler*.
- A process having its threads pinned to the cores of a processor integrated circuit (socket 1) of the host of the second compute node (Node B), designated *remote host scheduler*.

Thus, *geometry* is defined here to include more than just the topological aspects of the network connections, which would be the pattern of available connections, and, of those, the route being used, but includes also the physical conditions and types of electronic circuit along the route. So, this encompasses the type and length of network cables, and the network protocol being used, the model of the network adapters being used, whether network transmission is being used at all or rather inter-process communication via shared memory, whether there is any effect of CPU socket or NUMA region, how many PCI switches and interfaces there are on the route, and so on.

Only one of these locations for the scheduler was used in any particular run of the QR factorisation code. The long red arrows in the diagram show the paths of the messages between the computational kernels and the scheduler in each of those locations. Experimental runs were repeated in these different *geometries* for comparison.

Finally, a 5th location for the scheduler, a *control*, was used. This was the original QuickSched scheduler, so without the modifications of this project, and so the scheduler operates within the same process as the computational kernels. In this, each thread, when done with its latest task, becomes the scheduler object for a time while it works out which task the thread should perform next. The purpose of including this in the experiments was to provide a comparison basis for the experimental scheduler: i.e., is it any better than the original?

Communications between the computational kernels and the *experimental* schedulers, in contrast to the *control* scheduler, were conducted through a messaging library, which was of course the Argmessage library of this project. That library, however, requires a base messaging library to carry its messages. The main one used was OpenMPI, and within that use of UCX messaging was specified. These libraries make choices for themselves on exactly how to conduct the communication in any particular situation; however, messages between processes on the different host nodes were transported via the InfiniBand interconnect, with, in particular, the Connect-X InfiniBand adapter portion of the BlueField card being selected.

In the *local host geometry mode*, messages between socket 0 and 1 are transferred via shared memory. On the other hand, when messages are transferred between a host processor and a BlueField card or a remote host, that traffic is sent over the Infiniband network. However, for the *local BlueField scheduler geometry*, it was not clear whether the messages were transferred from the host processor to the Arm processor directly over the PCIe Bus, as might be expected for minimum latency, or went first from the host processor to the Connect-X Infiniband interface on the BlueField card and then from there to the Arm subsystem. The latter is what would be expected if the OpenMPI and UCX libraries do not have special code to take advantage of the direct route via the PCI bus only.[3]

Another communication library used, as an alternative to OpenMPI, underneath Argmesssage was the SNAPI messaging library, under development at Nvidia, which was made available to me. This is designed for communication between a BlueField and its host and so was only used for the geometry of having the scheduler on the local BlueField. It was designed to have a simple programmers' interface and is based on the InfiniBand verbs communication library [62]. It was used because it was plausible that it would be lighter weight than OpenMPI.

The *control* scheduler does not, by definition, make use of Argmessage, nor any messaging library at all. The equivalent of the messages is simply function calls, the original ones, to the QuickSched scheduler library.

The purpose of testing these various locations for the scheduler is to determine the extent to which their relative remoteness from the computational kernel process in terms of the messaging latency affects operation.

[3] An attempt to clarify this point with the BlueField manufacturers was not successful.

11.3. QR experimental parameters

The launch script for the experimental code deals with experimental parameters, which is detailed in in Appendix A at section 17.14.

The parameters passed to the QR factorisation code itself were as set out in Table 9. In the experiments these are all specified in a set for one run[4], being a line of the parameter file specified with the -g option of the launch script. The parameter file comprises many such sets for respective *runs* to build up a result data set, as explained in sections 11.5 and 11.6.

Parameter Name	**Description**
-T	The number of threads allocated to the scheduler program.
-t	The number of threads allocated to the computational kernels, i.e., to the application program.
-r	The number of repeats of the application program calculation, i.e., one repeat = 1 QR factorisation.
-k	The tile size of the QR factorisation algorithm. The tile has k × k matrix elements.
-n -m	The matrix factorised has n × m tiles. In all experiments n = m was used. So, the size of the matrix was (n × k) × (n × k) elements in total.
-S	"Strategy" – 0 = use experimental scheduler, 99 = use original QuickSched inside the application's client process, i.e., in the compute threads.

[4] A "run" comprises a number of repeats, as specified by -r.

-v	Run the verification part of the QR program – this is a test to verify the result of the QR factorisation using just a standard library function – this test was available in the original QuickSched library. This was used at first to check that the QR factorisation was running correctly but was turned off later to save time, thereby allowing more parameter sets to be tested.

Table 9 – QR factorisation example command line parameters

The parameters *m*, *n*, *k*, *r*, *t* and v were available in the QR factorisation test in the original QuickSched archive.

11.4. Machine Specifications

The experiments were carried out on compute nodes of the clusters at the HPC-AI Advisory Council [63], on two kinds of machine. The *jupiter* machines used in some experiments are stated [64] on their website to have this configuration:

- “Dual Socket Intel® Xeon® 10-core CPUs E5-2680 V2 @ 2.80 GHz
- NVIDIA ConnectX-4 EDR 100Gb/s InfiniBand adapter
- NVIDIA Innova-2 Flex Open VPI, dual-port QSFP28, EDR / 100GbE, KU15P (7 adapters)
- NVIDIA BlueField SmartNIC VPI MBF1L516A-ESNAT, 100Gb Ethernet / InfiniBand EDR (8 adapters)
- NVIDIA Switch-IB 2 SB7800 36-Port 100Gb/s EDR InfiniBand switches
- Memory: 64GB DDR3 1600MHz RDIMMs per node”

The BlueField part number MBF1L516A-ESNAT quoted there has a specification [65] of:

- “16 Core Arm Processor, with
- 16GB DDR4 RAM
- EDR Infiniband”

This BlueField card is part of the BlueField 1 Series, but in fact the *jupiter* machines contained BlueField-2 cards as noted in this section.

The Thor machines used in some of the experiments are stated [64] to have this configuration:

- “Dual Socket Intel® Xeon® 16-core CPUs E5-2697A V4 @ 2.60 GHz
- NVIDIA ConnectX-6 HDR100 100Gb/s InfiniBand/VPI adapters
- NVIDIA BlueField-2 SoC, HDR100 100Gb/s InfiniBand/VPI adapters
- NVIDIA HDR Quantum Switch QM7800 40-Port 200Gb/s HDR InfiniBand
- Memory: 256GB DDR4 2400MHz RDIMMs per node”

However, on all the BlueField cards used (*jupiterbf001,002,029,030,031,032 and thor-bf01,02,05,06,07,08*) the CA Type for the BlueField card is given as MT41686 by the *ibstat* command but *lspci* gave the number MT42822 for all the items found. The former of those numbers is used on the Mellanox website [66] for a Bluefield-2 card having the part number MBF2M516A-EEEOT [67] and the latter appears on the Nvidia website [68] also in the context of a BlueField 2 card. Therefore, the HPCAC website was out of date in respect of the BlueField cards in the *jupiter* nodes and the cards used in these experiments were all BlueField 2. In the results archive for the experiments in the provenance sections there is a log of the output of the command *ibstat*, and these give the CA type of the adapter used as MT41686.

The experiments reported in Chapter 13 were run on the nodes of these systems with these machine names:

Machine sets
thor005, thor-bf05, thor006, thor-bf06
jupiter001, jupiter-bf01, jupiter002, jupiter-bf02
jupiter031, jupiter-bf31, jupiter032, jupiter-bf32

Table 10 – Set C and set F experiments – machine sets

Each of these three sets of four nodes comprise two x86-64 architecture host nodes and two BlueField cards, each BlueField node being plugged into a respective one of the host nodes. The BlueField cards were set up to have separate network addresses from their hosts, so they appear as separate machines with separate machine names on the local network and in the cluster scheduler system, which is known as *separated host mode* [69]. The BlueField card nodes have “-bf” as part of their machine names, with the BlueField card having the number part of its name matching that of its respective host.

Each group of four nodes by itself provides the possibility for all the test *geometries* described in section 11.2. In the Table the lower numbered x86-64 host was the one running the client program that executes the computational kernels.

11.5. Parameter File Generation

A Python program *gen-args.py* (archive: expt0069/argmessageprivate/run-scripts-HPCAC/qr_args/arg_sets/gen-args.py – see in particular the functions *QR_args_set1()* etc. therein) was written to generate and format sets of parameters, stored in a file, for passing to the application program, the QR factorisation. To generate a parameter file, the user supplies a list of parameter names and, for each of those a list of values the parameter is to take, and the program then uses Python’s *itertools product* function to generate a Cartesian product of those lists (i.e., all combinations of the possible values, one from each list). For parameters that are to remain constant, a list comprising just that constant value is supplied. However, parameter sets with a relationship between parameters were needed; $n = m$, as mentioned in Table 9, is an example, so it is also allowed in the parameter generating program to specify a transform to each parameter set to be applied after the Cartesian product. So, for the $n = m$ example, only a single parameter x is specified by the user in place of n and m in the Cartesian product and then the transform is applied to the x parameter to derive the corresponding m and n parameters, giving both a value equal to that of x. The resulting parameter files are to be found at archive:expt0069/argmessageprivate/run-scripts-HPCAC/qr_args/qr_args/ and it is those files that are directly consumed by the launch scripts.

Some of these sets have, as an outermost repeat, the entire parameter set generated by the mechanism of the immediately preceding paragraph repeated four times in the file. The point of this outermost repeat is that if a *run*, which itself would repeat the experiment with the same parameters a number of times defined by the -r parameter, were to be adversely affected by some infrequent event in the computer then hopefully the outer repeat of that parameter set at another time would not be similarly affected.

11.6. Types of Parameter Sets Used

The main experiments conducted were:

(i) to vary the number of client application threads *t*. Increasing the number of application threads (for the same size of matrix) increases the number of tasks that may be processed in parallel. More threads should therefore decrease the execution time, but only up to the point where the number of threads exceeds the maximum number of ready tasks generated at any one point in time in a run.

(ii) to vary the number of tiles $n \times n$, (with $m = n$ always being used). Increasing *n* increases the size of the problem, i.e., the size of the matrix being factorised. It also increases therefore the number of tasks and hence the number of messages required between the kernel threads and the scheduler. Both of these will, at a constant number of threads *t*, increase the total execution time.

While, generally, both of these, *t* and *n*, were varied over various ranges, plots were made of execution time against *t* at constant *n*, and so therefore at a constant matrix size. Such a plot therefore is a *"strong scaling"* test [70], which is of the performance at constant problem size against the number of processors. Plots were also made against *n*, so against increasing problem size, at constant *t*. (That is however not a *"weak scaling"* test [70], which requires the problem size to be increased with an assumed *"constant workload per processor"*.)

In these experiments the number of scheduler threads, as opposed to the number of client threads, was usually fixed at 1.

The number of repeats of each experiment (the inner -r repeat) was usually 50 or 1. As explained in section 11.5, in some cases the same parameter combination was repeated four times, spaced over an extended period.

While the tile size was usually fixed, another kind of experiment was to keep the overall matrix the same size but to change the tile size. One would expect a complex interaction here since a larger tile size will make each task longer but reduce the number of them.

11.7. Timers used

The QuickSched library provided some timers for the execution various aspects of the code. These were extended to allow for the scheduler being in a separate process and were adapted take account of the RPC messaging loop.

So, the timers that were used in the present project are set out in Table 11.

Timers for control (Original QuickSched)	**Timers for experiment (Remote scheduler – Qsargm)**
clt_gettask - aggregate time inside _*gettask()* function clt_done - aggregate time inside _*done()* clt_prepare - aggregate time inside _prepare() “clt” = “client local timer”, meaning timers for the original control scheduler	cpt_qsargm_client_remote_gettask - aggregate time inside _*gettask()* function cpt_qsargm_client_remote_done - aggregate time inside _*done()* cpt_qsargm_client_remote_prepare - aggregate time inside _prepare() cpt_qsargm_client_kernels - aggregate time inside kernels “cpt” = “client proxy timer”, meaning that the timer is located in the proxy on the client (so on the left hand side in Figure 7) and so these timers include the calls across the messaging link, except for the kernels timer which does not involve an RPC call
Timers common to control and experiment	
taskgraph_build_time	taskgraph_run_time

Table 11 – timers for control and experiment runs – names after post processing

Thus, the new timers are: *taskgraph_build_time* and *task_graph_run*, which were provided in the *test_qr.c* code to provide a direct comparison between the experiment and control for those stages of operation. These were both measured with the *omp_get_wtime()* function, whereas all the others were measured with the *getticks()* function.

The original QuickSched library used the function *getticks()* function for all its timers. This function is to be found in the source file cycle.h, which contains many definitions of the function depending on processor architecture, compiler and operating system being used. For the *gcc* compiler and the x86-64 architecture this is defined as the x-86 assembler instruction *rdtsc*. (This version is selected by the presence of the macros __GNUC__ and __x86_64_ which are set by default by the GNU C preprocessor, as verified by the command *echo | cpp -dM -*, as suggested by [71].) The Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2, available at [72], at section 17.15, describes the timer accessed with the *rdtsc* instruction, which states that for newer Intel processors this clock runs at a constant rate and so may be used to indicate wall time. [73] states that on Linux the presence of this feature is indicated by */proc/cpuinfo* including the flag *constant_tsc*, and indeed on both the *thor* and *jupiter* host nodes that is present in the *flags* field. The duration of these ticks is set by the frequency of some clock signal and so that frequency needs to be determined if this timer is to be compared to the timers measured with other timer functions.

In order to relate times measured by *getticks()* to those of other timers given natively in seconds, i.e., (i) the OpenMPI message latency values and (ii) the timers in the experimental code measured with *omp_get_wtime()*, the frequency of the *getticks()* clocks was measured for *jupiter* and *thor* hosts and for the BlueField cards. The method was to sample all of the clocks involved, once each, in quick succession spaced before repeating after intervals of a nominal second, over a total of 20s. The results are shown in Figure 11.

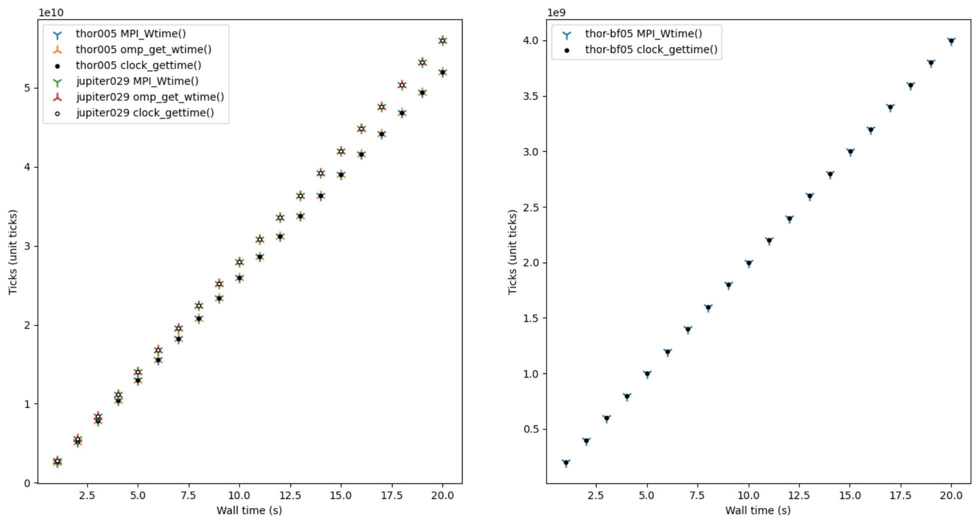


Figure 11 – Samples of getticks() against other clock functions, which are generically labelled "Wall time"

The measurement recorded the ticks given by *getticks()* and the wall time given by each of the functions *omp_get_wtime()*, *clock_gettime()* and also *MPI_Wtime(),* the latter being used by the message latency benchmark. The ticks were plotted against the other wall time measurements. The relationship was highly linear, confirming that the rate of ticks was indeed constant, so for example not being changed in response to processor load. The rate of the ticks was found, from the slope of the graph, to be as follows:

Machine	**getticks() frequency against other wall time functions (MHz)**		
	omp_get_wtime()	gettime()	MPI_Wtime()
thor005	2599.98588 ± 0.00004	2599.98599 ± 0.00004	2599.98595 ± 0.00004
jupiter0029	2799.91905 ± 0.00033	2799.91905 ± 0.00033	2799.91876 ± 0.00005
thorbf005	Not measured	199.965974 ± 0.000008	199.965975 ± 0.000008

Table 12 – getticks() frequency

While these values are highly precise, they are only for one particular one of each nominal type of machine and only for a particular occasion; it is not known what the variation was over time and between machines of the same kind. It was however also noticed that the measured rate of the *getticks()* clock was extremely close to the nominal clock rate of the Intel processors as revealed by the *model name* field of /proc/cpuinfo. In view of the potential variation stated, the nominal cpu frequencies, with their much lower precision, were used to scale the *getticks* values to seconds in the experimental plots and these uncertainties were kept in mind should the need to compare different machines or measurements on different occasions, which was in fact mostly avoided.

The clock for BlueField at 200Mhz is much slower than the nominal clock speed. That would be important to note if the timers on the BlueField were used, but as mentioned the timers on BlueField were not in the analysis made.

11.8. Automation Frameworks for Computational Experiments

The workflow that developed for the experiments conducted was as follows:

- Build code
- Generate parameter files
- Batch job for a set of parameter sets and geometry
    - Record provenance
    - Heterogeneous machine launch
    - Run experiments for each set of parameters from a parameter file
- Post-processing
    - Extract result directory logs to structured JSON file
    - Aggregate JSONs
    - Query aggregate results
    - Plot

The creation of scripts to perform, at least some of these steps undoubtedly repeated work done elsewhere, since relevant frameworks do exist and should provide the benefits of quality software including ease of use and reproducibility. One such, which has been suggested, is the FabSim3 framework [74], [75], [76]. This is aimed, according to the emphasis in its literature, to running physical simulation codes on large HPC sites. Through templates and plugins it provides mechanisms for generating and submitting job scripts for cluster schedulers and, further for executing ensembles of jobs with difference input parameters. The job script generation would have been useful in this project and would have made the experiments more portable to other systems (although the latter was not a requirement of this project itself). The ensemble of cluster jobs feature would have been useful for running the experiments in the different geometries but would not have been so for stepping through the number of threads and tile parameters, which occurs too frequently - relaunch of the code that often would not have been efficient. The recording of provenance it provides was of the software environment, which was done so that would have been useful, but in this project hardware configuration was also interrogated and recorded. There is no reason to think that the framework would address the particular difficulties faced in launching the code components in this novel and heterogeneous hardware environment as outlined in section 17.14.

The documents cited seem to indicate that postprocessing steps are included in the scope of the framework but the details of extraction of results from output logs is not discussed, but perhaps that is left to plugins for individual codes.

The conclusion is that while using existing frameworks for running experiments must be desirable, it may, for this kind of project, be impractical in terms of recognising the requirements *a priori* and finding (possibly multiple) frameworks that fit all the needs of the project. In the case of FabSim3, it would appear of course to meet the needs of running well known simulation codes on large HPC facilities. Further the usefulness of this kind of framework is multiplied where there is a large community of users to share plugins and other configuration files.

12. Performance Models

The operations of building and executing task graphs are complex. The performance of those two phases are considered here, particularly with regards to the effects of messages, which of course are introduced by Qsargm, before the results of the experiments with the QR factorisation example are presented and analysed in Chapter 13.

12.1. Performance of the build phase.

The number of tasks in a task graph is relevant to performance in both the build and execution phases.

Working from Chapter 8 explaining the QR factorisation example, or equally from the references cited there, and from the code of the test QR example (file qsargm_test_qr.c in Appendix D), and counting up the number of tasks created (calls to the function *qsched_addtask())*, that number, N_G is:

$$N_G = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Equation 1 – number of tasks N_G in QR factorisation for a n x n

where the number of tiles in the matrix is $n \times n$.

The number of tasks is cubic in *n,* and since it has all the coefficients of all its powers of *n* positive, it has the general form of increasing with *n* and that with a positive and increasing gradient.

A further detail relevant to the build phase is that for each call to *qsched_addtask()* to create one task in the task graph, around 4 calls are made to the other functions for building other aspects of the task graph, namely to the functions: *qsched_addlock(), qsched_addunlock(), qsched_adduse()*, the exact number of such calls depending on the task type; however since as *n* increases one type of task becomes strongly the most numerous, it would suggest that the variation in the number of those extra calls between task type will not change the general form.

As noted at section 7.12, the task graph build phase in the original QuickSched was a single thread and this was preserved in the Qsargm scheduler of this project. The time to build the QR factorisation task graph will therefore be generally the number of tasks as given in Equation 1 times the time to build each task. The time to build a task is the sum of the times to calculate the parameters of the task, make the calls (which will be OpenMPI messages in the case of Qsargm), and then process the calls by adding data to the relevant tables of the scheduler's representation of the task graph. In the QR factorisation the parameters are cheaply calculated from simple rules and adding data to tables is not complex. This means that messaging time could well be significant. How significant that is reported in Chapter 13.

12.2. Run time performance in the many cores limit

[16] mentions that a strategy to optimise the overall task graph execution time is to prioritise the tasks on the predicted longest path to the final task; indeed, the longest path through a task graph is an irreducible minimum time for the overall execution of a task graph. This of course applies to a *static task graph* – if a task graph may have further tasks added to it then the longest path is not determined.

In the limit of a large number of cores there will always be an idling core available to process immediately any new ready task; that of course includes tasks on the longest path, so when a task on the longest path finishes, the next task on the longest path will execute immediately. That next task will always be a ready task at that point since, if it had to wait for another task, then this other task would in fact be its predecessor on the longest path, while the just finishing task in question would not. In this regime, therefore, the length of the longest path is the time taken to execute the task graph overall.

Moreover, the effect of messaging will be felt between the tasks that are on that path, while for tasks off the longest path it may not. Further, in general for short messaging delays, the effect of those messaging delays is proportional to the number of tasks on the longest path, since each junction between tasks on the longest path will add the time for the one *taskdone* and the one *gettask* message, with their also being one *gettask* at before the first task and one *taskdone* after the last. Figure 12 illustrates this. This shows the times taken to process the same exemplary task graph for different, increasing cases of message latency, Cases A, B and C. (The exemplary task graph is not for the QR factorisation and is specially constructed to illustrate the points made here.)

Cases A and B are shown in the Figure twice, once at marks A and B and again at marks A’ and B’. At A and B, only the tasks on the longest path are shown, but at A’ and B’ the tasks off the longest path are shown as well. At A and B there are dotted lines marking off the messaging overhead sections of the tasks, while at A’ and B’ (and also at mark C for Case C) the dotted lines are omitted for clarity. The arrows represent dependencies between the tasks. Tasks on the longest path through the task graph are marked in red.

So, Case A is for a first, shortest, length of those messages, while Case B is for a longer length of the messages. Overall, the longest path has, between those two Cases, increased in length by 4 times the length of the message delay increase, since there are 4 tasks on the longest path and each task has a *gettask* message at its beginning and a *task done* message at its end.

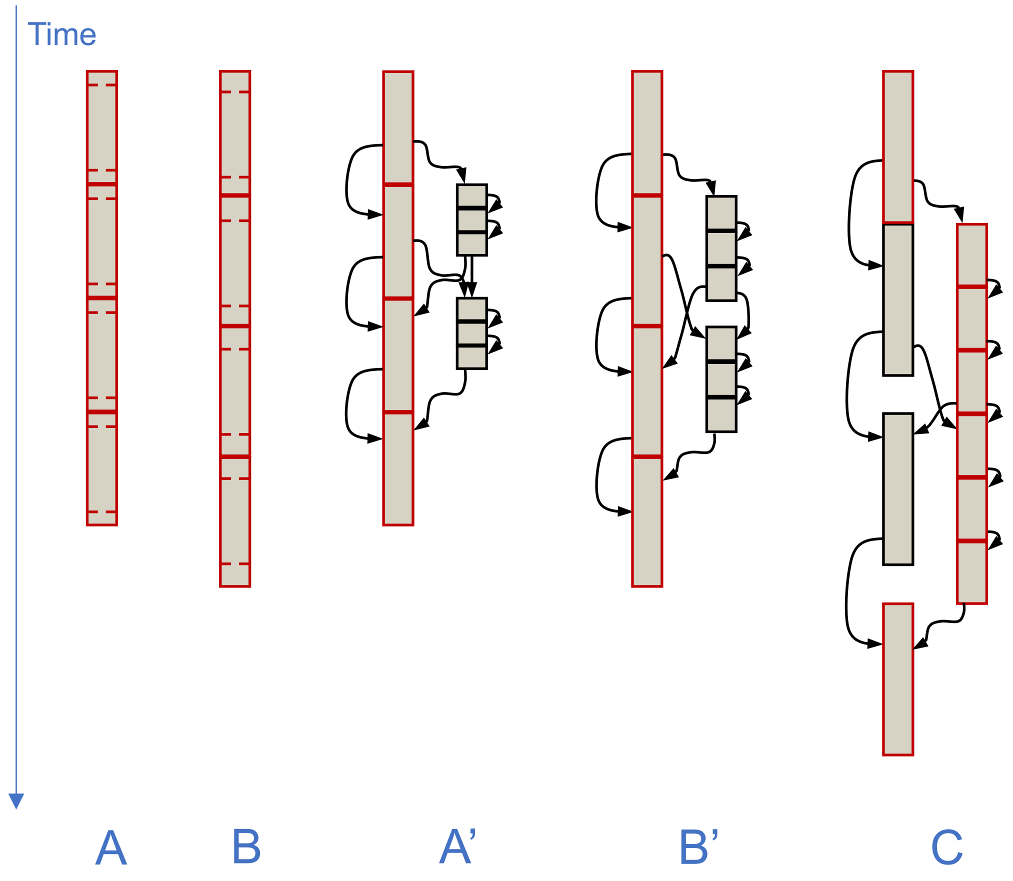


Figure 12 – Messaging time effect on longest path

Consider now the full task graphs for Cases A and B, so at marks A' and B'. Away from the first and last tasks, each task on the longest path has, in this exemplary task graph, in parallel with it, a section of 3 serial short tasks (i.e., the set of 3 and its parallel longest path task are both (i) dependent on the previous longest path task, and (ii) also dependent where it exists, on the previous set of 3).

For both these Cases, A and B, the total length of the set of 3 serial tasks is comfortably shorter than the parallel task on the longest path. So, the increase in message latency between the two cases has caused no change to the longest path. It remains that the additional length of the longest path due to messaging is proportional to the messaging time and the number of tasks on the longest path.

Case C of Figure 12 shows, however, what happens when the messaging time is increased even further for this task graph example. Because there are more tasks in each series set than in the single task in parallel with it, these have grown faster and have now taken over as the longest path. Tasks on the original longest path of Cases A and B, for shorter messaging times, have been forced apart, i.e., there is a gap in time between them.

So, there are some conclusions from this, i.e., for the case of a large number of cores as is being considered here:

1. The increase to the longest path length caused by message latency is proportional to both the number of tasks on the longest path and the latency itself.
2. The route of the longest path through a task graph is, in general, not independent of the messaging latency; it can switch to another path when tasks not on the longest path for a case of shorter messaging force apart the tasks that were on the longest path.

Thinking about larger task graphs, one can see that this switching effect to a different longest path, as messaging delay increases, could happen more than once and could happen for sections of the (current) longest path where either or both of the endpoints of the changing section are not the first and last tasks of the task graph (as was indeed the case in Figure 12). The former would make the addition to the task graph execution time piecewise linear as a function of increasing messaging latency, with the discontinuities of gradient occurring when the longest path switching occurs, so the additional task graph execution time ΔT due to the messaging latency L for each task (so modelled as a constant for all tasks) is given by:

$$\Delta T(L) = \sum_{i=1}^{k} (L_i - L_{i-1})N_{i-1} + (L - L_{i-1})N_k$$

Equation 2 – Addition to task path execution time caused by messaging latency in the case of longest path switching

where k is the number of longest task path switches that there are between no messaging latency and the latency in question L, $L_0 = 0$; L_i is that latency at switch k; and N_i is the number of tasks on the longest task path after i switches of the longest path, with N_0 being the number of tasks on the longest path at zero message latency.

Given that in Figure 12 the switching of the longest path was caused by the alternative path being greater in number of tasks than the related part of the original longest path, one might guess that $N_i > N_{i-1}$ at each switching point, so the gradient always gets steeper with increasing latency L. However, this task graph is probably unusual in that it has short, repeated sections of only a single long task on the longest path and each only has a single parallel path, and so on, so many more cases would in fact have to be examined to discover that that is a general rule.

Further, another reason for not getting too involved here with this analysis, is that for the QR factorisation the number of tasks on the longest path was, initially, determined empirically and had an outcome relevant to this. This determination was for the longest path as estimated by the task scheduler in the *prepare* stage (so the method was in the original scheduler). This determination was done here however with a separate Python program using the same method to it, using the same estimates for the individual task kernel execution times, while trying a range of message latencies. (archive: QSSim/src/quicksched/quickschedsim.py)

So, for the QR factorisation, the number of tasks, N_L, on the longest path was found to be given, always, exactly by:

$$N_L = (3n - 2) \ \ (= \ N_0)$$

Equation 3 – number of tasks on longest path for QR factorisation

where, as usual, n is the matrix tile count parameter (the matrix having $n \times n$ tiles). Moreover, this held true up to the largest n tried, of 512, and from very short messaging times (less than one thousandth of the estimated task duration) up to very large messaging times comparable in length to the estimated task execution times. (Note that in actual experiments as opposed to simulation the message time cannot be varied continuously, so simulation would provide more information, as well as being a quicker route to it.) Also, the total of the estimated time in the longest path task kernels was a constant for each n, meaning that if there was any switching of the longest path it was between task sections of the same length[5]. Therefore, it is thought from this that for the QR factorisation task graph the longest path, and certainly its length, probably does not switch from the ideal as a result of increased messaging latency and that in any event, in this simple model, the expected effect of the messaging delay is proportional to the individual messaging delay and to N_L, as given in Equation 3. Note that this concerns the longest path in the *task graph per se*, which, in the case of a large number of cores, determines the overall task graph execution time.

It was also subsequently realised that *3n-2* is the number of vertices on any path through a cubic grid of $n \times n \times n$ vertices from the top northwest corner vertex to the bottom southeast corner vertex if only unit moves east, south and down are allowed (so, equally the Manhattan distance plus 1, since on a path the number of vertices is one more than the number of edges). From that and recalling that the QR task graph can be laid out on such a 3D grid with many of the

[5] which is not a switch in longest path but a split of it into parallel longest paths of the same length.

dependencies being those unit moves (see Figure 8), it may well be that there is no freedom for the longest path for the QR factorisation graph to have any different length in terms of task count.

Then further, inspecting the QR task graph for the *n = 4* case in Figure 9, it was noticed that this has been automatically laid out, by the *graphviz* program, in 10, i.e., *3n - 2*, rows of tasks down the page with all the dependencies being in the direction down the page. Here, most of the dependencies move forward down the page just one row, none move forward up the page, and, further, there are paths from top to bottom that pass through 10 vertices, one per row. It cannot be therefore that, for this case of *n = 4* at least, a path has more than 10 (= *3n - 2*) tasks. Although it has not been confirmed, it seems unlikely that the possible paths from top to bottom that have fewer tasks will include the longest path. If so, the longest path will always have 10 tasks in this case, agreeing with the empirical result noted as Equation 3.

So this analysis has shown that the tasks on the longest path are the ones where message latency takes effect, and that the effect is generally proportional to the number of tasks on that path, at least for small message delay increases, (rather than being determined, perhaps, by some more alarming number of tasks that one could cast around for, such as the total number of tasks in the graph, given by Equation 1). Further, in the present case of the QR factorisation example, piecewise linear increases in the effect following are not expected.

Note also that conclusion 2 listed in this section would imply that strictly one should, in general, take into account the messaging time before determining the longest path in the scheduler. Such an adjustment is of course not onerous as it means only adding a constant to the already constant kernel execution time estimates, but while in fact this was not done for the estimates used in the original and experimental schedulers, in view of the matters discussed in this section, that does not seem likely to have caused the scheduler to have found the wrong longest path.

12.3. Run time performance, for fewer cores

At the other extreme from a very large number of cores, there is the case of just a single core to execute all the tasks. In this case, the off longest path tasks in the task graph cannot, of course, be processed in parallel with the on longest path tasks but obviously will have to be processed at some point by the single core to allow child tasks, including those on the longest path, to become ready tasks. The result therefore is that these off longest path tasks will force gaps between the execution of the on longest path tasks, thus making the overall execution time for the task graph longer than the length of the longest path. That is perhaps an overly elaborate explanation for this case of the tasks all just being processed serially on the single core – however, the "forcing gaps" language helps with the intermediate regime also discussed in this section, as well of course with the high messaging latency case for many cores that was discussed in the previous section. So, in this other extreme, serial one core case, the number of *taskdone* and *gettask* message effects to count towards the slowdown is therefore actually now equal to the alarming number of the total of the tasks, so as given by Equation 1, $N_G = (n+1)\ (2n+1)\ /\ 6$, rather than the number on the longest path of the task graph *per se*, as given by Equation 3, $3n\text{-}2$, as was the case in the previous section for when there were a large number of cores available.

Returning for a moment to the limit of a large number of cores, after a certain number of cores, having extra cores will be superfluous, because, for a finite task graph, both the number of cores occupied by tasks and the maximum number of tasks that can become ready tasks on completion of a task, and thus requiring further cores, are finite at any point in time. Below that number, there may, however, still be a regime down to some minimum number of cores in which the tasks on the longest path are still processed without pause, i.e., without the off longest path tasks forcing gaps between the on longest path tasks.

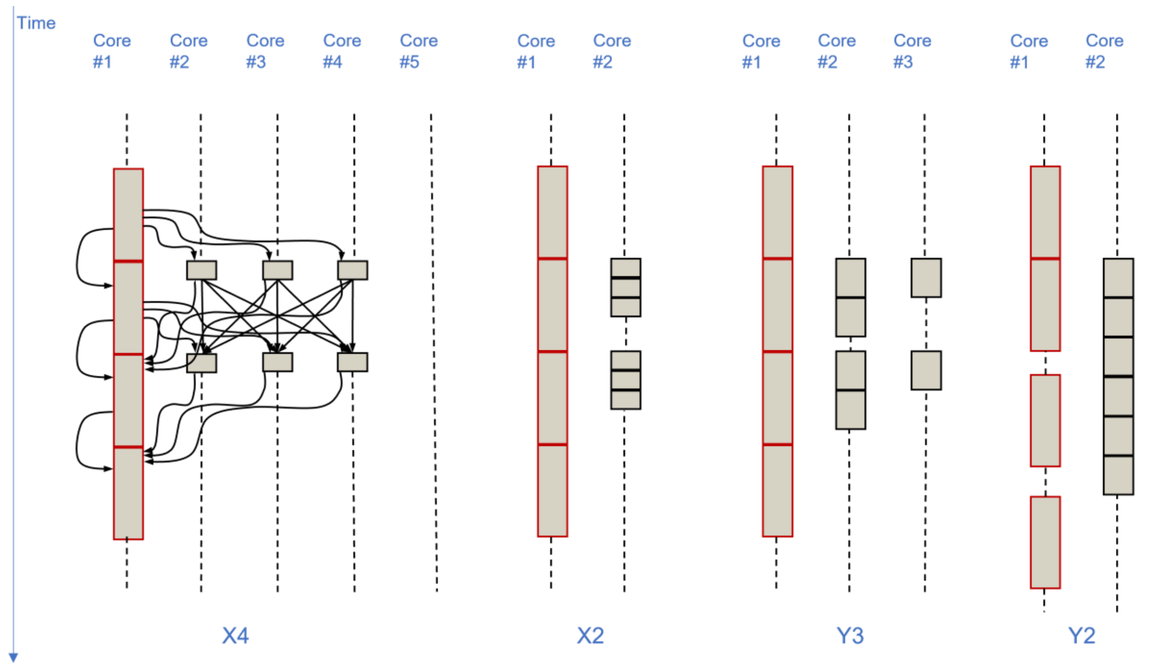


Figure 13 – Task graphs squeezed by not enough cores

Figure 13 illustrates this. Another exemplary task graph is shown at mark X4 with arrows for its dependencies. This task graph has tasks on the longest path, again in red, that are each of 5 units in duration and, and away from the end tasks, those tasks each have 3 tasks of 1 unit duration constrained by dependencies to run in parallel with that longest path task and are not constrained with respect to each other so are able to run in parallel with each other. i.e., these 3 off longest path tasks all just have a dependency on the previous on longest path task, and the next longest path task is dependent on all of them.

The task graph is laid out at mark X4 to show how it would execute in time on a large number of cores; at each stage both the longest path task and its 3 parallel tasks become ready tasks at the same time and are all immediately run on respective cores. It can be seen that a fifth core, or any further additional core, is superfluous. The number of messaging effects to take into account here for the overall task graph execution time is now again equal to the number of tasks on the longest path, as discussed in section 12.2.

The execution with just 2 cores is shown at mark X2 for the same task graph. The dependency arrows are omitted for clarity. Here, for each of the on longest path tasks, one core will process the on longest path task and the other core will process its respective off longest path tasks, in some order or other, and the latter will finish first. So, when an on longest path task completes, the next on longest path will execute immediately on one core, since it has priority[6], and one of its respective 3 off longest path tasks will execute on the other core, while the other 2 will queue. This second core will then execute those 2, in some order, but will then again idle after because the queue is empty, again until the current on longest path task finishes and the next on longest path task and its 3 parallel off longest path tasks become ready tasks. Thus, for this task graph it will be processed overall in exactly the longest path time from using a large number of cores right down to only 2 cores. 2 cores is the lower limit for such processing in the time of the longest path since, of course, for a single core all the tasks must process serially, in which, as discussed, off longest path tasks force gaps between at least some of those on the longest path.

A slightly different example is given at marks Y3 and Y2. This is as the last example but where the off longest path tasks are now of 2 units in duration, rather than 1. As it has the same dependencies as for example X, at each stage the longest path task and its parallel task become ready tasks at the same time. The longest path, is still favoured by the scheduling algorithm, i.e., each 5-unit duration task is immediately scheduled on becoming a ready task. In the case of 3 cores, shown at mark Y3, the other two cores immediately process one each of the 3 off longest path tasks and then, next, one of them then processes the remaining one. The 3 off longest path tasks are all completed before the 5-unit task on longest path task completes. So, again, the task graph is executed overall in the length of the longest path. However, 3 cores is the minimum number for that to happen for this task graph.

With just 2 cores, as shown at mark Y2, at each stage when the on longest path task and its parallel tasks become ready, the on longest path task is favoured and so executes immediately on one core. The other core executes the 3 tasks in parallel with it, but of course this is the only core left, so they will execute serially

[6] in QuickSched and presumably in the rules of any other sensible scheduler.

on it and will take a total of 6 units of time. The result is that the next 5-unit duration task on the longest path can therefore only start 1 unit after the end of the current 5-unit task.

So, where there are only 2 cores this task graph will take longer than the length of the longest path to execute. The longest path tasks have been forced apart, having a gap of 1 unit between them. Thus, for this task graph the minimum number of cores at which the task graph executes in exactly the longest path time, and at which adding more cores makes no difference, has been found.

For task graph Y for 3 cores the number of messages to take into account for the effect on the overall execution time is again the number of tasks on the longest path. However, for 2 cores this is no longer the case. To determine the number of messages affecting the overall execution time, one now has to trace a path down the tasks contiguous in time, i.e., avoiding any gaps on a core’s timeline, but where one may move between cores from one task immediately to the next. In the case of Y2 this starts with the red task on core #1, then passes through all 6 black, off longest path tasks on core #2 and then back to core #1 for the final task.

Determining the number of messages on this contiguous critical path would, in general, require a simulation, or other modelling, of the scheduling. However for this small and repeatingly structured task graph it is interesting to note the similarity in layout between Figure 13 at mark Y2 and Figure 12 at mark C. Remember that Figure 12 is concerned with the task graph *per se*, unconstrained by the number of cores. However, one could draw in in Figure 12 two dashed vertical lines for two cores through the two columns of tasks, which have been conveniently arranged for the purpose, to complete the analogy. In Figure 12, what constrains each set of three tasks to run serially within itself is the serial dependencies of the tasks between them, while in Figure 13 there are, however, no such dependencies (they may run in parallel with each other) and what constrains the tasks to run serially is that there is only one core to run them on. So, the lack of cores has in a sense created dependencies between the parallel tasks – they must run serially (but their order is not constrained).

(Note that the colouring of the sets of 3 tasks differs between Figure 12 and Figure 13. In Figure 12 they are on the longest path of the task graph *per se* but in Figure 13 they are not, and it would not help in the scheduler to somehow designate them as such; if given priority over the unit task, two of them would occupy the two threads, with the 5-unit task following, resulting in longer overall operation.)

This perhaps suggests an alternative to step-by-step event simulation to obtain a prediction for the overall execution time of a task graph. Small sets of parallel tasks could, for the calculation be amalgamated into larger tasks (which in effect is adding serial dependencies between them). But then again, for example, 3 parallel tasks would have 3 alternative forms by width, in terms of number of cores, and corresponding duration. In the cases previously mentioned in this section the number of cores available is known; it is set by the total available, less 1 for the longest path. But for larger task graphs the number of cores available would be set by parallel blocks of tasks on other cores, which would have their own set of possibilities for their shape. Further the process would be hierarchical, amalgamating these in turn into larger blocks, while fitting them into the number of cores available and taking into account the different possibilities for the shapes of each block. So, at first sight it appears that the complexity of this could well be greater than the step-by-step event simulation.

Another interesting case is for forms of task graph that are not particularly constrained by dependencies, to the extent that the available cores always have ready tasks that can be run on them, so the cores are always occupied and never idle. A large number of embarrassingly parallel tasks of the same size, unconstrained by dependencies between themselves, and having a number that is a multiple of the number cores would clearly be such a case. Here, the time taken to run the task graph is equal to the total number of tasks times the execution time for one task, divided by the number of cores. If this behaviour remains true as the number of cores increases, then this is perfect strong scaling. In this case, something strange has happened to the longest path through the task graph *per se*: it is no longer single and has as many alternative branches as there are tasks, with each longest path consisting solely of a respective task.

Fortunately, the number of messaging effects to take into account for the overall execution time is trivial to find: each core will process, without gaps, a number of tasks equal to the total number of tasks divided by the number of cores. That number is the number of tasks to take into account and it is the same for each core. (So, for small departures from the tasks all having the same size but where all cores are kept occupied, the number of messages to take into account would be the largest of the number of tasks processed by each core. However, that would be harder to predict with increasing latency since that would cause rearrangement of the tasks on the cores.)

12.4. Performance modelling conclusions

The models of task graph building and execution here are applied in the discussion of the results of the experiments with the QR factorisation example in Chapter 13, where the importance of the messaging time borne out. So messaging time is an important aspect of such a model. It was noted at section 8.1 that the QR factorisation has task kernels that will execute in a very stable amount of time; a more general model of application to other problems should include a model for the distribution of the execution times of each kind of kernel. These run time models may perhaps form the beginnings of predicting progress through the task graph as discussed in the final chapter.

13. Experimental Results and Discussion

13.1. Parameters for main experiments

The main experiments were runs of the task based QR factorisation code using my remote scheduler Qsargm based on my Argmessage RPC library and using the general arrangements set out in Chapter 11.

Two different parameter sets were used, defining two kinds of experiment, as discussed in section 11.6. In a first, the “**set C**” experiments (designated “C” and “C3” in the archive), both the number of threads and the matrix size parameter *n* were both varied according to the parameter specification:

```
[Opt('T', [1]), Opt('r', [1,50]), Opt('t', list(range(1,11))),
 Opt('x', list(range(1,9))),
 Opt('k', [32]), Opt('S', [99,0]), Opt('v', ['no'])][7],
```

and transform:

```
{('x', N): [('m', N), ('n', N)] for N in range(1,129)},
```

Therefore, this specifies the performance of 1 and 50 repeats of the experiments, for each of 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10 computational threads, with a tile size of 32 by 32 elements, for each of the original and experimental schedulers. Each of these combinations were generated for each of a dummy parameter *x*, having values of 1, 2, 3, 4, 5, 6, 7 and 8 which controlled the tile count parameter *n* so that the matrix had *n* by *m* tiles where *n = m.*

Further the argument set used for each run of set C was that (in the file *QR_args_set9.txt)*, had 4 *outer repeats* of that.

With the *S* parameter being the rightmost in the specification, that meant that each experiment having otherwise the same parameters was first done with the control scheduler and then was immediately repeated with the experimental scheduler. This attempts to provide the control and experiments with the same environment, particularly the same CPU clock speed (see section 8.1).

[7] The Python *range* function does not include its upper limit argument in the range produced, so these ranges are the same as those given in the table below.

So, in summary, this parameter set is a grid of experiments, of kernel thread counts against numbers of tiles, with fixed tile size and one server thread. This provides a general survey of the performance of the code.

A second, "**set F**" of the experiments explored the effect of size of the tiles used in the QR factorisation on the performance of the scheduler. This was generated using a parameter set file (QR_args_set21.txt), which was 4 long term repeats of the parameter set generated with the specification:

```
[Opt('T', [2,4]), Opt('r', [50]), Opt('t', [1,2,5,8,10]),
              Opt('y', list(range(3,8))),
         Opt('S', [99,0]), Opt('v', ['no'])]
```

with transforms applied of:

(i) `{('y', N): [('yy', N), ('k', 2 ** N)] for N in range(2,11)}`

and then

(ii)
```
{('yy', N): [('m', 2 ** (log2elements//2 - N)),
        ('n',  2 ** (log2elements//2 - N))]
             for N in range(2,11)}
```

where `log2elements` is a constant 20.

Therefore, this specifies the performance of 50 repeats of the experiments for each of 2 and 4 server threads, for each of 1, 2, 5, 8 and 10 computational threads, for each of the original and experimental schedulers. Each of these combinations were generated for each of a dummy parameter *y*, having values of 3, 4, 5, 6, 7 which controlled both the tile size parameter *k* and the tile count parameter *n*, in order to keep the total number of matrix elements constant. So, the tile size parameter *k* means that each matrix tile has *k* × *k* elements. The *(k, n)* pairs generated are below in Table 13, together with the total number of tasks and the number of tasks on the longest path through the task graph.

k	n	Total task count (n(n+1)(2n+1)/6)	Longest path task count (3n-2)	Tile size (kiB)
8	128	707264	382	0.5
16	64	89440	190	2
32	32	11440	94	8
64	16	1496	46	32
128	8	204	22	128

Table 13 – Set F (k,n) pairs and associated task counts

So, in all set F experiments, the matrix being factorised had 1024 × 1024 elements (hence the value of the constant *log2elements*). Thus, the problem size is a constant for all the experiments in Set F and all that differs is how the problem is divided up into tiles. The experiment was run on the *jupiter* and *thor* systems.

The resulting graphs for the various timers are discussed in the remainder of this Chapter.

13.2. Timer taskgraph_build_time for Set C

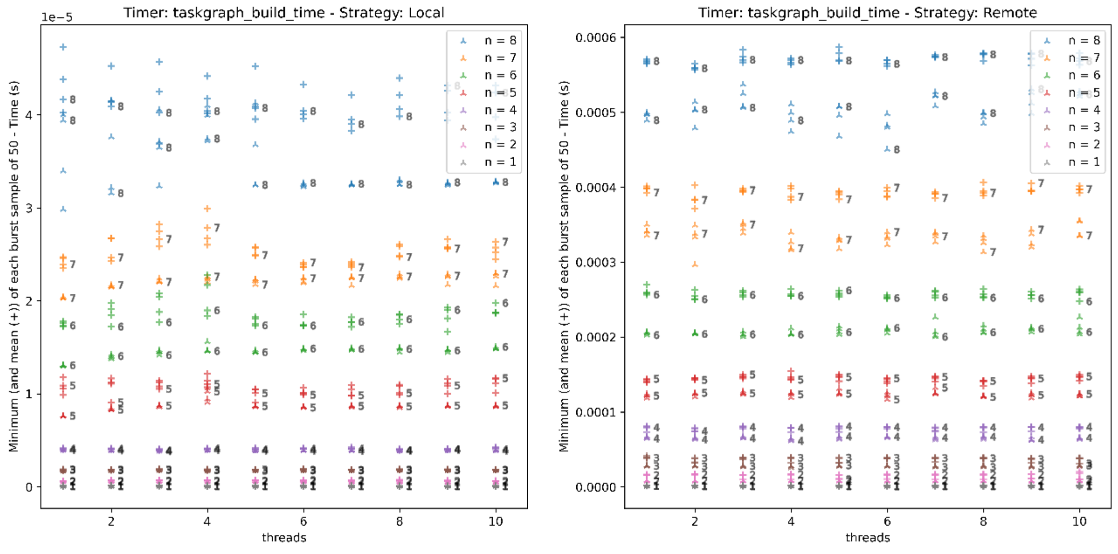
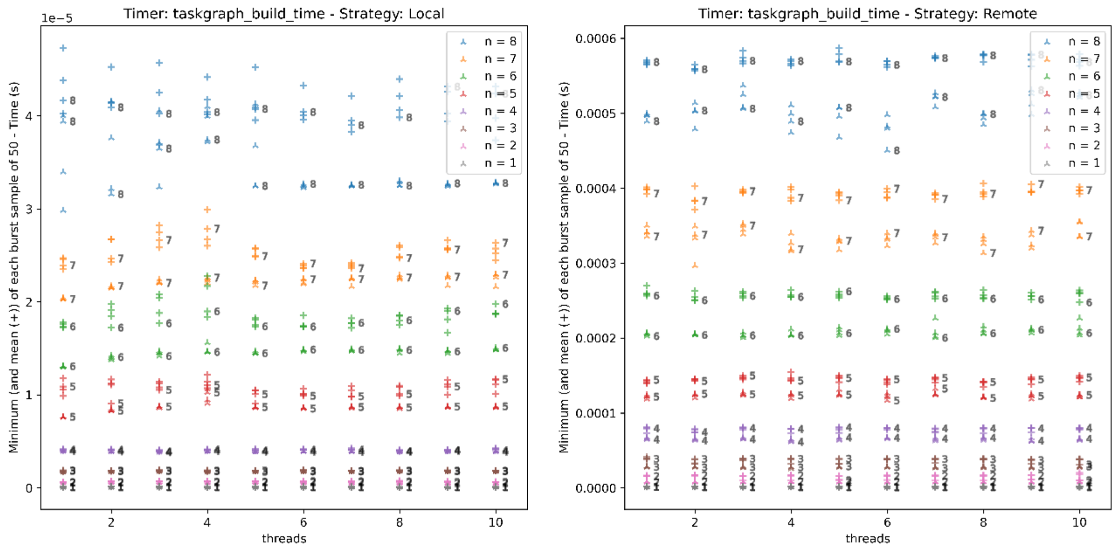


Figure 14 – task graph build time - local host geometry (thor005) – against threads

Figure 14



is a plot of the task graph build timer of the set C results against the number of threads allocated, with two makers for each set of 50 repeat sets, which are a + sign for the mean and a three-point star for its minimum. The tile count parameter *n* is indicated both by colour, as noted in the legend box, and a label equal to *n*. The control experiment of the original experiment is in the left panel and the right is for the Qsargm scheduler.

The reason for plotting the minima of the result sets is that in reviewing the performance of an algorithm the minimum is sometimes regarded as the best that an algorithm can do, with variation from that being dismissed as caused by extraneous events. On the other hand, in this work the code is multithreaded and many of the delay events are or could be caused by the code’s own operations, so the noise is intrinsic, and, generally a scheduler is for practical application, so mean performance figures are more appropriate for determining that.

It can be seen that both the original and remote schedulers, the task graph build time is generally constant, independent of the number of threads allocated, for any particular tile count *n*. This is to be expected because, in the QR example, the code, in the client, that calls functions to build the task graph is only single threaded. The parameter *threads* plotted along the abscissa means only that that many cores are allotted to the code by the launch scripts; the client code itself would have to take them up. (For other timers, discussed later, the client code does take up these threads.)

It is notable that, in general, the outer long-term repeats gave similar results to each other and so were reassuring that the experiment is generally stable and repeatable. It was also reassuring that the single repeats (not shown here) appeared to be consistent with the 50 repeats, indicating that there is no effect of repeating, or not, the experiment.

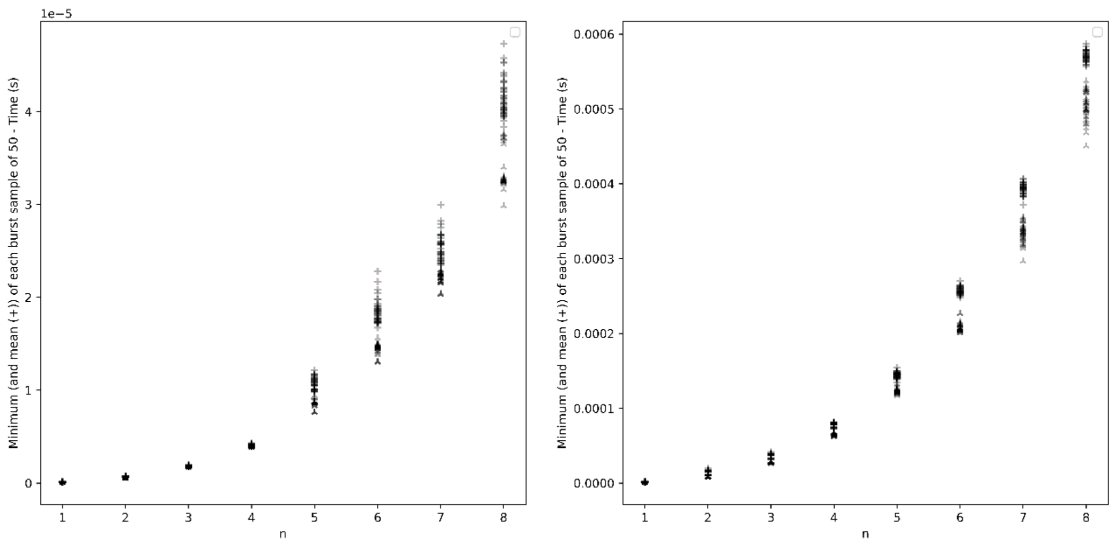


Figure 15 – task graph build time – local host geometry (thor005) – against tile count parameter n

Figure 15 is for the same data as Figure 14 but plotted against the tile count parameter *n*, (so including experiments for all thread counts). Again, there is a + sign for the mean of each set of 50 and a three-point star for the minimum. The general form for both the control and experimental scheduler is increasing, and, with the exception of the control (*row 2, column 3*) between *n = 6* and *n = 7*, increasing faster with *n*. The nominal number of client threads is not distinguished in the graph because the task graph build was single threaded. Again, this general form is to be expected, scaling with the number of tasks, as was explained in section 12.1, with reference to Equation 1.

The most significant feature of these plots is, however, that for the experimental Qsargm scheduler the task graph build time is much slower than the original scheduler, more than 10 times slower. (Note the different scales on the ordinates between the left and right panels.) That this is due to the much-increased messaging time is confirmed in detail in the next section, section 13.3. The trend with increasing message time associated with changing the client-server geometry is, however, apparent from Figure 16.

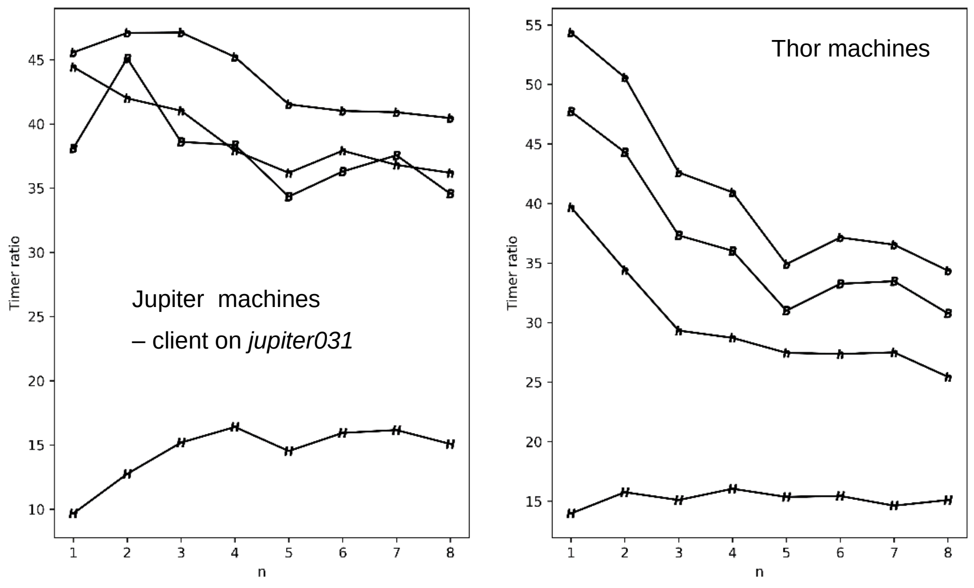


Figure 16 – Set C – Collected taskgraph_build_time – ratio to control

Figure 16 combines, respectively in the left and right panels, the results for the third and first machine sets of Table 10, so for the client on *jupiter031* and *thor005* respectively, for all the *geometries*. (Some of the runs for the *jupiter001* set failed to complete, i.e. exceeded the maximum 12 hours allowed for a job on the cluster, and were omitted from the results on that ground. The failed results were not inspected and so were not rejected on the basis of the values they contained, and those that did complete, although then are not shown, are consistent with the *jupiter031* set; they have the same machine specification.

The value plotted is now not the absolute value of the task graph build timer but the ratio of the minimum of that timer for the experimental Qsargm scheduler over the control QuickSched scheduler. The graphs are plotted against the number of tiles parameter *n* (i.e. there are *n* × *n* tiles). Different markers have been applied for the different *geometries* as follows in Table 14.

Geometry	Marker
Local host	H
Local BlueField	B
Remote host	h
Remote BlueField	b

Table 14 – Geometry marker letters

The *thor* system machine set, in the right panel, shows some clear separation by *geometry*, with the geometries occurring in the order of their message latencies as shown in Table 8. For the *jupiter* system, in the left panel, the separation is between the *local host geometry*, the lower graph, while the *remote host* and *local BlueField* graphs are similar to each other, with an upper graph of the remote BlueField being separated again, which is more or less how the OpenMPI latencies for this system vary – to fit the latencies exactly the remote host (‘h’) marker should be at the same level as the remote Bluefield (‘b') . These graphs therefore further confirm that the task build appears to be dominated by the messaging overhead. A more detailed confirmation is given in section 13.3.

The *thor* machine set, for the non-shared memory messaging *geometries*, shows a decrease in the ratio with the size of the task graph. (For the *jupiter* machine sets this is not so clear.) The shape suggests amortisation of a one-off cost in the experimental scheduler. Curiously, this is not there for the *local host geometry* (the bottom band), which has a roughly constant ratio except for the smallest values of *n* where the ratio is a little less, suggesting that the one-off cost is related to setting up the messaging across an actual messaging link, rather than in shared memory.

A final point is that the ratio is a little less for the BlueField cards for *n = 5* than for its neighbouring values. This might suggest perhaps that the pattern of messages for building that particular task graph may have a more efficient pattern of overlap of messages in these cases. Better overlap of messages would be a potential explanation of the shape of the for the non-shared memory messaging *geometries.*

This problem of a vastly increased time to build a task graph is clearly not tolerable in a practical system (for relative durations of the task graph build and run times see section 13.5) and how the problem can be resolved has been considered and set out in the next Chapter.

13.3. Task graph build time dominated by messaging

Considering now just the experimental Qsargm scheduler, for each task the messages sent from the client to the server are one for each *qsched_addtask()* and one for each of the additional calls listed in section 12.1. It might be expected that the additional to calls do not have such a significant latency, seen from the client, compared to *qsched_addtask()* itself, because, in contrast to the latter, no reply message from the remote task scheduler server is involved. (See in the server adapter code file for Quicksched (files qsargm_adapter0.h/c in 0) that function *qsargmadapter0_addtask()* has a return type of *struct argmessage_message**, whereas *qsargmadapter0_addlock()*, *qsargmadapter0_addunlock()*, and *qsargmadapter0_adduse()* all have *void* return types, and so the scheduler server does not return a message to the client for those.) This is borne out by the results as follows.

Therefore it could be guessed that the calls to *qsched_addtask()* would be the most significant element in the task graph build time, and this turns out to be correct. Taking a matrix of 8 × 8 tiles as an example, 204 calls are made to *qsched_addtask()*. In Qsargm, the buffer size for messages between the client and server was generally between 24 and 48 bytes. The Argmessage code used here used MPI_Send / MPI_Recv for the messaging. According to the measurements made with the OSU point to point message latency benchmark (Chapter 10), which uses those same MPI message functions, each message round trip for a *qsched_addtask()* should take 2 × 0.5 μs for that size of message. In building the task graph for an 8 × 8 tile matrix the OpenMPI messaging for *qsched_addtask()* alone (not counting the small time for processing of the call at the server) should take 204 calls × 2 × 0.5 μs (Table 8, Chapter 10), so about 200 μs in total. In the right panel of Figure 15, the minimum task graph build time for a matrix of that many tiles is around 500 μs (the three-pointed stars marked with an '8'), meaning that roughly nearly half the task graph build time is taken up by the OpenMPI messaging in *qsched_addtask()*. Given the number of calls to the other task graph building functions, i.e., the ones that do not need a reply, the conclusion is that those calls do not contribute quite so strongly to the total build time, probably overlapping with each other.

Table 15 tabulates rough figures for the minimum task graph build time for the 8 × 8 tile example against a total of an estimated round trip message time for the calls to the *qsched_addtask()* function alone, estimated from the benchmark round trip time, for the various geometries.

Experimental scheduler location (Client was on thor005)	**Minimum task graph build duration 8 × 8 tile example, read from subplot(1,4) (μs)**	**Estimate of time for *qsched_addtask()* calls: 204 × 2 × benchmark message latency for 32 byte message (μs)**	**Roughly estimated proportion of build time taken by *qsched_addtask()* messaging**
Local host	500	200	40%
Local BlueField	1200	600	50%
Remote host	1000	500	50%
Remote BlueField	1400	700	50%

Table 15 – Task graph build time estimates

Since that for each *qsched_addtask()* there were also several messages from the client to the server that do not have a reply, which also contribute, but each not as strongly, the conclusion to be drawn is that in all *geometries* the task build time is dominated by messaging, and within that the main contribution is from the round trip necessary for the *qsched_addtask()* calls.

13.4. Prepare Timer

The next stage in the task graph processing is the *prepare* stage; once the task graph is built the user code calls *qsched_run()* and that in turn calls *qsched_prepare()* as a preliminary step. *qsched_prepare()* begins with respective sorts of three tables of the task graph's representation (the tables *deps, locks, and uses*), which proceed in parallel on respective OpenMP threads. That is followed by several stages that are all processed serially on a single thread. Quoting from the comment lines in the code of the original QuickSched:

> *"link the locks and unlocks",*
> *"Init the queues",*
> *"set the waits",*

"sort the tasks topologically",

"set their weights, re-setting the waits while we're at it", and

"enqueue the non-waiting ones".

So, there is a portion, the sorts, that should benefit from the scheduler being allocated multiple threads, which can, of course, happen for both the original and experimental schedulers.

For the original scheduler, the timer is between the beginning and end of the *qsched_prepare()* function, inside the function. For the experimental scheduler the timer is inside the function *qsargmproxy0_run_openmp()* on the client sidearound the prepare message sent to the server and its reply. The *prepare* action does not need a reply but one was provided in the code so that this **did not** overlap with the subsequent task execution giving the timers for both of those a clear meaning. In practice removing the message would allow such an overlap, which may save a little time.

The next section, 13.5, concludes that the prepare stage is only as small part of the build-prepare-run process, and further there was also a bug in the code that meant that the table sorts for the experimental scheduler only operated on a single thread. Nonetheless there was some useful information to be had.

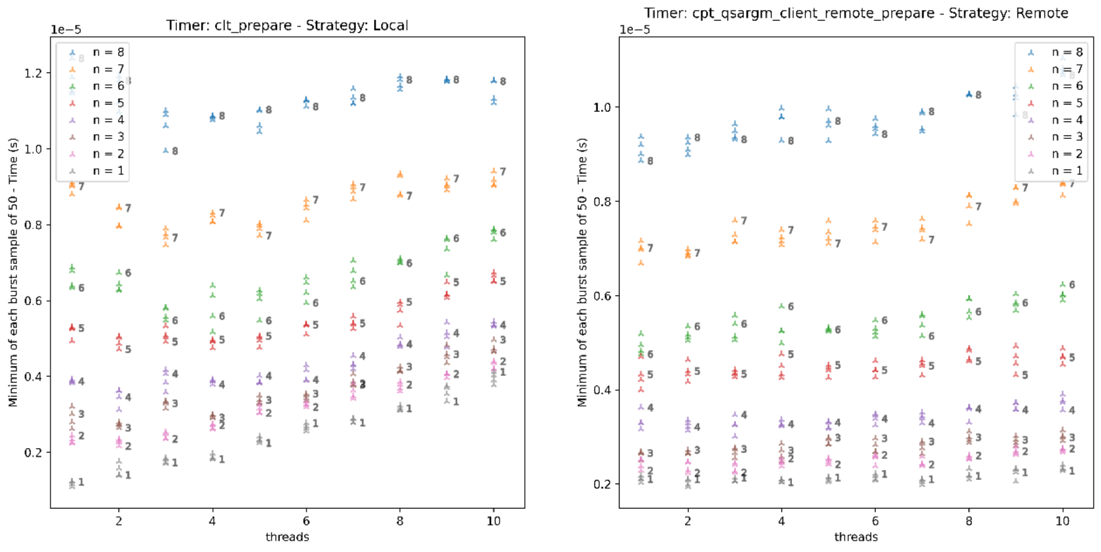


Figure 17 – prepare timer – thor system - local host geometry

Figure 17 shows the prepare timer for the *local host geometry* for the *thor* system from the Set C results. The times in this plot were scaled to seconds using the ticks per second values presented in section 11.7. Each marker is the minimum of a set of 50 repeats. Markers for different values of the tile count parameter *n* are distinguish by colour and by the value of *n* plotted next to the markers. The results for the original scheduler are in the left panel and those for the experimental. These show that for each tile count parameter *n* the prepare timer is roughly constant with a small increase with the number of threads. It was also noted (from a plot of the variances – not shown) that the experimental scheduler is less noisy for this timer than the original.

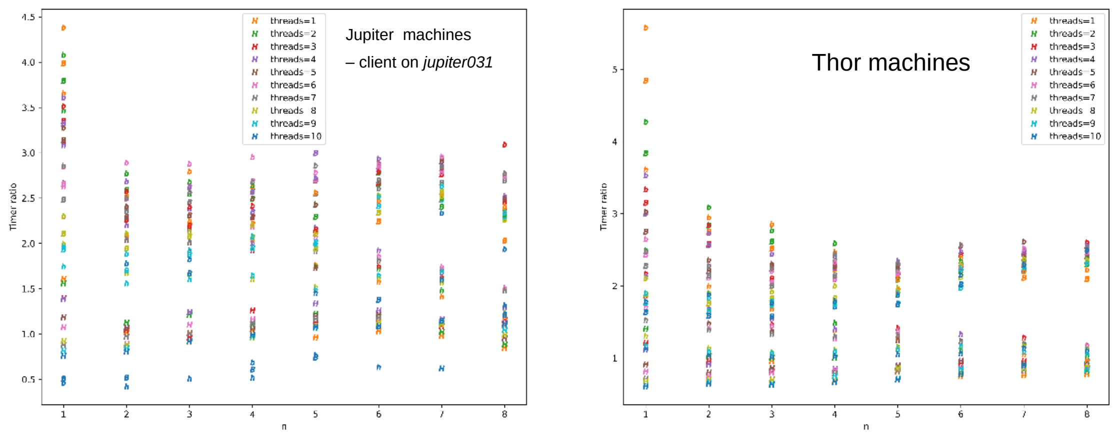


Figure 18 – Set C – Collected prepare timer – ratio to control

Figure 18 is the ratio of the experimental prepare timer to the control and gathers the results from the *jupiter* and *thor* systems, in the same manner as Figure 18 except there are markers per thread count in view of the weak dependency on thread count here. The markers are again as given in Table 14.

An interesting outcome here is that for the *local host geometry* ('H'), the remote scheduler is a little faster than the control for the *thor* system and for smaller problem sizes for the *jupiter* system.

The BlueField *geometries* are slower. For the thor system, at large *n*, there is a clear banding between the *Bluefield geometries* and the *host geometries*. This will be because the prepare stage is computationally expensive and the BlueField cores are less powerful than those of the hosts. (The single message pair only is a small addition to the larger problem sizes.) However, the insignificance of this slowdown is discussed in the next section.

13.5. Relative lengths of build, prepare and run stages

The typical durations of the build, prepare and run stages of the QR factorisation are set out in Table 16, taking as an example the values from the set C data, for 8 × 8 tiles, for the *thor* system. (The values were read off graphs by eye; the data for the run stage is discussed in the next section.)

Stage duration for 8 x 8 tiles (s)	**Control**	**Local host remote scheduler**	**Approximate thread count dependency**
Build stage	5×10^{-5}	1×10^{-3}	Independent of thread count
Prepare stage	2×10^{-5}	1.2×10^{-5}	Independent of thread count
Run stage	$6.5\times10^{-3} \rightarrow 1.5\times10^{-3}$	$7\times10^{-3} \rightarrow 2\times10^{-3}$	For 1 and 10 threads

Table 16 – Relative durations of build, prepare and run stages – thor system -set C

This shows that for the original, control scheduler the build and prepare stages are insignificantly small compared to the run stage, being around 100 times shorter. The performance failure of the build stage for the experimental scheduler makes the build stage of significance, being almost of the same order of magnitude as the run stage. This however means that the schemes for reducing the build time for the experimental scheduler set out in section 14.3 would be worthwhile. Moreover, those arrangements could achieve a build time, without messaging, similar to that of the control scheduler; to that would then have to be added some O(1) messaging time for the *en bloc* arrangements, which has a cost of O(10^{-6}s), so in total that would still be very much less than the O(10^{-3}s) for the run time, making the build time insignificant again.

Insignificant build and prepare times compared to the run time are the sensible regime in which to operate, because build and prepare are overhead to the actual calculation, but even if building the tasks was a much more complex operation than for the QR factorisation (for example as in the Swift code) and so were more of an overhead, the messaging here will be relatively less of a burden (because more complex means more calculation to make the same size of task graph). So, the revised approaches of the experimental scheduler discussed remain valid for such codes.

The results discussed in the next section, section 13.6, concerning the task graph run time, are therefore the most important to considering usefulness of the approach of this thesis.

13.6. Task Graph Run Timer for Set C

The time taken to run the task graph is the other key performance measure for the experimental scheduler. (As discussed in section 13.5, the time taken to build the task graph will be small by comparison for many application codes, if improvements mentioned there are made to the experimental scheduler, and so the task graph run time could be *the* key performance measure.)

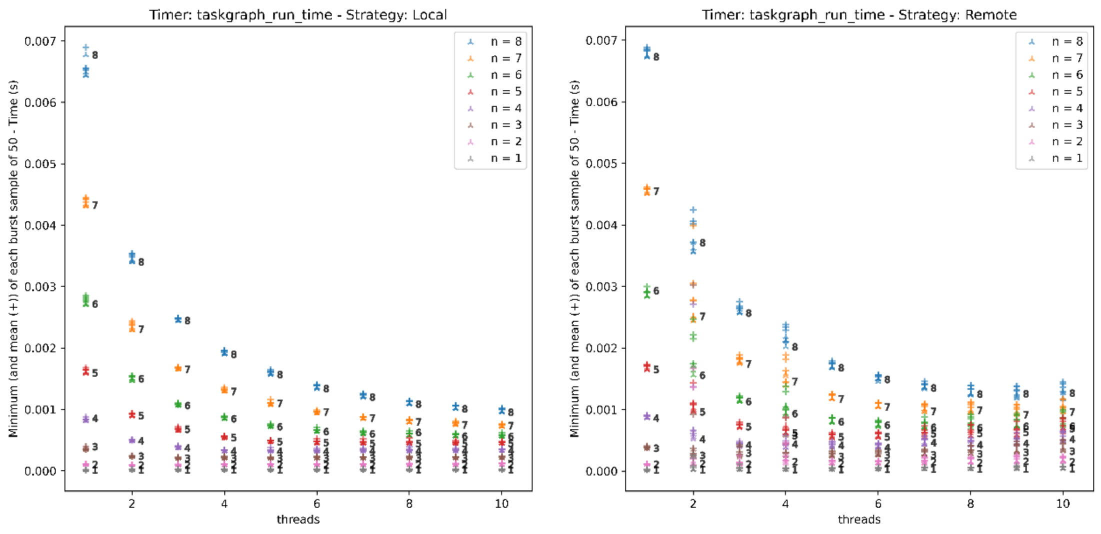


Figure 19 – Set C3 – task_graph_run_time timer – thor – local host scheduler

Figure 19 shows the run timer against number of threads, for a range of the number of tiles parameter, with the results for the original scheduler shown in the left-hand panel and those for the Qsargm scheduler in the right, for the *local host geometry*. The minimum of each set of 50 repeats has a three-point star marker and the mean has a plus sign. The tile parameter *n* for each marker is distinguished both by colour and by the value of *n* marked close by.

The code for the running of the tasks is, unlike the task graph build code, multithreaded, and the general form of the graph is as expected for that: as the number of threads executing tasks increases, for a fixed tile count parameter *n* and hence for a fixed initial task graph, the overall time taken to execute the task graph falls, on inspection perhaps approaching a limiting value. In detail, the original control scheduler perhaps reaches the limit for smaller values of n.

The left-hand panel for the other *geometries* (not all shown) showed very similar results, because, of course, the control experiment operates in exactly in the same way for all *geometries*.

There are, of course, general reasons to expect an approach to a limiting value. This is the situation considered by the well-known Amdahl's Law [77]. Amdahl observed that if a program has a serial part (always true) and a parallel part then, while the execution time of the parallel part might be reduced further and further by using more and more processors, the serial part will not be, and so the time to execute the serial part must be a theoretical lower limit to the execution time of the whole program. (One will not practically achieve that limit because breaking down the work into more and more parallel pieces will generate organisational overhead.)

Section 12.2 discussed the minimum time to execute a task graph being that of the longest path through the task graph. Thinking about that in the context of Amdahl's Law, it is the longest path from the root task to the last task that is that kind of minimum for the parallel processing of tasks in a task graph. (While the tasks along that path may be executed on different cores, they still have to occur in order, each starting after the last has finished, so in that sense serially.) So, Amdahl's law may well account for the minimum that approached by each size of task graph.

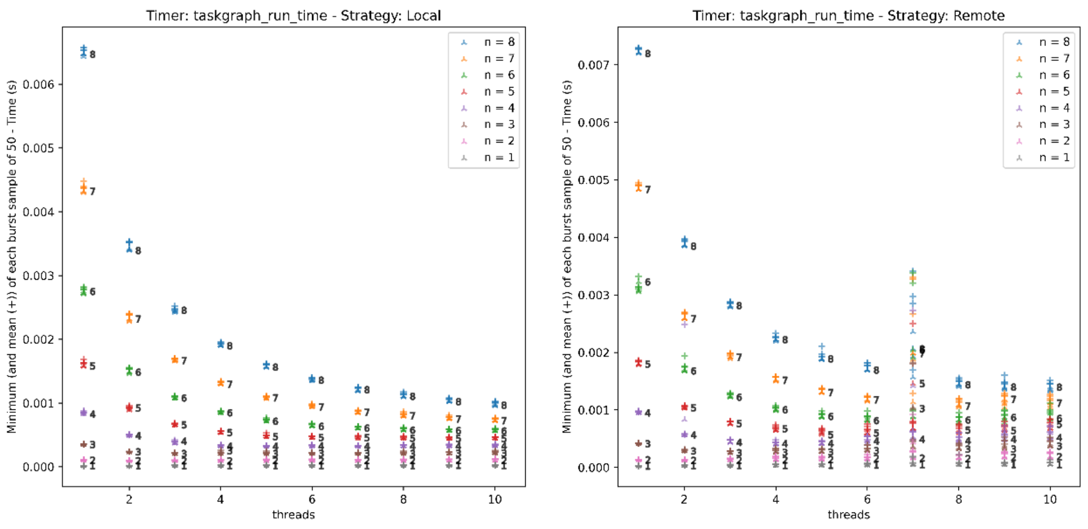


Figure 20 – Set C3 – task_graph_run_time timer – thor – local BlueField scheduler

Figure 20 is as Figure 19 but for the *local BlueField geometry*. The main point to note is that, for these and the other two *geometries* (not shown), the form of the plot for each fixed size task graph is the same form as for the original scheduler, but slower, in general just a little slower. This is quantified in section 13.8.

A detail of both Figure 20 and Figure 19 is, however, increasing variation in the run time at or approaching 10 threads for the experimental scheduler, which is not present for the control scheduler. At this point there will be more threads than tasks available so a lot of redundant messaging and many potential conflicts between requests for tasks. To gauge the number of tasks, consider the diagrams in Chapter 8 showing the task graphs for the QR factorisation; in particular, in Figure 9, which is for the case of $n = 4$; on inspection it appears from the dependencies that for each of the 4th and 5th rows, the longest rows, all the tasks of the row could fall to be being processed at roughly the same time, if there are enough threads, and these rows are 5 and 6 in number, so a little larger than n but the other rows are 4 and less in number. 10 cores is, of course, greater than the largest n, $n=8$, tested in these set C experiments.

However, later experiments go well beyond $n = 8$, so it was checked how the maximum number of tasks on the row of the QR factorisation task graph scales with n. This was checked with the expedient of having the *dot* program, which drew Figure 8 and Figure 9, output a list of coordinates on the page of where it will plot the tasks. From that a short script collected the coordinates and grouped them into rows, so that they could be counted. The results are in Table 17:

Tile count parameter *n*	**Tasks in longest row of graphviz plot**
4	6
8	21
16	80
32	297

Table 17 – Tiles in longest row of graphviz plot of QR factorisation task graph

This shows that the longest row of the task graph found, so a rough estimator of how many threads could operate in parallel on the task graph without any being unused, scales faster than *n* (so, that it was roughly equal to *n* for *n* = *4* was a coincidence). On this simple model it appears possible to keep a great many threads occupied.

An oddity in Figure 20, for the local BlueField *geometry* (and for the other *geometries* but not the *local host geometry* Figure 19) is that the case of 7 threads, for the experimental scheduler, there is a very large variation in the time taken to run the task graph. Contrastingly, for the *local host geometry* there is increased variance for both 2 and 4 threads; there is a hint of this visible in Figure 19, but it is clear in scatter plots and plots of the variance of the timer (not shown).

Such operation may be highly inefficient but of course in practical situations the usual objective would be to use a large number of threads, so this is not much of a concern.

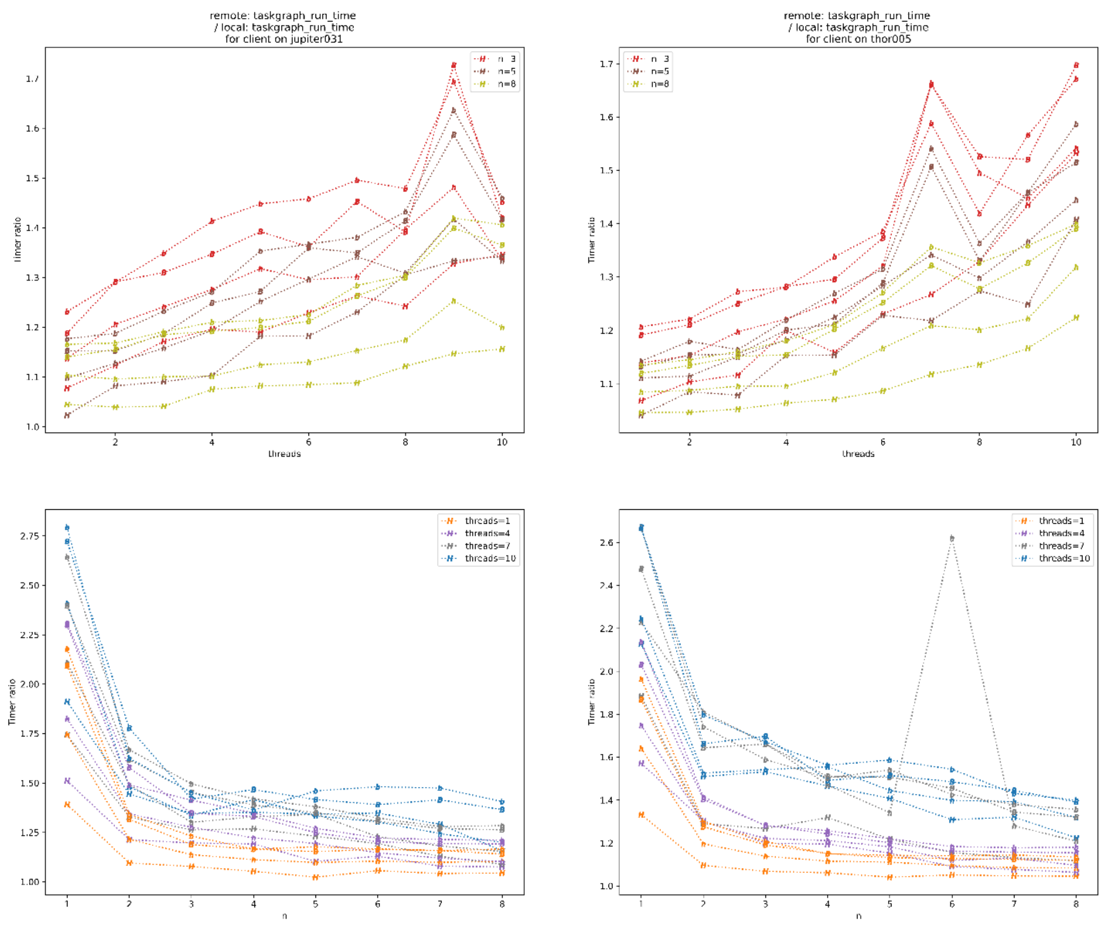


Figure 21 – Set C3 – Collected task graph run time – ratio to control

Figure 21 quantifies the slowdown compared to the original scheduler. In the same way as with Figure 16, this collects the *thor* machine set, on the right, along with the similar data for one of the two *jupiter* machine sets, on the left, and shows the ratio of the run timers for the experimental and control schedulers. Again, the markers are as listed in Table 14. The top row is the plot against threads, while the bottom row is against the matrix tile count parameter *n*. As can be seen in the legends only a subset of the *n* parameter values, or a subset of the thread counts have been plotted to provide visual clarity.

This Figure shows that the slowdown ranges from over 2.5 times in some cases down to only a few to 10 or 15 percent in others. The former is of course not practically useful at all, but the latter is very encouraging. The graphs show that the larger task graphs (see the top row against *n*) have better performance, i.e., less poor for the experimental scheduler. This region is more what would be wanted in practice, where task graphs can be much larger. (The task graph sizes used on the experiment were limited by consideration of repeating each experiment 200 times in order to gather significant numbers of samples in time spaced bursts.)

The upward trend in slowdown with number of threads is more clearly brought out in this Figure. This is a disadvantageous phenomenon in practice, since more modern processors have even larger numbers of cores than used here, but again closer inspection of the graphs suggests that the trend is less strong while the number of threads is suitable to the problem size, i.e., not more than the number of tasks available at one time.

It is also clear that the slowdown is strongly dependent on the messaging time, so the *local host geometry* (marker *H*), with the smallest latency, being better than the others. The future advance of technology will be favourable here also. BlueField cards are new and are undergoing active development to new generations (another generation will at the time of writing be available shortly) and latency between the card and the host is doubtless being worked on, so one would have an expectation that the next generation of BlueFields would perform better.

There is also of course the hope that optimisation of the arrangement of the messaging in the experimental scheduler could improve things further, especially for larger numbers of threads.

These results mean that the central proposal of this thesis, moving the scheduler from being within the compute threads to a remote separate process, was a success, or at least not a discouraging failure. However, it is clear that an appropriate region of operation needs to be used, and indeed the set F data, discussed in section 13.10, pursues that ***and produces greater success***.

The run time performance and its relationship to the messaging latency can be quantified further and that is done for the set C data in section 13.8.

13.7. Run timer results using sub optimal scheduler hints

An initial bug in the correction of the QR factorisation code provided some code which preformed the QR factorisation code correctly – it had the expected task graph – bit which did not have the intended *locks* and *uses* specifications, which are there to promote allocation of a task to a core that already has data for the task. Results were obtained for this code and the following were noted.

- the absolute task graph run times are very similar
- for the *thor* system, with the unintended *locks* and *uses* there was larger variation in run time for 9 and 10 threads
- for the *thor* system, with the unintended *locks* and *uses* the larger variation for $n = 7$ was not present

An explanation for the first point become apparent in section 13.9. The latter two points suggest that some combinations of task graph size (determined by n) and thread count can have an unstable layout of the tasks in time and across the cores between runs and that instability is sensitive to the hints given to the scheduler as to where to place a task.

13.8. Quantification of messaging overheads for run time

The performance models of Chapter 12 help analyse the effects of messaging delays on the execution of a task graph.

Starting with the case of a single core for executing the kernels, in this case the total task graph run time is the sum of the time to execute all of the tasks plus the overhead of allocating the next task after each of those and recalculating the queues. This provides a basis for numerical comparison between the original QuickSched and the experimental Qsargm remote scheduler runs, because the messaging adds to that overhead. Also, the case of a single core is quite clean in that there are messages only from that core, so clashes between messages from different cores do not occur.

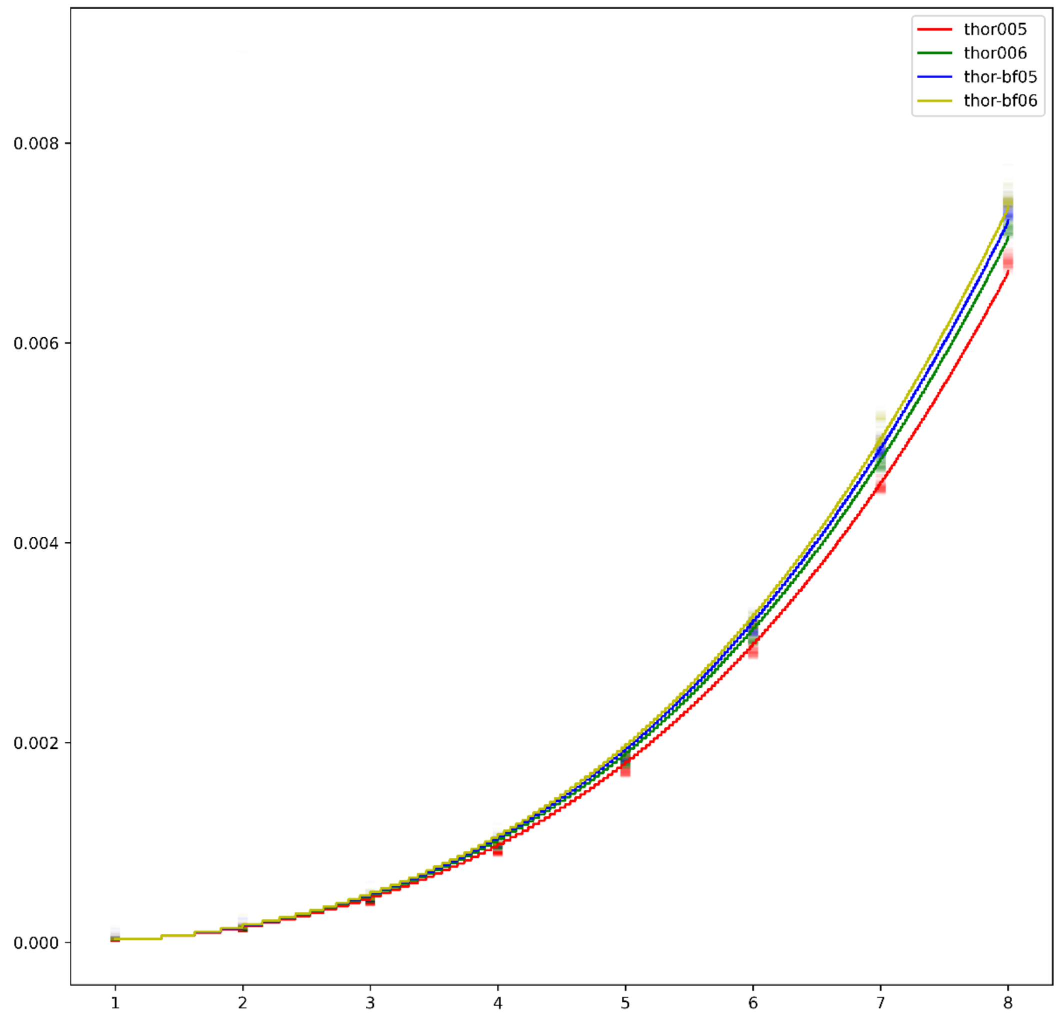


Figure 22 – Set C single thread run timer (s) against n – data for thor system fitted function of Equation 1

Figure 22 is a scatter plot of the run timer of the Set C data for just the single thread data for the *thor* system fitted to the function of Equation 1 for the number of tasks in the QR factorisation task graph, N_G.

The constant of proportionality needed to fit the results was noted for this and the other machine sets for both the original and experimental schedulers; this constant is the mean run time per task. The results for the original scheduler (not shown) were very closely the same for each *geometry* because, of course, the original scheduler does not communicate with the remote process, so the *geometry* is not relevant. For both the original and the experimental scheduler the fit in all cases is extremely good. That however is not miraculous given the cleanness of the one core operation. The fit was carried out using the *curve_fit* function from the *scipy.optimize* library [78].

The difference of the mean run time per task value given by the fit for the experimental and original schedulers is given in the final column of Table 18 for the different machine sets and *geometries*. That Table also includes the single message latency data from Chapter 10. For the *thor* system the machine sets were the same ones for the latencies and the fit result, but for the *jupiter* system the machines used were not the same (the systems being unavailable), but of course the machines within the *jupiter* system are similar, so the data can be used for comparison. In each case, it can be seen that this excess of the fitted value for the remote scheduler per task is something over twice the message latency. (The 32- and 64-byte times are similar and as noted previously are in the region of the message lengths used in Argmessage.)

This confirms therefore that the remote scheduler excess overhead is largely accounted for by the messaging between the client thread and the server, as follows. Recall that at the end of each task the thread first sends a *task done* message and then a *get task* message and then waits for the scheduler to process those and messages back the identity of the next task for it to execute. The *get task* messages out and back will cause a slowdown of two message latency times. The *task done* message does not cause an entire third one because it overlaps with the outgoing *task done* message. There may also be a contribution to the overhead caused by the more complex decoding of these messages than the simple function call equivalent in the original scheduler and, for the BlueField scheduler, this will not process the *task done* and *gettask* calls as quickly as the host processors since the Arm processor used is not as powerful as the hosts.

Single message latency or, final column, scheduler excess per task, in µs **Cluster and geometry**	**Minimum**	**32-byte messages**	**64-byte messages**	**Remote scheduler excess per task[8] over original scheduler**
Local host scheduler				
thor005 to thor005	0.38	0.50	0.51	1.11
jupiter029 to jupiter029	0.30	0.48	0.49	
jupiter031 to jupiter031				1.24
jupiter001 to jupiter001				1.27
Local BlueField scheduler				
thor005 to thor-bf05	1.39	1.48	1.56	3.80
jupiter029 to jupiter-bf29	1.35	1.45	1.54	
jupiter0031 to jupiter-bf31				4.25
jupiter001 to jupiter-bf01				4.97
Remote host scheduler				
thor005 to thor006	1.20	1.23	1.31	2.88
jupiter029 to jupiter030	1.66	1.73	1.81	
jupiter31 to jupiter32				3.90
Remote BlueField scheduler				
thor005 to thor-bf06	1.64	1.73	1.81	4.31
jupiter029 to jupiter-bf30	1.60	1.72	1.81	
Jupiter031 to jupiter-bf32				4.50

Table 18 – Excess run time per task for experimental remote scheduler over original scheduler (µs) for set C for single threaded task processing

[8] Output of the program: archive:aftermath/aftermath/analyse/H0_run_timer_strategy_differences.py

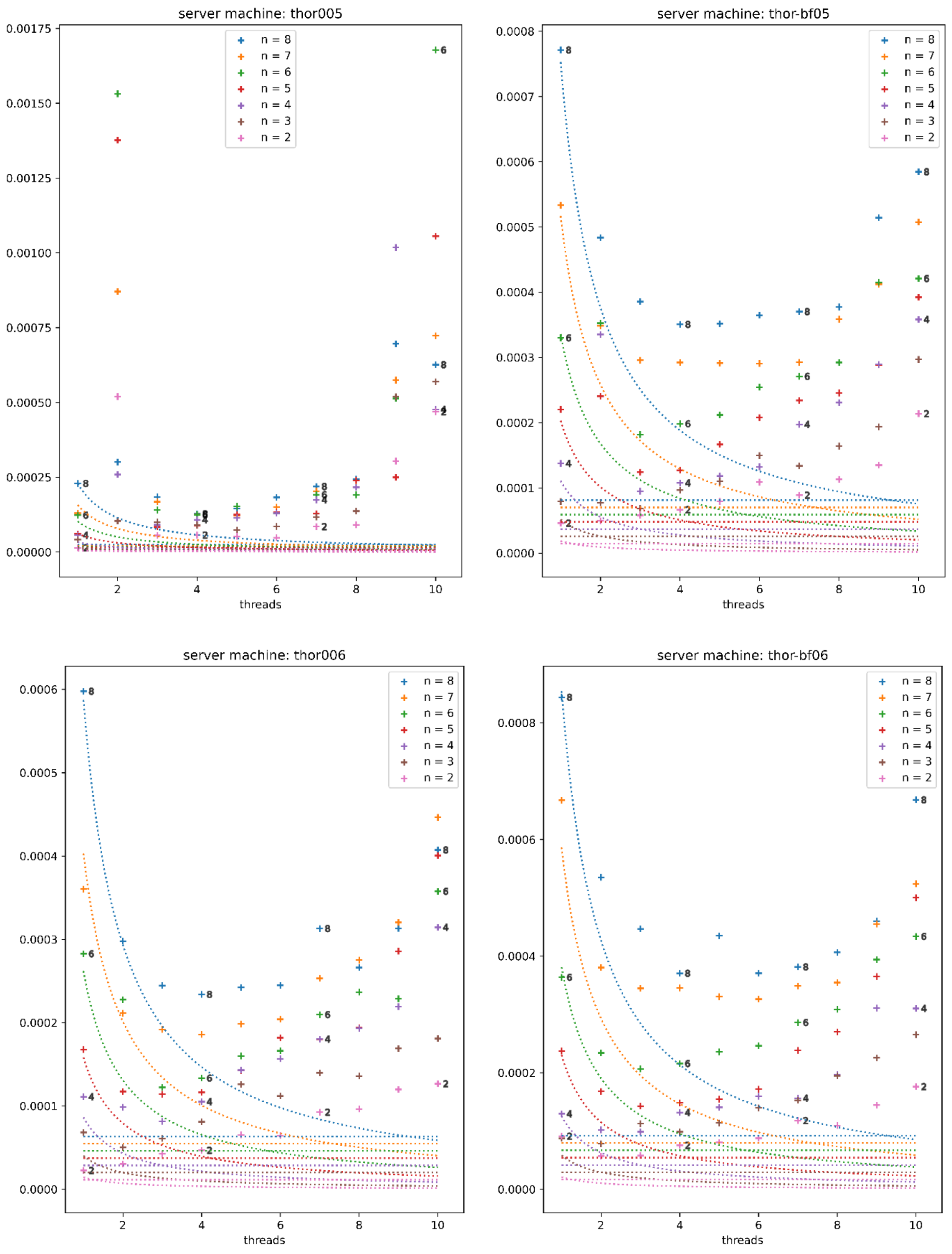


Figure 23 – Set C, thor – excess task graph run time – overlain with performance limits

These excess remote scheduler values per task were then applied to the data collected for the task graph run time for all the numbers of cores/threads employed, not just the single core case, as follows. Figure 23 and respectively show excess run time (in seconds) for the Set C data, for the *thor* and a *jupiter* machine set; i.e. the mean, marked with a ‘+’, for all the samples taken of that for the experimental scheduler minus that for the original scheduler.

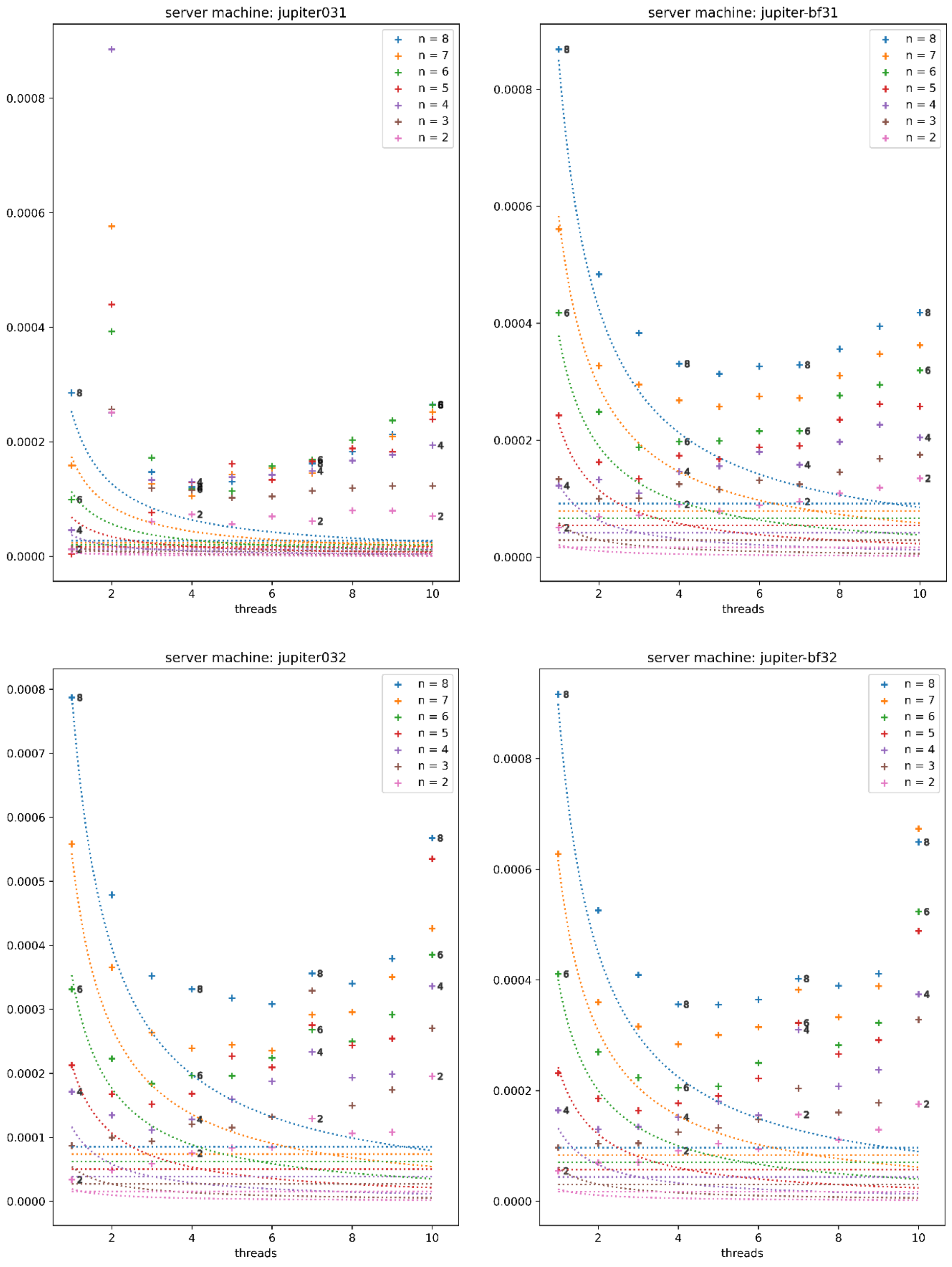


Figure 24 – Set C, jupiter – excess task graph run time – overlain with performance limits

Each plot is for a different one of the *geometries*. Each plot is overlain with model graph lines for two of the operational regimes discussed in sections 12.2 and 12.3. The curved model graph lines are for what would be expected for perfect strong scaling, i.e., for the case of a task graph whose tasks are so unconstrained by dependencies that the threads are always occupied, so the form is proportional to *1/t*, where *t* is the number of threads. This is plotted for each case of the matrix tile count parameter *n*. Each curve is scaled so that the point for one thread has the excess **per task** given in Table 18, that being the difference for the experimental and original schedulers' values found by the curve fits for the single thread case (Figure 22), **times** the number of tasks as given by Equation 1, since for one thread that thread executes all tasks. (Note these curves do not for a particular value of *n* meet the respective single thread point exactly, because the fit was for all values of *n*, 1 to 8, and not that particular value of *n*.) Of course, perfect strong scaling is not expected for the case for the present QR factorisation task graph, but this model is the best that could occur for any kind of task graph and so provides a lower bound to compare the result to. A model line for the actual, dependency constrained, QR optimisation for smaller numbers of threads is not plotted because to obtain such a model would require some simulation or complex estimation, as mentioned in section 12.3, which was not undertaken.

So, while the data points, for small numbers of threads and larger values of *n*, do track this overlaid, strong scaling, curve downwards, they of course do depart from it, i.e., not performing as well as the ideal strong scaling case. This is intrinsic in the task graph and so is not a bad thing. While in theory the downward trend should continue, that does not occur in practice; something is degrading performance at higher numbers of threads. This degradation in performance may not belong solely to the experimental scheduler. Earlier it was discussed how it does not seem likely that the pattern of execution of the task graph is not affected by the increased messaging latency, at least for the QR factorisation. So, the original scheduler may also be degrading owing to its own messaging and then the degradation of the experimental scheduler is a scaled up version of that.

The horizontal model lines were plotted as the remote scheduler excess per task value, which of course was for the single thread case, times the number of tasks on the longest path N_L as given by Equation 3, for the various values of *n*. This represents the best the scheduler could perform for the QR factorisation task graph at a high number of threads, so where the task graph runs in exactly the longest path time, if the excess overhead for the experimental scheduler per task for large numbers of threads stays the same as it is for the single core case, which is plainly not being achieved.

Note that the performance cannot be better than the strong scaling case, so only the section of each horizontal line to the right of its intersection with its respective strong scaling curve is valid as a minimum conceivable time for overall execution of the task graph. That does not mean the intersection point is one that could be approached for the QR factorisation, since its task graph cannot occupy all threads throughout the execution, i.e., cannot achieve strong scaling. So, the point at which the longest path limit could be achieved would be at some point to the right of the intersection.

Earlier it was observed that a cursory inspection of the QR task graph for *n = 4* suggested that it has a maximum number of parallel tasks available of around 5 or 6, which would be some indicator of the maximum number of threads that are useful to process it. Therefore, if the experimental scheduler where working well at large numbers of threads, it would approach the horizontal model line at some number of threads close that. Table 17 for the number of tasks in the longest row of the *graphviz* taskgraph plot has been recalculated for the values of *n = 2 to 8* to give Table 19.

Tile count parameter *n*	**Tasks in longest row of graphviz plot**
2	2
3	3
4	6
5	9
6	11
7	17
8	21

Table 19 – Tiles in longest row of graphviz plot of QR factorisation task graph for smaller values of n

However, it is clear from the experimental points that the minimum of the longest path in the task graph, the horizontal line, is nowhere near being achieved.

For an empirical observation from the experimental points in Figure 23 and Figure 24, noting the point where the excess run time actually starts to increase with extra threads, this point occurs roughly, as a trend for this range of *n*, at where *t* is about half *n*. Up to that point the remote scheduler appears reasonably efficient, and so its performance is probably determined simply by the message latency. This could be the point at which the task graph is supplying the number of threads it can and any more threads above this point are superfluous and causing inefficiency in the scheduler with unnecessary requests. Again, this state of operation might be confirmed with a simulation of progress through the task graph plotting its performance on those Figures. (If a simulation did not take into account the effects of excess requests to the scheduler the performance would become constant above that number of threads.)

If this number of threads of *t* being about half *n*, is the maximum number of threads that can be fed by the task graph then it is less than number in the longest row. This might simply reflect that for a lot of the time the task graph has fewer tasks in its rows than in the longest row, or there may be some interaction in the task graph that means only part of a whole row can become ready tasks at one time. There is tension between this and the idea that an infinite number of threads should ensure, with appropriate priority among ready tasks, that processing of the task graph in the length of the longest path. It would be interesting therefore to plot progress of ready and executing tasks on taskgraph plots produced by graphviz, such as Figure 9, to see if there are any more complex patterns than a simply advancing front down the page.

For the rising section of the data points, it could be that the scheduler is simply becoming swamped by the number of messages that it is receiving. As the number of threads increases there are increases in both the number of messages, leading to the increased chance of messages interacting, and in complexity for the messaging library to sort messages onto the correct thread. This interaction between messages is exacerbated by there being only one channel for the messages to travel along, so messages from the different threads will have to be serialised along it – if the channel is occupied with a message from a first thread, a message from another wanting to use it will suffer extra delay. There may also be an inefficiency in the scheduler of repeated failed attempts to find a task for threads that cannot be filled.

So, how can performance at higher numbers of threads be improved? Reducing the message latency *per se* would of course reduce the chances of interaction between messages and so improve performance at high thread numbers. Doubtless the technology is moving in that direction. Another approach is to reduce the number of messages by reducing the number of tasks, i.e., larger tasks, but these might have difficulties as being too big for the cache, or being too few to keep the threads supplied. Some more detailed ideas for improving the messaging are as follows. On the other hand, a lesson to take is not to try to supply more threads than can be used.

From the task graphs discussed in section 12.3, particularly Figure 13, or equally from Figure 9, it is apparent that when a task is done it may well cause several tasks to become ready at the same time. In the present experimental scheduler this will result in several *get task* reply messages being generated at essentially the same time. These will collide in that only one of these can be transmitted by OpenMPI at the same time. If a *gettask* reply message for a task on the longest path is among them, it could be delayed by the other *gettask* replies being transmitted first; in the worst case it could be delayed until all of them have been transmitted. There does not appear to be a mechanism in OpenMPI to give priority to one message over another.

One might perhaps try to account for this in the scheduler by scheduling the respective *gettask* message reply calls at a very fine scale with the reply for the longest path task first and then delays before sending the others, but even then it might not be guaranteed that the underlying OpenMPI library and the hardware would send the messages in the right order (as opposed to the promise to deliver them in that order). Certainly, this more detailed scheduling could be tried with the extra processing power allowed by the remote scheduler.

Another superficially attractive idea would be to combine these multiple "simultaneous" *get task* reply messages into a single message. This would eliminate collision between them. However, this just then creates a problem at the receiving end, which is the kernel processing threads. If the thread receiving the combined message is that which is going to process the longest path task, then before it can do that it must first communicate with each of the other threads to notify them of their new tasks. If the receiving thread is another thread, then the longest path task thread must wait for that thread to notify it. In either case an extra communication hop has been added to the wait before the longest path task can begin processing.

What would be useful here is to change the OpenMPI messaging for a more low-level library that is capable of sending a single message, containing all the new task IDs, and respective parameters, but which scatters them, using DMA, to a respective block of locations for each thread concerned, with the threads waiting for a new task polling their respective block, until the new task information arrives. This would be very quick, avoid message collision and, one would imagine, would be controllable to scatter the longest path task first. The User Mode Registration of RDMA Core [79] might provide such a facility. Its detailed suitability has, however, not been investigated, but if it is then it is a very attractive prospect for Qsargm.

13.9. Time spent on kernel processing

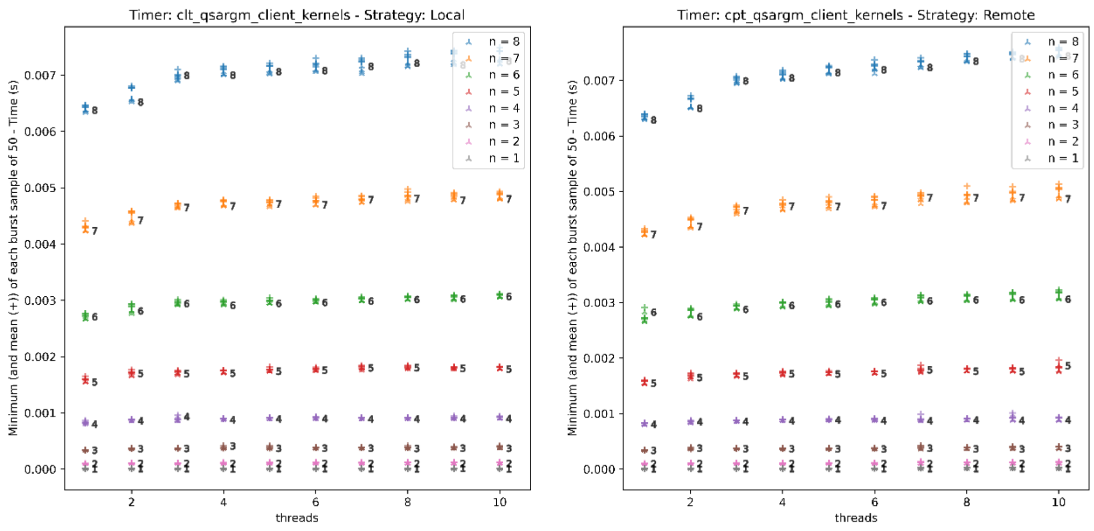


Figure 25 – Set C3, thor – time spent in computational kernels – local host scheduler

Figure 25 shows the time spend in the computational kernels for the thor machines in the *local host geometry*. In fact all the *geometries* have the same form of plot as this, which is the minimum time spent processing kernels is for the single threaded case, then, as the number of threads increases, the time spent processing kernels increases, with the rate of increase generally decreasing, or at least there is an elbow at *n = 3*. Any difference in form between the experimental and original control scheduler is not easily discerned and the magnitudes of the timer appear quite similar between the two.

Comparing this Figure to those for the task graph run time, Figure 19, it can be seen that for the single thread case that nearly all the run time is accounted for by the time inside the kernels. This is of course a sensible regime to be operating in, since scheduling should be a small overhead to the actual calculation.

That both all *geometries* and original and experimental schedulers have the same graph form for the time spent in the kernels and have similar values for this timer is not that surprising since, on the face of it, the scheduling does not explicitly interact with the internal operation of the kernels; inside the kernels the same code operates on the same values no matter how it is called. However, how is it that time spent in the kernels differs between using 1 thread and using many threads, by a few percent? Recall that the scheduler favours allotting a task to a thread that has recently operated on some of the data that the task will utilise and that this is to reduce cache misses. This is achieved by having a respective queue for each core and allotting a new ready task to a queue that has tasks with data in common with it. However, recall also the scheduler also has a competing rule, which is that if a core does not have any ready tasks available in its queue, then it will obtain one from the queue for another core. On being moved from this other core it is more likely that this core will not have data for its new task in its own cache and a cache miss will occur, degrading performance. The effect continues with increasing numbers of threads because the more threads there are the more often, in general, a core will not have any tasks in its queue, therefore causing a task to be stolen from another core, with a chance of another cache miss. The process has a theoretical upper limit, of course, which is that of all tasks suffering a cache miss for all of their data sets and correspondingly there will be an upper limit on the time spent in kernels. (In some cases, it might not be possible for all tasks to cause cache misses, with the result that the upper limit on the time spent in kernels would be lower than the case of all allocated tasks having total cache misses.) The form of the graphs of Figure 25 is clearly consistent with there being an upper limit and cache misses from allocating tasks to waiting cores rather than waiting to allocate for a cache hit is believed to be the cause. Further evidence could of course be obtained by using cache miss counters that are available on the processor. This was not done, however.

(Note that it is not concluded that all task graphs would have a monotonic increase in the kernel timer with the number of threads, only that that would be the overall general form; it could be that in some cases the pattern of distribution of tasks could dramatically change with a small change in the number of threads and hence have another effect on cache misses. Perhaps an example of that, some zig-zagging in the kernel time ratio, is discussed later in this section.)

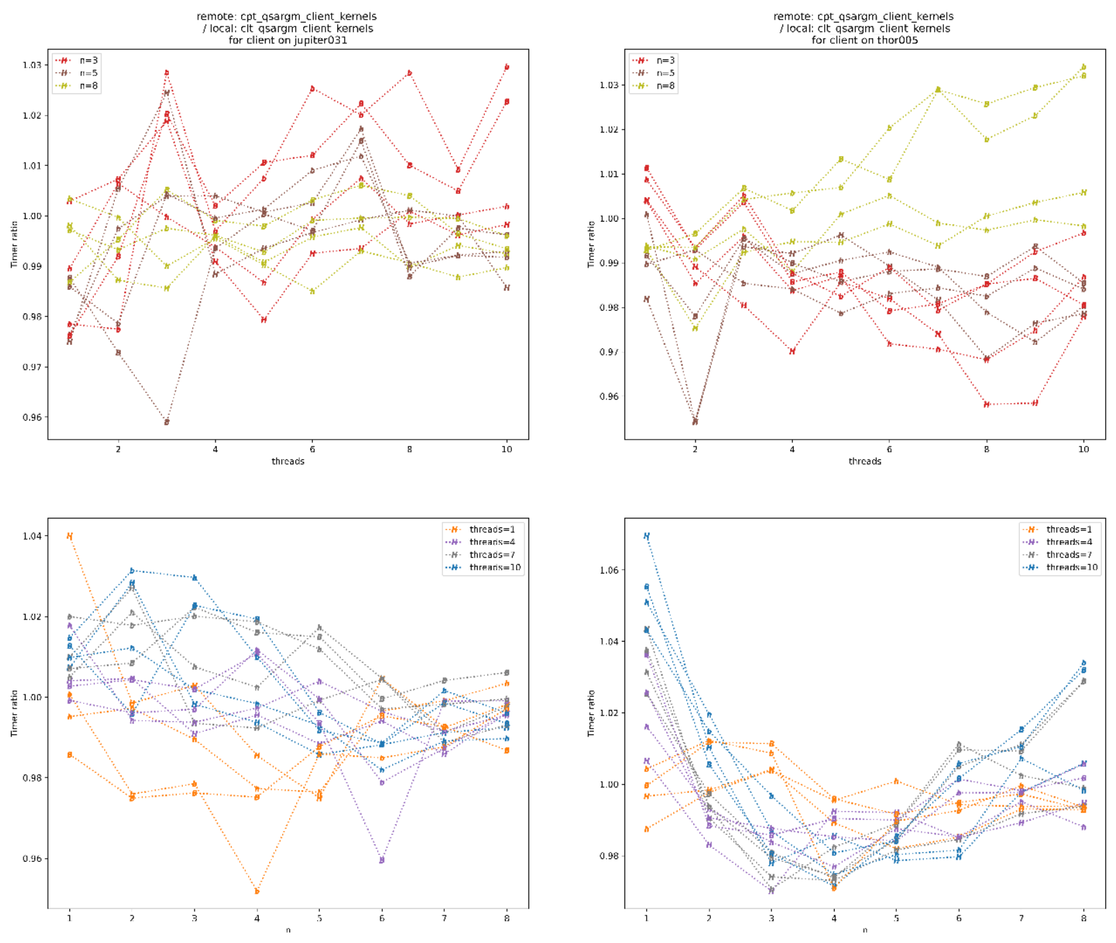


Figure 26 – Set C3 – ratio of time in kernels between experimental and original schedulers

As the differences in the time spent in kernels between the experimental and control schedulers were not easily apparent from plots like Figure 25, they were investigated by plotting their ratio. This is shown in Figure 26, which again combines Set C3 results available for two of the machine sets of Table 10; again one of the *jupiter* machine sets is on the left and the *thor* machine set is on the right; the top row of plots is against threads and the bottom row is against the tile count parameter *n*, with the *geometry* being indicated with the marker letter as given in Table 14. For clarity of the plots only a subset of the points for certain *n* or thread count was used, but the same features are apparent in plots with a full set and indeed the following comments are made in respect of the full set.

The behaviour is quite complex. While there are differences between the *jupiter* and *thor* systems there are some common features as well. For both, the ratio shows that, while the kernel timer ratio is generally around 1, there is a significant proportion, in fact a majority, of the test cases, where, for the experimental scheduler, less time is spent in the kernels than for the original scheduler. Reduced total kernel time does not always guarantee faster overall run times, since there are the overheads to consider, but it is generally encouraging.

Comparing the two machine systems, the *jupiter* system has a concentration of its ratio results in a central horizontal dense band for both the plots against threads and tile count parameter *n*, down from equality to 2% better for the experimental scheduler. For larger numbers of threads, the larger problem sizes of $n = 4$ to $n = 8$ are included in this band, which is the useful region in practice, i.e., lots of threads and large problems. For the *thor* system the band is not so well defined and in the plot against the tile count parameter *n*, the band is no longer horizontal but bows down, having a central portion favouring the experimental scheduler. So, here for a more limited range of tile count parameters, from $n = 3$ to $n = 5$ *or* 6, but again for a wide range of threads, including up to the maximum of 10, the experimental scheduler again has the advantage.

Another observation from the graphs is that, for both systems, and for both against threads *t* and against the tile count parameter *n*, the graph lines linking points for the same geometry exhibit much zig-zagging; i.e. there is a dependency on $(-1)^{n+t}$.

There are perhaps two possible mechanisms which might cause the advantage to the experimental scheduler in total kernel time, when it exists:

1. A different layout of the execution of the tasks in time and across the cores, which might be caused (i) by the different messaging latencies between the schedulers. Clearly, from the zig-zagging, there must be a different layout in tasks caused by (ii) increments in *n* or *t*. A different layout of tasks may well have a difference in cache misses involved and hence a difference in total core time, and this layout could be different between the experimental and control schedulers.
2. Because, in the experimental scheduler, the scheduler is in a separate process on a separate core, the scheduler itself does not use memory belonging to the cores processing the kernels and so will not upset its

cache. In contrast, in the original scheduler, a core processing the scheduler, after its task has finished, on becoming the scheduler will have to bring all, or parts, of the various of the scheduler's objects (the task graph tables, the core's queue and probably also other cores' queues (for work stealing)) into its cache, which may well displace some of the data that is needed for the next task allocated to the core, thereby causing a cache miss when that next task is executed by the core, when the application data might have otherwise been in the core's cach, negating the effect of the schedulers' preference for there to be such application data in the core's cache.

Now, these do not necessarily result in an effect on the task graph run time. First, in addition to the kernel time, there are the overheads to consider, and second, just because the total time spent in the kernels is reduced it does not necessarily mean that particular tasks relevant to the overall task graph execution time are affected, namely those on the longest path, or if relevant those on the contiguous critical path discussed in sections 12.2 and 12.3 for the case where off longest path tasks force apart longest path tasks. Further data concerning cache misses for particular tasks and a log of the particular cores they executed on would be needed to link the cache misses to overall run time quantitatively. However, both these effects, 1 and 2 listed immediately before this paragraph, on the cache are very plausible as the cause of the experimental scheduler having less kernel time and in turn that producing a faster overall run time than the original control scheduler, in certain circumstances yet to be discussed. The effect of reduced kernel time on the overall run time could be quite a significant one because, of course, in any sensible arrangement the time spent on processing a task is much larger than the time spent scheduling between tasks.

For the *jupiter* system, the advantage of the experimental scheduler for 1 and 2 threads, at smaller job sizes *n*, is particularly strong. It was discussed how, for smaller numbers of threads, the potential for cache misses caused by other effects remains high because the task stealing rule will not be operating to its full extent, so this may be evidence consistent with that.

In general, the mechanisms discussed here could well take effect differently between the two systems, leading to the different behaviour: they have different message latencies (as measured and noted in this thesis) and have different L3 cache sizes (and potentially structures): the Xeon E5-2697A processors used in in the *thor* system have a 40MB Intel Smart Cache [80], whereas the *jupiter* use Xeon E5-2680 V2 has only a 25 MB Intel® Smart Cache, and there of course may be many other differences in the memory system between machines of the two systems.

Now, it was true that for the Set C3 data none of the cases showed better performance in overall task graph execution time for the experimental scheduler. That does, however, mean that these cache miss effects are not significant in working against the direct effects of message latency. The section 13.10 sheds more light on the issue.

Another interesting point arising from this applies to the original scheduler *per se*: if there are a very large number of cores, for example, as exists in modern processors, which means that a cache miss is very likely for each task, it may not be worth the cost in the scheduler of maintaining priority queues for the gain of saving only an occasional cache miss. Some simpler rule of a single ordinary queue, or random selection from a pool, might be much quicker and outweigh the loss of a few avoided cache misses; indeed, it might also reduce cache misses caused by operating the scheduler in the computational thread, since much less data will have to be consulted by the scheduler to make its choice of the next task.

13.10. Optimisation of tile size – run time and kernel time for Set F

In general, task-based algorithms will have an internal parameter of the size of the tasks. Dividing the problem into larger tasks means fewer tasks and fewer messages; on the other hand, using smaller tasks may mean more opportunities to fit the tasks into spaces available on the threads. In the case of the QR factorisation studied here, the task size is directly related to the tile size. So, the effect of tile size on the task graph run time was investigated.

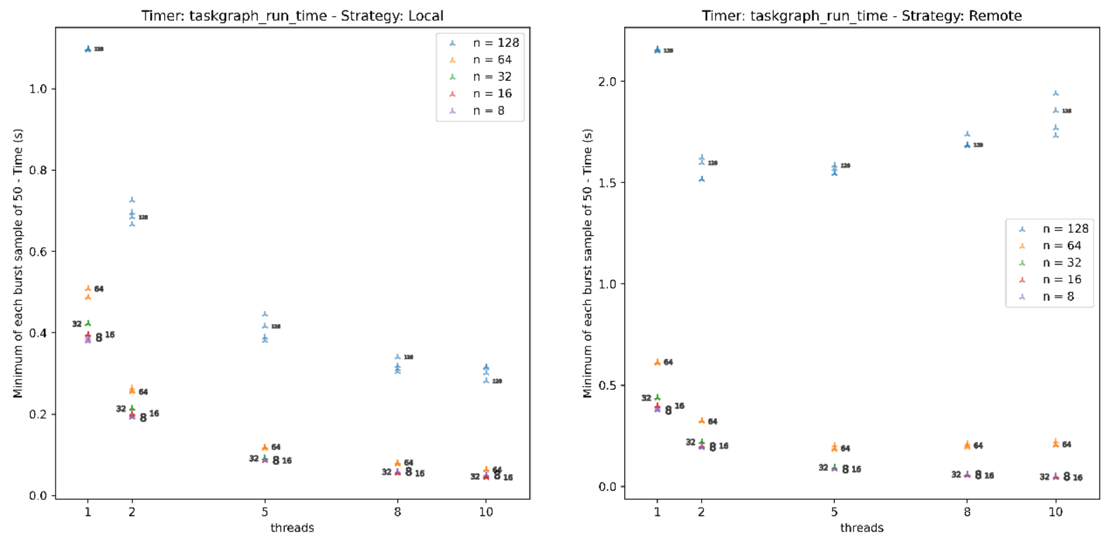


Figure 27 - Set F2, thor - task graph run time - local host scheduler – against threads

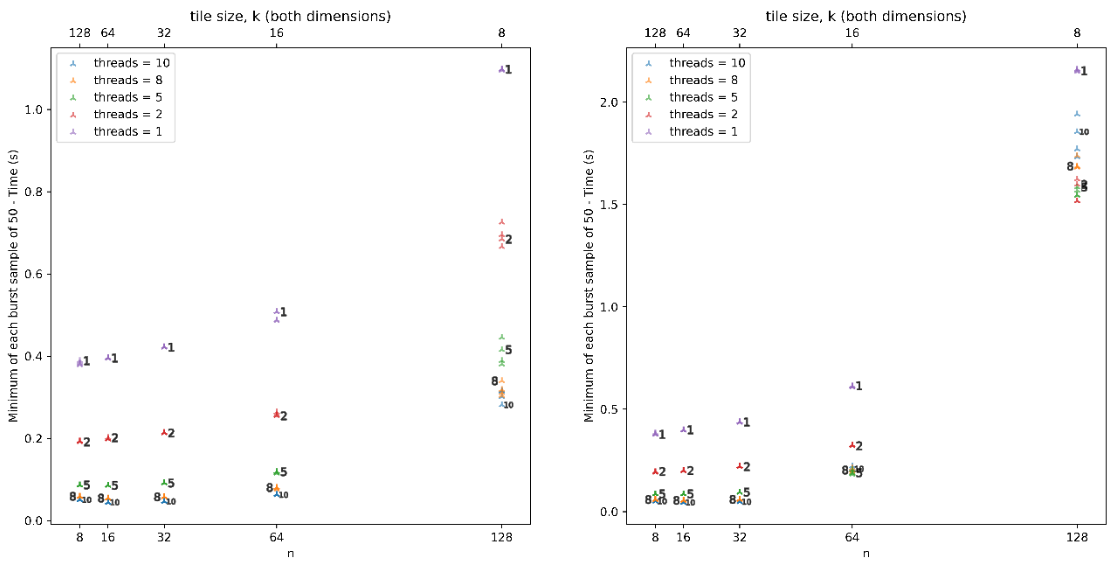


Figure 28 – Set F2, thor - task graph run time – local host scheduler – against tile count parameter n or equally tile size parameter k

Figure 27 and Figure 28 are plots of the task graph run time for the results of the Set F2 experiments, which were defined at section 13.1, for the *local host geometry*, for the *thor* system. Each marker is the minimum of each set of 50 results. In Figure 28, as it is dependent on the tile count parameter *n*, the tile size parameter *k* is marked along the top axis. The colours of the markers are supplemented with labels nearby of the *n* or threads values respectively. The low variation in the minima for the same parameters means that in many cases some or all of their four respective markers are overlaid.

From these Figures, it can be seen that many general features of the graphs for the original scheduler, on the left, and the experimental scheduler, on the right, are similar. Apart from the *n = 128* and *n = 64* cases for the experimental scheduler, the performance increases all the way to 10 threads. These larger *n* cases involve the greatest number of messages so one would expect that as the performance return for more threads diminishes, the overhead of more and more messages would eventually degrade performance. For both schedulers, at each of the higher thread counts of 5, 8, or 10 threads, generally both schedulers perform reasonably well, although from these Figures one can already see that for larger *n* the experimental scheduler is slower. Considering the tile count parameter *n,* the best performing values for both schedulers are *n = 8*, *n = 16*, *n = 32*; these all have similar performance, as indicated here by the plotted minimum of each set. To that group one might add *n = 64* for the original scheduler, since it has almost as good a performance as the others, but not *n = 64* for the experimental scheduler where performance is clearly beginning to degrade.

It is notable that for both schedulers the standard deviation in the run time (not shown) for *n = 8* is generally larger than for the other best performing cases. Here there are many fewer tasks, so it is imagined that any variation between runs in how the tasks get laid out on the cores have less opportunity to average out. If the most stable, repeatable run time were to be needed, then *n = 16* and *n = 32* for both schedulers are to be preferred. (This is not the case for the two BlueField *geometries* where *n = 16* and *n = 32* have larger standard deviation, that being there the same as for the *n = 8* case.) Although lower numbers of threads are not the general practical use case, it is interesting that *n = 8* is better performing for just 1 and 2 threads, for both the original and experimental schedulers, compared to the other cases of *n*.

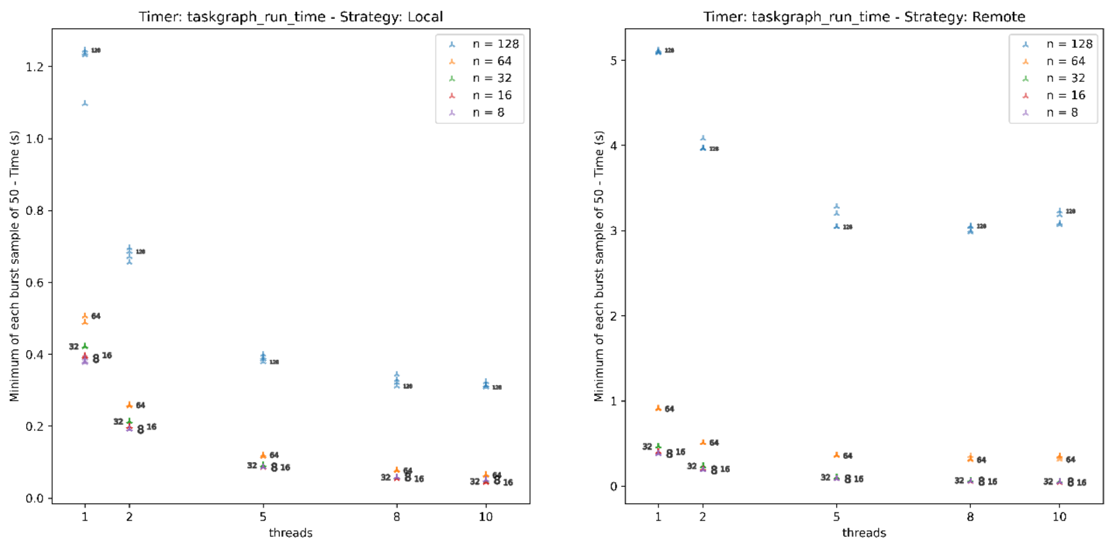


Figure 29 – Set F2, thor - task graph run time – local BlueField scheduler – against threads

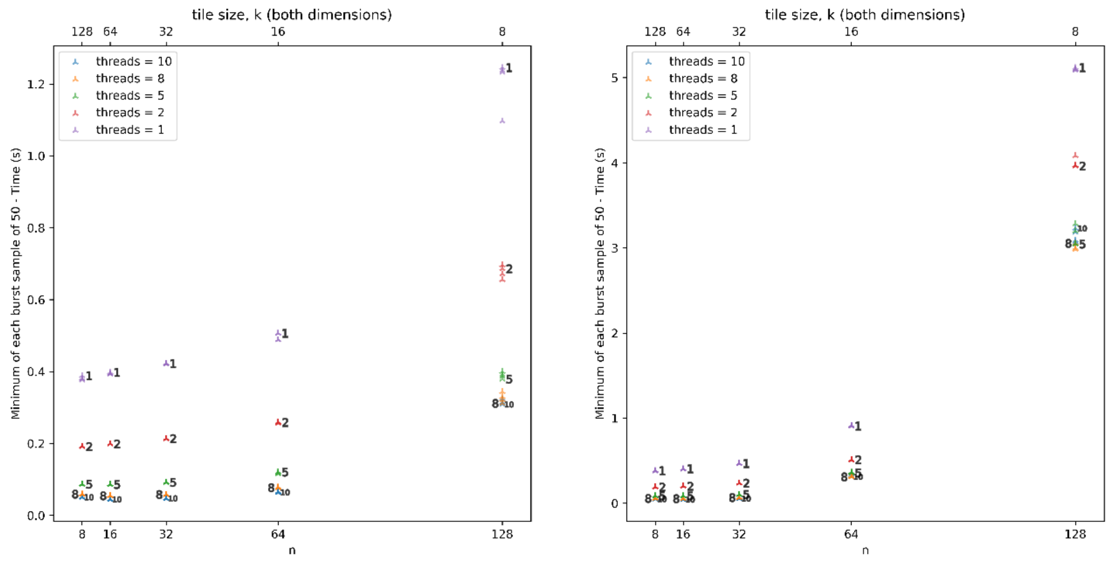


Figure 30 – Set F2, thor – task graph run time – local BlueField scheduler – against tile count parameter n or equally tile size parameter k

In the *local BlueField geometry* case of Figure 29 and Figure 30, the high-performance groups are the same as for the local host *geometry*. However, the poor performing case of *n = 128* is clearly much slower again for the experimental scheduler than for the *local host geometry*. That of course will be caused by the increased message latency given the large number of messages involved. These patterns are the same for the *remote host* and *remote BlueField geometries* as well (not shown).

It is quite striking that the overall best performing group (i) is the same group for the experimental and remote schedulers, and (ii) is the same group for all the *geometries* of the experimental scheduler.

These observations are, however, explained by all the experiments being, for a particular value of *n,* for the same task graph, and so the tasks for these may well be, for a particular number of threads, generally, laid out across the cores and in time, in the same pattern. What differs between the experiments of different *geometry* or between the original and experimental schedulers having the same values of *n* and thread count is, on the face of it, only the message latency. This will increase the overall run time in proportion to the number of relevant messages, as discussed in previous sections, and the number of relevant messages will be the same because that pattern is generally the same. The original scheduler is unified in this since the calls to the scheduler are equivalent to low latency messages. Accordingly, all *geometries* for the experimental scheduler and the original scheduler have their minimum run times in the same place.

In sections 12.2 and 12.3 it was explained how reducing the number of threads will after a certain point force apart the on longest path tasks and increase the number of relevant messages from the linear dependency on *n* indicated in the fourth column of Table 13Table 13. The third column of that Table is for the upper limit case of the number of relevant messages for a single thread, i.e., the total number of tasks, which is cubic in *n*.

The same forcing apart effect will also occur for a fixed number of threads and increasing the number of tasks: while, at low task numbers, the runtime will be the length of the longest path in the task graph, as the number of tasks in the graph is increased, the on longest path tasks will be forced apart, and in Figure 28 and Figure 30 (and similar Figures, not shown for the other *geometries)*, the run time does indeed increase rapidly for *n* beyond the best performing group (even though the size of the individual tasks is decreasing). However, it remains to explain why increasing the number of tasks does not within this high performance group, degrade performance, as is indeed the fact.

In sections 12.2 and 12.3, it was also stated that for the QR factorisation the number of threads at which the run time becomes that of the longest path may be around where the number of threads was about equal to a moderately increasing function of the tile count parameter *n* – see Table 17. But it was also shown that execution in the longest path length time might occur down to a smaller number of threads than that, but that that point would be hard to find without simulation. So, it is not clear where that happens.

Now, looking again at Figure 28, against *n*, for the 8 and 10 threads points, we can see that the point for where *n* is approximately equal to those thread counts, i.e., for *n = 8,* performs well, but those for *n = 16* and *n = 32* are not only not degraded in performance, but are in fact slightly better. Those are respectively roughly for 7 and 56 times as many tasks as for *n = 8* and, further, from Table 17, they have task graphs that are at the widest roughly 10 and 30 times the number of threads available. Therefore, it seems unlikely, from the sheer number of tasks, that the longest path execution time still applies, and yet the times are in fact slightly better than for *n = 8*. Here are some possible causes that one could entertain:

A. It might nonetheless be that the limit where the on longest path tasks have not been pushed apart has not been reached, despite the sheer number of tasks.
B. Another possible cause might be because tasks are now queuing. (As noted, for *n = 16* and *n = 32* the “width” of the task graph in the sense discussed in relation to Figure 9, could be around 10 and 30 times the number of threads available.) Queuing tasks may mean that the scheduler rule aiming to allot tasks to cores having relevant data in their caches comes back into play; when a core becomes free it is more likely that there is, among the queuing tasks, a task that is in the particular queue for the free core and hence that new task already has some data in the core’s

cache. One might expect opportunities for such cache hits to increase with *n*.

C. The intrinsic efficiency of the kernel code, related to the number and type of its internal operations, as a function of *n* would be relevant, if any increase as a function of *n* is not compensated for elsewhere.

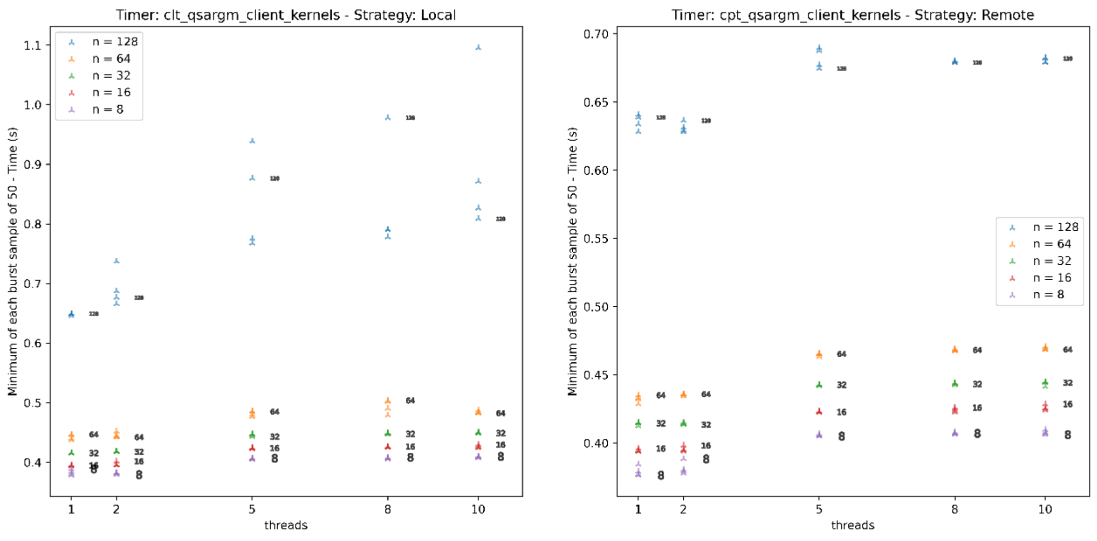


Figure 31 - Set F2, thor - total time in kernels - local host scheduler - against threads

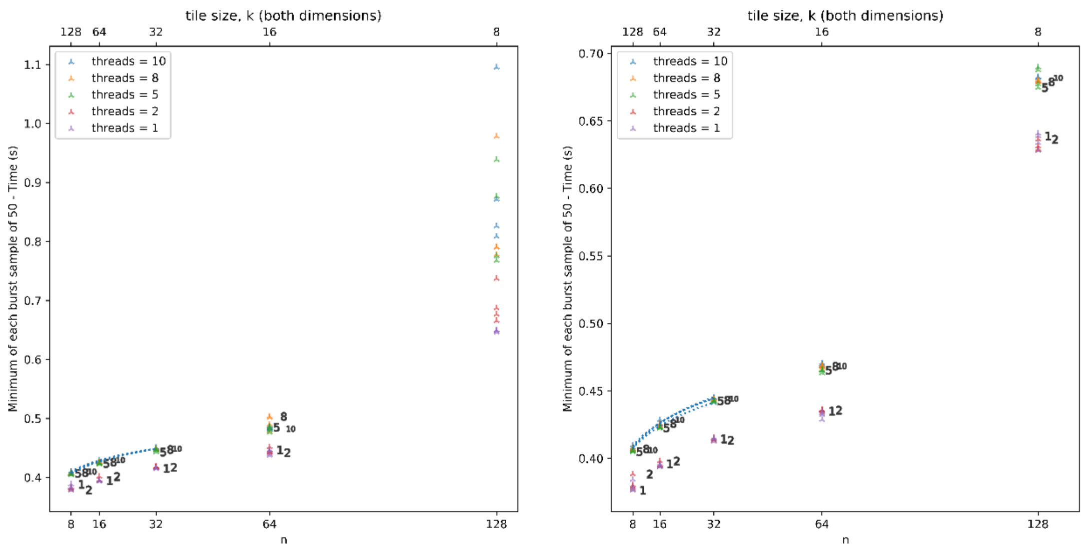


Figure 32 – Set F2, thor – total time in kernels – local host scheduler – against tile count parameter n, or equally tile size k

The data available does not identify the particular tasks on the actual critical path. However, graphs of the total time spent in all the kernels are still productive to consider. Figure 31 to Figure 34 show the total time spent in kernels for the same set F2 data from the *thor* system plotted against threads and tile count parameter *n* in the same way as for Figure 27 to Figure 30 for the task graph run time.

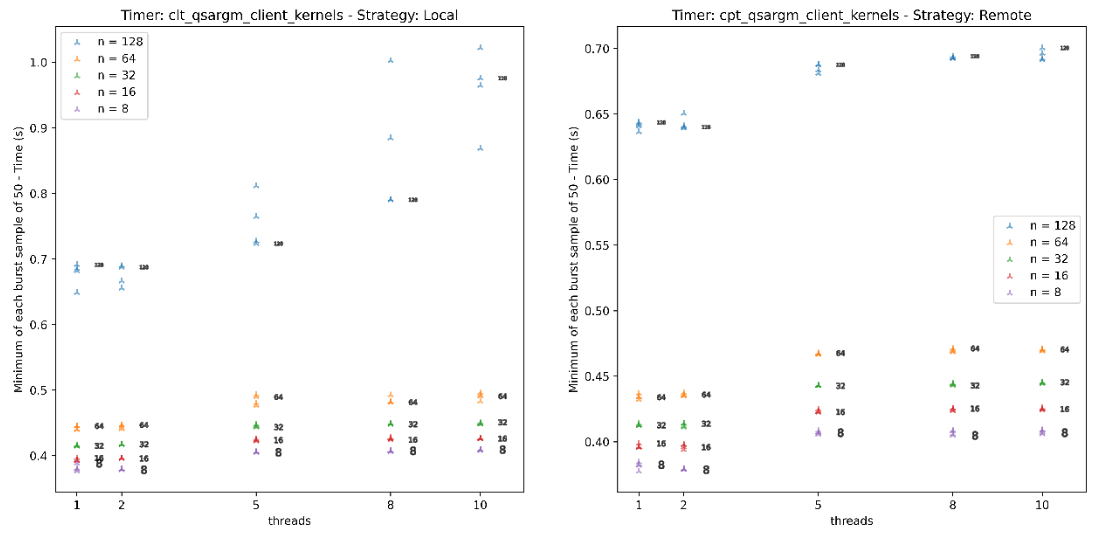


Figure 33 – Set F2, thor - total time in kernels - local BlueField scheduler – against threads

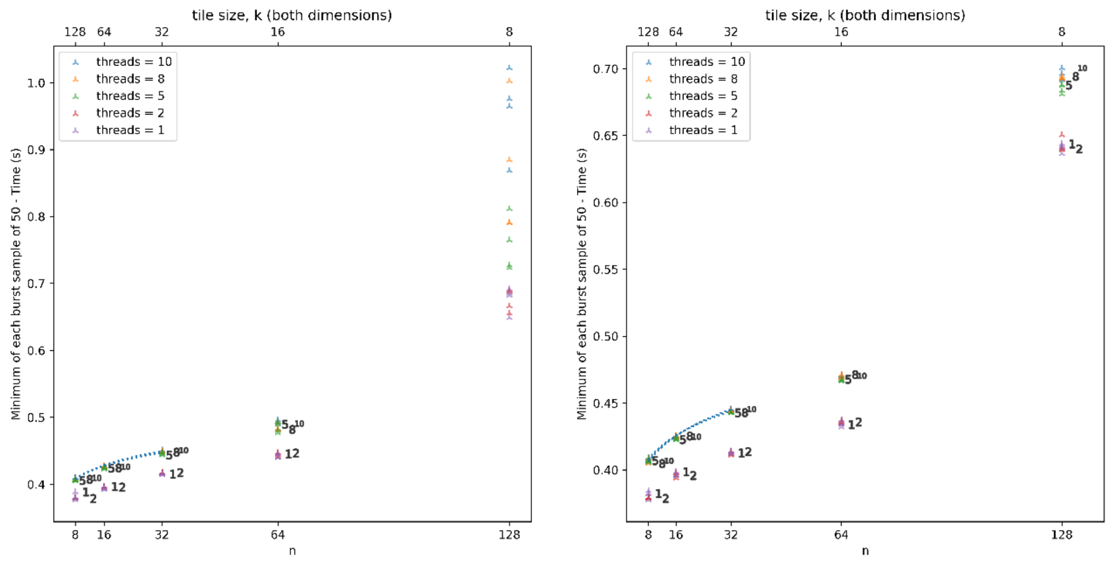


Figure 34 – Set F2, thor - total time in kernels – local BlueField scheduler
– against tile count parameter n, or equally tile size k

Looking first at the graphs against thread count, so Figure 31 and Figure 33, these show (the same is true for the other *geometries*, not shown) that for 1 and 2 threads the total kernel time has a first approximately constant value, for each value of *n*, and for 5 or more threads it has a second, higher, constant value. This is consistent with the cache miss effect between tasks of there being more frequent cache misses as the number of threads increases, but that saturating when a minimum number, perhaps but not necessarily zero, of cache hits between tasks remain. This would seem to rule out B because at high threads the cache missing has saturated, and, further, as *n* increases the time in the kernels increases, so if the effect exists it is not strong.

Turning now to the dependency on *n*, so Figure 32 and Figure 34, these again show the same general pattern for both the original and experimental schedulers and for all the *geometries* (including for the other *geometries* that are not shown), which is that the **total** time spent in the kernels increases slightly with *n*.

If it is also true, which seems likely, that the **particular** tasks on the critical path through the task graph therefore also increase just slightly in kernel time with *n*, then questions are: how does this increase with *n,* and does that differ when using the experimental scheduler compared to the original scheduler, and, then, how does that compare to the other effects on the critical path, i.e., the messaging and scheduler functioning?

To consider this the run time T_r can be broken down as follows:

$$T_r \approx N(n).\left[\frac{6.T_k(n,g)}{n(n+1)(2n+1)} + 2.t_m(g) + t_s(n,g)\right]$$

Equation 4 – a task graph runtime analysis

where,
T_k is the total time spent in kernels,
n(n+1)(n+2)/6 (= N_G*)* is the number of tasks in the QR factorisation task graph,
N is the total number of tasks on the actual critical path,
t_m is the time messaging time, and
t_s is a typical time for the scheduler to schedule one task.

Now, t_s should be small compared to the time for a task, unless scheduler is swamped at high *n*, and was not investigated. t_m should also be smaller than a task but could be more significant for the experimental remote scheduler. *N*, as discussed in sections 12.2 and 12.3, can range between $N_L = 3n-2$ and $N_G = n(n+1)(2n+1)/6$ depending on how confined the task graph is by the number of threads.

The dependency of T_k on *n* is examined in detail because clearly it will have a strong influence on the total run time. Figure 32 and Figure 34 show that the dependency is weak, almost constant. The points on these graphs are the minima of each set of experiments and the blue curves are for a simple model, the function $T_k = a.n^p$ fitted to those for *n = 8*, *16* and *32*, for the 10 threads points. The maximum of 10 threads was chosen because as it is realistic that a user would want to use a large number. (A fit was also tried including the *n = 64* points as well, but the curve did not fit the *n = 32* point so well; as can be seen, performance at *n = 64* is beginning to deteriorate, before it very badly deteriorates *for n = 128*, pulling the fitted curve away from the *n = 32* point.) The values of *a* and *p* found for the fit are as set out in Table 20, which gives values for the *thor* machine set discussed and also a set of *jupiter* machines.

T_k = aN^p fit parameters			
client *(so, covering all 4 geometries)*	**Scheduler – original or experiment**	**mean a (seconds)**	**mean p**
thor005	original	0.3538	0.0685
	experiment	**0.3591**	**0.0612**
thorbf005	original	0.3542	0.0681
	experiment	**0.3572**	**0.0632**
thor006	original	0.3548	0.0669
	experiment	**0.3574**	**0.0628**
thorbf006	original	0.3545	0.0677
	experiment	**0.3609**	**0.0599**
jupiter031	original	0.3228	0.1193
	experiment	**0.3301**	**0.1108**
jupiterbf031	original	0.3194	0.1223
	experiment	**0.3273**	**0.1138**
jupiter032	original	0.3195	0.1221
	experiment	**0.3288**	**0.1123**
jupiterbf032	original	0.3192	0.1229
	experiment	**0.3282**	**0.1132**

Table 20 – Parameter fit values for $T_k = an^p$

A point about this, however, relates to the total time in the kernel being fairly constant. (While there is a difference in p for the *jupiter* and *thor* systems the values are both in the region of 0.1, therefore that is true for both.) So, this is a property of the overall computational algorithm, in this case the tiled QR factorisation. If it scaled poorly with n then of course it would not be known as a useful algorithm. (How poorly could be tolerated in general would depend on how N in Equation 4 scales for a particular algorithm.)

Now, in this case, we also know that, in this region of interest, the run time T_r is also roughly constant. For that to be true that the number of tasks on the critical path N must nearly scale as $n(n+1)(2n+1)$ – *see* Equation 4 – so only differing approximately by the factor of $p \approx 0.1$, to cancel the coefficient of T_k. This dependency of N is not so surprising; as discussed when expressing less enthusiasm for A, for $n=16$ and $n=32$, it was suspected that these are in the region of on longest path tasks being forced apart and so will be on the way to the limit of N of $n(n+1)(2n+1)/6$ tasks (remembering that limit is of course for the single thread case). So, indeed, this eliminates A.

The values in Table 20 also show differences between the experimental and original schedulers. It is particularly notable that p for the experimental scheduler is always smaller than that for the original scheduler, which is in the direction of giving an advantage to the experimental scheduler with increasing n. So, to see whether this is significant, the values of a and p and the model of $T_k = an^p$ were used to produce the difference in T_k between the experimental and original schedulers scaled down by the total number of tasks, to give per task values, and then the mean of those were then scaled in units of 2 message times relevant for the geometries from Table 8, to give the results in Table 21.

Difference of experimental to original scheduler total core time, over task count, in units of 2.t_m (-ve advantageous for experimental) **Scheduler server Machine**	**n = 8 (k = 128)**	**n = 16 (k = 64)**	**n = 32 (k = 32)**
thor005	-0.84	-1.56	-0.41
thorbf005	-1.12	-0.47	-0.11
thor006	-1.13	-0.48	-0.11
thorbf006	0.82	-0.32	-0.10
jupiter031	9.53	-0.42	-0.32
jupiterbf031	4.69	0.11	-0.07
jupiter032	4.72	0.12	-0.06
jupiterbf032	4.43	0.08	-0.07
Approximate mean task time (both systems)	2 ms	300 µs	40 µs

Table 21 – difference between experimental and original scheduler's total core time over task count based on T_k=an^p model units of 2t_m

At the bottom of Table 21, there is also given an approximate value for the duration of a single task, for comparison with two messaging times, which of course is around 1 µs. For the case of *n = 8*, the duration of one task is around 2 ms, and so over 1000 times the duration of a message; therefore, for such a case, which is in the high performance region, the messaging introduced by the remote scheduler *per se* is not practically significant at all. (On the other hand, for the jupiter system the total core time has actually increased, by around 0.5% or 0.25% depending on the geometry.)

However, for the *n = 16* and *n = 32* cases, which have shorter task execution times, adding the messaging time to each task would be considered practically significant. However, the evidence of the Table, for many cases, is that when the experimental scheduler is introduced there is a decrease in kernel time and that that is on a scale similar to the extra messaging time. For one particular case, *n = 16* for the local host *geometry* (so *thor005* being the scheduler), the value is -1.56, so for each task the extra messaging time is more than compensated for by the improvement in the kernel time.

For the other *geometries* for the *thor* system, for *n = 16,* the compensation provided by the core time reduction is partial but still enough to mean that future improvements to messaging latency for those geometries would help significantly. Of those, large improvement to the local BlueField geometry message latency is perhaps the more likely since the card is newer technology and is directly attached to the kernel processing host, while the other geometries involve shared memory messaging or messaging over the local area network, which are older technologies which will have benefitted from long periods of optimisation.

In the *jupiter* system the compensation does not appear to exist on the same scale but the position generally improves across the Table, so with increasing *n*. For the thor system, the reduction in core time peaks at $n = 8$ (or before), or $n = 16$, and then declines with increasing n.

So, what effect could there be that would improve the kernel performance according to these trends against *n*?

Differences in cache misses in the tasks' data resources caused by differences in task allocation between the original and experimental schedulers seem less likely to be large. A difference in operation between the experimental scheduler and the original scheduler is, however, that in the original scheduler, each time a core becomes the scheduler, the scheduler data will, to some extent be read into the cache of that core. Just before that, the data produced by the previous task on that core will have been in the cache, or at least as much of it as will fit (or perhaps less than that depending on how the kernel processes the data). The reading in of the scheduler data will to some extent obliterate that result data, making it a cache miss if the task allocation is working to allocate a task to use that result data. The sorting of a core's queue will work over a lot of the data.

The amount of data left in a core's cache by each task that is obliterated by the original scheduler's operations to find the next task for the core is hard to quantify without a detailed model of the cache. However, we do know that the size of the result tile(s), so $k \times k$ elements, scales as $1/n^2$, since in these experiments $k \times n$ is a constant $(= 1024)$, so this might account for the decline in the effect for the *thor* system towards $n = 32$. On the other hand, the size of at least parts of the representation of the schedule that the scheduler will read will increase with *n*; for example, the number of tasks in the tasks table N_G scales as n^3, increasing the chances that scheduler operation will obliterate the cached results tiles.

One can check the sizes of the tiles relative to the cache sizes.

jupiter: E5-2680 V2 Intel Xeon processor [81]	
L1 data cache	32 KB per core
L2 cache	256 KB per core
L3 shared cache	25 MB

thor: E5-2697A V4 Intel Xeon processors [82]	
L1 data cache	32 KB per core
L2 cache	256 KB per core
L3 shared cache	40 MB

Table 22 – jupiter and thor processor cache sizes

Comparing the data size of a single tile from Table 13 to the cache sizes set out in Table 22Table 23, one can see that for *n = 8* a single tile would fit into the L2 cache but not the L1, so certainly not all the result will be in the L1 cache (in the QR factorisation the results can be more than 1 tile for some kernels). More of a result tile may fit in the L1 cache for *n = 16* and all of a result tile would fit for *n = 32*. So, if this effect is operating, say, at the L1 cache a similar amount of result data might be there for *n = 8* and *n = 16* so the increased scheduler data may be operating to obliterate more of it and result in better kernel time for the experimental scheduler as observed. This is, however, all very speculative. The cache operation will be very complex and it is not known here whether a kernel would even leave all of its data in the cache even if there is space for all of it. Moreover, this would apply to the *thor* system; for three of the *geometries* in the *jupiter* system the trend is of improving kernel time performance, so something else must be occurring her, or at least dominating the effect.

Nonetheless this differing in total kernel time between the experimental and original schedulers is a fact, and so some interaction with the scheduler must affect the speed of operation of kernels and that effect being via their cache behaviour seems highly likely, since how else could their speed of calculation be affected? (For example, more interruption in the case of the original scheduler does not seem likely.)

As an aside, this kernel result data obliteration mechanism also suggests a second similar kind of possible cache effect in the scheduler. In the original scheduler each core becomes the scheduler and calls into the cache at least some of its data tables, but this scheduler data in the core's cache will be obliterated to some extent by the operation the next kernel allocated or indeed, the scheduler coming into existence on some other core will not be able to make use of it because it is in the cache belonging to another core. In contrast, the experimental scheduler has a continuous existence separate from the cores processing the kernels, so it can keep its scheduler tables in its cache between the calls that are made to it. Such an effect might increase in size with n as the size of the scheduler data increases. The scheduler processing time was not however investigated.

13.11. Optimisation of tile size – relative performance for Set F

Figure 35 is a plot of the ratio of the mean of the run time for the two schedulers. The points are plotted against the tile count parameter *n* and the number of threads (t) on the same axis and are coloured and have marker symbols that distinguish the geometry (from Table 14).

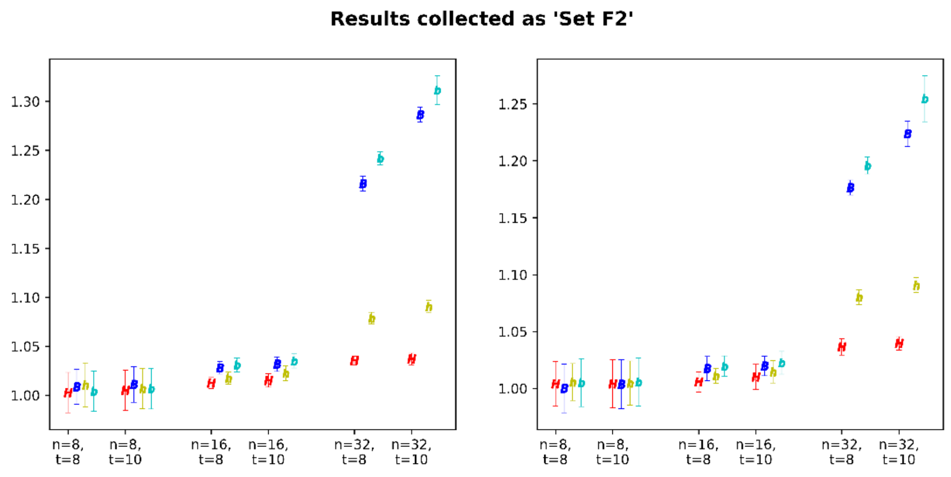


Figure 35 – Set F2 – ratio of run time for experimental and original schedulers

The error bars are calculated as the simple rule for error propagation for a ratio from standard deviations of the experimental and original schedulers respectively, therefore having the usual caveats. The left-hand subplot is for the *jupiter* system results and the right-hand is for *thor*, and as may be seen the data is selected to being that only of the high-performance region identified in section 13.10.

A ratio of less than unity means that the QR factorisation example runs faster with the experimental scheduler than with the original scheduler, for the same values of *n* and client thread count. As may be seen in the Figure, for *n = 8* and *n = 16* the points are within just few percent of unity. In fact, the standard deviation bars extend below unity, so while the mean ratio is above 1, the experimental scheduler does sometimes run faster than the original.

It can be observed that for *n = 8*, the ranges for the different *geometries* mostly overlap, while at *n = 16* and *n = 32,* the *geometries* with longer messaging times degrade progressively both with *n* and with longer messaging time, which is as would be expected. Further, the performance for the *thor* system is better than *jupiter* at low *n*.

The points in the Figure are tabulated in Table 23 and Table 24.

Client machine	Server machine	n	Threads (t)	ratio	Standard deviation
jupiter031	jupiter031	8	8	1.003	0.021
jupiter031	jupiter031	8	10	1.005	0.020
jupiter031	jupiter031	16	8	1.012	0.006
jupiter031	jupiter031	16	10	1.015	0.007
jupiter031	jupiter031	32	8	1.036	0.004
jupiter031	jupiter031	32	10	1.037	0.006
jupiter031	jupiter032	8	8	1.011	0.022
jupiter031	jupiter032	8	10	1.007	0.021
jupiter031	jupiter032	16	8	1.018	0.006
jupiter031	jupiter032	16	10	1.023	0.008
jupiter031	jupiter032	32	8	1.079	0.006
jupiter031	jupiter032	32	10	1.090	0.006
jupiter031	jupiterbf031	8	8	1.009	0.018
jupiter031	jupiterbf031	8	10	1.011	0.018
jupiter031	jupiterbf031	16	8	1.028	0.006
jupiter031	jupiterbf031	16	10	1.032	0.008
jupiter031	jupiterbf031	32	8	1.216	0.008
jupiter031	jupiterbf031	32	10	1.286	0.007
jupiter031	jupiterbf032	8	8	1.004	0.021
jupiter031	jupiterbf032	8	10	1.007	0.021
jupiter031	jupiterbf032	16	8	1.031	0.007
jupiter031	jupiterbf032	16	10	1.035	0.007
jupiter031	jupiterbf032	32	8	1.242	0.007
jupiter031	jupiterbf032	32	10	1.312	0.015

Table 23 – ratio of experimental to original scheduler run time - jupiter system

Client machine	Server machine	n	Threads (t)	Ratio	Standard deviation
thor005	thor005	8	8	1.004	0.019
thor005	thor005	8	10	1.004	0.021
thor005	thor005	16	8	1.006	0.009
thor005	thor005	16	10	1.010	0.011
thor005	thor005	32	8	1.037	0.007
thor005	thor005	32	10	1.040	0.006
thor005	thor006	8	8	1.006	0.016
thor005	thor006	8	10	1.005	0.019
thor005	thor006	16	8	1.011	0.007
thor005	thor006	16	10	1.015	0.010
thor005	thor006	32	8	1.080	0.007
thor005	thor006	32	10	1.091	0.007
thor005	thorbf005	8	8	1.000	0.021
thor005	thorbf005	8	10	1.004	0.022
thor005	thorbf005	16	8	1.017	0.011
thor005	thorbf005	16	10	1.020	0.009
thor005	thorbf005	32	8	1.176	0.007
thor005	thorbf005	32	10	1.224	0.011
thor005	thorbf006	8	8	1.005	0.021
thor005	thorbf006	8	10	1.006	0.021
thor005	thorbf006	16	8	1.020	0.009
thor005	thorbf006	16	10	1.023	0.010
thor005	thorbf006	32	8	1.196	0.007
thor005	thorbf006	32	10	1.255	0.020

Table 24 – ratio of experimental to original scheduler run time - thor system

Many points are within a very respectable 1% performance degradation. The original objective was not to be better than the original scheduler, which seemed unlikely given the substantial messaging latency that was being introduced, so it is gratifying to discover that, through other interactions in the system, there are some cases where the experimental scheduler is as good as the original for the mean run time.

If these results can be repeated for use cases for which the experimental scheduler was envisaged, involving more complex scheduling, for example to provide other performance benefits, then cases where the experimental scheduler is slightly worse, for example, within 1% performance degradation are likely to be considered useful.

Of course, to find the fastest scheduler, one has to carry out an optimisation, but that is true also of the original scheduler. While outside the window of good performance, the experimental scheduler is very poor, that is of no consequence if the high performance region is suitable for the application. Usefully the region of high performance is wide in the *n* parameter governing the task size and is only a little narrower than that of the original scheduler. A wide window is useful as it is easier to find and stay within if there are other constraints. For example, if there are other optimisations in the system to be carried out that depend on data size, for example, transferring blocks of data to other nodes in a multi-node calculation; there is more chance that the useful data size regions will coincide.

Forming the ratio of the results for the same number of threads *t* and tile count parameter *n* is appropriate comparison of the two schedulers, but they do not tell one which is better if there is a free choice of *n* and *t*. Figure 36 is a scatter plot of the same data as for Figure 35 but showing the absolute run times (in seconds) for all the *geometries* of the experimental scheduler and now also for all the results for the original scheduler, and still filtered again down to the high performing group of *n = 8*, *n = 16, n = 32* and *t* = 8, *t* = 10. The same colour coding is used, so red for the local host geometry, blue for the local BlueField, yellow for the remote host and teal for the remote BlueField, with the original scheduler being black. The plot of course confirms the information that at *n = 8*, there is a high degree of overlap in performance between all the experimental geometries and the original scheduler, that at *n = 16,* the performance of the experimental scheduler is close to the original and that at *n = 32,* the performance of the experimental scheduler is degraded and strongly dependent on the message latency.

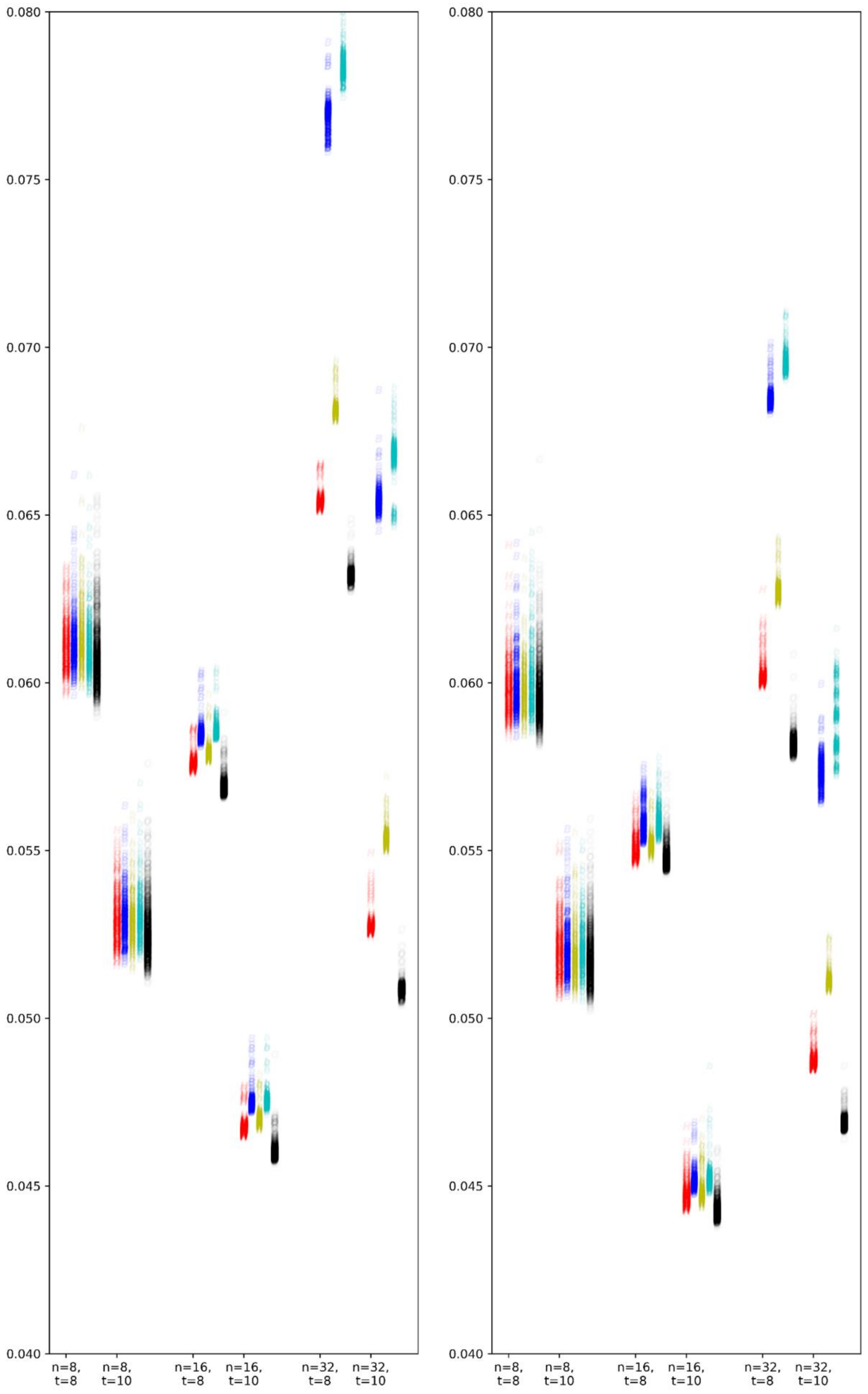


Figure 36 – Set F2 task graph run time (s) for high performance region of tile count parameter n and threads (t) with markers by geometry, for jupiter (left) and thor (right) systems

However, this plot further reveals that *n = 16* is clearly faster than *n = 8* and *n = 32*, and that that is equally true for the experimental scheduler. There is a small advantage for the original scheduler at n = 16, but it is far more important to optimize the tile size. The choice between the original and the experimental scheduler makes a very small difference. In this case at least, the approach of the experimental scheduler, of moving the scheduler to a separate process and communicating by messaging, is vindicated.

14. Improvements and Applications

14.1. Considerations for larger matrices

The experiments were for a matrix of 1024 × 1024 elements. The size was set by the need to gather many data points in a reasonable amount of time. This size of matrix may well suffice for many problems, but in some applications it is necessary to process matrices of the size of the RAM of a node. 1024 × 1024 elements equate to 8 MiB of storage (for double precision elements of 8 bytes each). 128GiB, so typical of node RAM sizes in HPC nodes, would store 2^{17} x 2^{17} elements. In the set F experiments the optimised tile count parameter *was n = 16*, for which the tile size was 64 x 64 elements. If one scaled the QR factorisation to a matrix of 128 GiB, while keeping to the same tile size out of a desire to keep the inter-task cache hits, then this would have n = 4096. This is not practical. Guessing at 32 bytes per task, the size of the scheduler representation would be 683 GiB, which is typically too large to fit into a node and is much more than the payload matrix data being processed and so will take significant time to process to make the scheduling decisions.

A redesign would be needed. One could bring the number back in bounds by significantly increasing the size of tasks. This abandons having inter-task cache hits. As discussed, large tile sizes will reduce the cache hit effectiveness, but the evidence of this work is that the cache hits are reduced anyway by the scheduler rule to use other cores if they are available. With large tiles the optimisation to use the processor cache effectively will have to be inside the kernel functions. Processor caches are optimised to keep data processed in the immediate past; inter task cache hits stretch this beyond that design intention. Larger tasks, also as discussed, make messaging overhead much less significant.

Table 17 shows that $n = 32$ may well, or $n = 64$ certainly, keep a modern processor of 40 or 80 cores supplied with tasks. These would be tile sizes of 4096 × 4096 or 2048 × 2048 elements, so of 128 MiB or 32 MiB. Messaging time would be very insignificant (see Table 21) but these are too large for the processor cache, which would be a problem if efficiency inside the kernels strongly depended on the size of the tile relative to the cache. The largest tile sizes used in Set F were 128 × 128 elements, which is 128 kiB, which is still within the L2 cache. Up to that point increasing tile size was reducing the total kernel time, but the next size, in the doubling scheme used, of 256 × 256 elements, which would be 0.5 MiB, would be larger than the L2 cache but would still be inside a core's share of the L3 cache, so it is hard to say whether the core time would continue to improve. Messaging time would certainly be insignificant, but the *n* would be 512, so the task graph representation estimate would still be over 1 GiB; this is still very much larger than the tile size and so seems likely to be burdensome to process for each task. Whether there is a sweet spot lying beyond there would require further experimentation.

Another approach to redesign would be having the task creation run in parallel with the task execution to keep the size of the task representation down. This again points to a separate process for the scheduler (to be discussed in section 14.3). Here the size of representation for the scheduler would be traded against use of cores to calculate the tasks on the fly. Whether that could be made efficient was beyond the scope of this work. For the QR factorisation calculating the next tasks beyond the working front in the task graph is easy, there are very simple rules for generating the tasks, but what is lost is finding the longest path, which uses the entire task graph. Some thought would have to be given whether the regularity in the structure of the task graph and the invariance of length of the longest path are clues to finding a rule to keeping to, or approximately to, the longest path that could be applied more locally in the task graph.

In this light, the extreme example of a task graph with no dependencies is quite straightforward when it comes to a problem taking up a large, or indeed any amount of memory. The work should be divided equally among the threads, as single task for each. Dividing it into more tasks would result in messaging and task scheduling overheads being incurred. One might then say that each task kernel will be too large for the processor cache, but this is then a problem for within the kernel – the work can be ordered within the kernel in cache sized chunks, bit scheduling is not required. In general, this is an indication to consolidate parallel tasks where possible within the task graph, without of course compromising the ability to feed the threads.

14.2. Multithreaded server experiments and proposed improvements

After a simple bug had been identified which caused the scheduler on the server to run in only a single thread this was corrected, and experiments were tried again.

However, once multithreaded server operation was reinstated and the extra complications were dealt with as described in Chapter 7, the performance was worse. There were several possible reasons for that. First, it may have taken longer for the OpenMPI library to sort out the messages onto the different server threads. Second, while the server from QuickSched is inherently multithreaded, transmission of calls via OpenMPI will serialise them, wherein contrast in the original more than one thread, to a certain extent, could become the scheduler at the same time. With the messages serialised, the calls to enter the scheduler are spaced out, and in the extreme may not even overlap. Third, multiple threads may be moving scheduler data between cores on the server unnecessarily.

Further investigation of the problems here, may be needed to decide if the performance of the server part of the scheduler can be improved, However, a suggestion is to separate the threads available on the server into different tasks. A first group of threads, perhaps just one, would deal with communication with the client. This would then launch the original functions of QuickSched on **ones** of another set of threads asynchronously, so checking from time to time for replies while otherwise dealing with other messages, and then sending those back to the client when they are ready. The number of threads in the second group could be varied to find the most efficient number.

On the other hand, the threads in the second group, those processing the actual calls to the QuickSched functions, are not now limited to a single thread. In the original the scheduler could only use its own thread because the other threads are being used to process kernels. That does not apply with the scheduler in a remote process, so the QuickSched functions could be parallelised. Two places come to mind: (i) resorting of the queues (especially if they are redesigned to be fully sorted, rather than being a priority queue), and (ii) searching other queues for work stealing when a thread’s own queue is not supplying a task; each queue could be searched in parallel with the first to supply a task terminating the search.

However, the one queue per thread was an aspect of the original design where it was allotted tasks that may reuse data in the cache, while still be a quick to access data structure. The whole issue of the queuing and its data structures is therefore worthwhile reconsidering. For example, some more elaborate data structure recording which data resources are actually in the cache of each core (inferred from the tasks that have been sent there) could be kept, together with a complete record of which tasks in the whole task pool would want to use those resources. The server now has more power to interrogate such data and so may be able to make a cache hit more often.

Another feature inherent in the design of the original QuickSched is that the scheduler must process its call quickly and then the thread must return to kernel processing. Although there is parallelism in that calls from different threads can be processed to some extent in parallel, it is not possible for the scheduler to work on the questions put to it while the thread in question is processing kernels. So, the new arrangement could make use of background processing. In one example the scheduler could keep ready an answer, revising it as *task done* messages arrive, to the question from a thread of what is its next task, *gettask*, with the answer being sent back immediately it is asked, while processing to maintain the scheduler data, so processing the consequences of the *task done* message, namely finding new ready tasks, selecting a queue for them and resorting the queues is done in the background. This does, however, alter the question being asked by a thread from what is the next task, given that I have just told you which task I have just completed, to what did you think was the best task a moment ago? This difference may or may not lead to less efficient task allocation. It will mean when few tasks are about that queues are starved, but then one just processes the *task done* to see if that generates new tasks, which is the same as the original scheduler and so is not a detriment. Even if the new question is sub optimal, it may be that its quicker answer is a sufficient payoff.

14.3. Proposed improvements to task graph building in Qsargm

A clear outcome of the set C experiment, discussed at section 13.2, was that the task graph build time for the Qsargm scheduler is more than 10 times slower than for the original scheduler, in fact, 30 to 40 times slower for larger task graphs and *geometries* having actual messaging links, this is clearly not helpful to the utility of the experimental Qsargm scheduler and so alternatives are now considered. The problem is that the operation in the remote scheduler version to record a new task is a simple matter of recording a few values in a couple of tables but each such operation is burdened with its own messaging overhead; in fact, several messages are used per task as noted in sections 13.2 and 5.10. In the case of MPI messaging the overhead is much greater than the recording operation itself. Nonetheless, this suggests that the following changes to the client-server architecture will be improvements:

A. Instead of building the representation of the task graph in the process of the remote scheduler as is done in the experimental code, the task graph representation could be built first in the client compute process, and when the tables representing the task graph are complete those should then be transferred *en bloc* in a single (or just a few) messages to the remote scheduler process. Once the tables are there, the remote scheduler can then perform the execution of the task graph as normal. (So again, this is a *static task graph*.) Owing to the very significant reduction in the number of messages involved, from $O(n^3)$, for the QR factorisation example, to $O(1)$, this method will be much quicker.

   This method is illustrated in Figure 37, which shows the task generation code on the host processor, i.e. the processor that will later execute the tasks. The cores of the host processor are again shown as purple squares and a white oval on them means that they are processing the task generation code. Two of these are shown because, in general, task generation could be performed in parallel (as indeed it is in the Swift code). The task generation will again take $O(n^3)$ steps for a QR factorisation but those are much quicker because now each task generation does not involve any messaging. The built task graph representation is then copied *en bloc* to the remote scheduler. Once there the task graph can be executed by the remote scheduler exactly as explained with reference to Figure 1 or Figure 4.

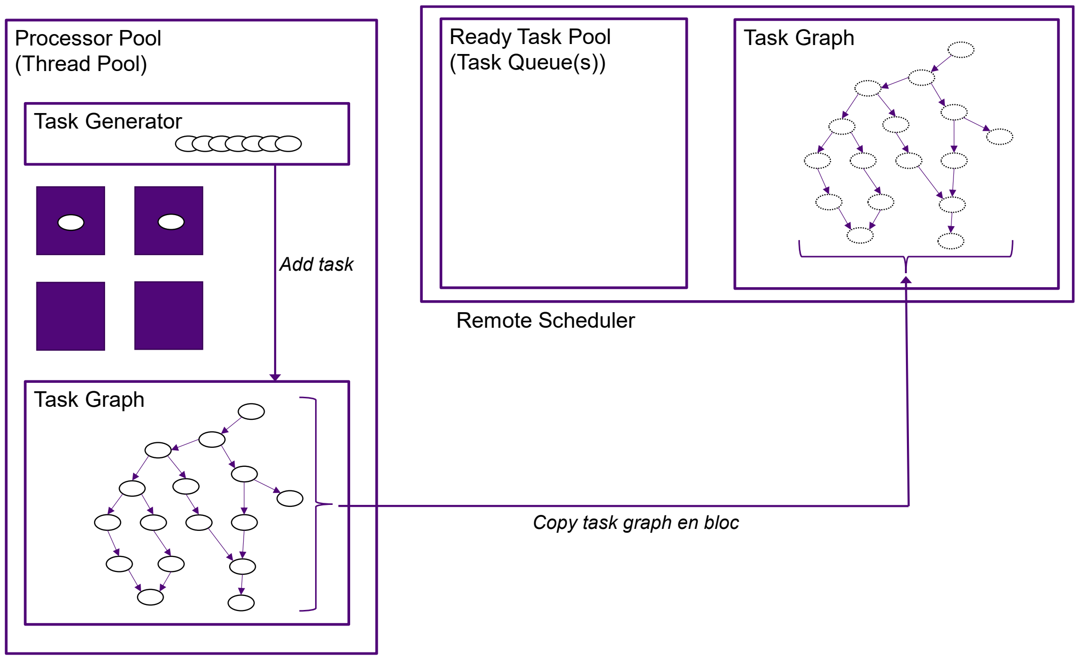


Figure 37 – Creating task graph en bloc on host processor

B. In conversation with Richard Graham of Nvida Networks with me about the problem, he suggested very generally overlapping task creation and execution. So, this is now a *dynamic task graph*, in that the task graph gains tasks after execution of it has begun. In this section I have worked through how that might be done. So, in an alternative to calculating the tasks *en bloc,* once the task graph has been compiled, the tasks could be sent one at a time as in the experiments, but the cost of the messaging could be hidden by starting the task graph execution before the task graph is complete. In fact, that could be started well before completion of the task graph build, i.e., shortly after the building of the task graph is commenced. So, once some tasks have been sent to the scheduler and are in the task graph representation there, execution of the task graph can be started by allocating some of those tasks to the compute cores. This would require the task build code to create first those tasks that are to be executed first. In many cases, that would not be a burden to the application programmer because, of course, it is common, for algorithms to be thought about, and/or be expressed, in the order of the operations to be executed. So, this will be quite natural for the application programmer to organise.

Figure 38 shows the arrangement. In this particular example, the task generator is located on the host processor that will execute the tasks. (Other locations, for example, on the remote processor would, I think, also allow the overlap of task generation and execution.) The host processor has multiple cores/threads and at least one is allocated to the task generation code. Here, however, the task generator does not build the representation of the task as in the arrangement of Figure 37, but reverts to the original method of sending small messages to the scheduler, which interprets them to build the representation of the task graph there. As shown, tasks at and near the root of the task graph are the first to be generated, queued, and executed.

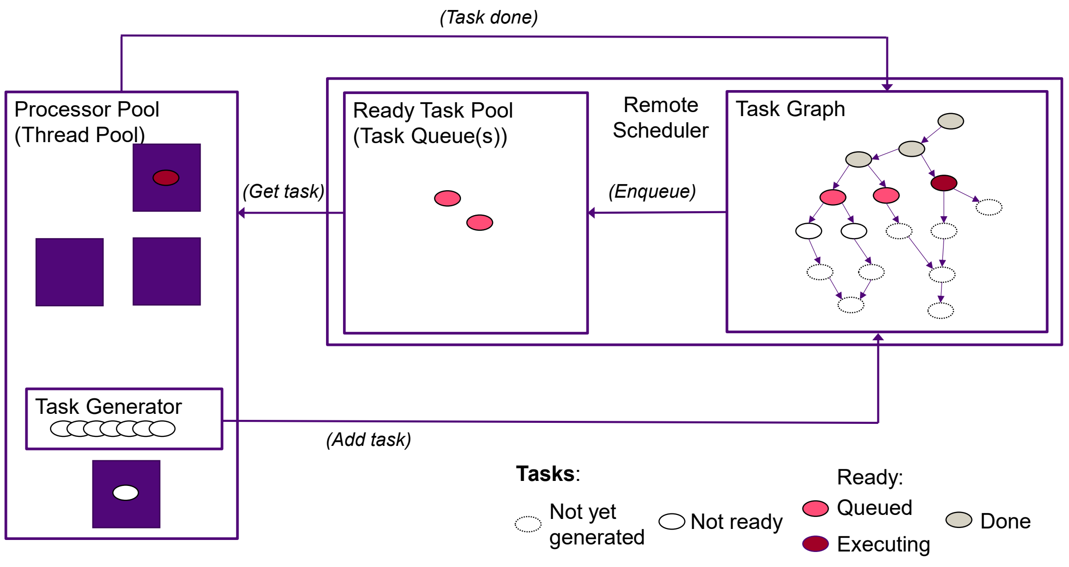


Figure 38 – Contemporaneous task generation and execution

- A house-keeping point to be dealt with in this arrangement is that a task is in danger of being allocated to a queue before the representation of that task is complete. This problem arises from the calls subsidiary to *qsched_addtask()* that are used to add details to a task; for example, the declaration of the parents of a task may not be complete but the child task may look apparently ready to be queued because its declared parents have all been executed. A simple fix would be to provide the task representation with a flag to mark its building complete, which the task building code only sets when that becomes true.

- A more substantial issue is that in QuickSched some post processing is applied to the whole task graph once all its tasks have been declared to it. This is the *prepare* stage. Since this stage calculates some values to use later in the determining of priority among ready tasks, that can be worked around. One possibility would be to ignore the priority when allocating the early tasks and use some arbitrary order, since priority is a matter of efficiency rather than correctness (Chapter 5). The usual efficiency priority order could then be calculated and used only once all the tasks were declared. (It could also be considered whether a calculation of the usual priority can be made on a partially built task graph is meaningful and useful.) Note that when the task graph is complete it has become a static task graph and so predictions about the longest path can be made again.
- In this method some thought should also be given as to how the task generation and task executing would be shared amongst the cores of the host processor. One might make special provision, by allotting a number of cores to task generation and leaving the remainder to task execution; once task building is complete the core(s) allotted to that would sensibly then be transferred to task execution. The choice will be informed by the relative amounts of time it takes to build the task graph and to execute it, and, within that, the need not to run out of tasks simply because they have not been created.

- One way to make this allocation of cores on the host processor to task generation would be to create task generation tasks. It is noted that the original QuickSched scheduler does have a dynamic task creation function that could be used. This however adds tasks to the current task graph and looks expensive, in that it recalculates the queues on each call to it. An efficiency could be to have task generation being limited to building tasks that are in the next “round” of computation, which round would have a separate task graph representation on the server.
- Alternatively, it may well also work to have task generation and task execution as separate Linux processes on the host and allow the operating system’s process scheduler to determine the dynamic allocation of cores between the two, rather than making special provision in the user code. However, this may not be compatible with the current approach of pinning threads in the computational process to specific cores of the host processor, and also it would cause data access issues in the cases where the tasks created are based on the data created by tasks in an earlier round of computation.
- If sending a message to create each task again proves inefficient in this case, the messaging overhead could again be reduced by compiling batches of tasks before sending a batch rather than a single task over in a message.

These methods of sharing of the host’s resources between task generation and task execution may, I think, be quite efficient, in that many parallel algorithms start with a serial set of initial tasks or just a small set of few parallel computational tasks that have no other peers, so only a few threads/cores will be occupied with calculation at that time, leaving the others free to calculate the rest of the task graph. (For example, in the case of the exemplary QR factorisation algorithm only one task may be executed initially, that on the top left tile, and the number of ready tasks grows slowly – see the width of the rows in Figure 9 – so it fits this pattern.)

When it comes to codes that operate in rounds of first task creation and then execution, this overlapping task creation and task execution spreads the task graph execution over a longer period (while probably not increasing the total time for both the creation and execution), which means that any data movement tasks moving data between nodes will be more spread out in time, which is an advantage if the speed of operation of the code is limited by the network traffic.

C. Another possibility is to have the server scheduler process build the task graph by itself, i.e., run the task generation code. This is shown in Figure 39, which is quite similar to Figure 1 except that the task generator now has a definite location. Running the current QR example in this way would be a bit slower in the case of the scheduler on a BlueField card compared to the original QuickSched because its processor and memory were slower than that of the host, at least where a complete static task graph is generated. In this location the task creation messages are eliminated. Further, this arrangement could be combined with the starting of the task graph's execution before it is complete from arrangement B and then there would be a high degree of overlap between task generation and execution.

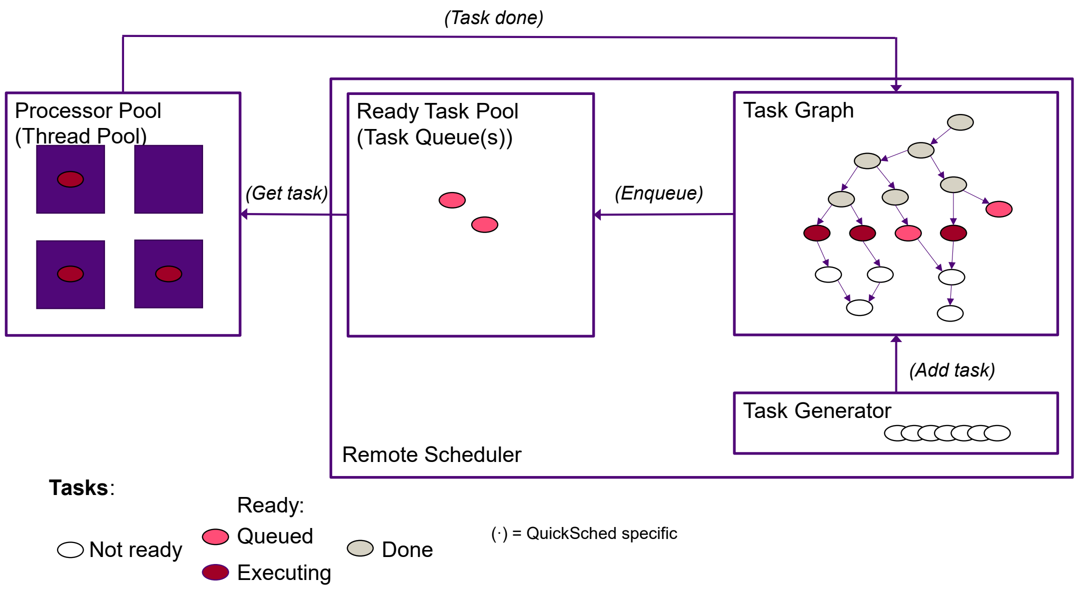


Figure 39 – Task generator at the remote scheduler location

This arrangement C would be a very pure form of the philosophy proposed here of putting the 'auxiliary' card in charge of the calculation in place of the host processor. Here the host x86-64 processor has now become just a processor of computational kernels and thus is now, in effect, a specialised ‘accelerator’ card.

The disadvantage of this arrangement is when the generation of tasks is based on data produced in an earlier round of computation; the task generator would have to obtain those results from the host, possibly by sending messages to ask for it – a better method for dealing with that that could be used here would be pre-emptively forwarding the needed data from the host to the scheduler. That is used in the next arrangement, D.

D. Another overlap between task generation and task execution which may be possible, is for the server process to generate a new task graph for a next round of calculation while the task graph for the previous round is being executed. The arrangement is shown in Figure 40.

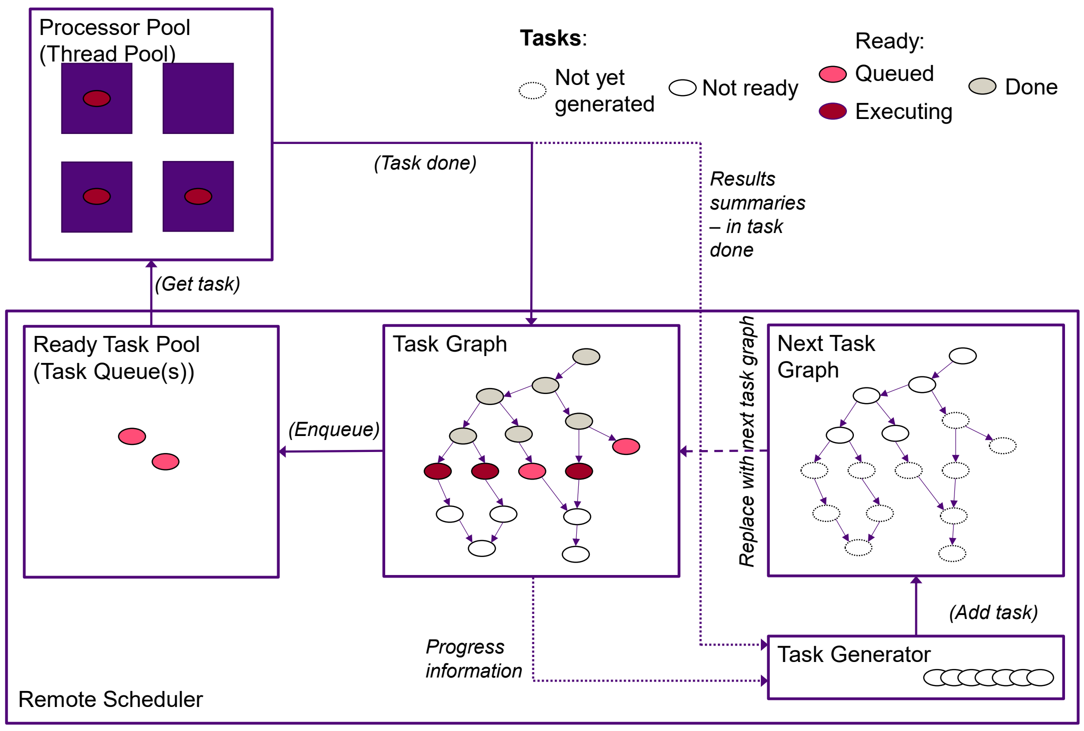


Figure 40 – Task generator at scheduler with overlapping build of next task graph

The actual opportunity here will depend on the application. If the task graph for the next round is not particularly new, so in the extreme example would just apply the existing task graph to new data, or the results of the previous round, then it may be possible to determine the handles for those new data items immediately and put them into the tasks of a copy of the existing task graph to make the new task graph.

As mentioned, in more complex cases, however, the form of the new task graph, or equally each new task thereof, may be determined by the results of the previous round, so building the task graph early would require careful analysis of when sufficient data becomes available to generate each part of the task graph. That point may well be a fixed point in the task graph, (but the location of that point could in some cases perhaps depend on some of the results as well). Now, when the task graph generation is located at the task scheduler on the server it will always have progress information immediately available to it and so will immediately know when new tasks may be generated. This awareness is marked as the dotted arrow *progress information* in Figure 40Figure 40.

The newly calculated data on the host, or some suitable data derived from it, needed to determine the details of the new tasks (and perhaps also needed to know when they may be created) does need to be transferred to the scheduler. That data may be tacked on to a *task done* message, as shown in Figure 40, marked with the dotted arrow labelled *results summaries*. This is an appropriate time in that the data is in a defined state and will not have gone out of date through the action of some other task. If the data on which new task generation is based requires some processing from the data that is the primary concern of the tasks, the balance between deriving it on the host or the remote scheduler should be examined.

The conclusion is that while the task graph build in Qsargm performed very poorly, there are in fact many opportunities to improve performance, while keeping the task graph and scheduler in the remote location, so retaining the advantages of a remote scheduler.

14.4. Further optimisations

Any opportunity to improve the latency of messaging, especially for the *gettask*, and *taskdone*, messages should provide benefits. If this it does not provide significant improvement at the optimal task size for a given problem size, lower latency should nonetheless improve the range in task size over which a remote scheduler is comparable in performance to the original one.

While a change of base messaging library could be an option, there are some different arrangements in the OpenMPI messaging that could be investigated:

- The messaging on the client side posts an MPI Send and then immediately waits for an answer to that particular message with MPI Recv. MPI Sendrecv is a single call designed to do both of those in one, more efficient, operation, so would be worthwhile trying.
- The present Qsargm sends a *task done* message and then immediately sends a *get task*. These could be combined into a single operation with a combined call sending the identity of the task completed and the reply to that giving the next task to be executed. This may save some messaging time, (but the *task done* message does not have a reply and so may have aspects of its transmission that are already in parallel with the *gettask* message).
- The rules of the scheduling could be changed so that the scheduler does not ask the current question of: now that the identity of the last task completed has been used to update the queues, what is, therefore, the highest priority task to execute? Instead, the scheduler should always have ready a new task to give to the thread and then process the task done message. This hides the time taken to process the task done message from the client, but it does have the scheduler answer instead the question of what the highest priority task was just before the completed task was known. This does change the priorities and so could well have effects on the task graph execution time, not necessarily for the better.

14.5. Complex scheduling applications

The scheduling algorithm of QuickSched, and equally that of this work's Qsargm, concentrates on ensuring as far as possible that the tasks on the longest path in the task graph are the priority.

As previously noted, in modern processors the cores are not all equal in processing power. This is not in accordance with the assumption in the scheduler used for calculating the priority, which is that each task has a single particular duration. Clearly the duration of a task will depend on the type of core on which it is allocated. Further, in QuickSched this information is processed once before a run, but the cores to which a task will be allocated are not known ahead of time. A central feature of QuickSched is that the tasks are dynamically allocated. So some more complex rules may be needed to set the priority and worse, these might have to be (re)calculated dynamically during the run. Freeing up a separate process with its own cores addresses this issue; these are more resources that would allow more complex scheduling rules to deal with the issue to be calculated.

Another issue that received attention in this thesis is the extent of the effect of cache misses. These may well not be the same for different types of core in the same microprocessor, so again this could be addressed in the scheduling rules, i.e. giving preference to tasks and cores for where there will be a bigger cache hit.

A related point is the compromise, noted by Chalk, of QuickSched only sorting the queues into a heap so when the queue is searched beyond its head, a less high priority task may be found than if it were fully sorted. The extra resources of the scheduler in a separate process could be used to rectify that.

Another potential type of complex schedule processing might be the dynamic splitting or aggregation of tasks. It has been noted that there is an optimal size for tasks when it comes to scheduling. In the optimisation experiment, this task size, or more correctly its data size, was the same for all tasks. However, it will be that at different times in the execution of a task graph there are different numbers of threads available and different kinds and numbers of task to fill them. So, for example if there are many threads available but only one large task, it may be that that task can be converted into a number that can be processed in parallel. (Whether that is possible will depend on the particular kernel of the task.) Such calculations in the scheduler would require resources.

Another example of complex processing that may well need more resources for the scheduler is the problem, noted in [83], of memory controllers in modern processors not having sufficient capacity to feed all the cores. The authors say, *"Algorithms have to avoid that all tasks access the main memory controllers concurrently and thus become bandwidth bound."* Potentially a scheduler could have rules to mitigate that also.

14.6. Cooperative compute node applications

The complex scheduling functions in the previous section relate to better rules for working through the task graph. Beyond that there are further functions that the scheduler could take on now it is provided with more resources, relating to cooperation with other compute nodes in the same cluster.

A first application relates to data transfer between compute nodes that are cooperating on the same computation under the control of task-based schedulers on each node. From time to time, data will be produced on one compute node that will need to be transferred to another compute node. This may be done, as is done, for example, in the Swift application, by providing in the task graph dedicated data movement tasks. These are placed in the task graph with dependencies so that the data they move is ready to be moved when the data movement task is allotted to a core. The kernel of this data movement task does not perform calculations but rather makes the OpenMPI calls needed to move the data. (The process is more complicated in that the data resource at the other end has to be ready to receive the data and a matching data receive task is provided at a suitable point in the task graph at the other end to do the receiving.)

It is now proposed that in Qsargm, particularly for the local BlueField geometry, that a data movement task is not allotted to one of the compute cores but is carried out, or rather commanded, by the BlueField. I was given access to a library in development at Nvidia, which allows such commands to be issued by a program running on the Arm processor of the BlueField, so the scheduler, to the RDMA system of the card to move data from the RAM of the host node to the RAM of another host node, or indeed in the opposite direction. The arrangement is as shown in the diagram of Figure 41.

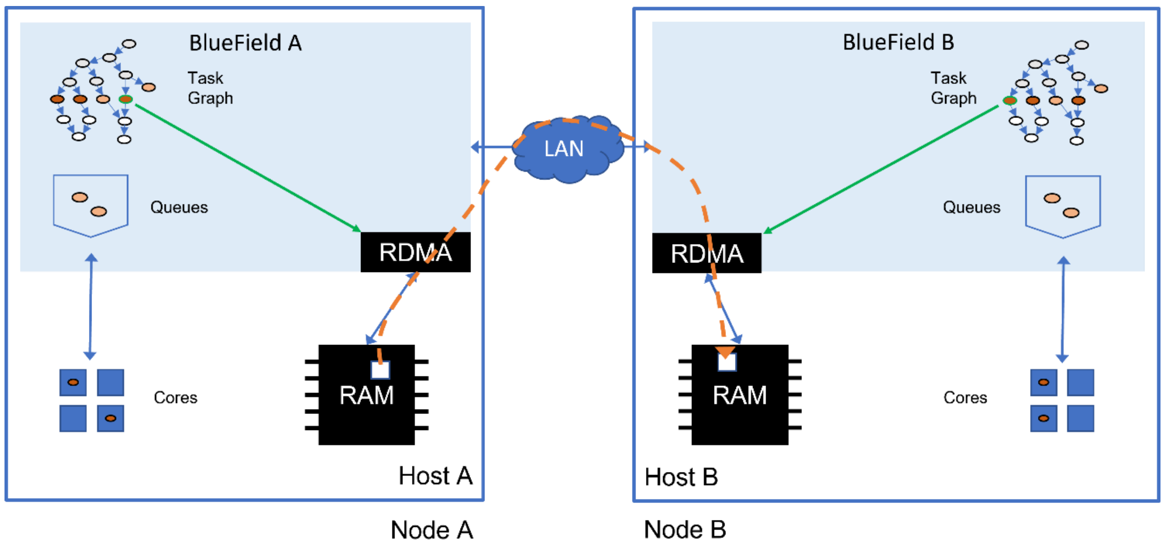


Figure 41 – Data movement tasks commanded from BlueField

Each node in the cluster has a respective BlueField card running a Qsargm scheduler, which in general is allocating compute tasks to the cores of its host computer. On reaching a data transfer task, the task oval with the green border, at the base of the green arrow, the schedulers command, through the library, their respective RDMA hardware on the Connect-X network adapter portion of the card to carry out the transfer, indicated by the green arrow. The RDMA hardware then transfers a specified memory block in the host RAM, the white block, to the RAM of the other host. Not shown here are any messages needed between the schedulers to time the transfer and which of the BlueFields actually makes the final instruction is glossed over. There is also the need to generate memory keys for the memory blocks. This is done on the relevant hosts and then the keys have to be transmitted to the BlueFields. However, the keys are long lasting and so may be generated ahead of time, for example when generating the tasks, so this preparation will not affect the task graph run time. There is no need to check that the data is indeed ready or that whether it is going to be written to by some other task; the dependencies and locks of the task graph have already taken care of those issues. The host cores have, however, not been involved at the time of the data movement task and therefore remain free to be allocated other computational tasks. So, in this arrangement not only has the task scheduling been offloaded to the BlueField but the network operations as well.

I did not implement this within Qsargm, but I did check that the proposed data transfers could be made with this library, so this test involved generating and distributing the memory keys, putting test data in the memory blocks, issuing the data transfer commands from a program running on the BlueField and then verifying that the test data was now in the other location.

Another proposal is for the schedulers on BlueField cards to organise “work stealing”. Work stealing is the process of redistributing tasks from a processor that is falling behind on its schedule of tasks to another that is ahead. One such work stealing arrangement, discussed in section 3.5, was proposed in reference [26]. Here the situation addressed was tasks on one node blocking tasks on another. Another need for work stealing between nodes arises from the estimates for when the work will be done being inaccurate and the initial division of the work between processors being made on the basis of that – the cause of the discrepancies in the times nodes finish their work seen in [16] and [20] and discussed at section 3.4. Whatever the cause or the scheduling rules used to mitigate the problem, because the scheduler has information about progress and because cooperation with other nodes is needed to organise the response locating such a scheduler on the BlueField is a good fit.

In the case of inaccurate work division between nodes, the scheduler is in a position to assess whether its node is ahead or behind schedule, or when that will occur in the near future. A node that is ahead of schedule will also need to find a partner node that is behind schedule. Further it will have to identify tasks that are ready (or predict when they will become so) and whose data can be shipped to the cooperating node, be processed there, and be shipped back ahead of the time when their data is needed by the next task (i.e. the point when that next task’s other dependencies are predicted to have been fulfilled). Now, this round trip time is dependent on the size of the data to be shipped and processed. This requirement on task size may in general be different to that found for optimising the task graph run time. These competing requirements could of course be taken into account in some overall optimisation, but dynamic splitting of tasks could also be considered – offloading a smaller task is more likely to be possible given the finite window before its results must be returned. All of this takes more processing power to organise, so locating the scheduler on the BlueField is again indicated.

One aspect of this is predicting the times of stages in the task graph execution being reached. This might perhaps be achieved through rules involving the weight of tasks as exist in QuickSched; another option might be simulation by the scheduler of its progress in near future from its current state of progress. Finally, it might be possible to make such predictions using machine learning techniques using the task graph as an input – it is notable that the roadmap for the BlueField card includes an on-board GPU. It would of course have to be checked whether such calculations could be made on the timescales needed by fine-grained scheduling.

A similar but milder intervention might be as follows. In the papers concerning the use of QuickSched on multiple compute nodes, tasks that had input data and/or results on different nodes were dealt with using data communication tasks. For each of these cases alternative task subgraphs could be generated in which the task is carried out on one or the other of the nodes respectively. The schedulers of the nodes cooperating with each other would cause one or other of these to actually happen at the appropriate time depending on which was ahead, and which was behind schedule. This holds out a promise of helping to even up progress, but I have not worked out the details of whether it would do so at the cost of overall delay, nor tried any examples to see if it is a significant enough effect to reduce typical discrepancies in finishing time.

Work stealing is of course remedial action. Better may be for nodes to cooperate on generating future tasks, contemporaneously with executing existing ones, in a continuous process and assigning them to nodes, and transferring the relevant data to them, according to rules designed to keep the workload balanced, without having to stop execution for task generation and mass data repartitioning periods. Such schemes will not be trivial to compute but a separate scheduler, for example, on the BlueField card (i) will have its own resources to compute that, (ii) will have its own access to the network to send control messages to its peers on other nodes, (iii) be able to control data movement between the host RAMs, so offloading that from the host, and (iv) and still be able to schedule efficiently fine grained tasks on its host processor. A separate scheduler located on a BlueField card is again indicated.

15. Conclusions

15.1. Key conclusions

The main conclusion of this thesis is that messaging latency does not always kill the performance of a remote scheduler. Latency remains important, however, and should be attended to. Optimisation of task size is needed to find acceptable speed of operation in the task graph execution phase. Indeed, the performance of the remote scheduler was very similar to that of the original. In the case studied, it was found to be more important to perform this optimisation than to choose between the locations tried for the scheduler.

All task-based algorithms have a free parameter of the size of the tasks, so this optimisation will be of general application to other computations having other forms of task graph. With a remote scheduler being a usable alternative to the original scheduler, the field of using task-based schedulers for more complex and augmented scheduling operations beyond the intentionally simple approach of the original scheduler is opened up.

The task graph build phase must be organised to avoid poor performance caused by the overhead of many small messages.

The motivation for even attempting such an unpromising arrangement was that the brief had been to find a way to control operations of a high-performance code from the network. In that context, the task-based scheduler, while in its original form does, on a time divided basis, mix up scheduling and task execution operations, it does separate those concerns, and therefore the scheduling, which is the control aspect, could be moved to the network. That was achieved

Thus, the prejudice that remote scheduling would increase latency and steal processing power from the actual objective of executing the tasks, has been overcome.

15.2. Task size interaction mechanisms

The task size optimisation for the run phase was found (or in some cases speculated) to operate through a highly complex set of interactions, including,

- Growth of the task graph with problem size:
    - Ability to supply the threads with ready tasks; probably related to the task graph width.
    - Consequent time for the scheduler to answer the question of which task to run next.
    - Tasks may become ready tasks simultaneously and cause delay through serialisation of the communication between scheduler and task processing threads.
- Time to run a task:
    - Relative size to messaging times for communicating a task as complete and the next task to start (including also to processing those events in the scheduler). Messaging times vary by location of the remote scheduler.
    - Scaling with task size - how the intrinsic number of operations to execute the kernel increases.
- Layout of the tasks on the threads and in time generated by the scheduler:
    - There was a theoretical discussion of some reasons for stability of this pattern and reasons for abrupt change and some evidence for both kinds of behaviour.
- Cache interactions
    - Self kernel interaction – how do the kernels perform if the task's data fits within the cache, or only partially does so?
    - Dilution of inter-task cache hits with increasing thread count.
    - Task layout pattern changes affecting inter task cache hits.
    - Scheduler processing fitting or not in the cache, e.g. sorts of the ready task queues.
    - Scheduler processing overwriting task data in the cache intended for an inter-task cache hit, in the case of the original Quick Sched.

    - Kernel processing overwriting scheduler data in the cache, in the case of the original QuickSched.
- Absolute size of the problem
    - The optimum task size does depend on the overall problem size, through at least the effect of the problem size on the size of the task graph.

15.3. Task graph building

The task graph and queue scheduler component location decided on, together with a desire for ease of development, led to an almost complete preservation of the interface functions of the scheduler library. The results of that clearly showed that such a structure was naïve for building the task graph.

With this problem having come to light, schemes for overcoming it were generated and were considered in detail. These solutions involved not only ways to batch up these messages to reduce their number, but ideas for how to implement overlapping task graph building and execution to hide messaging costs.

15.4. Argmessage and base transport

It was realised that the remote scheduler needed an RPC messaging system, and one was implemented appropriate to the timescale of the problem and to the intended HPC applications, so in the C language and using OpenMPI messaging as the base transport layer. Argmessage is unusual in that it provides for RPC calls that do not have a return type, as well as those that do. Argmessage was implemented successfully and would be applicable to remoting a wide range of C libraries.

OpenMPI messaging was used because, as well as being applicable to the application, it is well established and mature and therefore likely to be reliable and to contain optimisations making a wide range of messaging scenarios efficient. Certainly, OpenMPI caused no reliability problems relating to the messaging *per se*. Nonetheless, it would be worthwhile implementing Argmessage over any message transport more optimised for this particular application. A candidate library for that, in development at Nvidia, “SNAPI”, was tried but was not quicker. Improvements to MPI arrangements were suggested.

15.5. Qsargm

In order to implement this new location for the scheduler, the existing task-based scheduling library, QuickSched, was analysed into its components and examined for where it might be split between the compute threads and the remote process provided by Argmessage. It was concluded that a split where the remote scheduler had both its task graph and all its queue objects in the remote process would be the preferred implementation.

The QuickSched scheduler was reimplemented using that split and using a messaging loop provided between them provided by Argmessage. In particular, in general each function call of the original library was provided with its own RPC message. Details, like how to use tags to address the messages to individual compute threads, and how to prevent lock up of the scheduler, particularly near the end of the task graph, caused by the serialisation of messages between the compute threads and the scheduler that had been introduced, were worked out.

15.6. QR factorisation test example, and simulating other task graphs

The difficulty with the *locks* and *uses* specification in the QR factorisation example from the original scheduler’s repository required very detailed study to resolve. However, this was fortuitous as it led me to a much greater appreciation of the structure of the task graph and of the scheduling of tasks, for the QR factorisation and by extension for task graphs in general. In the end, fixing the problem made no difference to the results obtained, except some different values of thread count and tile count parameter *n* where affected by larger variation in task graph run time; this gives rise to an interest in future work exploring the pattern of the layout of tasks across the cores and in time to see if these patterns vary and if that is related to that phenomenon.

Future work should also include trying various other algorithms beyond the QR factorisation, chosen for diversity in their task graph structure, in particular ones having greater numbers of tasks that are more freely in parallel with each other would provide a contrast to the QR factorisation.

While such work could be carried out using Qsargm directly, such experiments do take a long time to run. The process could be much speeded up using a simulation. Noisy variation in task and message duration may lead to rearrangement of the pattern of the tasks across the cores against time, in algorithms that are susceptible to that. A model for the effects of cache missing may well be needed. Direct measures (e.g. cache miss counters) of the extent of that (e.g. how much of a tile is left in the cache even when there is not a miss) were not obtained, but simulation of a range of such extents, by adding to the task graph execution time when there is a miss, could be used to explore the effects.

Another benefit of such simulations would be the ease of visualisation of the front of progress, the line between completed and ready tasks. This would be interesting to see if there are patterns which hold up progress. For example, in plots (not shown) for larger *n* values of the task graph of the QR factorisation made by *graphviz* there are clear repeats in the occurrence of red, green and blue tasks every three rows of the plot and the yellow tasks are grouped in the plots rows with a high degree of fan out from a few tasks in the preceding row. I can imagine that the propagation of the front of completed tasks through the task graph may be affected by such more macro structures in the task graph. In turn that may inspire more advanced scheduling rules.

A visualisation could also be obtained of the layout of the tasks across the threads and in time. This is a 2D space and the placement of new ready tasks in it is, in QuickSched, governed by the cache miss rules competing with the rule to feed waiting threads. The former bind a new ready task to preferred sites in the existing arrangement of completed and executing tasks. This is reminiscent of the physical process of crystal growth, so one might expect to see regions of more and less order and perhaps grain boundaries between ordered regions. It would be interesting to see the effects of changing parameters like the number of tasks and threads on this structure and, of course, on the task graph execution time. Another parameter to vary would be the number of parents each task has in the graph and the mix of that for different tasks. In real cases that property of a task graph is fixed by a particular actual algorithm, but in a simulation this task graph can be made abstract, with the actual work being done by the kernels being ignored.

15.7. “From the network” – meaning

The goal set for this work was to perform the control of a code “from the network”. Clearly control from somewhere else has been achieved, but where that is requires some examination. The scheduler was placed in a variety of locations. On the same host was some distance away from the compute threads in terms of message latency and the local BlueField and the remote host and remote BlueField locations were further away again. However, the network as distinct entity did not appear – those are all just other locations on the edge of the network and are not that dissimilar in distance from any particular host. While these locations for the scheduler on remote machines seemed possible, no compelling reason emerged to use them – the better latency of the local locations would therefore favour them.

Assuming that one is operating in a regime where the messaging time of 1 μs is significant, distance to a remote controller would come into play at the larger data centre level, since 1 μs added by actual transmission in longer cables, rather than processing and propagation in network electronics is equivalent to about 300 m, so at that size of network the concept of control from the network might conceivably emerge.

Another aspect of the phrase would be about access to the network. The BlueField cards do not seem to have faster access to the network; both the host CPU and Arm subsystem on the BlueField are connected to the network adapter via the PCI switch on the card. What is important about the Arm subsystem is (i) that it has access to the network so that it may communicate directly with its peers on the network, (ii) that it is provided with its own computational resources to perform complex calculations to support cooperation with its peers, (iii) that it can of its own motion effect transfer of payload computational data between the RAMs of hosts, and (iv) it can direct operations (execution of tasks) on its own host. Taking all those into account the BlueField has become the centre of control of operations of the local machine, both for internal operations and cooperation with other hosts. It remains, however, at the edge of the network and a thing of the local machine.

What has happened, however, is that the host CPU, although it still has the same physical level of, and quality of, connectivity to the network, has conceptually moved a step away; the BlueField is its interface to its peers in all matters as it has become a dedicated computational engine. The model of control has remained local and cooperative between machines, and no central organisation has emerged, which at first sight the phrase “from the network” might imply.

16. References

[1] Nvidia, “NVIDIA Completes Acquisition of Mellanox, Creating Major Force Driving Next-Gen Data Centers,” 27 April 2020. [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-completes-acquisition-of-mellanox-creating-major-force-driving-next-gen-data-centers#:~:text=NVIDIA%20today%20announced%20the%20completion,performance%20and%20dat.

[2] TOP500 Project, “November 2021,” November 2021. [Online]. Available: https://www.top500.org/lists/top500/2021/11/.

[3] Nvidia, “Datasheet: Nvidia Bluefield-2 DPU,” [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf. [Accessed 28 April 2022].

[4] Nvidia, “What is RDMA?,” 1 January 2020. [Online]. Available: https://www.youtube.com/watch?v=lu78_C-9jvA.

[5] Nvidia, “NVIDIA® Mellanox® Scalable Hierarchical Aggregation and Reduction Protocol (SHARP),” [Online]. Available: https://docs.nvidia.com/networking/display/sharpv214.

[6] The Open MPI Project, “Open MPI: Open Source High Performance Computing,” [Online]. Available: https://www.open-mpi.org/.

[7] Swift Project, “Swift - SPH WIth Inter-dependent Fine-grained Tasking,” [Online]. Available: https://swift.dur.ac.uk/about.html. [Accessed 28 April 2022].

[8] SchedMD, “sbatch command reference page, version 21.08,” [Online]. Available: https://slurm.schedmd.com/sbatch.html (see “--dependency” option). [Accessed 28 April 2022].

[9] S. Shuler, N. Bloch, O. Hayut, R. Graham and A. Shahar.United States Patent 2016/0072906 (Application), 2016.

[10] J. Pinkerton, “The case for RDMA,” 29 May 2002. [Online]. Available: http://www.rdmaconsortium.org/home/The_Case_for_RDMA020531.pdf. [Accessed 14 March 2023].

[11] Nvidia Corporation, “ConnectX-6 Datasheet,” [Online]. Available: https://nvdam.widen.net/s/5j7xtzqfxd/connectx-6-infiniband-datasheet-1987500-r2. [Accessed 14 March 2023].

[12] MPI Forum, “Collective Communication - Introduction and Overview,” [Online]. Available: https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node64.html. [Accessed 14 March 2023].

[13] University of Tennessee, “MPI: A Message-Passing Interface Standard (Version 1.1),” 12 June 1995. [Online]. Available: https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node64.html#Node64. [Accessed 02 January 2024].

[14] C. Mellor, “Mellanox SoCs it to NVMe over Fabrics with BlueField platform - JBOF made easier,” 9 August 2017. [Online]. Available: https://www.theregister.com/2017/08/09/mellanox_bluefield_soc/.

[15] P. Gonnet, A. Chalk and M. Schaller, “Source Forge: QuickSched,” 2014 December 2014. [Online]. Available: https://sourceforge.net/projects/quicksched/.

[16] A. B. G. Chalk, “Task-Based Parallelism for General Purpose Graphics Processing Units and Hybrid Shared-Distributed Memory Systems,,” 5 September 2017. [Online]. Available: http://etheses.dur.ac.uk/12292/.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, “Chapter 6 Heapsort,” in *Introduction to Algorithms, Fourth Edition*, Cambridge MA, MIT Press, 2022.

[18] Regents of the University of Minnesota, “METIS (Github),” 1998-2020. [Online]. Available: https://github.com/KarypisLab/METIS. [Accessed 2 January 2024].

[19] P. Gonnet, M. Schaller, T. Theuns and A. B. G. Chalk, “SWIFT: fsat algorithms for multi-resolution SPH on multi-core architectures,” 15 September 2013. [Online]. Available: arXiv:1309.3783v1.

[20] P. Gonnet, “Efficient and Scalable Algorithms for Smoothed Particle Hydrodynamics on Hybrid Shared/Dsitributed-Memory Archtectures,” *SIAM Journal Of Scientific Computing,* vol. 37, no. 1, pp. C95 - C121, 2015.

[21] T. Theuns, A. Chalk, M. Schaller and P. Gonnet, “Swift: task-based hydrodynamics and gravity for cosmological simulations,” 1 August 2015. [Online]. Available: arXiv:1508.00115v1. [Accessed 17 March 2023].

[22] P. Gonnet, A. B. G. Chalk and M. Schaller, “Quick Sched: Task-based parallelism with dependencies and conflicts,” 20 January 2016. [Online]. Available: arXiv1601.05384v1.

[23] M. Schaller, P. Gonnet, A. B. G. Chalk and P. W. Draper, “SWIFT: Using Task-Based Parallelism, Fully Asynchronous Communication, and Graph Partition-Based Domain Decomposition for Strong Scaling on more than 100,000 Cores,” in *Proceedings of the Platform for Advanced Scientific Computing Conference 2016 (PASC)*, Lausanne, 2016.

[24] J. S. Willis, M. Schaller, P. Gonnet, R. G. Bower and P. W. Draper, “An Efficient SIMD implementation of Pseudo-Verlet Lists for Neighbour Interactions in Particle Based Codes,” 17 April 2018. [Online]. Available: arXiv:1804.06231v1. [Accessed 18 March 2023].

[25] J. Borrow, R. G. Bower, P. W. Draper, P. Gonnet and M. Schaller, “SWIFT: Maintaining wek-scalabilty with a dynamic range of 10^4 in time-

step size to harness extreme adaptivity," in *Proceedings of the 13th SPHERIC International Workshop,*, Galway, Ireland, 2018.

[26] P. Samfass, T. Weinzierl, D. E. Charrier and M. Bader, "Leigthweight Task Offloading Exploting MPI Wait Times for Parallel Adaptive Mesh Refinement," 14 April 2020. [Online]. Available: arXiv:1909.06096v2. [Accessed 18 March 2023].

[27] P. Samfass, T. Weinzierl, B. Hazlewood and M. Bader, "TeaMPI - Replication-Based Resilience Without (Perfomance Pain)," in *ISC High Performance*, Online, 2020.

[28] Excalibur Reseach Programme, "Excalibur Research Programme Home Page," [Online]. Available: https://excalibur.ac.uk/. [Accessed 15 March 2023].

[29] Excalibur Research Programme, "Exposing Parallelison: Task Parallelism," [Online]. Available: https://excalibur.ac.uk/projects/exposing-parallelism-task-parallelism/. [Accessed 15 March 2023].

[30] A. Tuft and A. Chalk, "Otter project at GitHub," [Online]. Available: https://excalibur.ac.uk/projects/exposing-parallelism-task-parallelism/. [Accessed 2023 March 2023].

[31] UCF Consortium, "Unfied Communication X," [Online]. Available: https://openucx.org/. [Accessed 4 May 2022].

[32] opensource.com, "CFS: Completely fair process scheduling in Linux," 5 February 2019. [Online]. Available: https://opensource.com/article/19/2/fair-scheduling-linux.

[33] H. Hoogendoorp, "Extraction and visual exploration of call graphs for large software systems," 15 February 2018. [Online]. Available: https://fse.studenttheses.ub.rug.nl/9153/.

[34] J. L. Hennessy and D. A. Patterson, Computer Architecture A Quantitative Approach, Sixth Edition, Cambridge, MA: Morgan Kaufmann, 2019.

[35] F. Burton and M. Sleep, “Executing functional programs on a virtual tree of processors,” in *Proceedings of the 1981 conference on Functional programming languages and computer architecture;*, New York City, Association for Computer Machinery, 1981, pp. 187-194.

[36] R. J. Halstead, “Implementation of multilisp: Lisp on a multiprocessor,” in *Proceedings of the 1984 ACM Symposium on lisp and functional programming*, New York City, Association for Computer Machinery, 1984, pp. 9-17.

[37] R. Rudolph, M. Slivkin-Allaouf and E. Upfal, “A simple load balancing scheme for task allocation in parallel machines,” in *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, New York City, Association for Computer Machinery, 1991, pp. 237-245.

[38] OpenMPI Project, “FAQ: What kinds of systems / networks / run-time environments does Open MPI support?,” [Online]. Available: https://www.open-mpi.org/faq/?category=supported-systems. [Accessed 4 May 2022].

[39] Oracle Corporation, “ONC+ Developer's Guide,” [Online]. Available: https://docs.oracle.com/cd/E19683-01/816-1435/index.html. [Accessed 2022 May 4].

[40] Oracle Corporation, “ONC+ Developer Guide, What IS TI-RPC,” [Online]. Available: https://docs.oracle.com/cd/E19683-01/816-1435/rpcintro-fig-1/index.html. [Accessed 4 May 2022].

[41] P. Krzyzanowski, “Remote Procedure Calls &Web Services,” 18 September 2023. [Online]. Available: https://people.cs.rutgers.edu/~pxk/417/notes/rpc_cases.html. [Accessed 3 January 2024].

[42] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns : elements of reusable object-oriented software, Boston: Addison-Wesley, 1995.

[43] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard Version 4.0,” 9 June 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[44] Open MPI Project, “MPI_Init_thread man page,” [Online]. Available: https://www.open-mpi.org/doc/v4.1/man3/MPI_Init_thread.3.php. [Accessed 4 May 2022].

[45] G. R. Andrews, “Glossary,” in *Concurrent programming – Principles and Practice*, Redwood City, The Benjamin/Cummings Publishing Company, Inc., 1991, p. 604.

[46] Wolfram, “Wolfram Langauage & System: Documenation Center QRDecomposition,” [Online]. Available: https://reference.wolfram.com/language/ref/QRDecomposition.html. [Accessed 5 June 2022].

[47] Eigen Project, “Eigen: Linear algebra and decompositions,” [Online]. Available: https://eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html. [Accessed 5 June 2022].

[48] S. Blackford, “LAPACK User Guide: Orthogonal Factorizations and Linear Least Squares Problems,” 1 October 1999. [Online]. Available: http://netlib.org/lapack/lug/node39.html. [Accessed 5 June 2022].

[49] A. Buttari, J. Langou, J. Kurzak and J. Dongarra, “Parallel tiled QR factorization for multicore architectures,” *Concurrency and Computation: Practice and Experience,* vol. 20, pp. 1573-1590, 2008.

[50] P. Gonnet, A. B. G. Chalk and M. Schaller, “QuickSched: Task-based parallelism with dependencies and conflicts,” 20 January 2016. [Online]. Available: https://arxiv.org/pdf/1601.05384.pdf.

[51] S. Freeman and N. Pryce, Growing object-oriented software, guided by tests, Boston: Pearson Education, Inc., 2010.

[52] Network-Based Computing Laboroatory, The Ohio State University, “MVAPICH: MPI over InfinDand,Omni-Path, Ethernet/iWARP, and ROCE,” [Online]. Available: https://mvapich.cse.ohio-state.edu/benchmarks/. [Accessed 5 May 2022].

[53] J. Duato, S. Yalamanchili and L. Ni, “Chapter 1 - Introduction,” in *Interconnection Networks - An Engineering Approach*, San Francisco, Morgan Kaufmann, 2003, pp. 1-41.

[54] Y. Chen, Z. Liu, S. R. Sandoghchi, G. T. Jasion, T. D. Bradley, E. N. Fokoua, J. R. Hayes, N. V. Wheeler, D. R. Gray, B. Mangan, R. Slav´ık, F. Poletti, M. Petrovich and D. Richardson, “Multi-kilometer Long, Longitudinally Uniform Hollow Core Photonic Bandgap Fibers for Broadband Low Latency Data Transmission,” *Juornal of Lightwave yechnology,* vol. 34, no. 1, pp. 104-113, 2016.

[55] OpenMPI Project, “ompi / ompi / mpi / c / wtime.c on gitHub,” [Online]. Available: https://github.com/open-mpi/ompi/blob/eb87378cd8c55c08fde666c03a04aa2811da395a/ompi/mpi/c/wtime.c. [Accessed 1 July 2022].

[56] OpenMPI Project, “ompi / opal / util / clock_gettime.h on GitHub,” [Online]. Available: https://github.com/open-mpi/ompi/blob/eb87378cd8c55c08fde666c03a04aa2811da395a/opal/util/clock_gettime.h. [Accessed 1 July 2022].

[57] GNU project, “clock_gettime(3) - Linux man page,” [Online]. Available: https://linux.die.net/man/3/clock_gettime. [Accessed 1 July 2022].

[58] GNU Project, “gettimeofday(2) - Linux man page,” [Online]. Available: https://linux.die.net/man/2/gettimeofday. [Accessed 1 July 2022].

[59] Intel Corporation, “How 12th Gen’s Intel® Core™ Hybrid Technology Works,” [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/how-hybrid-design-works.html. [Accessed 14 June 2022].

[60] Intel Corporation, “Get to Know Intel® Turbo Boost Max Technology 3.0,” [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-max-technology.html. [Accessed 15 June 2022].

[61] Arm, “big.LITTLE Technology,” [Online]. Available: https://developer.arm.com/documentation/den0024/a/big-LITTLE-Technology. [Accessed 15 June 2022].

[62] Nvidia Corporation, “VPI Verbs API,” [Online]. Available: https://docs.nvidia.com/networking/pages/viewpage.action?pageId=34256565. [Accessed 30 June 2022].

[63] HPC AI Advisory Council, “HPC AI Advisory Council - homepage,” [Online]. Available: https://www.hpcadvisorycouncil.com/. [Accessed 21 June 2022].

[64] HPC AI Advisory Council, “HPC AI Advisory Council - High-Perfomance Centre Overview,” [Online]. Available: https://www.hpcadvisorycouncil.com/cluster_center.php. [Accessed 27 Janauary 2022].

[65] Mellanox (now Nvidia Networks), “BlueField Infiniband Specifications,” [Online]. Available: https://docs.mellanox.com/display/BFVPIDPU/Specifications . [Accessed 27 January 2022].

[66] Mellanox, “ConnectX-6DX/Bluefield-2 IPsec HW Full Offload Configuration Guide,” [Online]. Available: https://support.mellanox.com/s/article/ConnectX-6DX-Bluefield-2-IPsec-HW-Full-Offload-Configuration-Guide. [Accessed 21 June 2022].

[67] Nvidia, “NVIDIA BLUEFIELD DPU FAMILY SOFTWARE 3.6.0.11699 DOCUMENTATION - Supported Platforms and Interoperability,” [Online]. Available: https://docs.nvidia.com/networking/display/BlueFieldSWv36011699/Supported+Platforms+and+Interoperability. [Accessed 21 June 2022].

[68] Mellanox, “BlueField-2 Infinband/VPU DPU User Guide,” [Online]. Available: https://docs.nvidia.com/networking/display/BlueField2DPUVPI/Hardware+Installation. [Accessed 27 January 2022].

[69] Nvidia Corporation, “BlueField DPU SW Manual - Modes of Operation,” [Online]. Available: https://docs.nvidia.com/networking/display/BlueFieldSWv35111601/Modes+of+Operation. [Accessed 14 July 2022].

[70] Herausgegeben im Auftrag des Rektors der Rheinisch-Westfälischen Technischen Hochschule (RWTH) Aachen, “HPC Wiki - Scaling Tests,” [Online]. Available: https://hpc-wiki.info/hpc/Scaling_tests. [Accessed 3 January 2024].

[71] GNU Project, “The C Preprocessor System-specific Predefined Macros,” [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/System-specific-Predefined-Macros.html#System-specific-Predefined-Macros. [Accessed 1 July 2022].

[72] Intel Corporation, “Intel® 64 and IA-32 Architectures Software Developer Manuals,” [Online]. Available:

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. [Accessed 1 July 2022].

[73] Wikimedia Foundation, “Wikipedia - Time Stamp Counter,” [Online]. Available: https://en.wikipedia.org/wiki/Time_Stamp_Counter. [Accessed 1 July 2022].

[74] D. Groen, A. Bhati, J. Suter, J. Hetherington, S. Zasada and P. Coveney, “ FabSim: facilitating computational research through automation on large-scale and distributed e-infrastructures,” *Computer Physics Communications,* no. 207, pp. 375-385, 2016.

[75] D. A. H. S. D. Groen, W. Edeling, E. Raffin, Y. Xue, K. Bronik, N. Monnier and P. Coveney, “FabSim3: An automation toolkit for verified simulations using high performance computing. Computer Physics Communications,” *Computer Physics Communications,* no. 283, p. 108596, 2023.

[76] D. e. a. Groen, “FabSim3 - An automation toolkit for complex simulation tasks - Readthedocs,” [Online]. Available: https://fabsim3.readthedocs.io/en/latest/. [Accessed 5 January 2024].

[77] G. M. Amdahl, in *AFIPS Conference Proceedings (30): 483–485*, 1967.

[78] S. S. Council, “scipy.optimize.curve_fit,” 2022 October 19. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html. [Accessed 2022 October 23].

[79] Nvidia Corporation, “Nvidia RDMA Core (MLNX_OFED v5.0-1.0.0.0) - User-Mode Memory Registration (UMR),” [Online]. Available: https://docs.nvidia.com/networking/pages/viewpage.action?pageId=25138119. [Accessed 19 February 2023].

[80] Intel Corporation, “Intel® Xeon® Processor E5-2697A v4,” 2022. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/91768/intel-xeon-

processor-e52697a-v4-40m-cache-2-60-ghz.html. [Accessed 09 Decemeber 2022].

[81] “CPU World - Intel Xeon E5-2680 v2,” [Online]. Available: https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2680%20v2.html. [Accessed 12 February 2023].

[82] “CPU World - Intel Xeon E5-2697A v4,” [Online]. Available: https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2697A%20v4.html. [Accessed 12 February 2023].

[83] D. E. Charrier, B. Hazelwood and T. Weinzierl, “Eclave Tasking for DG Methods on Dynamically Adaptive Messages,” 24 February 2020. [Online]. Available: arXIv:1806.07984V3. [Accessed 18 March 2023].

[84] “man page for bash,” [Online]. Available: http://www.manpagez.com/man/1/bash/ at "INVOCATION". [Accessed 4 May 2022].

[85] Open Grid Scheduler Project, “qsub man page,” [Online]. Available: http://gridscheduler.sourceforge.net/htmlman/htmlman1/qsub.html at option "-v". [Accessed 2020 May 4].

[86] SchedMD, “sbatch man page,” [Online]. Available: https://slurm.schedmd.com/sbatch.html at option "--export". [Accessed 4 May 2022].

[87] University College London, “UCL_RITS rcps-buildscripts,” [Online]. Available: https://github.com/UCL-RITS/rcps-buildscripts. [Accessed 4 May 2022].

[88] Spack project, “Spack,” [Online]. Available: 2022. [Accessed 4 May 2022].

[89] Nvidia, “Mellanox HPC-X Tollkit,” [Online]. Available: https://support.mellanox.com/s/productdetails/a2v50000000XcPdAAK/hpcx-toolkit. [Accessed 4 May 2022].

[90] The GNU project, “Bash Manual - Bash Satrtup Files,” [Online]. Available: https://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html. [Accessed 27 January 2022].

[91] Oracle Corporation, “Sun HPC ClusterTools 8.1 Software User’s Guide - About the mpirun Command,” [Online]. Available: https://docs.oracle.com/cd/E19923-01/820-6793-10/ExecutingPrograms.html#50524166_62903. [Accessed 26 June 2022].

[92] University of Tennessee; University of California, Berkeley; University of Colorado Denver; and NAG Ltd., “LAPACK - Linear Algebra PACKage,” [Online]. Available: http://www.netlib.org/lapack/. [Accessed 28 June 2022].

[93] The NumPy Project, “Numpy - The fundamental package for scientific computing with Python,” [Online]. Available: https://numpy.org/. [Accessed 30 June 2022].

[94] Wikipedia, “Object (computer science),” [Online]. Available: https://en.wikipedia.org/wiki/Object_(computer_science). [Accessed 28 April 2022].

17. Appendices

Appendix A – Operation of Argmessage – further details

17.1. Argmessage – initialisation (and finalisation)

There are several aspects to starting an application that uses an Argmessage library:

- launching the processes that will execute the client and the server
- initialising the underlying messaging library used by Argmessage
- initialising the Argmessage client and server
- initialising the user library supported by Argmessage on both the client and server
- initialising the user application itself, on the client

17.2. Launching processes and initialising the software environment

In this project the client and server are initialised together and run during the same period, which is usually the lifetime of the client application. It could be envisaged to have a long running server that clients connect to and disconnect from as necessary, but that is outside the scope of this project. Such an arrangement would face additional security and reliability issues.

In the main application of Argmessage in this project the underlying messaging library is OpenMPI and some of the initialisation steps make use of OpenMPI facilities. These therefore would have to be adapted if another underlying messaging library was used.

17.3. Launching with mpirun

For the main MPI case, launching the client and server processes is preferably carried out using an *mpirun* command that comes with MPI libraries, which is, of course, the standard approach for MPI programs. Ordinary users of MPI codes will be familiar with the *mpirun* command and its use to launch multiple copies of the code spread across multiple compute nodes. It is also possible to use the *mpirun* command to launch different codes on different nodes, and that is used here to launch the Argmessage client and the server. This is a less common use, so an example is given in Table 25.

Note, however the details of *mpirun* are implementation specific, which means not only the particular MPI code project, of which there are several, but the details of the installation on any cluster. The points made in this Appendix relating to MPI relate in particular to OpenMPI and, further, may depend on the particular cluster I was working on. The reader will therefore have to apply the points made to their particular setup.

The *mpirun* command first launches a part of the communications library (the ORTE daemon in the case of OpenMPI) on each node referred to by the *mpirun* command, and then starts the specified programs, which will be the client and server of this project. (The client and server programs will initialise the messaging library further by each calling *MPI_Init()* from within their own program.)

So, an example *mpirun* command, for one client-server pair, is:

```
mpirun -np 1 –host client_hostname client_application : -np 1
                --host server_hostname server_application
```

Table 25 – Argmessage application launch mpirun example

The colon character separates the parameters used for the different programs being launched. So, this launches one process ("-np 1") of the application called "client_application" and one process of the "server application" respectively on the machines called "client_hostname" and "server_hostname".

The *mpirun* command may be issued in various contexts. One is that it may be included in the batch script of a scheduled job of a cluster scheduler. This script is of course executed on the one of the nodes allocated to the batch job by the cluster scheduler selected to be the head node. This is how it is used in the experiments of this project. A requirement on the batch script is of course that it obtains from the cluster scheduler, or has provided to it otherwise, access to the machines referenced in the *mpirun* command.

17.4. Use on heterogeneous architectures

For the common use of *mpirun* of running multiple copies of the same code, the user does not usually have to specify the hosts, which are instead communicated to *mpirun* using a special file listing the hosts (sometimes called a "machine file") or using an environment variable. Whether or not a machine file is employed, in Argmessage the user will have to make some arrangements to sort out which machine will run which of the client and server. For example, in the particular case of using a BlueField card for the server, one way will be for the job batch script to work out which of the available nodes is the BlueField card and which is the host and launch the server and client on those respectively. Helpfully most cluster system administrators give nodes of similar specification the same name differentiated by a serial number, so the BlueField cards will have different names from the hosts and so the script can use those to differentiate.

Implicit in this is that the cluster scheduler has been set up to have the server (e.g., a BlueField card) as a separate, allocatable node for cluster jobs, so that a batch job is able to request it amongst its list of required nodes. In order to be able to be set up as a separate node in the scheduler, a BlueField card should be set up in its *separated host mode*.

The user will of course have to add further options to the *mpirun* command to optimise, or simply make the communications work, on the cluster system, for example covering things such as selecting the network cards to use and options that determine which components/options of the underlying messaging library are used. For example, for a host to communicate with its own BlueField card, one should preferably set MPI to use the BlueField card as its network card, rather than some other network card in the host computer. The latter might not fail but messages between the host and its BlueField may well take an indirect route.

OpenMPI is designed to allow programs running on different architectures to communicate with each other. The different binaries of these programs needed for the different architectures can be specified using the colon syntax of mpirun shown in Table 25.

The client and server programs *per se*, and their libraries including those of OpenMPI, can require further painstaking set up. In Linux systems, where to find the libraries is determined by the environment, in particular the environment variable LD_LIBRARY_PATH. This can be set in various ways, for example by direct assignment or by using *environment modules* provided by the *module* command, or possibly also by the *Spack* command. It can also be set or modified at various times; these may include:

- start-up scripts called when the cluster scheduler contacts the head node of the batch job,
- commands inside the batch script itself, and
- start-up scripts called when the client and server are launched (launching through *mpirun* causes this).

A further complication is that, potentially, different start-up scripts can be called depending on just how any login to establish these processes is made [84] – for example, value of shebang of the batch script and any scripts invoked by *mpirun* will be important. Another complication is that the user's environment at the time of submitting the batch job to the scheduler may be considered. (I was unfortunate enough to have to learn that one popular cluster scheduling system, SGE, has an **opt-in** system for doing this [85], while another, SLURM, has an **opt out** [86].)

The user's strategy for getting all this right is to either to diagnose what is happening to the environment, for example by logging relevant items to a file in the various scripts that may come into play, and go with it, or to override the LD_LIBRARY_PATH environment variable at the last moment possible, which may involve using *mpirun* not to call the client and server programs directly but to call scripts that do launch them but only after first setting LD_LIBRARY_PATH. In the latter case the user should ensure not to remove unintentionally something set by the log in scripts but actually needed – this especially applies to the scripts controlled by the system administrator rather than the user. The best strategy may depend on the quality and breadth of the setup provided by the system administrators. The experiments section of this thesis records how this was all achieved in the particular case of the experimental runs.

17.5. Initialisation within the client and server applications

Once the correct client and server applications have been launched with the correct environment, the remaining aspects of the initialisation are dealt with within the code of the Argmessage client and server applications.

The underlying messaging library on which Argmessage depends will need to be initialised. In the case of OpenMPI explicit action is required in the form of calling the function *MPI_Init()*. This is included both in the Argmessage server code and client code. *MPI_Init()* contacts the ORTE daemons mentioned earlier and establishes the connections between the processes and their *rank* numbers, which are their abstract addresses between which MPI messages are sent.

To start Argmessage on the client and server they respectively call *argmessage_proxygetobject(), argmessage_serverenginegetobject()*. These in turn both call *MPI_Init()*.

17.6. Using Argmessage in a client that has MPI peers

It is noted that there may be an inconvenience in the case where the client application is itself an OpenMPI application with peers on other host nodes of a cluster, with which it would want to communicate using OpenMPI. In that case, a refactoring of moving *MPI_Init()* out of the Argmessage library and into the client application *per se* will help. Here, once the client application has called *MPI_Init()*, it could then sort out first which *rank* numbers belong to its counterpart Argmessage server and which to its peer client applications. The client will only have a single *rank* number, so to separate the communications of the client application with its peers from those with Argmessage server respective *MPI communicators* should be defined for these two purposes. All OpenMPI calls in the client application and in the Argmessage library will then have to be amended to refer to the correct one of those *communicators*, rather than for example the general default *communicator*, *MPI_COMM_WORLD*.

The Argmessage function *argmessage_serverrank()* negotiates which *rank* plays the role of the Argmessage server and its logic might need to be reviewed in such a refactorisation; as it stands it is expected that one rank will declare itself to be the server, so if *communicators* were to be defined these should include just one rank that has been launched to execute the server application.

17.7. Further initialisation of Argmessage

An initialisation step on the server, which could either precede or follow the OpenMPI initialisation, is the initialisation of the *proxy* and *adapter* objects' function tables.

These need to be populated with entries for both Argmessage's internal housekeeping functions and the remote functions of whichever application the server is here to support. Finally, the server and client are then provided with further items of its own initial state and initial state for the library that it supports.

17.8. User library initialisation

With the Argmessage library initialised, the user application of course needs to complete any other initialisations it needs. The *user library* programmer should provide initialisation functions, if required, for both the client-side and the server-side. On the server-side there may be a need for either or both of automatic initialisation and client-requested initialisation.

The former should be done once the *adapter objects* have been created, since the state created by this initialisation should usually be attached to the relevant adapter object for retrieval when a *remote user function* is called. For this purpose, an *adapter* is already provided with a pointer to a state object, which can be set to the initial state of the library so that the state may be retrieved, if needed, by subsequent *remote library function* calls. Indeed, the first parameter of all *remote library functions* is the identity of the *adapter*, which allows the *remote library function* to access the pointer and hence access the library's state.

Client-requested initialisation is for where the initialisation of the *user library* needs some value from the client side, which is sent in an Argmessage message calling this initialisation function. Because client-requested initialisation uses an Argmessage message and a remote library function, it can only be called after Argmessage has been placed in the run mode.

17.9. Argmessage's run mode

With the initialisation complete (with the possible exception of client-requested server-side initialisation of the *user library*), the Argmessage server is set to "run" using the function, *argmmessage_serverenginerun()*, which means it will now accept messages from the client, which it will interpret as RPC requests. The client does have a corresponding "run" function, *argmessage_proxyrun()*, which is called at this point, but it only initialises a precautionary maximum for the number of messages it will process, guarding against an application that gets stuck in a loop; the *proxy* on the client is only actually in execution during the run mode when public interface functions are called by the client application. On the other hand, the server, when it has nothing to do, is waiting to receive more RPC messages.

The details of how Argmessage messages operate to achieve RPC calls has been described in section 6.4, with reference to Figure 6.

17.10. Finalisation

Once everything has been initialised, the client application can then of course make calls to the *user library* via its public interface to achieve whatever is desired from it. Finally, after the desired processing using the library is completed, in a finalisation step, the client application should both make a call to the *user library* to finalise that, and then finalise the Argmessage library. For the former the *user library* programmer should provide a special function. The latter is achieved with calls to *argmessage_proxykillserverrequest()* and as many calls to *argmessage_proxykilladapter()* as the number of adapters that the client application was using. The call to *argmessage_proxykillserverrequest()* is noted by the server and is acted on once all the adapters have been deactivated with *argmessage_proxykilladapter()*. This avoids unexpected termination of the service for the other clients.

While it is desirable in some cases that using Argmessage appears transparent to the application program, i.e., it just calls functions of the public interface, Argmessage does requires initialisation and finalisation which are explicit from that perspective; this is a modest complication for the application programmer. Nonetheless between initialisation and finalisation it was possible to maintain that transparency of presentation. (The more demanding complication of where Argmessage and the client application share use of the same underlying message library has been noted above.)

Appendix B – Compilation of Argmessage and Qsargm

17.11. Compilation of components for heterogeneous architectures

As will become apparent, the software in this project has server and client applications that run in different processes, with the intention being, in general, that they are run on separate processors. Indeed, the main target arrangement has the client application running on an x86-64 compute node and the server running on a BlueField card plugged into the motherboard of that compute node. In that case, the two processors have different architectures.

It is therefore necessary to be able to compile the client application and the server for their respective architectures. This leads to the following difficulties. While administrators of HPC systems often provide a selection of dependencies precompiled for compute nodes against which users may compile their own code, unsurprisingly, since BlueField cards are relatively new and not commonplace, they do not provide similar for the BlueField cards. The need for such libraries will not only be a few system libraries, but also computational dependencies may well also be needed on the compute node (and potentially the BlueField card but that did not arise).

Compilation of libraries is often, of course, not that straightforward: each has its own dependencies and options for compilation, and various build systems may be preferred for each. This can lead to a complex set of possibilities, only some of which work. System administrators distil their experience of getting a large set of working dependencies to compile into a collection of build scripts. For example, those used for the central HPC systems at University College London are to be found on GitHub [87]. One route to providing for dependencies for different architectures might have been to use such scripts adapting them as necessary. This would have been a lengthy and risky process, for example working down the tree of dependencies only to find some incompatibility. Certainly I found the process unwieldy. I hoped to address this problem in this work by using the Spack package manager [88], but this was only partially successful. Nonetheless it may find better application in future as Spack develops.

17.12. The Spack package manager

In order to tailor the binaries for the system being used, Spack downloads the source code, applies patches and compiles it using its own compile script. Further and importantly, it also compiles any required dependencies, and dependencies needed by them and so on, in a consistent manner. The build scripts for each package are kept in a global central repository and are contributed and maintained by both system administrators of many HPC and other systems and by the authors themselves of the applications and libraries being built. In this way much more experience of which build options and dependency versions are compatible is built up. Spack provides build scripts for wide range of packages used in scientific computing and so was suitable for use in this project.

In normal use of Spack these build scripts are fairly hidden from the user and so two users on different systems, having the same or different architectures, will get highly similarly built code. To provide some flexibility however in what is built, options are provided to allow that. Those options, together with the version numbers and options of the dependencies and even also those details for the compiler, used to build a package are recorded, resulting in a complete specification of how that package and its dependencies have been built. That specification can be later used to, more or less exactly, reproduce the complete build, which is of great assistance to, for example, another user who wants to reproduce computational results generated by the package. These build operations are not instant, as Spack finds a wealth of dependencies to be built for many packages, but it was of course far more methodical in doing that than I could have been.

It was a goal in this project therefore to use Spack to build as many dependencies as possible. In the end, it was only used to build the *netlib-lapack* package, which was used in the main example of a computational program using the Qsargm library of the project.

In the main this project used MPI, in particular OpenMPI, for the messaging between the Argmessage client and the server. Spack has build scripts for OpenMPI but these did not always work, particularly for more recent versions (at the time of building them), failing when building its dependency on the lower level UCX communication library [31]. Fortunately, (i) compiled versions of OpenMPI for many different architectures and operating systems are provided by Nvdia/Mellanox in the HPCX distribution [89], which are often, but apparently not always, easy to install, and (ii) Spack does provide facilities for declaring a dependency that exists on the system to be a package within its system, so that it is what I used generally.

One useful feature I found in Spack was on an earlier system than used for the final experimental system, which did not allow Internet download to the BlueField cards, which was inconvenient since Spack downloads the source code files. Spack allows different installations (a set of sources and compiled software made by Spack – Spack "site") to be marked downstream of one another, with the downstream installation reusing items available in the upstream one. So, I was able to declare Spack sites for nodes of different processor architectures, one of which had Internet access available for its nodes. Builds on that downloaded the sources and these sources would then be reused by Spack when compiling packages on the BlueField card.

17.13. Build system for Argmessage

The Argmessage build is not complex and could be achieved with various build systems. The Argmessage client is intended to be compiled as part of the executable of the application it serves, so a user of Argmessage should include it in the build specification files of whatever build system is being used by the client application. In this project it was applied to QuickSched, which uses the Autotools build system, so the build specification files for that, the files Makefile.am, were extended to cover the Argmessage files as well. One important aspect of Argmessage, as noted in section 17.11, is that it may well need to be built for different architectures. In-source builds are always messy and do not work for multiple architectures as object files would be overwritten when compiling for the second architecture. The build script to build the application therefore uses a respective build directory for each architecture. The application was built for each architecture on a machine of that architecture, so the build script determines the architecture, and hence where to do the build, simply from the hostname of the system. One further complication of this was that the login node on the cluster where the code was developed has an architecture (an AMD one) different from both that of the compute nodes (an Intel one) and that of the BlueField cards (an Arm one); so, to allow for frequent compilation during debugging a build for the login node's architecture was also provided. The build script of course also set the environment to include the correct versions of dependencies for the architecture. Owing to the difficulties noted in this section, that comprised a mixture of user *environment modules* made for the hand built messaging libraries and *Spack modules* for the computational kernels.

17.14. Compiling messaging libraries and launching experiment programs

Section 17.2 discussed launching Argmessage applications generally. More detail is given here on how the QR factorisation experiments were launched, including how to launch for all the test *geometries* and including the arrangements for working through the sets of experimental parameters used.

Many HPC clusters are homogeneous, meaning that all machines in the cluster have the same processor architecture. If that is not the case, as in the present project, which has both x86-64 machines, i.e., the hosts, and Arm processors, on the

BlueField cards, care has to be taken to use not only the correct versions of the application code compiled for the respective architectures but also the correct versions of all the dependencies. Heterogeneous production clusters, for example with different generations of x86-64 processors with different instruction set extensions, do exist and there are a couple of tricks that can help a system administrator to deal with the problem of needing different versions of compiled codes.

One is to compile all the software modules for each architecture in matching directory structures and then mount on each machine the correct directory set for that machine in exactly the same place with respect to the root directory, “/”, independent of the architecture. In this way, selection of the correct version can be made entirely automatic: i.e., nothing has to be done in a launch script on different machines to make a selection since the same path for the code is quoted when launching an application program.

It is possible, I understand, to include variants of the object code for different x86-64 generations in the same object code file, but that is not relevant here as the machines here were more heterogeneous in that they include the Arm processors, to which that does not extend.

The cluster used here was, of course, experimental, and little software was provided. Only compilers and a few of the common computational libraries, MPI versions and a few common HPC applications were provided for the host systems; none were provided for the Arm system over the bare base operating system. So, since both Arm and x86-64 code was required, all the experimental code and its dependencies had to be compiled for each architecture separately. Not being granted administrator rights, mounting the code at the same path in the file system was not an option. Further, finding a compilation for OpenMPI that worked between the x86-64 hosts and Arm BlueField did not happen straight away, but the combination noted in Table 26 was found to work and was used in the experiments. It was tried to compile these with *Spack* but that failed to build the UCX dependency of OpenMPI.

OpenMPI 4.1.1	
Archive file:	openmpi-4.1.1.tar.gz
Configure flags:	--prefix=/global/scratch/users/cyrusl/placement/ software/openmpi-4.1.1/install-broadwell --with-ucx=/global/scratch/users/cyrusl/placement/ software/ucx-1.10.1/install_broadwell/ --enable-heterogeneous --without-slurm --enable-orterun-prefix-by-default=no --enable-mca-no-build=btl-uct --with-hwloc=/global/home/users/cyrusl/placement/ software/hwloc-2.4.1-broadwell
UCX	
Archive file:	ucx-1.10.1.tar.gz
Configure flags:	--disable-logging –disable-debug --disable-assertions –disable-params-check --prefix=/global/scratch/users/cyrusl/placement/ software/ucx-1.10.1/install_broadwell --enable-mt
hwloc (a dependency of OpenMPI)	
Archive file:	hwloc-2.4.1.tar.gz

Table 26 – Communication library compilation options

The compilation was done separately for each architecture: once for the Arm aarch64 architecture and once each for each of the particular x86-64 architectures of the different clusters used at HPCAC. The values in Table 26 for the OpenMPI and all dependency paths are for the Broadwell x86-64 architecture and were changed appropriately for the other x86-64 architectures. Linux modules were created for each version.

The options *--enable-heterogeneous*, *--without-slurm*, *--enable-orterun-prefix-by-default=no* for OpenMPI are all relevant to the launch flow described in Appendix A sections 17.2 to 17.4.

The launching of the client and server programs was achieved in these experiments as is now described, giving particular details of the MPI arrangements, as well as the organisation of providing the experimental parameters to test application programs.

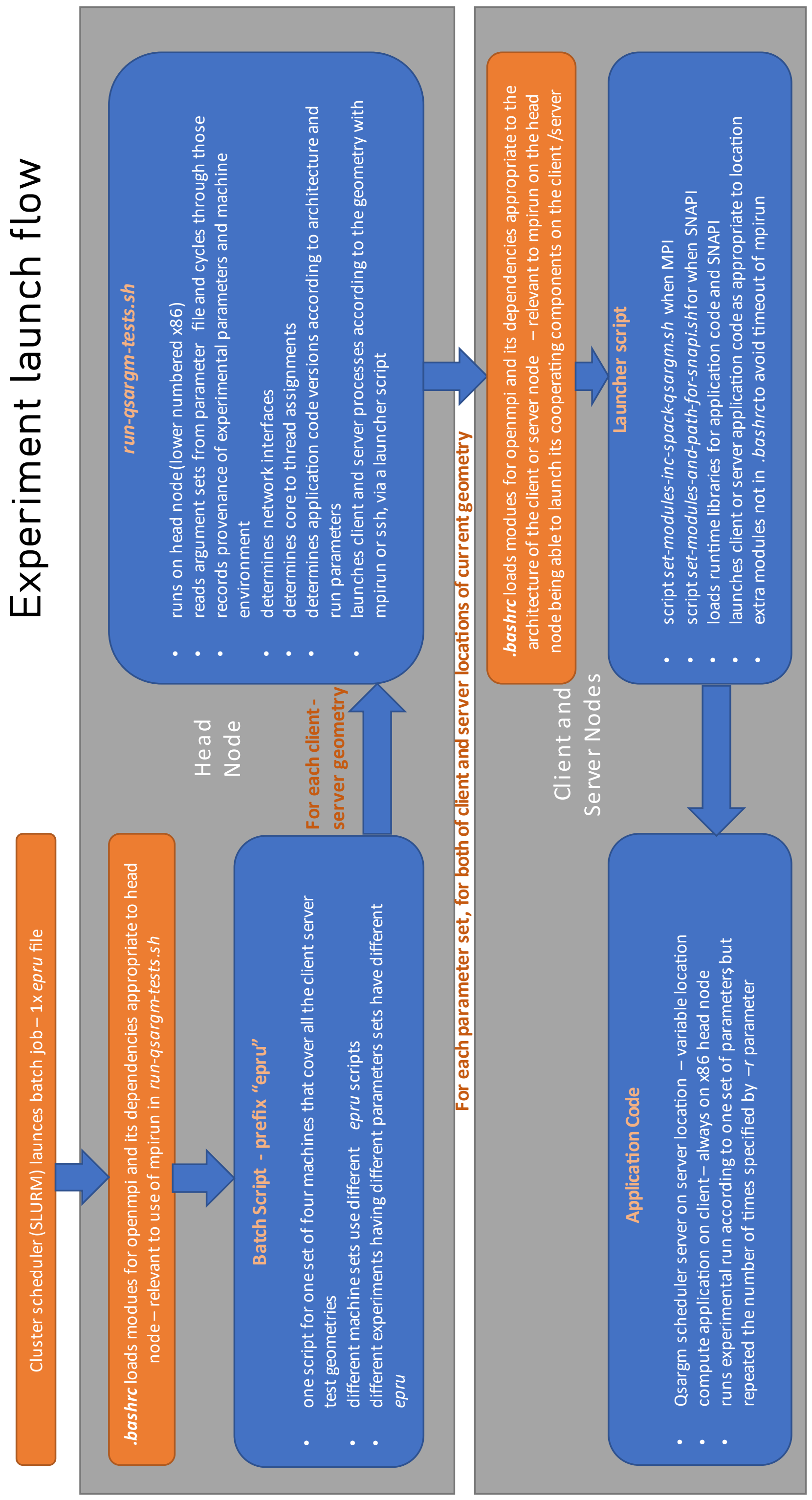

Experiment launch flow
Cluster scheduler (SLURM) launces batch job – 1x epru file
.bashrc loads modues for openmpi and its dependencies appropriate to head node – relevant to use of mpirun in run-qsargm-tests.sh
Batch Script - prefix "epru"
one script for one set of four machines that cover all the client server test geometries
different machine sets use different epru scripts
different experiments having different parameters sets have different epru
Head Node
For each client - server geometry
run-qsargm-tests.sh
runs on head node (lower numbered x86)
reads argument sets from parameter file and cycles through those
records provenance of experimental parameters and machine environment
determines network interfaces
determines core to thread assignments
determines application code versions according to architecture and run parameters
launches client and server processes according to the geometry with mpirun or ssh, via a launcher script
For each parameter set, for both of client and server locations of current geometry
.bashrc loads modues for openmpi and its dependencies appropriate to the architecture of the client or server node – relevant to mpirun on the head node being able to launch its cooperating components on the client /server
Launcher script
script set-modules-inc-spack-qsargm.sh when MPI
script set-modules-and-path-for-snapi.sh for when SNAPI
loads runtime libraries for application code and SNAPI
launches client or server application code as appropriate to location
extra modules not in .bashrc to avoid timeout of mpirun
Client and Server Nodes
Application Code
Qsargm scheduler server on server location – variable location
compute application on client – always on x86 head node
runs experimental run according to one set of parameters, but repeated the number of times specified by –r parameter

Figure 42 – Launch flow diagram for experiment programs

Figure 42 is a block diagram showing the flow of the launch process. A batch script is submitted to the cluster batch scheduler, in this case the *SLURM* scheduler, requesting a set of nodes of the form shown in Figure 10, namely a pair of x86 hosts and their respective BlueField cards, so covering all the *geometries*. These batch files in the archive have names prefixed with “epru”. The cluster scheduler allocates those nodes and launches the *epru* script on the head node, which was always the x86-64 node with the lower number in its machine name; so, for example, when *jupiter005*, *jupiter006*, *jupiter-bf05* and *jupiter-bf06* were specified, *jupiter005* was the head node. While it would have been more transparent to set the OpenMPI modules in the *epru* batch file, this was put in the *.bashrc* file, which is, of course, executed each time a new *bash* shell is started as a login shell [90]. So in the launch arrangements, *.bashrc* is executed in two places: (i) when the cluster scheduler starts a new process on the head node to execute the *epru* script and (ii) when, in the arrangement of the HPCAC clusters, *mpirun* starts new processes on the client and server machines to run the application program. The reason for setting the modules for OpenMPI in *.bashrc* comes from the second of those, which I shall come to in a moment.

So, *.bashrc* is first executed just before the *epru* script (since the cluster scheduler logs into the head node in order to run the latter). There *.bashrc* sets the version of *OpenMPI* needed on an x86 host, in particular to execute the *mpirun* program. Next, the *epru* script, in many examples, cycles through the four possible geometries and for each calls the script *run-qsargm-tests.sh*. In turn that script cycles through the parameter sets being used (one parameter set being one line of a parameter file), and for each of those it launches the task scheduler server program and compute client program in respective processes at their respective positions in the current *geometry*. For each launch it first determines the names of the network interfaces to use, the index numbers of the processor cores to be assigned to each of the two processes, the locations, in the file system, of the object code for the programs compiled for (i) the relevant architecture and (ii) for any specified compile time options for the test application programs (usually the QR test and the remote scheduler programs), and determines the launch method (depending on whether

OpenMPI or SNAPI messaging is being used), and further it passes on those of the experimental parameters that the test application programs need to consume.

.bashrc is also called, for the second time, before each of the test client and server programs is launched (since mpirun logs into their respective rank nodes in order to launch those programs) in their respective locations. Here *.bashrc*, in the same way as for the first visit to *.bashrc*, loads modules for OpenMPI and its dependencies appropriate to the architecture of the machine. It is necessary to do that here because the first thing that the *mpirun* command on the head node does once logged into an MPI rank node, before it gets round to launching the program requested in the *mpirun* command, is to launch its own component ORTE, which in turn launches the application programs [91]. This process failed unless the correct versions of OpenMPI for the respective architectures are already selected. Therefore, in this arrangement it is necessary to select the OpenMPI modules before that happens, therefore in *.bashrc*.

Now, the QR factorisation application program needed another dependency, in particular the *Netlib LAPACK* library [92], for the independent verification of its calculation. This library was prepared and loaded with the *Spack* package manager, but it proved impossible to load it in *.bashrc* because it caused the *mpirun* operation to fail, probably by timing out. *Spack*, especially when initialising, seems to require a large number of file accesses, which is a very slow process on most parallel file systems. So, the *Spack* initialisation had to be done later: *mpirun* was set to run a launch script which first initialised *Spack,* then with that loaded the extra dependency before finally launching the object codes of the client and server programs, which meant that the *Spack* initialisation could be removed from *.bashrc*. (The bash script was called *set-modules-inc-spack-qsargm.sh* or *set-modules-and-path-for-snapi.sh*. For the case of the *SNAPI* messaging the *SNAPI* library was also selected in this launch script.)

Splitting the selection of the dependencies like this, i.e., between *.bashrc* and a launch script, makes it less easy for a reader of the code to see how the modules are being loaded, but was necessary for the reason given.[9] The launch script also passes on to the client program the experimental parameter set it needed that came from the current line of the parameter file.

To compare performance between machines of different specifications, experiments were run on different sets of four machines. In some examples, the different *geometries* were, as described above in this section, run with respective *mpirun* commands in the same *epru* file; in others it was arranged by using a separate *epru* batch job file for each such set, which differed in the machines specified there to the SLRUM job scheduler. It depended on whether the maximum job run time for the cluster was going to be exceeded.

Different *epru* batch job files were also used to change the parameter file being used to provide parameter sets to the QR factorisation application. These batch files may be found in the project archive in the directory archive:expt0069/argmessageprivate/run-scripts-HPCAC/exptl_runs/.

In fact, the script run-qsargm-tests.sh had many parameters with the effects noted in Table 27.

[9] Having gained further experience with Spack more recently, I have noticed that part Spack can be used to generate traditional environment modules which may be invoked without loading Spack. These of course load much faster since they simply update environment variables. On the other hand they hide the provenance of the code, whereas Spack can provide a full tree of the versions, options and dependencies used to compile a code.

Parameter	Effect
-p	Selection of the application to be run, e.g. -p QR for the QR factorisation experiment but there are also options for the simpler test programs.
-m	Selects between OpenMPI and SNAPI messaging for the messages between the Argmessage client and server (distinct from the -m parameter in Table 9).
-u	Select for launch a previously compiled version of the QR application having many print statements for debugging purposes.
-c	Machine name for the client program.
-b	Machine name for the remote server program, so together with -c, these allow the *geometry* to be specified.
-w	Run warm-up programs before the experiment (not used).
-W	For internal use relating to the warm-up.
-g	Path of a parameter file for client application (those parameters are noted in Table 9), one line thereof being passed to the client at each launch of the client application made by the *run-qsargm-tests.sh* script.
-P	Introduces a set of parameters to be passed to the client program (an alternative to the -g mechanism), the set is to be put in quotes in the *run-qsargm-tests.sh* command line. This was used for interactive debugging and testing.
-T	Number of threads to be allotted to the server program, which may also be specified in the -g parameter file lines.
-o	Select the compile time code optimisations for the QR application, operates by selecting for launch among previously compiled versions of the QR application having the combination of those optimisations.

Table 27 – Parameters of the run-qsargm-tests.sh launch script

For most of the experiments, OpenMPI messaging was used. There was a difference in the launching for when SNAPI messaging was used, which was that, instead of a single *mpirun* statement being used to launch both the client and server, separate and respective *ssh* calls to the relevant client and server locations were made. This was done to make use of the *taskset* command to set the processor core assignment explicitly, rather than having to consider whether *mpirun* would set those in the case of an application program that did not use OpenMPI beyond that (i.e., has no MPI statements in the application code), such as the SNAPI version of the experimental code.

Finally, the *control* scheduler (the original QuickSched) does not, of course, by definition, need a separate scheduler process. However, in the experimental runs such a process with remote scheduler was launched to keep the launch arrangements the same, but that process was not used once launched, i.e., it is never contacted by the client with Argmessage RPC calls.

The selection in the application code between the original *control* and remote schedulers comes inside the running of the client program in the code of each *Qsargm* proxy interface function. This examines a parameter (-S, *strategy*, Table 9) passed from the launch script and thereafter, in the case of *control* scheduler, passes the call directly to the original QuickSched library, or, in the case of the modified, *Qsargm*, scheduler, to a function that sends a message to the experimental scheduler. In this way *control* experiments could be interleaved with those using an experimental remote scheduler, without having to change the version of the application programs being launched, avoiding any doubt as to the identity of the code, otherwise, between the two cases.

Appendix C – Results Archive Notes

17.15. Post Processing

Post processing of the results began with filtering the output logs, from both the client and server, for the timer values and their contexts. The 'contexts' were mostly the parameters used to generate the experimental results, and also the index of the -r repetition of the execution of the QR factorisation code for the same combination of parameters. This data was compiled into a JSON format file.

The results files so gathered have a filename extension of *.QR-2.json*, the "2" indicating that a second attempt was needed to optimise the format.

The program *collect_QR.py* performs the compilation of results form a results directory and makes use of the functions in the library *collectors.py*, both at archive: expt0069/argmessageprivate/aftermath/aftermath/collect/. The library functions find lines in the logs using *regex* matching to find the data lines, but this is supplemented with more *regex* to find surrounding context delimiting lines to deal with the data lines being similar between repeats.

Each *run* produced by a line of a parameter set file produces separate log files. Compiling the respective JSON file sections for these *runs* was an embarrassingly parallel problem and so was parallelised using Python's *multiprocessing* module.

The further processing of the JSON files, for example, in order to plot a graph, begins with conversion of the JSON files into a single flat file *numpy* [93] table, with a column for each timer and so repeating the values of outer context items in their columns. This was performed by a program *aggregate.py* (archive:expt0069/ argmessageprivate/aftermath/aftermath/collect/aggregate.py). This single table format was chosen as it is very straightforward to filter and sort as required, e.g., for plotting graphs. When it was needed to include results from more than one result directory, the conversion of each to a flat table is processed in parallel, again parallelised using Python's *multiprocessing* module, with the resulting flat file tables being concatenated.

The routines for plotting the results of the QR factorisation experiments are to be found in the at archive:

expt0069/argmessageprivate/aftermath/aftermath/plot/plotQR.py. The plots themselves are explained in the relevant sections discussing the results. A higher level script, archive:expt0069/argmessageprivate/aftermath/aftermath build_display_sets.py compiles the items mentioned in the next section.

17.16. Archive structure of “result sets”

The *epru* batch scripts described above at section 17.14 were the unit of initiating experiments. Each of these created one or more results directories, generally one for each parameter file consumed and so, e.g., one for each of the geometries addressed by the *epru* file. To organise these results directories into suitable groups for post processing and ease of reference *sets* were defined. The *sets* aggregate results directories for the different test *geometries* and for different specifications of host node. These *sets* are referred to in this document by the same letter designation as they are in the archive. The discussion of the results in the next chapter starts, however, at set C, since sets A and B were part of the software development process. To facilitate browsing of the results in the archive, the directory for each set contains:

- symbolic links to the constituent result directories,
- a Python notebook in respect of each result directory grouping for displaying the graphs in that result directory,
- image files of graphs based on data taken from across the result directories,
- a file *run_dirs_for_set.dat* listing the result directories for the set, which is used by the scripts that compile the other items, and
- usually, top level logs of the batch jobs that created the results in the respective result directories (file names are of the format *epruNNNN_JJJJJJ.out* and *.err*, where *NNNN* is an index for the batch script file and *JJJJJJ* is the job number for the system).

The gathered JSON results files remain in their respective results directories. Each of those also sometimes contains a file *args_with_resultant_omp_places.txt* that records the core index numbers actually allocated, recovered from the logs, for the computational and scheduler threads against each input parameter set.

Appendix D – Argmessage and Qsargm Code Files

This appendix is the code of the Argmessage and Qsargm modules written for this project. Code files of the original scheduler functions are not included; reference should be made to the original authors' code repository on Source Forge and their publications. These files are used unchanged. Files are included which provide an interface between the current project to original QuickSched functions or are the test QR factorisation example. These necessarily reproduce code or structures of the original but are included to show how they have been adapted to make use of the original library within the current project. These files are acknowledged individually.

Program Copyright Notice and Licence

So that the code may be published under this LGPL3 licence (that of the predecessor project), it has been redacted here but published on Github at https://github.com/cjlegg/qsargm-thesis. The filenames of the files intended for this Annex are listed below, with the original comments on the code remaining.

argmessage.h

argmessagehelpers.h

argmessagehelpers.c

argmessageproxy.h

argmessagesizes.h

packunpack.h

packunpack.c

waitfordebugger.h

waitfordebugger.c

qsargm_adapter0.h

qsargm-adapter0.c

qsargm_client.c

qsargm_common0.h

qsargm_common0.h

qsargm_common0.c

qsargm_proxy0.h

qsargm_proxy0.c

qsargm_server.c

qsched.h

This file has been adapted from the original QuickSched, but has been updated to include additional functions of this project.

qsched.c

This file has been adapted from the original QuickSched, but has been updated to provide an interface to the code of this project, principally the switch in each public interface function to call either the original code or the code of the present project.

qsargm_test_qr.c

This file has been adapted from the original QuickSched, but has been updated to use the Qsargm version of that from this current project.