Durham University

Durham E-Theses

Task-Based Parallelism for General Purpose Graphics Processing Units and Hybrid Shared-Distributed Memory Systems.

AIDAN BERNARD GERARD CHALK

How to cite:

CHALK, AIDAN BERNARD GERARD (2017) Task-Based Parallelism for General Purpose Graphics Processing Units and Hybrid Shared-Distributed Memory Systems. Doctoral thesis, Durham University.

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a https://etheses.durham.ac.uk/id/eprint/12292/ is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
https://etheses.durham.ac.uk

Task-Based Parallelism for General Purpose Graphics Processing Units and Hybrid Shared-Distributed Memory Systems.

Aidan Bernard Gerard Chalk

A Thesis presented for the degree of
Doctor of Philosophy

School of Engineering and Computing Sciences
Durham University
United Kingdom

September 2017

Abstract

Modern computers can no longer rely on increasing CPU speed to improve their performance as further increasing the clock speed of single CPU machines will make them too difficult to cool [48], or the cooling require too much power. Hardware manufacturers must now use parallelism to drive performance to the levels expected by Moore's Law. More recently, High Performance Computers (HPCs) have adopted heterogeneous architectures, *i.e.* having multiple types of computing hardware (such as CPU & GPU) on a single node. These architectures allow the opportunity to extract performance from non-CPU architectures, while still providing a general purpose platform for less modern codes [13].

In this thesis we investigate Task-Based Parallelism, a shared-memory paradigm for parallel computing. Task-Based Parallelism requires the programmer to divide the work into chunks (known as *tasks*) and describe the data dependencies between tasks. The tasks are then scheduled amongst the threads automatically by the task-based scheduler. In this thesis we examine how Task-Based Parallelism can be used with GPUs and hybrid shared-distributed memory, in particular we examine how data transfer can be incorporated into a task-based framework, either to the GPU from the host, or between separate nodes. We also examine how we can use the task graph to load balance the computation between multiple nodes or GPUs.

We test our task-based methods with Molecular Dynamics, a tiled QR decomposition, and a new task-based Barnes-Hut algorithm. These are problems with different dependency structures which tests the ability of the scheduler to handle a variety of different types of computation. The results with these testcases show improved performance when we use asynchronous data transfer to and from the GPU, and show reasonable parallel efficiency over a small number of MPI ranks.

Statement of Copyright

The copyright of this thesis rests with the author. No quotation from it should be published without the author's prior written consent and information derived from it should be acknowledged.

Declaration

The work in this thesis was carried out in the School of Engineering and Computing Sciences at Durham University between October 2012 and March 2017. All of the work was carried out by the author unless otherwise stated and has not previously been submitted for a degree at this or any other university.

Acknowledgements

I would like to thank my external advisor Dr Pedro Gonnet for his continued support throughout my PhD. It is only due to his expertise and direction I have completed this work.

I would also like to thank my supervisors, Dr Tobias Weinzierl and Prof Iain Stewart for their help during my PhD.

Many thanks to the members of the SWIFT project and particularly to Dr Matthieu Schaller for his help with the work for the Barnes-Hut simulation.

Additional thanks to the lecturers in the Innovative Computing group at Durham University for helping me with issues that I encountered during my PhD.

This work was supported by the UK Engineering and Physical Sciences Research Council.

This work used facilities provided as a part of the Durham University NVidia CUDA Research Centre. This work also used the DiRAC Data Centric system at Durham University, operated by the Institute for Computational Cosmology on behalf of the STFC DiRAC HPC Facility (www.dirac.ac.uk). This equipment was funded by BIS National E-infrastructure capital grant ST/K00042X/1, STFC capital grant ST/H008519/1, and STFC DiRAC Operations grant ST/K003267/1 and Durham University. DiRAC is part of the National E-Infrastructure.

Finally, I'd like to thank my family and Helen for their love and support in completing this work.

Contents

Chapter 1

Introduction

Modern computers can no longer rely on increasing CPU speed to improve their performance as further increasing the clock speed of single CPU machines will make them too difficult to cool [48], or the cooling becomes too expensive in terms of power. Until now, hardware manufacturers could rely on increasing clock rates to driver performance, now they must use parallelism to drive performance to the levels expected by Moore's Law.

The most popular method for parallel computing is the Message Passing Interface (MPI) [44]. In MPI, simulations are usually parallelised by decomposing the data into equal sized chunks, and executing the computation on each chunk on each processor in parallel. This is known as *data parallelism*. Each processor can communicate over an interconnect with any other processor in the system. This programming model is primarily designed for distributed memory systems, in which no two processors share the same memory. In a distributed memory system, each individual CPU is known as a node (MPI specifically calls each process a *rank*), and each node also contains the memory and network required by that CPU.

Parallelism has also spread within single CPUs, known as multicores (or shared memory machines). Machines with 8 or 16 CPU cores are now commonplace, and workstations with over 32 cores are available. The ever increasing size of multicores is already driving larger HPCs, with the Trinity supercomputer featuring 301,056 CPU cores, with 16 cores per node [1]. More detail on how these work is discussed in section 1.1 Additionally, massively parallel devices known as *accelerators* that feature many more cores than CPUs are becoming increasingly popular. The current number one in the top 500 list features an accelerator-style architecture only. Figure 1.1 shows how the number of cores and performance of the number one in the top 500 list has progressed since the top 500 list's creation.

More recently, OpenMP has become increasingly popular as a method of using processors that share memory. OpenMP allows a programmer to add directives or

pragmas to their code, to tell the compiler to parallelise certain sections with multiple threads (*multithreading*). In the simplest case, the *parallel for* is used, which allows a single for loop to be executed in parallel. To ensure correctness, the programmer also needs to make sure to declare which variables can be shared between threads, and which variables need to be copied for each thread. This fork-join parallelism can lead to large amounts of load imbalance for more complex loops, and can introduce a lot of synchronisation points. Getting good performance from OpenMP programs can be quite difficult.

1.1 Shared and Hybrid Shared-Distributed Memory High Performance Computers

Shared memory systems (*multicores*) feature one or more processors sharing overlapping parts of the memory hierarchy. An example of multiple processors sharing parts of the cache hierarchy is shown in Figure 1.2-a. Modern multicores often feature multiple CPU sockets, i.e. a 32 CPU multicore may feature 4 physical CPUs, with 8 cores each.

Older multiprocessor systems used a *Uniform Memory Access* (UMA) memory model, where all of the processors accessed memory through a single memory bus, and so the memory access time was uniform for all processors. As the number of cores and CPU sockets increased, it has become standard that the speed of access to different regions of memory is not uniform for each processor. This is known as *Non-Uniform Memory Access* (NUMA). Each group of processors (e.g. cores in a single CPU) have UMA access to some section of main memory, known as their *local memory*. These groups are connected by an interconnect, allowing slower access to non-local memory. This is shown in Figure 1.2-b.

Multicores have become increasingly common in HPCs. The distributed memory model used on larger HPCs has been replaced by a new hybrid, shared-distributed memory model, where the HPC consists of many multicore nodes connected by an interconnection network. Each node usually features between 8 and 128 cores.

1.2 Introduction to CUDA GPUs

Computing using General Purpose Graphics Processing Units (GPGPUs, or GPUs) has become increasingly popular in the last few years, most commonly GPUs that use NVIDIA's CUDA[35] architecture. CUDA GPUs have large numbers of cores (often over 2000), each of which are less powerful than those in a standard CPU, have more limited memory access, and a less rich instruction set. These cores are grouped into

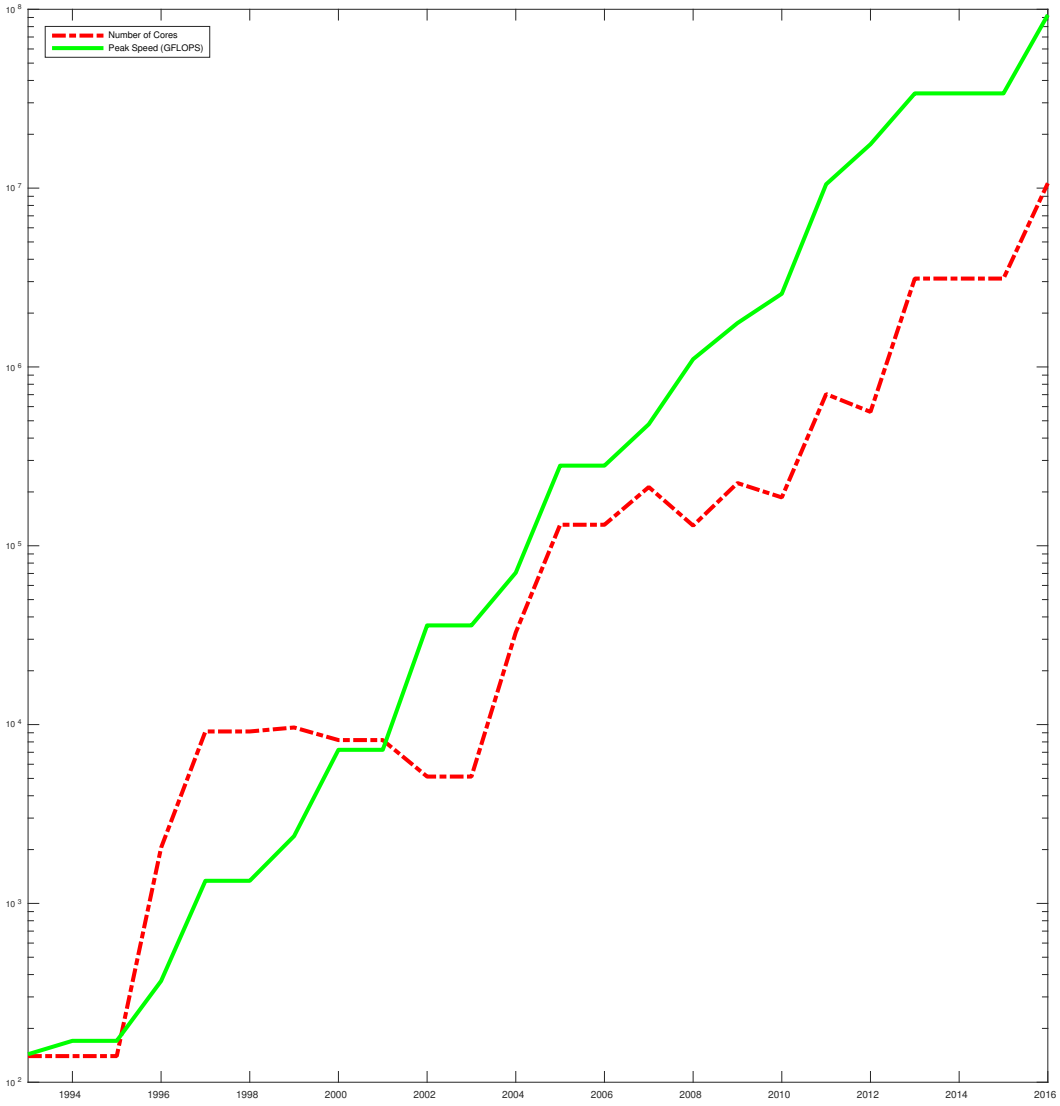


Figure 1.1: A chart showing the performance and number of cores for the number 1 machine in the top500 list in each year from 1993. The data was taken from the November lists in each year from 1993-2015, as well as the June 2016 list. Since 2007, the number of cores in the machines has increased at roughly the same rate as the performance.

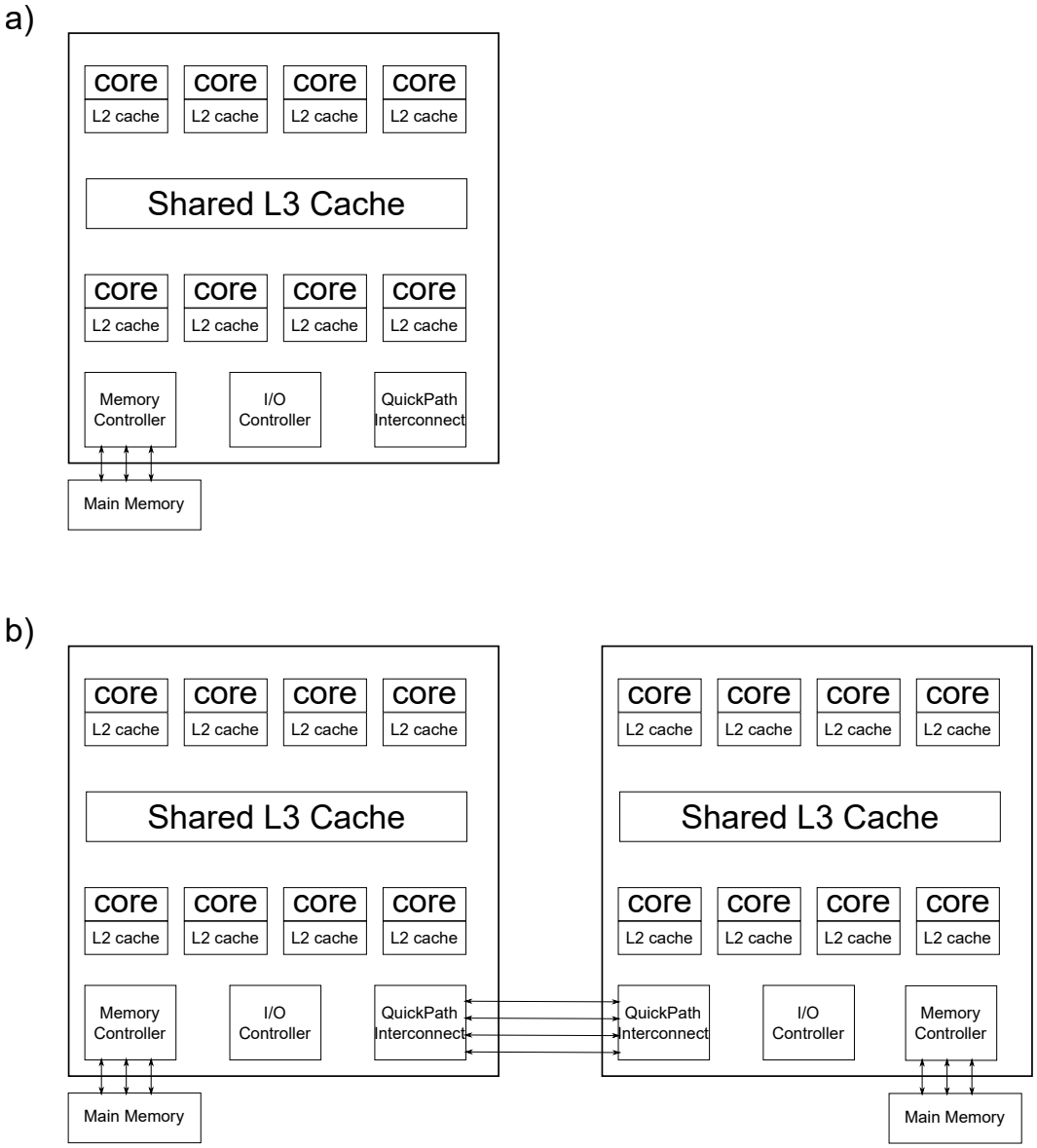


Figure 1.2: Figure a) shows a single multicore CPU (based on Intel Haswell). Each core will usually have its own L1 and L2 cache, however multiple cores will share the L3 as shown. The cores will all share the main bus to RAM. The shared nature of CPU caches varies on each individual processor.
Figure b) shows a 2-socket CPU setup, where each CPU will have similar cache hierarchy as in a). Each CPU has has its own direct bus to an area of RAM known as its local memory, and must use the interconnect between the CPUs to access the other CPUs local memory. The memory accesses through the interconnect are often slower than access to local memory.

streaming multiprocessors (SMXs or SMs), which currently contain 192 cores. In each SM, there are 6 *warp* schedulers, which each issue a single instruction to 32 threads to be executed in lock-step.

During computation up to $2^{31} - 1$ threads (known as a *thread grid*) can be set to execute a *kernel*, a sequence of instructions to be executed on the device. The thread grid is split into *blocks* of up to 1024 threads each. Each block is assigned to an SM when executed, and is only executed on that SM. Current SMs can have up to 16 blocks assigned to them at any time. Each block is further broken down into *warps* of 32 threads. The GPU executes each warp of threads in strict lock-step parallelism, i.e. all 32 threads execute the same instruction before any thread in the warp executes any other instruction. A schematic of a single SM is shown in Figure 1.3-b.

Cores have relatively slow access to the main GPU memory, however there is a small high-speed L2 cache shared between all the SMs (as shown in Figure 1.3-a), as well as an individual L1 cache for each SM. The L1 cache is split into two sections, the actual L1 cache and *shared memory*. The L1 caches are not coherent between SMs, so if 2 SMs are writing to the same area in memory they will not necessarily observe changes made on the other SM immediately. To help avoid these issues, CUDA has a set of `__threadfence` functions which can help enforce the observed ordering of memory operations by other threads in the system.

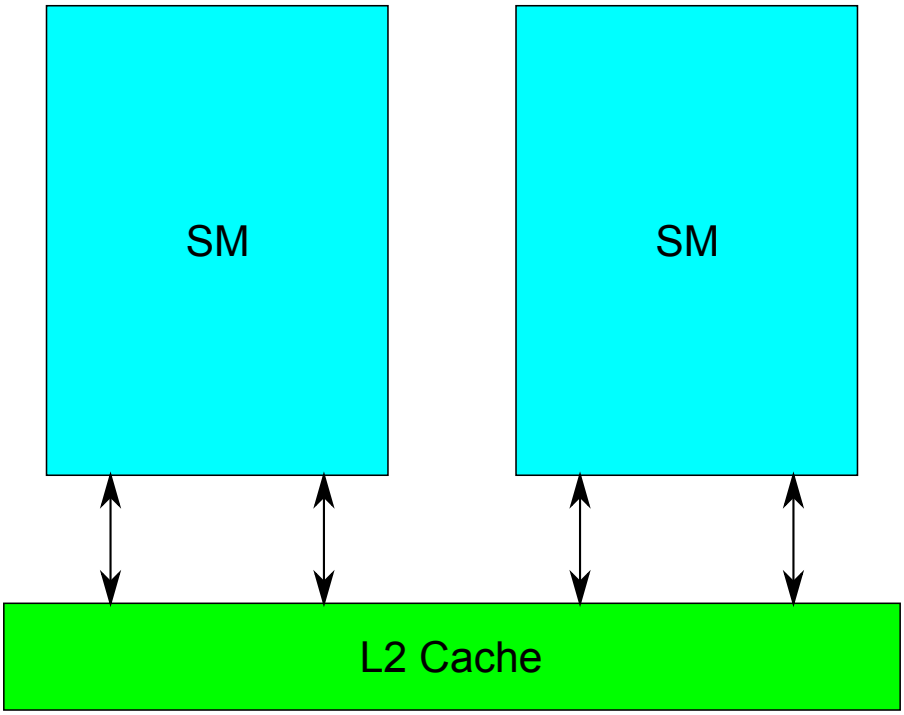
The traditional programming paradigm for GPUs has been to treat them as large vector machines, repeating the same set of instructions, known as *kernels*, on large datasets in lock-step, i.e. "'Single Instruction, Multiple Thread"' (SIMT) parallelism. This approach is effective for problems that vectorise easily. Unfortunately, many problems cannot be easily vectorised and thus currently cannot be efficiently ported to these devices.

Individual blocks have no explicit mechanisms to communicate with each other, synchronisation between blocks has to be managed explicitly by using atomic operation on values in global memory.

1.3 Heterogeneous Architectures (GPU Clusters)

Heterogeneous architectures are an increasingly common type of architecture in HPC systems, with 3 of the top 10 HPCs in the June 2016 top 500 [1] list utilising heterogeneous architectures. Heterogeneous architecture machines consist of nodes with both (multicore) CPUs and one or more accelerators or coprocessors. Most commonly these systems use either NVIDIA CUDA GPUs (accelerators) or Intel Xeon Phi coprocessors. As accelerators and coprocessors are currently more power efficient (per flop) than standard CPUs, these are likely to be a staple in future large machines, due

a)



b)

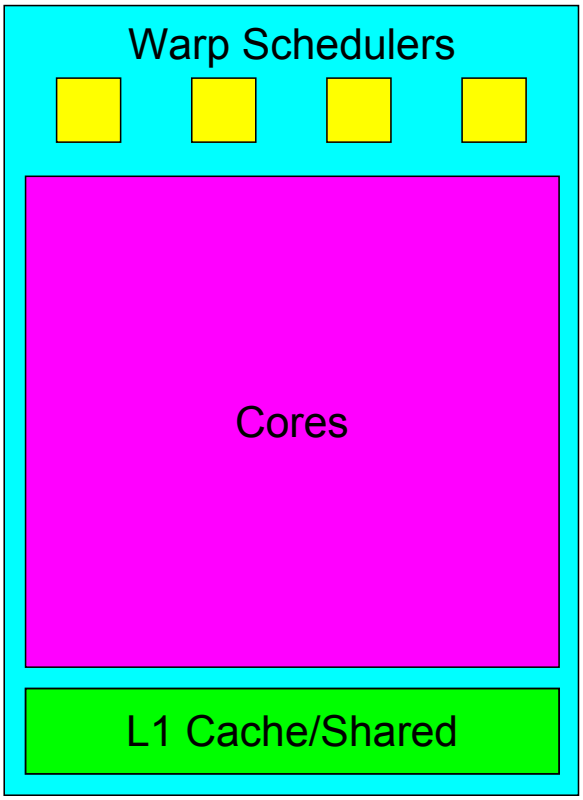


Figure 1.3: Figure a) shows a pair of SMs, each with their own access to the shared L2 Cache.
Figure b) shows a single SM in more detail, with multiple warp schedulers, a large block of cores, as well as the SM's own L1 Cache and shared memory region.

to the concerns about power consumption in exascale systems [13]. Accelerators and coprocessors are currently mainly only useful for specific types of tasks that vectorise well, e.g. Linear Algebra, Finite Element Methods.

Currently heterogeneous architectures are used less effectively than homogeneous architectures [13]. These architectures present many challenges compared to homogeneous architectures, including data movement between RAM and accelerator memory, load balancing between processor types, and avoiding the introduction of more synchronisation points.

Accelerators and coprocessors typically have their own integrated memory which cannot be accessed directly from the CPU. Current architectures require data transfer across the PCI bus, which is slow and can be power inefficient compared to RAM accesses by the CPU. The data is usually copied to the device manually by CPU operations, or by MPI calls, which is an additional cost to using these devices.

Another problem highlighted by these heterogeneous architectures is load balancing. While load balancing has improved on homogeneous architectures it becomes more difficult on these heterogeneous architectures, as often times they are not executing the same operations. If the CPUs in the system are waiting for accelerators or coprocessors to finish work, it introduces additional synchronisation points which can reduce performance.

1.3.1 Thesis Overview

This thesis examines an approach for using modern HPCs, known as *Task-Based Parallelism*. In Chapter 2 I give a historical overview of the method, and I discuss our implementation of Task-Based Parallelism in Chapter 3, called *QuickSched*.

Chapter 4, I discuss the test-cases and algorithms used to test our implementations, and introduce a new task-based algorithm for Barnes-Hut simulations in Chapter 5.

Task-Based Parallelism is an established paradigm for shared-memory CPU parallelism, however its ability to be used on GPGPUs is less established. I aim to provide a possible solution in chapter 6. In HPC, GPUs are often treated as accelerators for specific sections of code, usually sections that naturally parallelise to a large number of threads. Most task-based setups treat GPUs as a single, massively vectorised processor, which executes single tasks across the entire device. Instead, we recognise the ability of the GPU to perform many tasks in parallel inside a single GPU kernel, where each task needs to parallelise to a small number of threads. Our approach allows complicated algorithms to be executed in a task-based way using only the GPU, and allows us to perform computation in parallel with data transfer, an important topic in GPGPU computing.

Another topic attracting a lot of research interest for Task-Based Parallelism is

its use on Homogeneous Hybrid-Memory architectures. In Chapter 7, I introduce an extension to our QuickSched model for these architectures, and show its strengths and weaknesses on small numbers of MPI ranks.

Chapter 2

Task-Based Parallelism

2.1 Motivation

Traditional HPC methods are becoming increasingly difficult to use efficiently on large machines. MPI typically relies on a data decomposition approach, and communicating data between processors. As the number of processors increases the ratio of computation to communication falls, eventually resulting in communication dominating the computation in the system, as shown in figure 2.1. This can lead to a loss of strong scaling at larger numbers of nodes.

Recently MPI has been combined with OpenMP on these large machines to reduce the number of MPI ranks required (as OpenMP deals with the shared memory parallelism on each node). OpenMP can introduce new issues however. OpenMP computations usually consist of a number of parallel sections, and synchronisation at the end of each of these sections while the system waits for all of the threads to complete. This fork-join approach to parallelism can introduce a lot of synchronisation points in the system, amplifying any load imbalances and reducing strong scaling as the number of processors per node increases, as shown in Figure 2.2. NVIDIA have also developed OpenACC [33] whjch is an OpenMP-like paradigm for GPU programming, where the user can annotate loops to be executed on the GPU.

In addition to the massive increase in the number of CPU cores in HPC systems, accelerators such as NVIDIA CUDA GPUs or Intel Xeon Phi Coprocessors are becoming increasingly popular. These devices have very different architectures to traditional CPUs, so the standard MPI/OpenMP approach does not work on them.

One possible solution to the ever increasing complexity of HPC machines could be to find a programming model that is effective for both shared and distributed memory setups and that can support accelerators as well. One such possibility is *Task-Based Parallelism*.

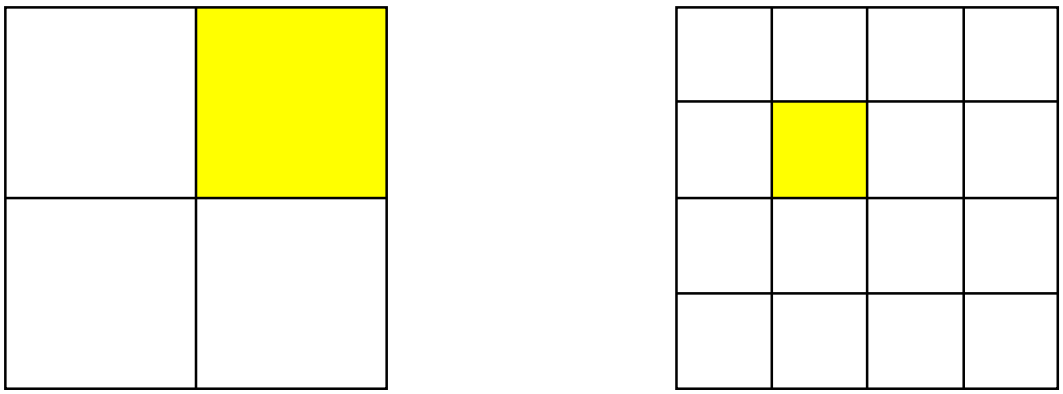


Figure 2.1: A diagram showing how increasing the number of ranks can lead to communication overheads with MPI. As the number of ranks increases, the amount of computation (the area in yellow) on each node decreases, however the amount of communication decreases more slowly (the edges of the boxes).

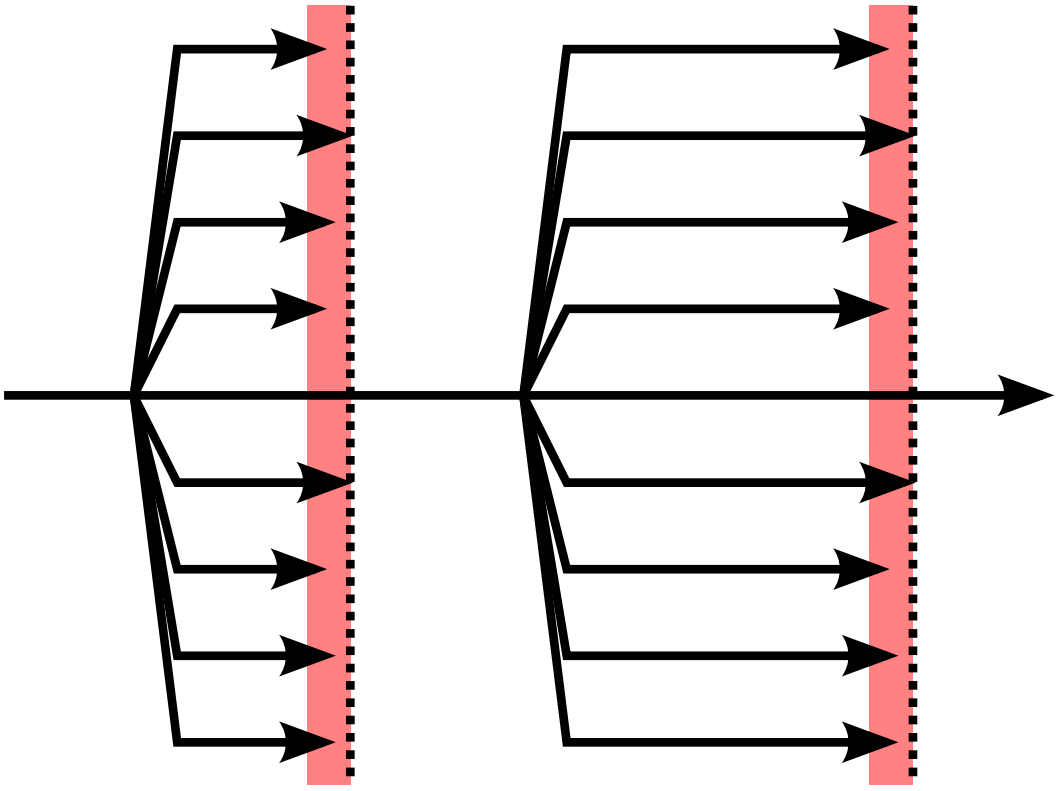


Figure 2.2: A diagram showing the fork-join method associated with OpenMP. The red highlighted areas show areas where there is load imbalance, so some threads are idling while waiting for other threads to complete.

2.2 Introduction to Task-Based Parallelism

Task-Based Parallelism is a method originally created for shared memory parallel programming, in which a program is split into a series of tasks. The tasks are retrieved and executed by the cores until the computation is completed. To avoid concurrency issues, the *dependencies* between tasks need to be considered, i.e. if there are 2 tasks, where task A produces a result needed to compute task B, we say B is *dependent* on A, or A *unlocks* B. The dependencies form a Directed Acyclic Graph (DAG). The cores can execute tasks with no unsatisfied dependencies without having to worry about concurrency or synchronisation, as these are handled implicitly by the dependencies.

Task-Based Parallelism has two major advantages over traditional techniques. Firstly, since the tasks are assigned to the cores dynamically, the work is automatically load balanced, i.e. no core will run idle if there are still tasks to be computed. Secondly, the task dependencies avoid the necessity of any explicit synchronisation between the cores, i.e. no core will sit idly waiting on data being computed by another core if any other work is available.

Task-Based Parallelism does have some downsides though. Many codes need significant rewrites to be able to use Task-Based Parallelism, which can be too expensive or disruptive for a project to be able to complete. Furthermore, its use in massively parallel codes is relatively unexplored, and it is not straightforward to use with heterogeneous systems.

2.2.1 Cilk and Other Implicitly Declared Task Dependency Based Libraries

One of the first task-based libraries was Cilk [6], introduced in 1995. The Cilk model for Task-Based Parallelism consisted of a group of *procedures*, each of which is broken into a sequence of *tasks* (which they called threads). Each task is a *nonblocking* C function, meaning it can run to completion without waiting once it has been invoked. When a task completes it can *spawn* a number of child threads which begin new procedures, as shown in Figure 2.3. The spawned task can execute concurrently with its parent, as well as spawn additional children. As all of the tasks are nonblocking, no task may wait on data from any of its children so a *successor* task is spawned to receive the children's return values when they are produced. A task and its successors are considered to be part of the same procedure. Values sent from one task to another induce *data dependencies* between the tasks. This computation is known as a *spawn tree*. Cilk is an extension to the C programming language, and linked with a runtime library at compile time.

Cilk also introduced a work-stealing scheduler [7] for task-based computations. Usu-

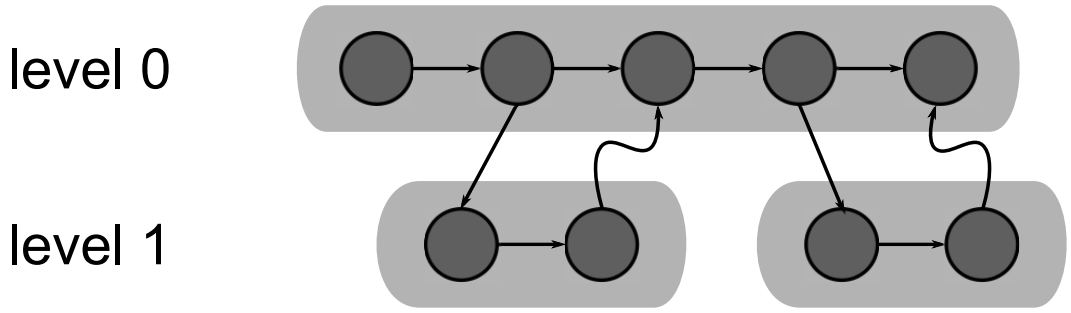


Figure 2.3: The Cilk model of multithreaded computation. Each task is represented by a circle, grouped into procedures. The downward edges correspond to the spawn of a child process. Horizontal edges represent the spawn of a successor. Curved, upward edges correspond to a data dependency. This figure is based on Figure 1 from [6]

ally task-based libraries have 1 task queue for each processor in the system. Without a work-stealing approach whenever a task queue becomes empty, the processor associated with that queue would idle. With work-stealing, instead the processor looks at the other queues to find a procedure to steal. In [7] they show that using a randomized work-stealing algorithm is provably efficient. In this method, each processor picks a random queue to attempt to steal a procedure from. If successful this procedure is executed by the processor. If not, the processor picks another random queue to attempt to steal a procedure from. This continues until all the computation is completed.

This work stealing setup is the basis for most of the scheduling in other tasking libraries, and leads to good load balancing on shared memory computers.

NVIDIAs Dynamic Parallelism [32] for GPUs has some similarities to Cilk. Rather than spawning tasks that are executed by a single thread, the CPU spawns a number of thread grids (i.e. kernels). These kernels can in turn launch additional kernels, and may block (using the `CudaDeviceSynchronise` function) to wait on child kernels. This allows a few methods of launching grids to enable various types of dependency structures. Dynamic Parallelism is only available on CUDA GPUs with compute capability of ≥ 3.5.

2.2.2 Libraries with Automated Dependency Generation

SMP Superscalar (SMPSs) [43], StarPU [4], QUARK [52] and KAAPI [16] allow the user to annotate functions as tasks, and specify the data input and outputs for each task function. They all use this information to automatically generate data dependencies between tasks. As of version 4.0, OpenMP can also compute dependencies in a similar manner [37].

SMP Superscalar

SMP Superscalar (SMPSs) was first released in 2007 [43]. SMPSs is written as a C language extension, allowing users to provide pragmas to the compiler to define functions that represent tasks, as well as the data requirements for those functions, such as input parameters, output parameters and inout parameters. The runtime library can then analyse the dependencies in the system at runtime and generate the dependencies automatically, meaning the programmer doesn't have to analyse the data flow in the computation, and allows the runtime to take advantage of the additional data when scheduling tasks.

The SMPSs model specifies 3 different types of dependencies: *Read after Write* (RaW) dependencies occur between a task that reads data and previous tasks to have written to that data, i.e. task B reads from data then task C writes to the data; *Write after Write* (WaW) dependencies are between tasks that write to a data location and previous tasks that also write to that data, and *Write after Read* (WaR) dependencies occur between a task that writes to data and a task that previously read that data. Examples of these dependency types are shown in Figure 2.4. The RaW and WaW dependencies are effectively the same data dependencies as specified in Cilk, while the WaR dependencies are similar to those implied by task spawning. The SMPSs library also uses variable renaming to help avoid WaR and WaW dependencies: If a task writes to an array, the runtime library creates a new array instead, and redirects all following reads of that array to the new array.

The SMPSs scheduler will execute dependent tasks on the same processor when possible. Since two dependent tasks must share some data in the SMPSs model, this should improve data locality.

As of version 3.0, OpenMP allows the user to specify tasks using the `#pragma omp task` directive. Initially, there was no way to specify dependencies, however in OpenMP 4.0 the `depend` keyword was added, allowing users to specify variable as *in*, *out* or *inout* for each task. Any task with an *in* dependence-type is dependent on all previously generated sibling tasks that reference any of the variables in an *out* or *inout* dependence-type. Any task with a *out* or *inout* dependence-type is dependent on all previously generated sibling tasks that reference any of the variables in an *in*, *out* or *inout* dependence-type. This has been extended further in OpenMP 4.5 to allow users to specify task priorities, though this is not implemented by any of the commonly used (gcc, intel) implementations. The draft of the OpenMP 5.0 standard currently allows reduction variables to be used inside task regions, and discussion is taking place about allowing concurrent (or commutative) dependencies.

OmpSs is an extension of SMPSs which aims to extend OpenMP with new directives to allow task-based parallelism similar to SMPSs. OmpSs extends to heterogeneous

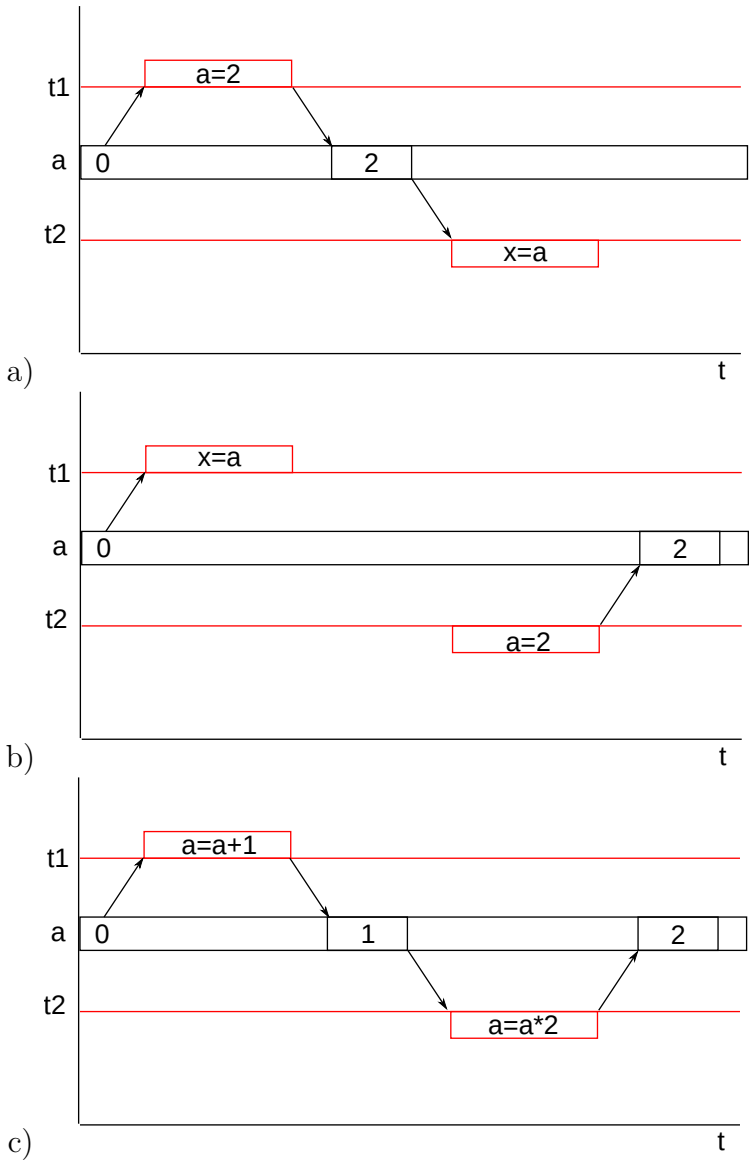


Figure 2.4: The three types of data dependency specified in SMPSs. Figure a shows thread 1 writing a value to the variable a, followed by thread 2 reading from it (RaW dependency). Figure b shows thread 1 reading from a before thread 2 writes to it (WaR dependency). Figure c shows thread 1 incrementing a, followed by thread 2 doubling its value (WaW dependency). These figures all assume the order in which these operations occur is critical for correct execution.

architectures by allowing the programmer to supply multiple functions for each type of task, where each function is designed for a different type of processor. The runtime system can then choose to execute certain tasks on accelerators or coprocessors when it thinks it would be better.

SMPSs and OmpSs both give the user a variety of scheduling policies to help tune the performance of their code.

StarPU

StarPU, first released in 2008 [4] uses a *codelet* model. A codelet is a kernel that may have multiple implementations of a task function, for CPUs, GPUs, and other coprocessors. The StarPU scheduler also allows *callback* functionality to help with synchronisation between different memories in the system. Once a task is completed, the scheduler executed the callback function supplied when the task is created, and can be used to enforce dependencies. Finally, the scheduler has some custom functionality to try to transfer data directly between some pairs of accelerators. StarPU has recently added commutative access to variables in tasks (version 1.2, released August 2016).

Similarly to SMPSs, StarPU uses real-time dependency generation based upon the data required for each task, as provided by the user. StarPU also has a variety of algorithms it uses for task scheduling, including a work stealing scheduler, a priority-aware scheduler, and a data transfer time-aware scheduler.

QUARK and DAGuE

QUARK (QUeueing And Runtime for Kernels) was developed for the PLASMA linear algebra library, and so many of the optimisations in the library are designed for linear algebra algorithms. These libraries do not require a compiler extension. QUARK requires the user to provide the runtime system with a series of tasks, and the data requirements for those tasks. The runtime system then schedules the tasks using one queue per thread. The queues are priority-aware, and the user can supply the library with task priorities when defining the tasks. Tasks are enqueued such that data reuse is maximised, and the scheduler uses a work-stealing approach to avoid threads idling.

As many linear algebra algorithms have $O(n^3)$ tasks, QUARK does not store the entire DAG in memory, instead using a sliding window of active tasks to reduce memory usage.

DAGuE (Directed Acyclic Graph Unified Environment) [8] is a more recent (2012) task-based library that uses a similar approach to QUARK for scheduling, in that it avoids storing the entire DAG in memory, and schedules to maximise data locality. DAGuE uses their own Job Data Flow (JDF) representation of the dependencies for

an algorithm. These are seperate files in which the user specifies the types of tasks in the system, the data required by the tasks and the functions used by each task in the computation. This is used by the compiler (and runtime system) to determine the data flow in the system and type of data associated with the various task types.

DAGuE also automatically handles data communication between MPI ranks based on the data dependencies between tasks, according to a data partitioning. All communication is handled by a separate thread, which takes commands from the compute threads and issues the data transfers using MPI's non-blocking point-to-point operations. When data must be sent, the sending node sends an activate message that contains information about the task that completed. Upon receiving this message, the destination node schedules the reception of the relevant data by evaluating the dependencies of the parent task, and then replies to the sending node to tell it to initiate data transfer. In addition to the data transfer, the system creates *control messages*, which are used to tell other nodes about the completion of various tasks.

DAGuE can also support heterogeneous architectures in a similar way to OMPSs and StarPU, where the CPU offloads tasks to accelerators when it deems appropriate.

2.2.3 Fully Declarative Task-Based Parallel Libraries

A third, less common approach to Task-Based Parallelism requires the user to specify the entire task graph manually This is much more demanding on the programmer, as they must fully understand the computation to be able to correctly specify the tasks and their dependencies. However, since the entire task graph is known at all points of the computation, the scheduler can potentially prioritise tasks along the critical path of the computation. Additionally, the scheduler can use this information to maximise memory-reuse.

This approach is used in Intel's Threaded Building Blocks (TBB) [39]. The programmer specifies C++ classes for each type of task they wish to use in their program. Each class overrides the `execute` method which is called by the executing thread. Each task object keeps track of a `successor` task, which can be set by the programmer, and the `refcount` which stores the number of children tasks. The programmer then spawns tasks to be executed, and uses the successor field plus the inbuilt synchronisation methods to control task dependencies. TBB does not prioritise tasks along the critical path of the computation however.

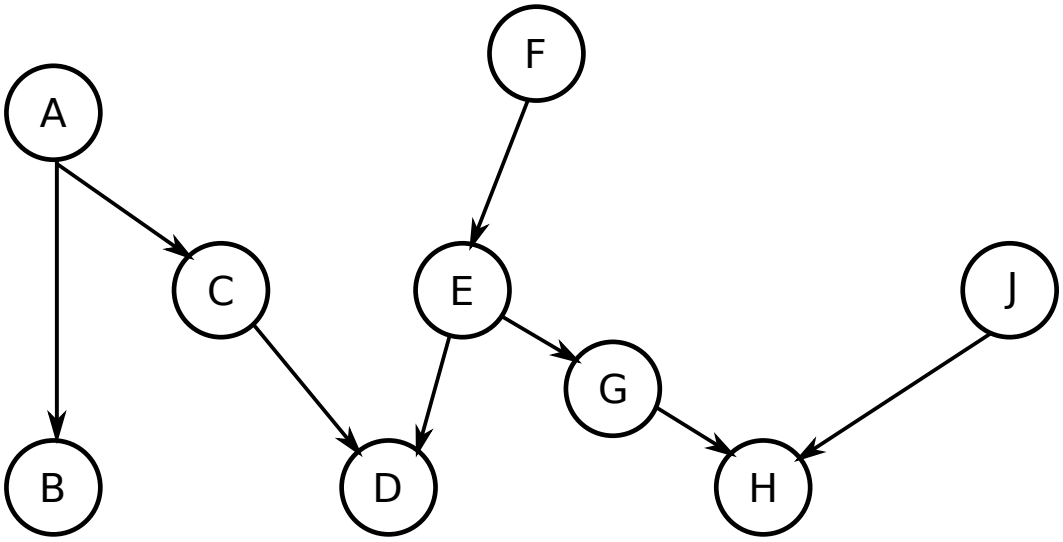


Figure 2.5: A diagram showing a difficult dependency structure to represent with a task-spawning model. Usually additional dependencies will be needed to handle this dependencies structure.

2.3 Strengths and Weaknesses of these Models

Most of the task-based libraries use either the task-spawn model (similar to Cilk), or automated dependency generation (similar to SMPSs). Each of these models have various strengths and weaknesses.

The spawn-based model is very easy to use, as the model works similarly to standard serial computation, with a few additional keywords to allow the spawning of tasks, and synchronisation of data from spawned tasks. This works well for simple dependency structures, however it can be difficult to accurately represent some dependency structures without the introduction of additional synchronisation points or dependencies, as shown in figure 2.5. To create the task hierarchy shown in figure 2.5, the program can start by spawning task D. Task D then spawns tasks E and C, and `waits` for the result. The C and E spawn tasks A and F respectively, and `waits` for their results. Once A completes it spawns task B. Once task E completes, it spawns task G, which in turn spawns task H. Before H can execute, it must spawn task J, and `wait` for it to complete. This ordering inherently introduces an extra dependency between task G and task J.

The automated dependency generation methods are a little more difficult to use - the user needs to provide the system information on each of the task functions regarding the utilisation of data, as well as providing the code to execute the tasks. From this, the library builds a dependency graph, and executes the code. One weakness of this automated dependency generation is that many of these libraries only store (or only generate) a section of the DAG in memory at any one time. While this saves on memory

usage, it has a cost with respect to scheduling. This sliding window approach can't know how many tasks will be spawned when a task is completed, so it is difficult to prioritise tasks on the critical path of the computation.

Task Conflicts

Most of these libraries are only aware of a single type of relationship between tasks, i.e. dependencies, which specify a strict ordering between two tasks. In many cases, the task ordering need not necessarily be this strict. If two tasks update some shared resource in an order-independent way then enforcing an ordering can be detrimental to scheduling, e.g. when accumulating forces on a particle, the order in which the forces are computed does not matter. The only requirement on such tasks is to avoid concurrent updating of that resource, so the two tasks must not execute simultaneously. Such a relationship between two tasks is called a *conflict*. In some of these previous libraries, conflicts are modeled as dependencies, however this enforces a pre-determined ordering on conflicting tasks. This restriction can severly limit the performance of a computation, as there are less options available. This can lead to sections of the computation being delayed unnecessarily. This has been noted in [31] and [2].

The QUARK scheduler allows conflicts to be modelled by explicitly marking dependencies as concurrent, whilst KAAPI and OMPSs allow marking access to certain variable as reductions, however this is only for a limited set of basic operations. StarPU has recently added commutative access to data between dependent tasks, resulting in a model similar to conflicts. The Cilk model cannot represent conflicts.

Chapter 3

The QuickSched Library

3.1 QuickSched

Most of the work in this thesis focuses on extending the QuickSched library [20] for Task-Based Parallelism to make use of GPUs and hybrid shared-distributed memory machines. QuickSched uses a fully declarative task model.

QuickSched's task scheduler consists of four main object types: *task*, *resource*, *scheduler*, and *queue*. The task and resource objects are used to model the computation, whilst the scheduler and queue objects manage how the tasks are executed.

3.1.1 Tasks and Resources

Tasks and *resources* are created by the user to model the computation. The tasks represent the units of work in the computation, and each one is required to have a *type*. When a task is executed, it will perform an operation that is defined by its type. During their computation, tasks are assumed to utilise a number of resources, and can either require exclusive access to that resource (known as `locking` a resource) or not (known as `using` a resources). The resources conceptually provide access to any resource in the system (such as a memory address, disk access, etc.). Locking a resource can be conceptually thought as read and write access to a memory address, while using a resource would be read-only access to that data. The resources cannot be directly associated with a memory location however, so the user must be careful when creating and storing the resources. If any pair of tasks lock the same resource, then those tasks conflict.

As well as declaring the tasks, resources, uses and locks, the user also needs to provide the scheduler with the task dependencies. If any Read after Write or Write after Read dependency exists in the computation, it has to be explicitly specified by the user before computation, unlike other libraries that can automatically generate

some dependencies.

Resources can also be declared as a hierarchical tree. When a resource is created, the user can provide a parent resource. When a task is executed, it must not only be able to lock all of its resources, but none of the parent or child tasks can be locked. To ensure this, each resource has both a lock and a *hold* counter. The hold counter is used to count the number of sub-resources that are currently locked. A resource can only be successfully locked if its hold counter is 0, and none of its parents are locked.

Quicksched tasks have a *cost* and *weight* associated with them. The cost of a task is a user-provided estimate of the task's runtime. The weight of a task is computed by the scheduler before execution, and is equal to the task's cost plus the cost of all of descendants of the task (a task's t *descendants* are all tasks u from which there is a path in the DAG from t to u).

3.1.2 Queues

The queues are stored as an array of task structures. Ideally, the tasks would be stored in a sorted list, from the highest to lowest priority, however this makes adding and removing tasks too expensive. Instead, the array is organized as a max-heap, i.e. where the k^{th} task has high priority than both the $2k + 1^{th}$ and $2k + 2^{th}$, and the task with the highest priority in the first position. Maintaining this heap structure is much cheaper ($O(log(n))$ vs $O(n)$ for a sorted list) when adding or removing tasks from the queue.

The downside of this structure is that there is no efficient way to traverse the tasks in priority order. Instead, the tasks are traversed as though it is a sorted list, and returns the first task that can be locked. While this doesn't necessarily result in optimal task selection, it proves to be sufficient for efficient task-based computations. Figure 3.1 shows an example of how the tasks may be organised using this heap.

The queues are locked using mutexes, which do not scale well. However, since QuickSched usually uses one queue per computational thread, contention will only occur when threads are work stealing, and this is rare enough that it will not have much effect on performance.

3.1.3 Scheduler

The scheduler object is used as the main interface to QuickSched, and as such contains the instances of the other three object types, as well as the number of tasks that have not yet been executed.

The main interface to begin a task-based computation is the `qsched_run` function. This requires a pointer to the scheduler to be executed, as well as a user-defined

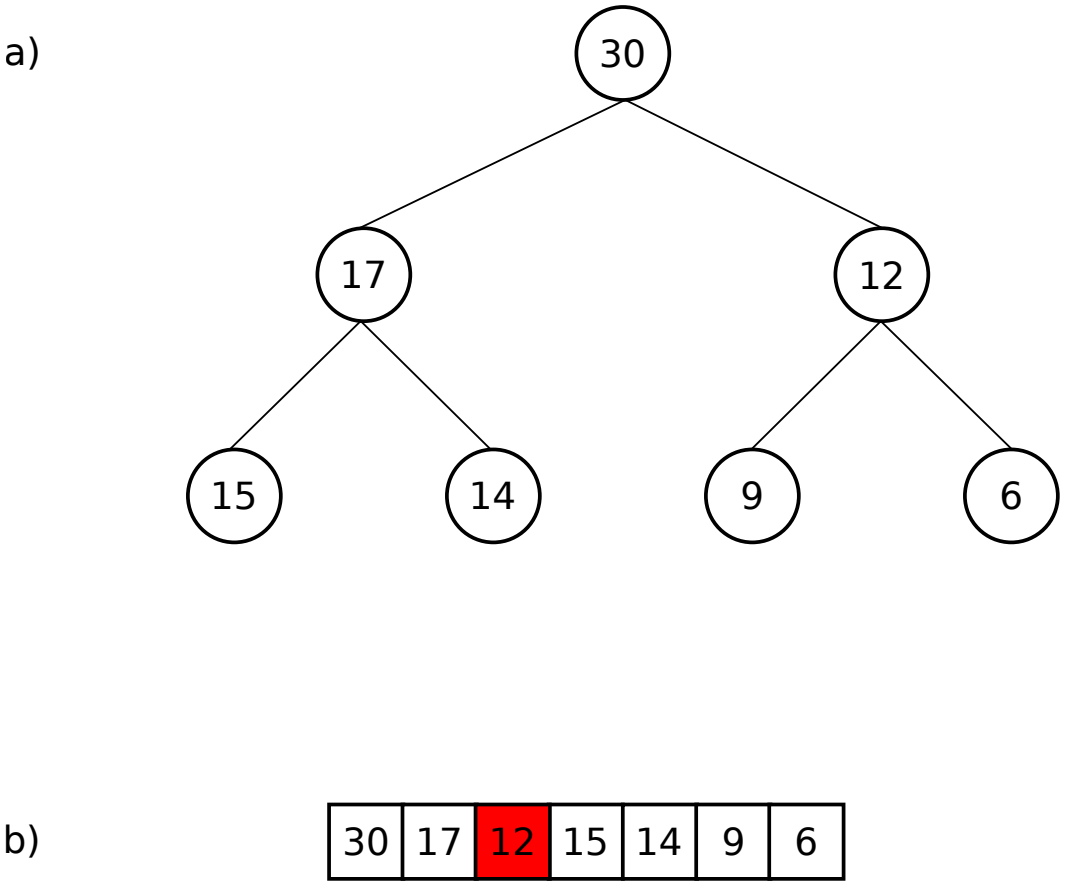


Figure 3.1: A diagram showing a task queue with task priorities. Figure a) shows the theoretical heap structure used to order the tasks, while figure b) shows how the array traversal can sometimes not follow the exact task priorities.

function pointer, which is defined as `typedef void(*qsched_funtype)(int,void*)`. This function is called whenever a thread in the system is ready to execute a task. The `int` is the `type` of the task to be executed, and the `void*` is the `data` associated with the task. This allows the programmer to specify the behaviour of the thread whenever a task is to be executed.

The QuickSched scheduler is capable of executing a task-based computation using both OpenMP or pthreads. With pthreads, each thread is created and start executing the task-based computation. With OpenMP, the entry call of the task-based computation is placed inside a parallel region, and the threads perform the same procedures as when using pthreads. Functions such as locks are controlled using preprocesser macros that wrap library calls or atomic operations.

3.2 Comparing Quicksched to Other Task-Based Systems

Table 3.2 shows a comparison of QuickSched to other commonly used task-based systems. OpenMP-like includes OpenMP 4.5 as well as the extensions available in projects such as OmpSs or StarPU. All of the systems highlighted here have similar basic ideas such as a work stealing scheduler, and many of them work on multiples architectures. QuickSched is the only system discussed that prioritises tasks that lie on the critical path, which can be done as it has the entire DAG in memory at the beginning of the computation. In Quicksched, each thread tries to prioritise tasks that access some of the same memory as the task it previously executed, which helps with memory reuse.

Task system	Manual DAG specification	Work Stealing Scheduler	Critical Path Scheduling	Target Architecture	Conflicts	Hierarchical Dependencies	Memory Reuse Scheduling
QuickSched	✓	✓	✓	CPU GPU MPI	✓	✓	✓
OpenMP-like	×	✓	×	CPU GPU OmpSs MPI	OmpSs StarPU	OmpSs	×
Cilk	✓	✓	×	CPU	×	✓	×
DAGuE	✓	✓	×	CPU MPI	×	✓	✓
Intel TBB	✓	✓	×	CPU	×	✓	×

Table 3.1: A table comparing the features available in different task-based runtimes. QuickSched has been designed to be applicable to a wide variety of problems, so it contains all of the features shown here, while other runtimes may lack certain other features.

Chapter 4

Example Problems

4.1 Molecular Dynamics

4.1.1 Introduction to Molecular Dynamics

Molecular Dynamics (MD) is a type of N-body problem used for studying the movement of atoms and molecules, usually in small biochemical systems. A simulation usually requires thousands to billions of iterations (*timesteps*). In each timestep the accelerations on the particles are computed then the particles are moved according to an integration scheme.

The accelerations of the particles are usually computed in 3 sections. The first section is the acceleration due to non-bonded interactions with neighbouring particles. The non-bonded section is expensive due to finding all the neighbours of each particle in the system, and this is the main cost of the simulation. The second section is the acceleration due to long-range electrostatic interactions. The final section is any interactions due to bonds between atoms. MD simulations usually use periodic boundary conditions, as they represent a small area of a larger tiled molecular system.

Non-bonded Interactions

The non-bonded interactions are the main computational cost of any molecular dynamics simulation, primarily the neighbour finding section of the interactions. During this step, the forces on each particle due to their interactions with each other atom in the system are computed. These forces are large between atoms close to each other, and tend towards 0 as the distance between atoms increases. To reduce the amount of computation, atom pairs further away than a cutoff distance (r_c) are ignored when computing these forces.

Short-Range Electrostatics

The short-range electrostatics are usually computed by the Smoothed Particle-Mesh Ewald (PME) algorithm [14], Particle-Particle/Particle Mesh (P^3M) [27], or by multi-level summation [42]. These forces are computed as:

$$\frac{\partial^2 x_i}{\partial t^2} = \frac{1}{m_i} \sum_{r_{ij}<r_c} -\frac{\partial v_{ij}(r_{ij})}{\partial r_{ij}}, \quad i = 1, \ldots, N$$

where x_i and m_i are the position and mass of the ith particle, and r_{ij} is the Euclidean distance between the ith and jth particles. The potential function $v_{ij}(\cdot)$ is dependent on the particle types and is truncated at the cutoff distance r_c, beyond which the interactions are considered insignificant.

Long-Range Electrostatics

The long-range electrostatics can be computed using the Particle-Mesh Ewald method [11]. To compute the long-range electrostatics, the particles in the system are interpolated onto a grid. The grid is transformed into fourier space using a FFT, and then the forces are computed. The sum of the long and short range electrostatics on the particles should sum to the periodic electrostatic potential. Other methods (such as Fast Multipole methods or Partial Differential Equation solvers) can also be used to compute the forces due to long-range electrostatics.

Bonded Interactions

The bonded interaction section of a MD simulation ensures any forces in the system do not break any of the bond, angle, dihedral or exclusion interactions specified in the simulation input. Exclusion is the negative of the non-bonded interaction, as this should not be computed for particles that are bonded [1].

4.1.2 Algorithms for Neighbour Finding

In MD, the non-bonded interactions involve each particle pair that are within the cutoff radius (r_c) of each other, known as *neighbours*. Neighbour finding is one of the most computationally intensive sections of the simulation. Many different algorithms can be used to compute each particle's neighbours in MD simulations. In this section I will

[1] Since particles that are bonded may be very close to each other, the potential for the non-bonded interactions is flattened off if particles are very close to avoid the potential becoming very large and causing large inaccuracies in the system

detail the $O(n^2)$ algorithm, Verlet lists, cell lists and the extensions of cell lists (sorted cell lists and pseudo-Verlet lists) used in this thesis.

The $O(n^2)$ algorithm

The simplest algorithm for neighbour finding is a double for loop over the particles, and checking if each particle is a neighbour of each other particle in the space. This algorithm is prohibitively slow for even small particle systems.

Verlet Lists

Verlet Lists[30] are one of the most popular algorithms for neighbour finding in particle simulations. In the first step, the all-pair interactions are computed, and for each particle a list containing all of its neighbours (*i.e.* all particles within r_c) is stored. In the next step we can reuse the same verlet lists to save time provided we have a shell, *i.e.* by storing all particles within $r_c + 2d$, and only rebuilding the verlet list when any particle has moved further than d from the initial position.

Verlet lists give better performance than the naive $O(n^2)$ algorithm (Verlet lists take $O(n^{\frac{5}{3}})$ runtime), and many codes use them for neighbour finding. However, building the verlet lists initially is very expensive, and they often just store the particle indices, so are not usually cache-friendly, as looping through the neighbours will involve loading most particles directly from main memory.

Cell Lists

The cell list algorithm for the short-range electrostatic involves dividing the space into a series of cells of size r_{cell}. If $r_{cell} \geq r_c$ then to find all neighbours of a particle, you only need to look inside the cell containing the particle and the 26 neighbouring cells (in 3 dimensions). The algorithm to then compute the short-range electrostatic between all particles in two neighbouring cells is:

1: **for all** $p_i \in c_i$ **do**
2: **for all** $p_j \in c_j$ **do**
3: $r^2 \leftarrow |\mathbf{x}[p_i] - \mathbf{x}[p_j]|^2$
4: **if** $r^2 \leq r_c^2$ **then**
5: Compute interaction between p_i and p_j
6: **end if**
7: **end for**
8: **end for**

This algorithm has a runtime of $O(n)$ with respect to the number of particles in the system, as the inner loop doesn't depend on the number of particles, only the

Type of neighbour	Percentage of particles in-range
Face-sharing pair	50.0%
Edge-sharing pair	16.2%
Corner-sharing pair	3.62%

Table 4.1: Average percentage of particles in a neighbouring cell that are in range of any given particle in a cell.

distribution of them in the space. This means it has theoretically better runtime than constructing Verlet lists.

However, for uniform particle distributions the percentage of particles in neighbouring cells that are actually within r_c of any given particle is actually quite low, as shown in Table 4.1. This leads to large numbers of misses when searching for neighbours (i.e the `if` statement in line 4 rarely triggers), which results in a net loss of performance compared to a Verlet list. This is also apparent from the volume being checked around the central cell. If $r_{cell} = r_c$, then the volume searched for neighbours is $27r_c^3$ (volume of the cube around a particle), which is 20.75 times higher than the ideal $\frac{4}{3}\pi r_c^3$ (the volume of the neighbourhood sphere for a particle) [51]. Figure 4.1 shows how the algorithm works for a set of particles in 2 dimensions.

Halving the size of r_{cell} dramatically reduces the searched volume to $(2.5)^3 r_c^3$, even though non-adjacent cells need to be searched for neighbours.

In [3] it was shown that that reducing r_{cell} to contain only a single particle in any cell resulted in better performance than the conventional algorithm where $r_{cell} = r_c$. However, [51] claims this method is not as fast as a Verlet List approach. They suggest an improvement to the cell list approach to do a "weak" sort of the particles in the system, meaning particles in the same cell are close to each other in memory, which results in an improvement in cache hit rate. However, they still use a mix of the cell list and Verlet list approach.

Sorted Cell Lists

[18] improves cell lists by sorting the particles in each cell along each of the 26 vectors between the cell's centre and each of its neighbours' centres.

To sort the particles along the cell axes ($\vec{r}$), each particle is first projected onto $\vec{r}$ using the equation $\mathbf{x}[p] \cdot (\frac{\vec{r}}{||\vec{r}||})$, where $\mathbf{x}[p]$ is the position of a particle p. The author proves that any pair of particles that have a pairwise distance larger than r_c along the axis shared between the cells that contain them are guaranteed to be further apart than r_c. Once we have a projection of the particles onto each of the cell axes, we create

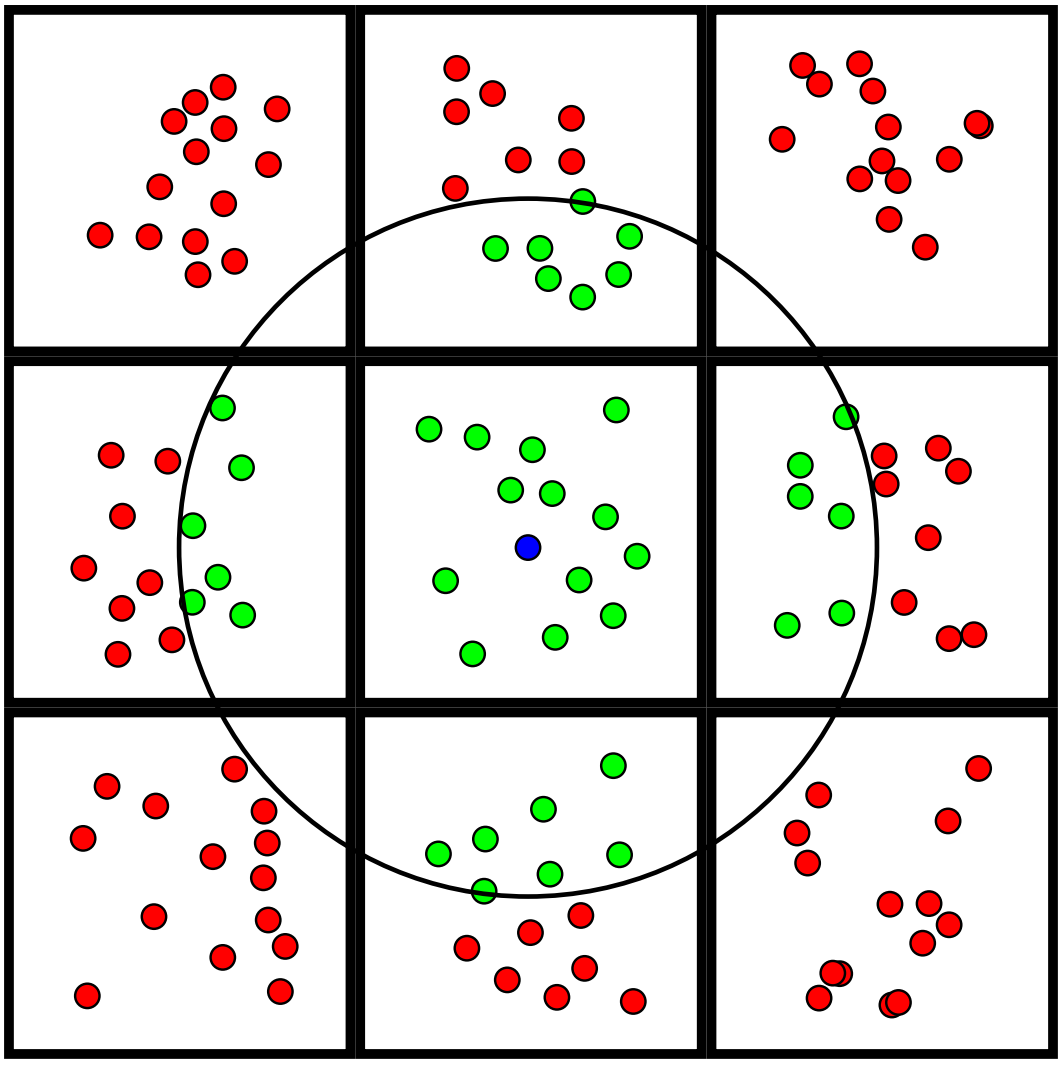


Figure 4.1: An illustration of the cell list approach. The circle shows the neighbourhood (r_c) around the particle in blue. All of the particles in green are neighbours of the blue particle, whilst the red particles are not neighbours, but would be checked by the neighbour finding algorithm.

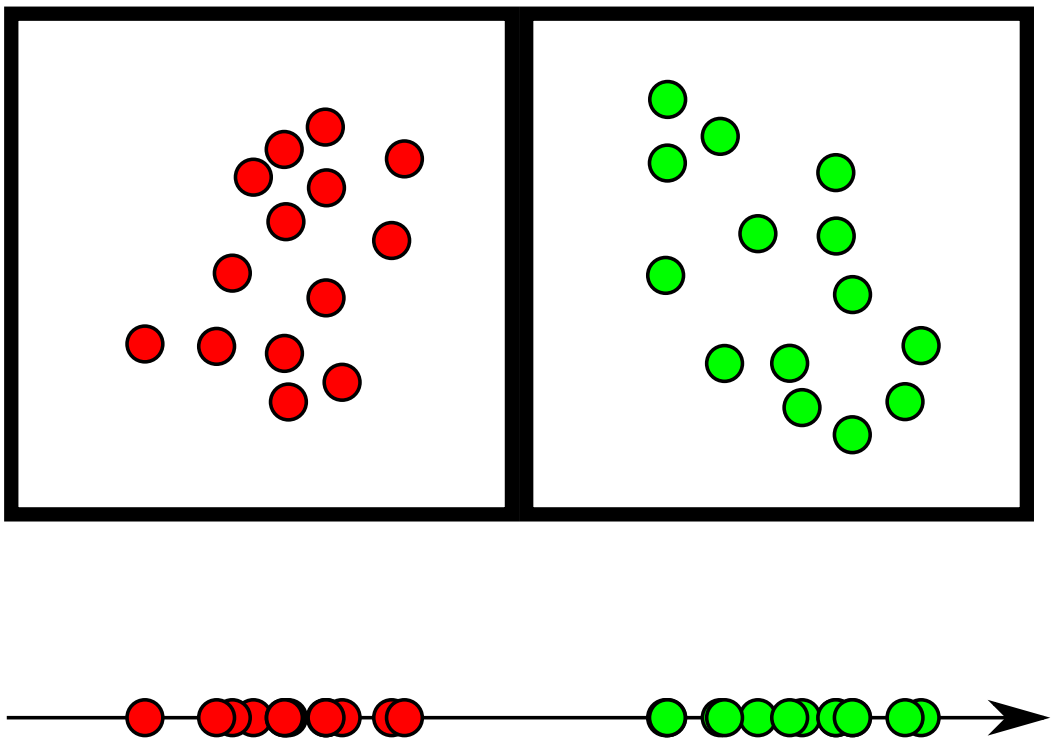


Figure 4.2: In the sorted cell lists, the particles are sorted along the axis between the two cells. When neighbour finding, any particles that have pairwise distance larger than r_c along the cell axis are also guaranteed to be further apart than r_c

sorted lists for each axes using any sort algorithm. Figure 4.2 shows how the result of this sorting for a pair of cells in 2 dimensions.

For any pair of neighbouring cells, c_i and c_j, we can calculate the forces on the particles by making use of the sorted lists (S_i and S_j) as follows:

$last_j \leftarrow count_j$
for i from $count_i$ to 0 **do**
  for j from 0 to $last_j$ **do**
    $d \leftarrow P_i[S_i[i]] - P_j[S_j[j]]$
    if $d \leq r_c$ **then**
      $p_i \leftarrow c_i[S_i[i]]$
      $p_j \leftarrow c_j[S_j[j]]$
      $r^2 = |\mathbf{x}[p_i] - \mathbf{x}[p_j]|^2$
      if $r^2 \leq r_c^2$ **then**
        Compute interaction between p_i and p_j
      end if
    else
      $last\_j \leftarrow j$
    end if

end for
end for

where P_i and P_j contain the projections of the particles onto $\vec{r}$ for cells c_i and c_j respectively, and requires that values of P_i are all less positive than the values of P_j.

This algorithm results in far fewer spurious neighbours being found (around 59% of particle pairs found by this algorithm will be neighbours, wheras 27% will be when using unsorted cells of size $\frac{1}{2}r_c$), especially as the majority of cell-pair interactions do not share a face, and is faster than any of the other cell list variants [18].

Pseudo-Verlet Lists

One drawback of cell lists is the need keep an up-to-date record of which particle is in which cell, and making sure the particles in each cell are contiguous in memory. If we use sorted cell lists (which have been shown to have the best performance) we also need to re-sort the cells along all 26 axis in every timestep which is expensive.

One alternative is to use Pseudo-Verlet Lists[19]. Pseudo-Verlet lists are similar in concept to cell lists, however the size of the cells (r_{cell}) is greater than r_c. The usual drawback of using larger cells to find neighbourhoods (*i.e.* searching a larger volume than necessary) is negated by the sorted cell list approach. Figure 4.3 shows how the area searched and areas containing neighbours differ for a particle in 2 dimensions. Until the red area is not completely contained inside the grey region for any particle, the cells do not need to be re-sorted.

We can then modify the algorithm for sorted interactions, by changing the projection distance check from $d \leq r_c$ (as above) to $d \leq r_m$, where r_m equals r_c plus twice the maximum displacement of any particle since the last regrid. This allows us to re-use the sorted particle indices in more than one timestep.

The criteria for regridding is then to regrid if any particle has displacement more than $\frac{r_{cell}-r_c}{2}$ (known as the skin radius r_s), as neighbours will always be found until a particle has moved more than $\frac{r_s}{2}$.

4.1.3 The Task-Based Algorithm in `mdcore`.

All of the work with MD in this thesis is a part of `mdcore` [23]. `mdcore` is an open source library that supports a variety of architectures including shared memory CPU, hybrid shared-distributed memory CPUs, and CUDA GPGPU processors. `mdcore` uses Task-Based Parallelism on all of the architectures and implements both sorted cell list and pseudo-Verlet list algorithms within the task parallel scheme.

During each timestep in an MD simulation, we have to compute the forces due to non-bonded interactions, electrostatics, and the bonded interactions in the system.This

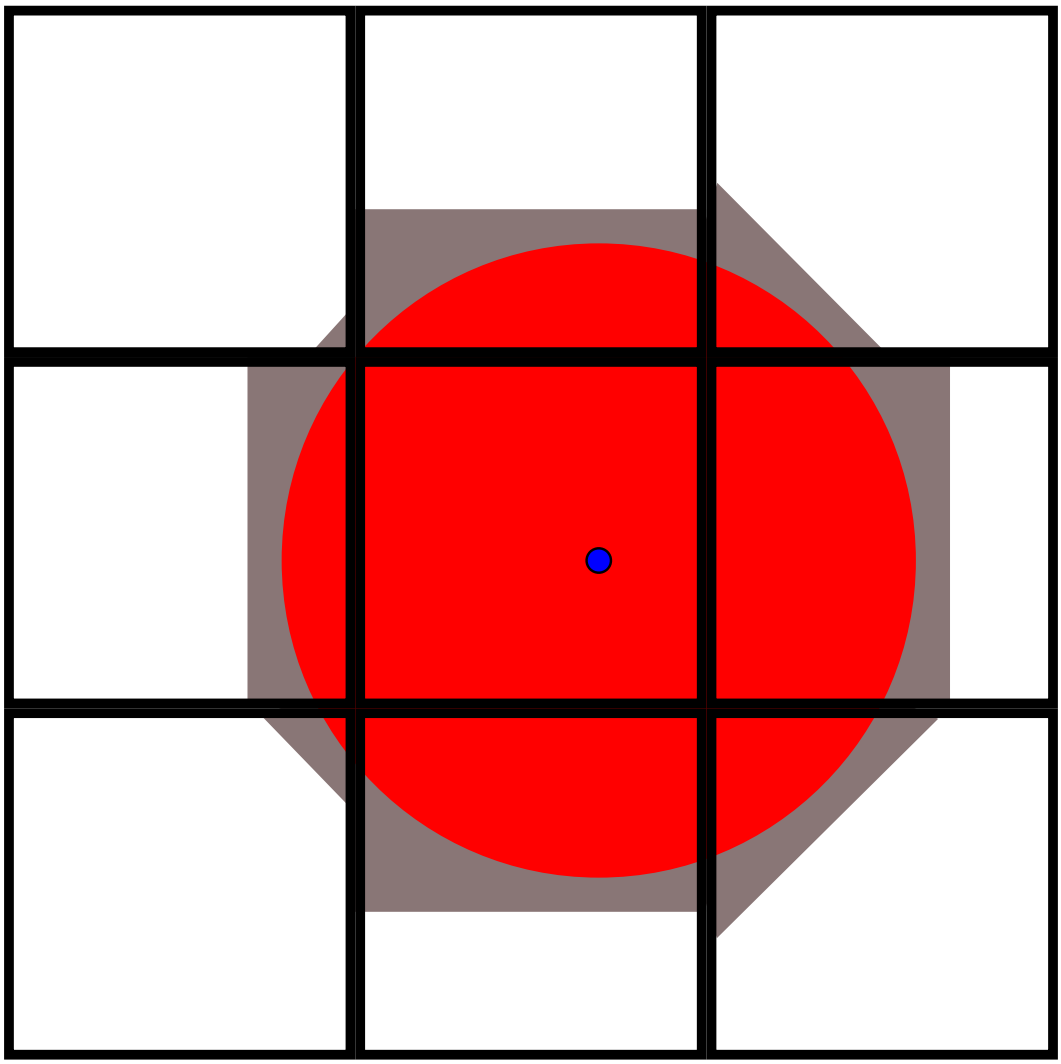


Figure 4.3: The psuedo-verlet list algorithm checks the area inside the grey polygon for neighbours, as the particles are sorted along the cell axes, and these are only computed occasionally. The red area is r_c distance around the particle. The advantage of the algorithm is that the cells do not need to be resorted until a particle moves more than r_s. At this point, the neighbourhood of the particle (the red circle) is no longer fully contained by the searched area (the grey polygon).

work has no requirement on any other data computed during the timestep, so there are no dependencies between tasks in the classical MD simulation. As `mdcore` uses either sorted cell lists or pseudo-Verlet lists, there is a requirement to sort the particles before we can compute the non-bonded interactions. Since 95% of the work is done to compute the non-bonded interactions, we turn this section into tasks. The remainder (such as constraint satisfaction, SHAKE, thermostat etc.) of the work is done in serial.

The task algorithm to compute the non-bonded and bonded interactions requires 3 types of tasks:

1. Sort tasks. The sort tasks sort the particle indices for a single cell on all 26 axes. We use symmetry to reduce this to only require 13 sort tasks per cell.

2. Non-bonded interaction tasks. These are actually further subdivided into *pair* and *self* tasks, depending on whether the task computes forces on a cell pair or a single cell. The pair tasks depend on the sort tasks for both of the cells involved.

3. Bonded tasks. These are also subdivided into bond, angle, dihedral and exclusion tasks.

As well as the dependency between sort tasks and non-bonded tasks, any pair of non-bonded tasks that share a cell *conflict*, *i.e.* they cannot be computed at the same time as they write to the same variables, which would cause a race condition.

A diagram showing the tasks and dependencies for a simulation of four cells is show in figure 4.4.

`mdcore` does not use QuickSched, but its task-based scheme was a precursor to QuickSched.

4.2 Smoothed Particle Hydrodynamics

4.2.1 Introduction to Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) is a particle-based method that is used to model compressible fluid flow[17]. It is commonly used in cosmological simulations to model the behaviour of the gas in the universe [46].

In a SPH simulation, each particle p_i has a position $\mathbf{x_i}$, velocity $\mathbf{v_i}$, internal energy u_i, mass m_i, and a smoothing length h_i. Note that unlike in MD, the smoothing length[2] is not fixed for all particles, and can change during the lifetime of the simulation for any given particle, which makes this more difficult to solve than MD as neighbour finding

[2]equivalent to the cutoff radius in MD

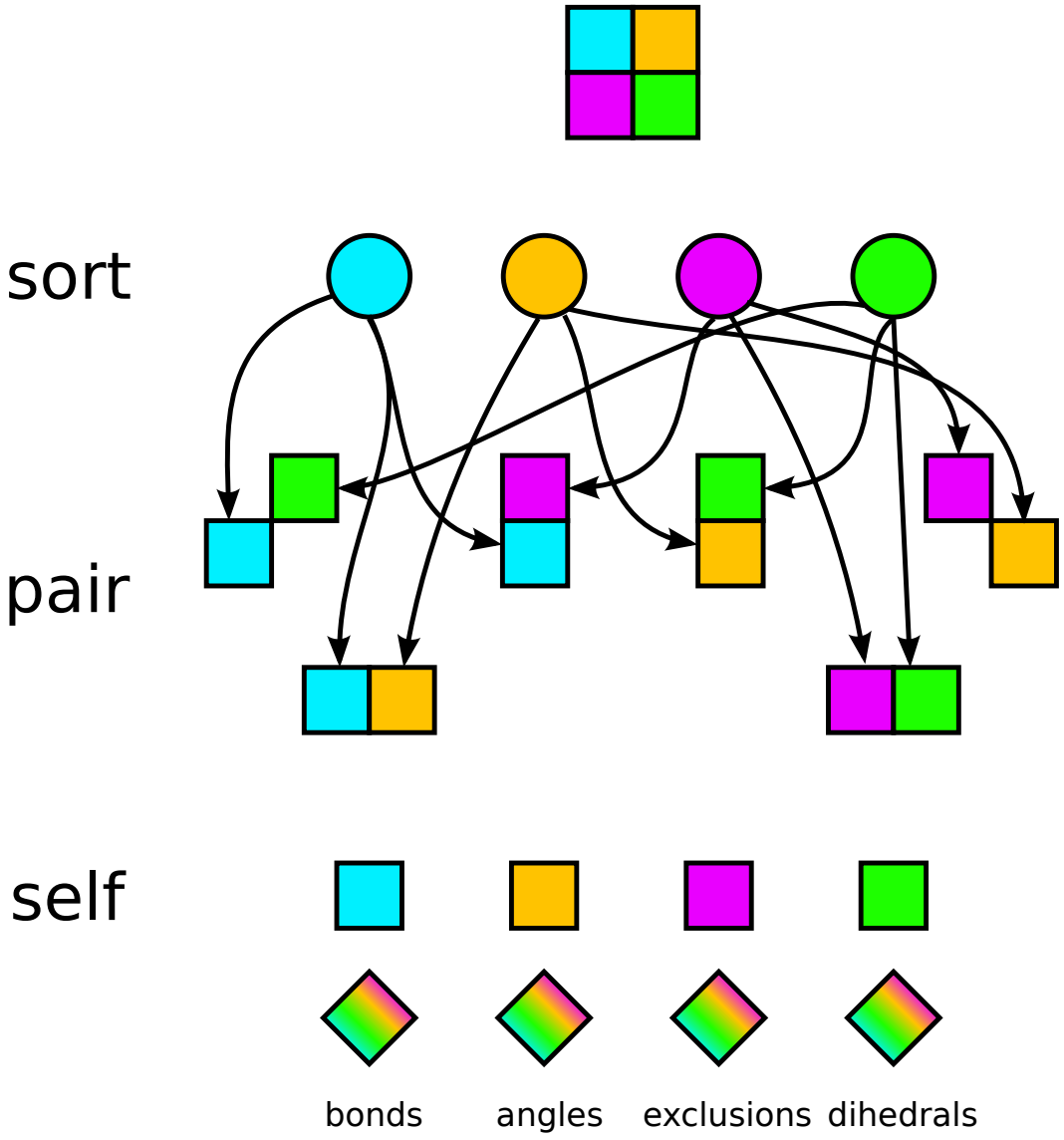


Figure 4.4: A diagram showing the task DAG for an example MD simulation of 4 cells. As well as the dependencies shown, no tasks which share a cell of the same colour can be done simultaneously, as they conflict.

is more difficult. The smoothing length is the interaction radius of the particle, and is chosen so all particles have a similar number of neighbours.

This explanation of SPH is adapted from [21].

The particles are used to interpolate any quantity Q at any point in the space as a weighted sum over the particles:

$$Q(r) = \sum_i m_i \frac{Q_i}{\rho_i} W(||\mathbf{r} - \mathbf{r_i}||, h), \tag{4.1}$$

where Q_i is the quantity at the ith particle and $W(r, h)$ is the *smoothing function*. In our code SWIFT [24], which implements SPH, $W(r, h)$ is defined using a piecewise polynomial:

$$W(r,h) = \frac{8}{\pi h^3} \begin{cases} 1 - 6(\frac{r}{h})^2 + 6(\frac{r}{h})^3, & 0 < \frac{r}{h} \leq \frac{1}{2}, \\ 2(1 - \frac{r}{h})^3, & \frac{1}{2} < \frac{r}{h} \leq 1, \\ 0, & \frac{r}{h} > 1. \end{cases} \tag{4.2}$$

The particle density ρ_i used in eq. 4.1 is computed as:

$$\rho_i = \sum_{j, r_{i,j} < h_i} m_j W(r_{i,j}, h_i), \tag{4.3}$$

where $r_{i,j} = ||\mathbf{r_i} - \mathbf{r_j}||$ is the distance between particles p_i and p_j. The smoothing length h_i of each particle is chosen such that the weighted number of neighbours N_{ngb} is kept roughly constant.

$$N_{ngb} = \frac{4}{3}\pi h_i^3 \sum_j W(r_{i,j}, h_i) \tag{4.4}$$

This is achieved by applying a single step of Newton iteration to solve eq. 4.4 for h_i in every time-step, where the derivative $\partial N_{ngb}/\partial h_i$ is computed alongside eq. 4.3. If the number of neighbours for any particle strays too far from the expected value, the smoothing length and number of neighbours for that particle is recomputed until a valid N_{ngb} is obtained.

Once the densities have been computed, the time derivatives of the velocity, internal energy and smoothing length (which all require ρ_i) are computed as follows:

$$\frac{dv_i}{dt} = \sum_{j, r_{i,j} < \hat{h}_{i,j}} m_j \left[\frac{P_i}{\Omega_i \rho_i^2} \nabla_r W(r_{i,j}, h_i) + \frac{P_j}{\Omega_j \rho_j^2} \nabla_r W(r_{i,j}, h_j) \right], \tag{4.5}$$

$$\frac{du_i}{dt} = \frac{P_i}{\Omega_i \rho_i^2} \sum_{j, r_{i,j} < h_i} m_j (\mathbf{v_i} - \mathbf{v_j}) \cdot \nabla_r W(r_{i,j}, h_i), \tag{4.6}$$

where $\hat{h}_{i,j} = \max(h_i, h_j)$, and the particle pressure $P_i = \rho_i u_i (\gamma - 1)$ and correction

term $\Omega_i = 1 + \frac{h_i}{3\rho_i}\frac{\partial \rho}{\partial h}$. The polytropic index γ is usually set to $\frac{5}{3}$.

The computations in eq. 4.3, eq. 4.5, and eq. 4.6 involve finding all pairs of particles within range of each other. Any particle p_j is in *within range* of a particle p_i if the distance between p_i and p_j is less than or equal to the smoothing distance h_i of particle p_i, as is done in eq. 4.3. Since in compressible SPH the smoothing lengths vary between particles this relationship is not symmetric. If the distance between p_i and p_j is less than $\max(h_i, h_j)$ then the particles are within range *of each other*.

Each timestep of the SPH computation is computed in three distinct stages that are evaluated separately:

1. *Density* computation: For each particle p_i, loop over all particles p_j within range of of p_i (*i.e.* $r_{i,j} \leq h_i$), and evaluate eq. 4.3. This computes the densities and smoothing lengths of all particles.

2. *Force* computation: For each particle p_i loop over all particles p_j that are within range of each other (*i.e.* $r_{i,j} < \max(h_i, h_j)$), and evaluate eq. 4.5 and eq. 4.6. This computes the forces that act on each particle.

3. *Timestepping*: For each particle, move the particle forward in time, using the forces on the particle to update the acceleration of the particle and move it in space.

4.2.2 The SWIFT Task-Based Algorithm for SPH

A task-based algorithm for SPH was introduced in [24] and [21]. It makes use of cell-list type structures for finding the neighbours. The algorithm is more complex than for MD due to variable smoothing lengths, so we can't use a single cell size to efficiently find all the neighbours of every particle.

At the beginning of the simulation (and whenever the system needs to be repartitioned) the space is partitioned into an initial grid of cells. The edge length of these cells is at least as large as the maximum smoothing length of any particle, h_{max}. Each of these cells is divided into 8 child cells if all of the particles contained within it have a smoothing length of less than half the cell edge length. This process is repeated until no cell in the system violates this constraint. In the SWIFT implementation a cell is also not further divided if it contains less than a certain number of particles (n_{min}), however this is not important to understand the algorithm. Figure 4.5 shows an example particle distribution, and how it may be divided into cells (note that in two dimensions, the cells are only split into 4 subcells).

This decomposition means all neighbours of any particle p_i will be located in either the smallest cell containing p_i, or any direct neighbours of the cell containing p_i.

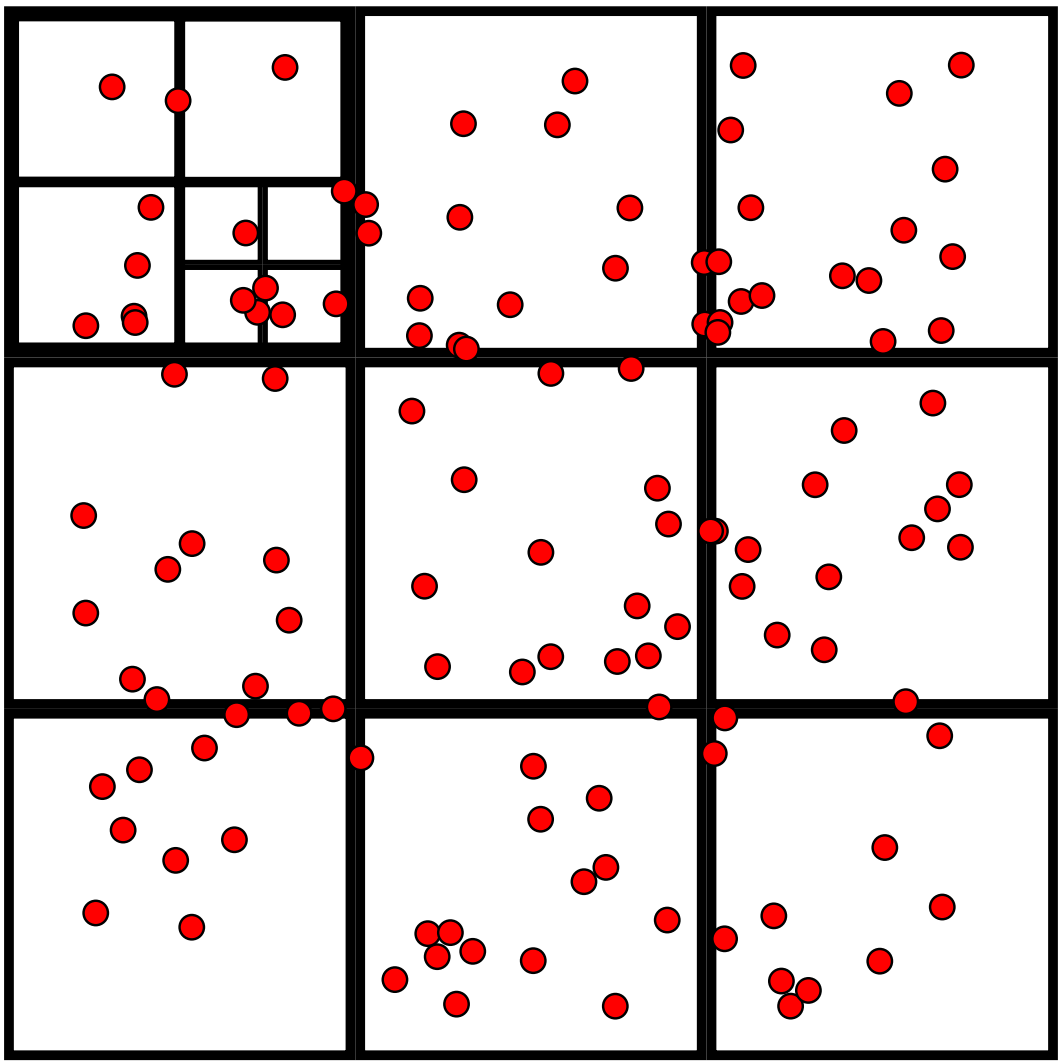


Figure 4.5: A diagram showing the cell partitioning for a 2D simulation. The top-left cell needed dividing into smaller sub-cells as all of the particles contained within the cell had small smoothing lengths.

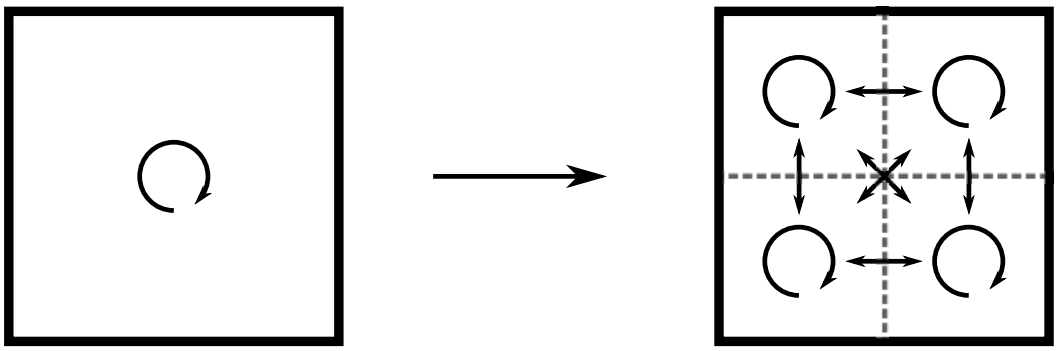


Figure 4.6: If all the particles within a cell have small smoothing lengths, the cell can be split, and its self interaction replaced by the self and pair interactions of its subcells.

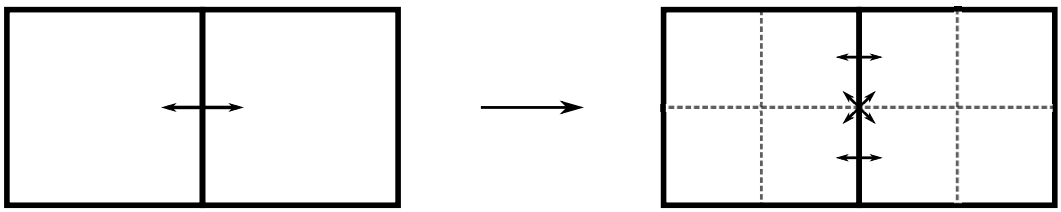


Figure 4.7: If all the particles in a pair of interacting cells have a smoothing length less than or equal to half of the cell edge length, both cells can be split. Instead of the pair interaction of the two large cells, the pair interactions of the subcells neighbouring across the interface are computed. This ensures distant particles that never interact do not need to be checked in the neighbour search.

For both the density and force computations cell-self and cell-pair tasks are constructed according to these rules:

- For any leaf cell (a cell that is not split), create a cell-self task for both density and force.
- For any pair of neighbouring cells of the same size, create a cell-pair task for both density and force provided one or both of the cells is not a leaf.

In addition, integrator tasks are created for each leaf that move the particles at the end of each timestep.

We construct the tasks by creating cell pairs at the highest level, and recursively splitting as shown in Figures 4.6 and 4.7.

Splitting cells can reduce the volume searched for neighbours for a particle, which should lead to improvements in neighbour finding. To speed up the neighbour finding, we used the cell sorting method described in section 4.1.2. One useful improvement for the sorting of particles is that once we have sorted the subcells of a cell, we can construct the sorted indices of the cell by merging the indices of its eight subcells (shown in 2D in Figure 4.8). This can significantly reduce the cost of sorting large cells containing many particles. Using this improvement, we can create a task-based hierarchical sort

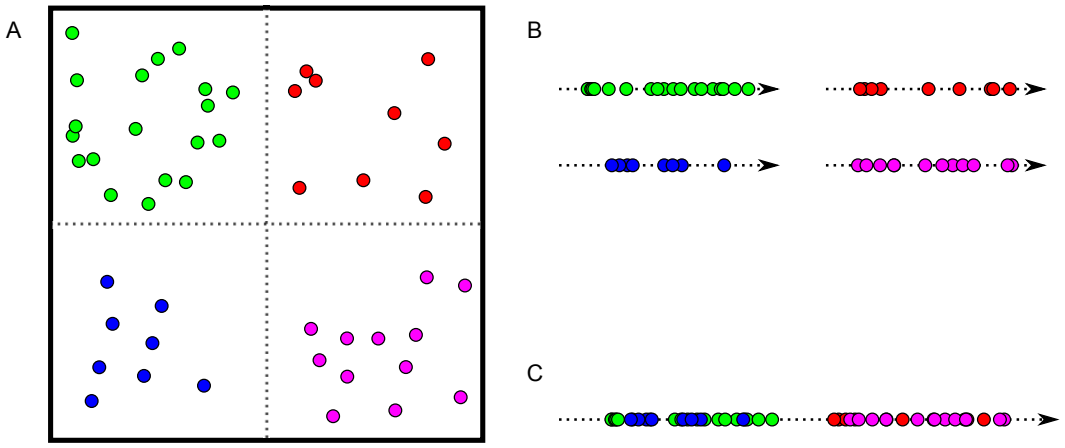


Figure 4.8: Hierarchical sorting of a large split cell (A). First each subcell is sorted along their individual cell axis (B), then the subcells are shifted and merged to produce the sorting for the large cell (C).

setup. We create a sort task for every cell in the system with dependencies depending on whether the cell is split or not:

- If the cell is a leaf cell, then it has no dependencies.
- If the cell is split into smaller cells, then its sort task dependent on the sort tasks of all of its children.

When executed, only the leaf cells actually need to sort the particles. Larger cells can merge the result of the sorting of their children.

This gives us the initial set of dependencies required for SPH:

1. The sort tasks of the leaf tasks which have no dependencies.
2. The sort tasks of the larger cells, which are dependent on the sort tasks of each of their children.
3. The density tasks, which are dependent on the sort tasks of the cells they act upon.
4. The force tasks, which are dependent all of the density tasks of all of the cells (and the density tasks of all the children of those cells) they act upon.
5. The integrator tasks, which are dependent on all of the force tasks of the cell they act upon. These are created only for leaf cells.

In order to avoid the large number of dependencies between the density and force tasks of each cell, we create ghost tasks. The ghost task for a given cell depends on all of the density tasks for that cell, and the force tasks are made dependent on the ghost

task. Additionally, the ghost task can do some computation that is required between the two operations, such as checking the accuracy of N_{ngb} for all of the particles in the cell, and correcting any that were computed incorrectly during the density tasks. This addition results in the task structure as shown in Figure 4.9.

As in MD, any pair of tasks that act on a shared cell or one of its subcells cannot be executed at the same time, so they conflict. This is solved using hierarchical locks, i.e. a cell can only be locked if none of its subcells or parent cells are locked.

SWIFT also uses the pseudo-Verlet list ideas introduced in [19] to avoid sorting the particles in every timestep (discussed in section 4.1.2).

4.3 Tiled QR Decomposition

4.3.1 The Tiled QR Decomposition

The QR decomposition is a linear algebra operation that decomposes a matrix A into two matrices Q and R such that $A = QR$ where Q is orthogonal and R is upper triangular. It is commonly used to solve the Least Squares Problem.

The problem is often solved using Gram-Schmidt or successive column-wise Householder Reflectors, but these do no parallelise well. To create a parallel version, [9] created a tile version of the algorithm, known as the tiled QR decomposition.

The algorithm divides the matrix into $m \times n$ square tiles of size $K \times K$, and performs $k = \min(m, n)$ sweeps over the matrix. Each sweep performs a series of operations on a decreasing sized submatrix of A, starting from the tile at position (k, k). The standard tiled-QR decomposition also uses Householder reflections [28]. By doing this, Q and R fit into a matrix the size of A, plus a small τ matrix. The τ matrix is required to recover Q from H (the Householder matrix).

The task-based algorithm uses four types of task, usually referred to by their LAPACK names: xGEQRF, xLARFT, xTSQRF, xSSRFT. The x in the names refers to whether the operation uses single precision (x=S), double precision (x=D), complex (x=C) or complex-double (x=Z). In this thesis I only consider the operations on single or double precision floating point numbers. The task structure (including dependencies) is shown in table 4.2. There are no conflicts between tasks in the tiled QR decomposition. An example of how the computation progresses on a 4×4 matrix is shown in Figure 4.10.

4.3.2 Implementing the Tiled QR Decomposition

The tiled QR decomposition has not been widely implemented, as most uses of the algorithm use library implementations such as LAPACK or PLASMA, though it is often

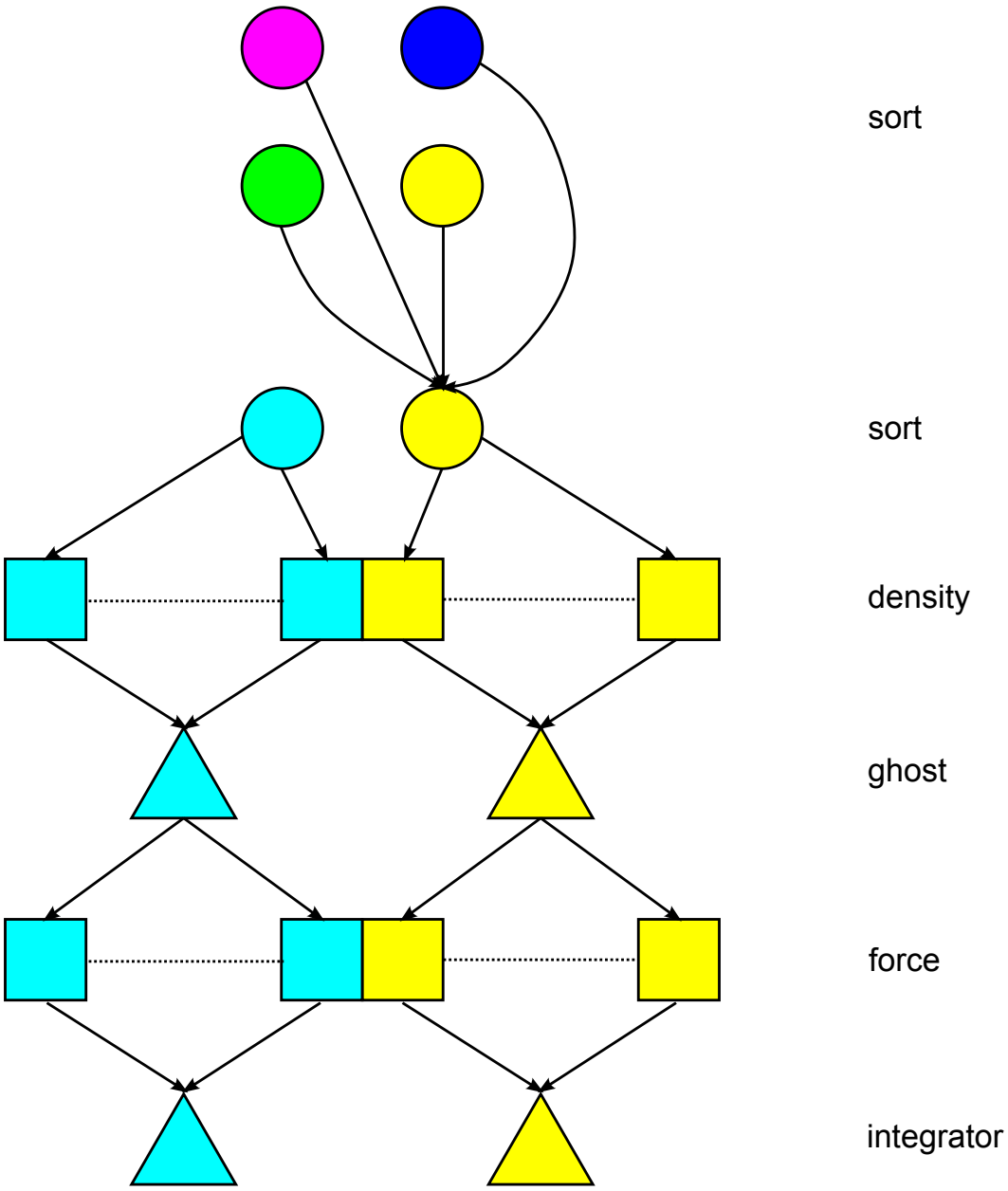


Figure 4.9: The task graph used for computing SPH. Arrows between tasks represent dependencies between different task types. Dotted lines between tasks indicate conflicts between tasks on the same data.

Task	where	depends on task(s)	locks tile(s)
SGEQRF	$i = j = k$	$(i, j, k-1)$	(i, j)
SLARFT	$i = k,\ j > k$	$(i, j, k-1)$, (k, k, k)	(i, j)
STSQRF	$i > k,\ j = k$	$(i, j, k-1)$, $(i-1, j, k)$	(i, j), (j, j)
SSSRFT	$i > k,\ j > k$	$(i, j, k-1)$, $(i-1, j, k)$, (i, k, k)	(i, j)

Table 4.2: The task structure for the Tiled QR-decomposition.

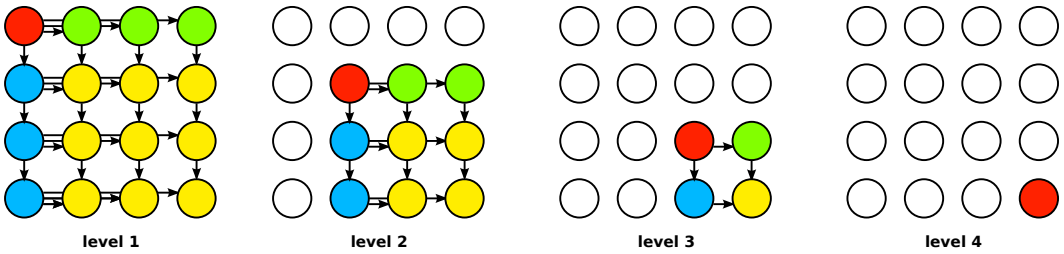


Figure 4.10: Task-based QR decomposition of a matrix consisting of 4×4 tiles. Each circle represents a tile, and its colour represents the type of task on that tile at that level. Empty circles have no task associated with them. The arrows represent dependencies at each level, and tasks at each level also implicitly depend on the task at the same location in the previous level.

used to benchmark task-based libraries. I implemented my own version of each of the operations, first in serial MATLAB, and then in both C and CUDA C. The section explains the operations and details my MATLAB implementation and the CUDA C implementation. Additionally I provide the function to compute Q and check correctness.

Our MATLAB function to compute the tiled QR decomposition of a matrix A (of even dimensions) in serial is as follows:

```
function [A, tau] = houseQR(A, tilesize)
[m,n] = size(A);
m = m/tilesize;
n = n/tilesize;
minmn = min(m,n);
tau = 0;
for k=1:minmn
  [A, tau] = SGEQRF(A, tau, k, tilesize);
  for j=k+1:n
    [A, tau] = SLARFT(A, tau, j, k, tilesize);
  end
  for i=k+1:m
    [A, tau] = STSQRF(A, tau, i, k, tilesize);
    for j=k+1:n
      [A, tau] =  SSSRFT(A, tau, i, j, k, tilesize);
    end
  end
end
end
```

xGEQRF

The xGEQRF function performs a Householder QR decomposition of the tile it is executed on, resulting in $\begin{pmatrix} R & R \\ H & R \end{pmatrix}$ and $\tau(k,(k-1)\times K)$ to $\tau(k,(k-1)\times K+K-1)$. The operation is performed on the top left tile in each step ($A(k,k)$), and must be completed before any other operations in that step can take place. The operation is dependent on any operations from previous steps on the tile being completed.

The operation computes each column of $H = I - \tau w w^T$, where w is the normalized column vector being operated upon. $R = w$ is stored in the upper right triangle on each row of the matrix, as each column of H is computed.

Our MATLAB function that computes xGEQRF is as follows:

```
function [A, tau] = SGEQRF(A, tau, k, tilesize)

block = A((k-1)*tilesize+1:k*tilesize, (k-1)*tilesize+1:k
    *tilesize );
for j = 1:tilesize
  normx = norm(block(j:end,j));
  s = -sign(block(j,j));
  u1 = block(j,j) - s*normx;
  if u1 ~= 0
    w = block(j:end,j)/u1;
  else
    w = zeros(size(block(j:end, j)));
  end
  w(1) = 1;
  if normx ~= 0
    tau(k,(k-1)*tilesize+j) = -s*u1/normx;
  else
    tau(k,(k-1)*tilesize+j) = 0;
  end
  block(j+1:end,j) = w(2:end);
  block(j,j) = s*normx;
  block(j:end,j+1:end) = block(j:end,j+1:end)-(tau(k,(k
      -1)*tilesize+j)*w)*(w'*block(j:end,j+1:end));
end
A((k-1)*tilesize+1:k*tilesize,(k-1)*tilesize+1:k*tilesize
    ) = block;
end
```

where `A` is the matrix A, `tau` is the τ matrix, `k` is the step being computed and `tilesize` is K.

xLARFT

The xLARFT operation is performed on the tiles $A(k, j), j > k$ at each step k. It cannot be computed until any operations from previous steps are completed on the tile it acts upon, and until the xGEQRF operation for step k has also been completed. The xLARFT routine applies the transformation H computed in the xGEQRF operation to

the tiles on row k, *i.e.* $(I - H\tau H^T) \times A(k, j)$. The input parameters for the operation are therefore the cells $A(k, k)$, $A(k, j)$ and $\tau(k, k)$.

Our MATLAB function to compute this operation is as follows:

```
function [A, tau] = SLARFT(A, tau, j, k, tilesize)
upperblock = A((k-1)*tilesize + 1:(k)*tilesize, (k-1)*
    tilesize+1:(k)*tilesize);
block = A((k-1)*tilesize + 1: (k)*tilesize, (j-1)*
    tilesize+1:(j)*tilesize);
for i = 1:tilesize
  w = upperblock(i:end, i);
  w(1) = 1;
  block(i:end, 1:end) = block(i:end, 1:end) - (tau(k,(k
      -1)*tilesize+i)*w)*(w'*block(i:end,1:end));
end
A((k-1)*tilesize + 1: (k)*tilesize, (j-1)*tilesize+1:(j)*
    tilesize) = block;
end
```

with the same naming scheme as xGEQRF.

xTSQRF

The xTSQRF operation is performed on the tiles $A(i, k), i > k$ at each step k. It cannot be computed until any operations on the tile it acts upon from previous steps are completed, and until the xGEQRF operation for step k has also been completed. Additionally, for each tile $A(ii, k), ii > k+1$, the xTSQRF operation at tile $A(ii-1, k)$ must have already been completed. The xTSQRF routine applies the Householder transformation to the cell $A(i, k)$ and updates the values of R in cell $A(k, k)$.

Our MATLAB function to compute this operation is as follows:

```
function [A, tau] = STSQRF(A, tau, i, k, tilesize)
upperBlock = A((k-1)*tilesize+1:k*tilesize, (k-1)*
    tilesize+1:k*tilesize );
R = getR(upperBlock);
block = zeros(tilesize*2, tilesize);
block(1:tilesize, 1:tilesize) = R(1:tilesize, 1:tilesize)
    ;
block(tilesize+1:end, 1:tilesize) = A((i-1)*tilesize+1:i*
    tilesize, (k-1)*tilesize+1:k*tilesize );
```

```
for j=1:tilesize
  w = zeros(tilesize - j +1,1);
  w(1) = block(j, j);
  w(tilesize+2-j:tilesize*2+1-j,1) = block(tilesize+1:end
     , j);
  normx = norm(w);
  s = -sign(w(1));
  u1 = w(1) - s*normx;
  if u1 ~= 0
    w(2:end) = w(2:end)/u1;
  else
    w(2:end) = 0;
  end
  w(1) = 1;
  if normx ~= 0
    tau(i,(k-1)*tilesize+j) = -s*u1/normx;
  else
    tau(i,(k-1)*tilesize+j) = 0;
  end
  block(j:end, j:end) = block(j:end, j:end) -(tau(i,(k-1)
     *tilesize+j)*w)*(w'*block(j:end,j:end));
  block(tilesize+1:end, j) = w(tilesize+2-j:end, 1);
end
for j=1:tilesize
  upperBlock(j, j:tilesize) = block(j, j:tilesize);
end
A((k-1)*tilesize+1:k*tilesize, (k-1)*tilesize+1:k*
    tilesize) = upperBlock;
A((i-1)*tilesize+1:i*tilesize, (k-1)*tilesize+1:k*
    tilesize) = block(tilesize+1:end, 1:tilesize);
end
```

where the `getR` function returns the upper triangular matrix from the tile supplied.

xSSRFT

The xSSRFT operation is performed on the tiles $A(i,j), i > k, j > k$ at each step k. The xSSRFT operation on a tile $A(i,j)$ for step k cannot be computed until the xTSQRF operation on tile $A(i,k)$ has been completed, and the xSSRFT or xLARFT

operation on tile $A(i, j-1)$ has been completed. As with all of the other operations, the operation on the tile from all previous steps must also have been completed. The xSSRFT routine applies the transformation computed by xTSQRF at $A(i, k)$ to the tiles $A(i, j)$ and $A(k, j)$.

Our MATLAB function to compute this operation is as follows:

```
function [A, tau] = SSSRFT(A, tau, i, j, k, tilesize)
tile = zeros(tilesize*2, tilesize);
tile(1:tilesize, 1:end) = A((k-1)*tilesize+1:k*tilesize,
    (j-1)*tilesize+1:j*tilesize);
tile(tilesize+1:end, 1:end) = A((i-1)*tilesize+1:i*
    tilesize, (j-1)*tilesize+1:j*tilesize);
lowertile = A((i-1)*tilesize+1:i*tilesize, (k-1)*tilesize
    +1:k*tilesize);
for m = 1:tilesize
  w = zeros(tilesize*2,1);
  w(m) = 1;
  w(tilesize+1:end, 1) = lowertile(1:end, m);
  tile = tile - (tau(i,(k-1)*tilesize+m)*w)*(w'*tile);
end
A((k-1)*tilesize+1:k*tilesize, (j-1)*tilesize+1:j*
    tilesize) = tile(1:tilesize, 1:tilesize);
A((i-1)*tilesize+1:i*tilesize, (j-1)*tilesize+1:j*
    tilesize) = tile(tilesize+1:end, 1:tilesize);
end
```

Checking Correctness and Recovering Q

One advantage of the QR decomposition as a test case is the relative ease of checking the correctness of the result. As the definition of the problem is $A = QR$ we can check correctness by taking the product of the Q and R matrices and comparing the result to the input matrix, A.

As the tiled QR decomposition outputs HR and τ instead of QR, we need to be able to recompute Q from H and τ.

H consists of a number of column vectors, denoted $H_1, ..., H_n$. Q can be recovered as $Q_1 Q_2 ... Q_n$, where $Q_i = I - \omega w^T$ and ω is a column vector, where for $j = 0 : n$, $\omega(j * K : j * K + K - 1) = \tau(j, k) * A(i, j * K : i, j * K + K - 1)$. This is similar to reassembling a matrix from its Householder reflectors, however since this

implementation doesn't normalise each column, we need to use the τ values stored to recover the column vector.

Our MATLAB script to recover Q is as follows:

```
1  function Q = getQFull(A, tau, tilesize)
2  Q = eye(size(A));
3  [numrows,numcolumns] = size(A);
4  numcoltile = numrows/tilesize;
5  numrowtile = numcolumns/tilesize;
6  for k = 1:numrowtile
7    for l = 1:tilesize
8      Qtemp = eye(numrows);
9      for i = k: numcoltile
10       w = zeros(numrows,1);
11       w((i-1)*tilesize+1:i*tilesize) = A((i-1)*tilesize
             +1:i*tilesize, (k-1)*tilesize+l);
12       w((k-1)*tilesize+l) = 1;
13       if (((k-1)*tilesize+l) > (i-1)*tilesize+1)
14         w(1:(k-1)*tilesize+l-1) = 0;
15       end
16       Qtemp = Qtemp * (eye(numrows) - tau(i, (k-1)*
             tilesize+l)*w*w')';
17     end
18     Q = Q*Qtemp;
19   end
20 end
21 end
```

This code is only to illustrate how to perform this computation, but is not optimized. Lines 16 and 18 can be implemented with fewer matrix multiplications to improve performance. Our MATLAB, CPU and CUDA implementations presented have all been checked to ensure $A = QR$ within rounding error.

4.3.3 Implementing the Tiled QR Decomposition on the GPU

On the GPU, we parallelise task-based computations by executing one task per block. This gives the coarse-grained parllelism for the QR decomposition on the GPU. The major hurdle in implementing the tiled QR decomposition in CUDA was implementing SIMT-parallel versions of each of the task functions. To parallelise the individual task functions, we spread the matrix-vector and vector-vector operations over the

threads in a block. This means we exploit SIMT parallelism to perform operations like `w=block(j:end,j)/u1;` as each thread in a warp reads in a single element of `w` and performs the necessary operations on it in parallel. For the SLARFT and SSSRFT we can further parallelise the operations by striding across the updates to the upper tile by the number of warps in the block, *i.e.* `blockDim.x/32`.

To avoid the need for synchronisation I therefore used tile sizes of 32 (the same as the warp size). This means all of the operations within a tile are performed in lock-step parallelism.

Larger tile sizes such as K = `blockDim.x` would have likely lead to improved performance, despite the requirement for synchronisation between warps and increased complexity of reduction operations. Our GPU implementation executes blocks of 128 threads, however the SGEQRF and STSQRF functions need to be executed a single column at a time, meaning these functions are restricted to only 32 threads.

On CUDA GPUs it is usually important to have coalesced memory accesses for all the threads in a warp. Since the QR operations primarily operate on columns of the tiles, it makes sense to have the tiles stored in a column-major format. Additionally, most implementations notice better cache-coherency if the tiles are stored as blocks in memory, though this is less relevant for the GPU. This results in a tiled column-major layout (also known as square blocked [25]), *i.e.* For a matrix A with tiles of size 32, $A(1:32)$ is the first column of the first tile, $A(33:64)$ is the second column of the first tile, and so on. This is the matrix layout I decided to use for my GPU implementation of the tiled-QR decomposition.

Calculating the Norm of a Vector

To calculate the norm of a vector we can use a reduction operation. CUDA Compute Capability 3.0 introduced shuffle operations, which allow threads within the same warp to communicate data stored in registers to other threads in the warp. It was shown in [12] that these shuffle operations can improve the speed of reductions within a warp when compared to using shared memory.

I implemented a reduction operation in CUDA as follows [3]:

```
1 __device__ inline void reduceSumMultiWarp( float* value ){
2 #if __CUDA_ARCH__ >= 300
3   #pragma unroll
4   for(int mask = 16 ; mask > 0 ; mask >>= 1)
```

[3] Our code was safe for any compute capability - see Appendix C for the variant that works when no shfl operations are available.

```
    *value += __shfl_xor((float)*value, mask);
  *value = __shfl((float)*value, 0);
#else
  //No shfl commands
#endif
}
```

where I used the shuffle operations if available. The `WARPS` variable is a predefined value that ensure safety when using shared memory reductions with compute capabilities pre-3.0. A vector norm operation can then be implemented as:

```
norm = 0.0;
norm = Tile[ i*tilesize + TID ] * Tile[i * tilesize + TID];
reduceSumMultiWarp( &norm );
norm = sqrt(norm);
```

where `TID` is `threadIdx.x % 32`, `i` is the column of the vector to be normalised, and `Tile` is a pointer to a section of the matrix.

Appendix C contains the CUDA implementation of the SLARFT code, as well as an alternate version that can cope with larger tile sizes.

4.4 Barnes-Hut Simulation

4.4.1 The Barnes-Hut Algorithm

The Barnes-Hut (BH) algorithm [5] is a hierarchical algorithm for solving N-body problems, *i.e.* computing all the pairwise interactions between a set of N particles, in $O(NlogN)$ operations. It is often used in cosmological simulations to calculate the forces between particles due to gravity.

The algorithm uses a recursive decomposition, and starts from an initial cell containing all of the particles. For each cell, if that cell contains more than n_{min} particles (in the original algorithm, n_{max} is 1), the cell is split in each dimension to create 8 subcells (in 3 dimensions), which are processed recursively. This process is repeated recursively until no cells contains more than n_{max} particles. This is known as an *oc-tree decomposition*. Figure 4.11 shows this decomposition for a set of particles in two dimensions. As the figure shows, not all leaf cells have the same edge width.

Each particle p_i interacts directly with every other particle $p_j, j \neq i$, where the ratio $\frac{s}{d} \geq \theta$ holds, where s is the edge-length of the cell containing the particle p_j, and d is the distance between p_i and the centre of mass of the cell containing p_j. θ

is the opening angle, a predefined constant whose value is chosen to give the required accuracy for a simulation. If n_{max} is greater than or equal to 1, particles within the same cell must interact directly regardless of the value of θ.

If $\frac{s}{d} < \theta$, then p_i interacts with the centre of mass of the cell containing p_j, rather than with the particles individually.

Altering the value of θ varies the cost vs accuracy of the algorithm, and if $\theta = 0$ it is the same as doing the $O(n^2)$ interaction. Figure 4.12 shows the interactions of the same particle set as in Figure 4.11 for two different values of θ. The green particle interacts directly with the blue particles, and with the centre of mass (magenta particles) for each other cell. In some cases when the cell only contains a single particle, the centre of mass is just the particle (these are the magenta particles with a black border). The hollow red particles are interacted with only by the centre of mass of one of the cells that contains them. As θ is increased, fewer direct interactions are performed, and centre of mass interactions cover larger groups of particles.

The tree-walk can be created by a recursive algorithm, by initially interacting each particle with the root cell as follows:

```
function TREEWALK(p, cell, theta)
    s ← cell.s
    com ← cell.com
    d ← |p.pos - com|
    if s/d ≥ θ then
        for Each nonempty child cell c of cell do
            Treewalk(p, c, theta)
        end for
    else
        Interact p with cell.com
    end if
end function
```

If we ignore symmetry (*i.e.* for each particle pair p, q we compute the interaction twice), then we can theoretically parallelise this algorithm easily, by dividing the particles into N chunks (where N is the number of processors to parallelise over) and having each processor compute one of the chunks in parallel.

The task-based algorithm used for the BH is discussed in chapter 5.

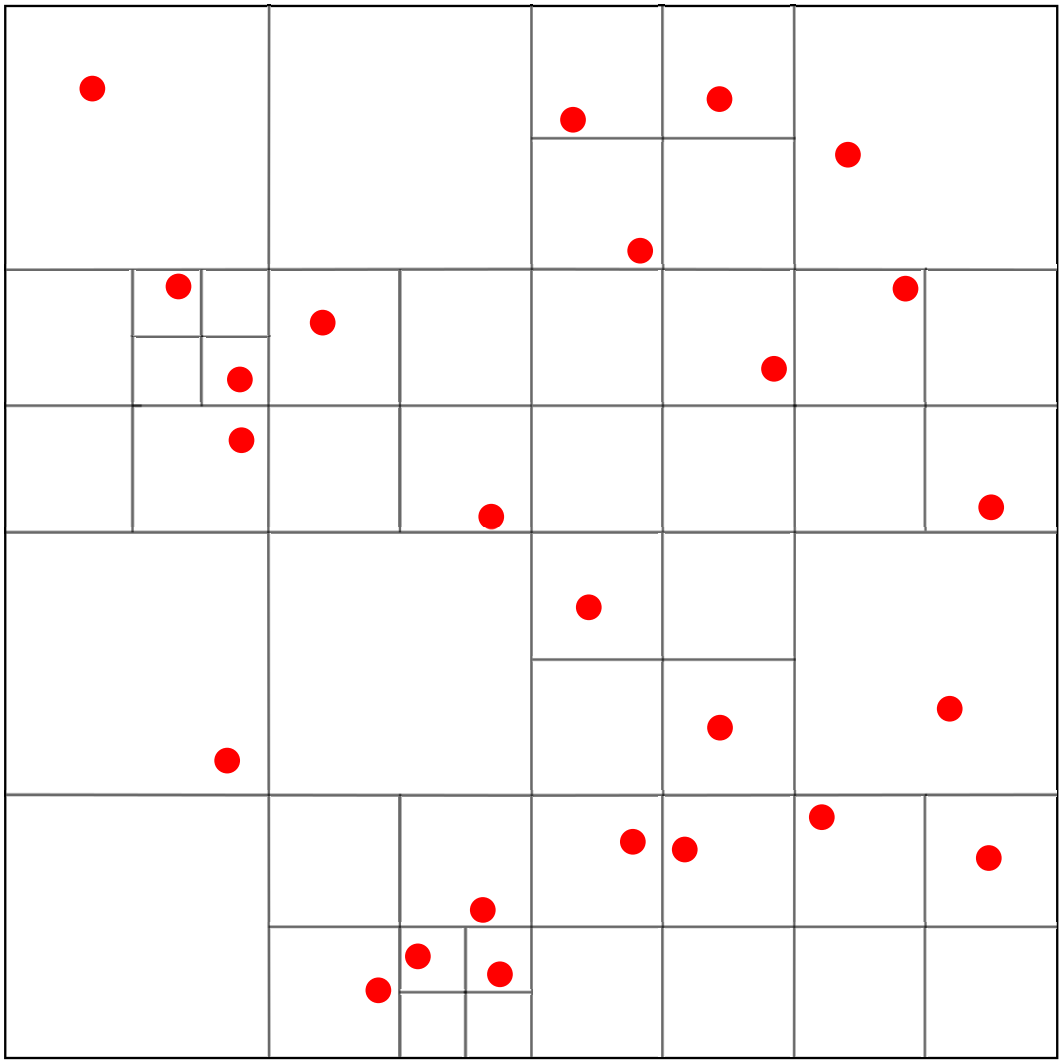


Figure 4.11: An image that shows the 2D Barnes-Hut cell decomposition of a set of particles.

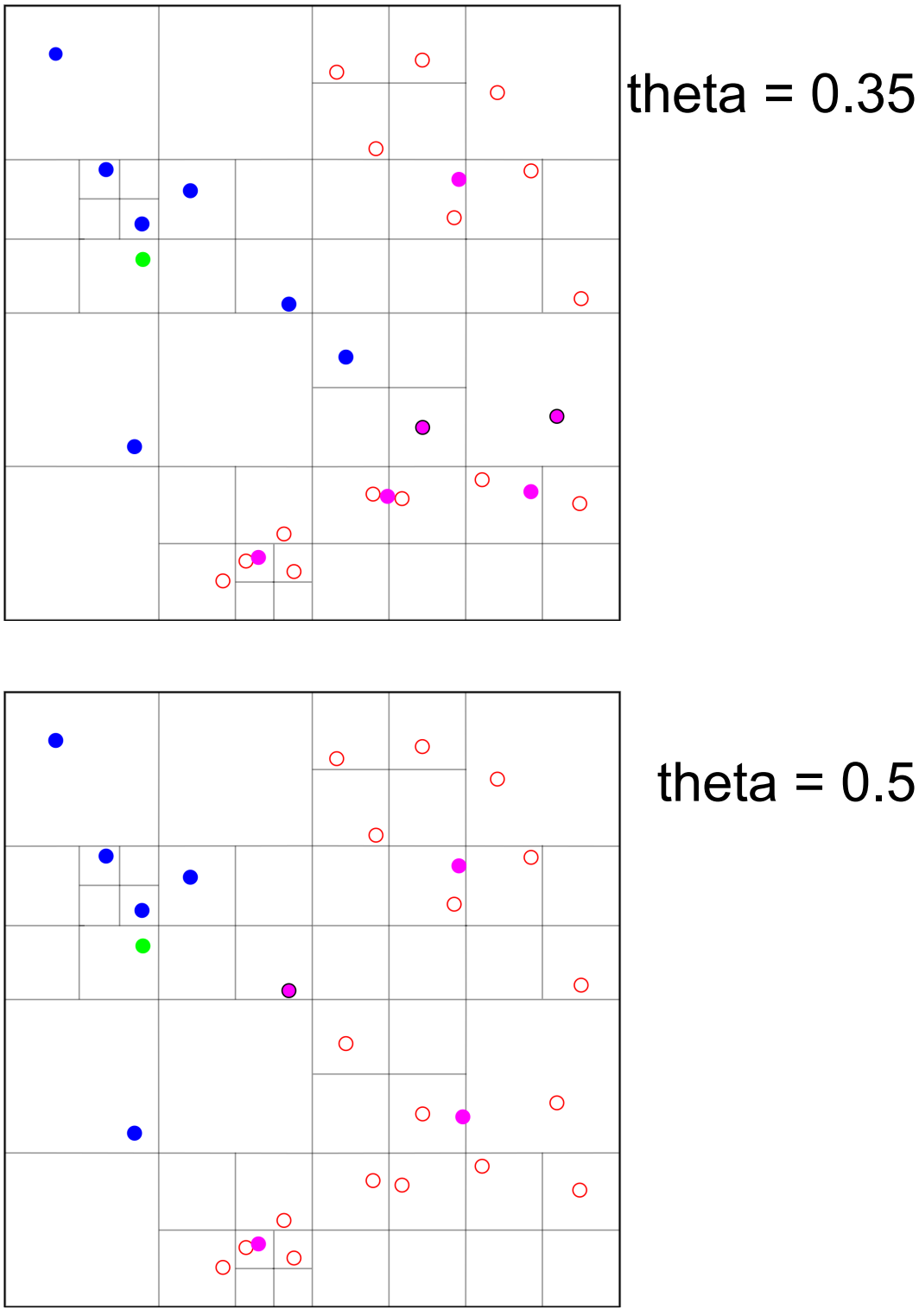


Figure 4.12: This figure shows the interactions of the same particle introduced in Figure 4.11 for two different values of θ. The green particle interacts directly with the blue particles, and with the centre of mass (magenta particles) for each other cell. In some cases when the cell only contains a single particle, the centre of mass is just the particle (these are the magenta particles with a black border). The hollow red particles are interacted with only by the centre of mass of one of the cells that contains them. As θ is increased, fewer direct interactions are performed, and centre of mass interactions cover larger groups of particles.

Chapter 5

Task-Based Barnes-Hut for Gravity.

Sections 5.2.1 and 5.3 contain work done by Matthieu Schaller (Institute of Computational Cosmology, Durham University) and myself

5.1 The Shared Memory CPU Task-Based Algorithm

The standard octree approach for solving Barnes-Hut (BH) simulations uses a treewalk through the tree to perform neighbour finding and compute the interactions. This can lead to performance issues as this treewalk is performed for every particle in the system, despite many of the particles interacting with similar sets of particles. Additionally, if the particles are not contiguous in memory, this can lead to poor cache behaviour, though using a space filling curve or similar to sort the particles may alleviate this. Using a task-based approach to perform the Barnes-Hut allows us to avoid repeating the tree-walk for every particle.

In a naive task-based implementation of the Barnes-Hut, every tree-traversal could be a task, i.e. for every cell c_i in the system, we create a task for every other cell c_j that c_i interacts with. This method is acceptable if we have cells containing $10 - 1000$ particles when performing the particle-particle interactions, however when performing particle-monopole interactions the scheduling overheads of the task would dominate the computation.

To reduce this issue, the algorithm needs to be modified to increase the amount of computation in each task. Rather than performing the octree decomposition until every cell contains at most one particle, cells were split if they contain more than a certain number of particles, n_{max}, usually 50 to 200.

We also change the criteria for particle interactions slightly from the original BH algorithm. While in the original BH, particles may interact with particles in non-

neighbouring cells if θ is low enough, we enforce that any direct interactions are between pairs of leaf cells that are direct neighbours. In both the original BH and our new algorithm (referred to as task-based BH), a particle's interaction radius is governed by the particle density of the space around it. In our algorithm, the number of neighbours, and thus accuracy of the simulation, is roughly controlled by n_{max}. The two algorithms will potentially compute slightly different interactions, so all of the results in this section are only comparisons with the shared-memory CPU version of the task-based BH algorithm.

The algorithm used 3 types of tasks:

- Single cell (aka *self*) direct interaction tasks that compute the forces between all particle pairs p_i, p_{ii} in a single cell c_i.
- Direct interaction tasks that compute the forces between all particles p_i, p_j in a pair of cells c_i, c_j (pair tasks).
- Particle-cell tasks. This computes all of the monopole interactions for the particles p_i in a single cell c_i. This function performs the recursive treewalk from the root cell to find all of the monopoles that are required to interact with the cell c_i.

The direct interaction tasks are built using a recursive function that is called on the root cell:

```
1:  function CreateTasks(ci, cj)
2:      if cj = ϵ then
3:          if ci is split & ci.count > limit then
4:              for Each child of ci, cp do
5:                  CreateTasks(cp, ϵ)
6:                  for Each sibling cz of cp not already looped to do
7:                      CreateTasks(cp,cz)
8:                  end for
9:              end for
10:         else
11:             Create self task for ci
12:         end if
13:     else
14:         if ci and cj are neighbours then
15:             for Each pair of children of ci and cj, cx and cy do
16:                 CreateTasks(cx, cy)
17:             end for
```

```
18:         end if
19:         Create a pair task for ci and cj
20:     end if
21: end function
```

The function initially checks if cj was passed to the function. If not, then it checks if ci is split (i.e. has subcells) and contains more particles than a specified $limit$. If so, the function recurses on each child of ci, and each combination of pairs of children of ci. If ci is not split or contains too few particles, a self-interaction task is created for ci.

If cj was passed to the function, and ci and cj are neighbours then the function recurses for each pair of children from ci and cj. If ci and cj are not neighbours, then a pair-interaction task is created for ci and cj.

This function performs (some of) the tree-walk required for the BH, so is a more expensive initialisation than the standard BH. However, since we don't walk all the way to each individual particle, and only have to perform this once per tree rebuild, the cost over an entire simulation would be much lower than having to perform

The particle-cell tasks require the root cell and a leaf cell for execution. The function starts by finding the child of the root that contains the leaf, `cp`. The function recurses with `cp` and the leaf cell, and then calls the `iact_pair_pc` function on each other child of the root and the leaf. This continues until all non-leaf cells have interacted with the leaf:

```
1: function IACT_SELF_PC(cell, leaf)
2:     for Loop over children of cell, cp do
3:         if leaf is inside cp then
4:             Break out the loop, remembering cp
5:         end if
6:     end for
7:     if cp is split then
8:         iact_self_pc(cp, leaf)
9:         for Each other child of cell, cps do
10:            if cps is split then
11:                iact_pair_pc(cp, cps,leaf)
12:            end if
13:        end for
14:    end if
15: end function
```

The `iact_pair_pc` function requires 3 cells, ci, cj and the $leaf$ cell, and ci must contain the leaf cell:

```
1: function IACT_PAIR_PC(ci, cj, leaf)
2:     for Loop over children of ci, cp do
3:         if leaf is inside cp then
4:             Break out the loop, remembering cp
5:         end if
6:     end for
7:     if cp and cj are neighbours then
8:         for Loop over children of cj, cps do
9:             if cp and cps are neighbours then
10:                if cp and cps are both split then
11:                    iact_pair_pc(cp, cps, leaf)
12:                end if
13:            else
14:                Interact leaf and cps directly.
15:            end if
16:        end for
17:    else
18:        for Loop over the children of cj, cps do
19:            Interact leaf and cps directly.
20:        end for
21:    end if
22: end function
```

The function initially searches for the child of ci that contains the $leaf$ cell, cp. If cp and cj are neighbours, the algorithm loops over the children of cj, cps. The function recurses for cp and each cps that are neighbours, provided both cells are split. If cp and cps are not neighbours, the function interacts the particles in the $leaf$ cell with cps directly.

If cp and cj are not neighbours, the $leaf$ is interacted with each child of cps directly.

Unlike the traditional BH algorithm, we do not control the accuracy by altering the value of θ (the opening angle). Instead, to tune the accuracy of the code the maximum number of particles (n_{max}) can be changed. If the maximum number of particles in a cell is reduced then it is similar to increasing θ in the traditional algorithm. Inversely, if the maximum number of particles in a cell is set to be the number of particles in the space, it computes the n^2 interaction.

5.2 Task-Based Barnes-Hut on the GPU

The previously defined `iact_pair_pc` function has recursion even if `task_limit` is 0, and recursive functions often perform poorly on GPUs. While newer versions of CUDA support recursion, our experience was that the algorithm developed for use on the CPU was not ideal on the GPU, though I did implement a SIMT parallelised version of the algorithm discussed in section 5.1 for the GPU. The results with this algorithm are shown in Table 5.1. These results were worse than we had aimed for, so we attempted to modigy the algorithm to improve the performance.

Type	Runtime
GPU, 128 parts per cell	8.206ms
CPU, 1 thread	82.729ms
CPU, 4 threads	21.988ms

Table 5.1: Results with 15000 randomly distributed particles for the recursive task-based algorithm of the Barnes-Hut on one GPU of an NVIDIA GeForce GTX 690 and a 4-core Intel i7 (Sandy Bridge) CPU. The CPU setup uses 100 particles per cell and a `task_limit` of $1e8$. The GPU performs roughly 2.5× faster than the full CPU.

5.2.1 Modifying the Algorithm to be GPU-Friendly

To improve the performance of the algorithm on the GPU we need to remove the recursion in the task functions. Additionally, increasing the number of particles in the leaves to be approximately equal to the number the number of threads in a block should improve performance. For the results shown in this section the maximum number of particles in a cell was set to 128.

To avoid recursion in the direct interaction tasks, I removed the `limit` when constructing tasks (line 3 in the CreateTasks function pseudocode above), and modified the interaction functions accordingly.

To avoid recursion in the particle-cell tasks, we separate the particle-cell tasks into two types:

- Normal particle-cell tasks. This type is used when tasks are created on small cells (`cell->count <= 64 * cell_maxparts`, where `cell_maxparts` is the maximum number of particles in a cell).
- Split particle-cell tasks. If particle-cell tasks would be created on large cells, 8 of these split tasks were created for the large cell's children instead of creating the single large task.

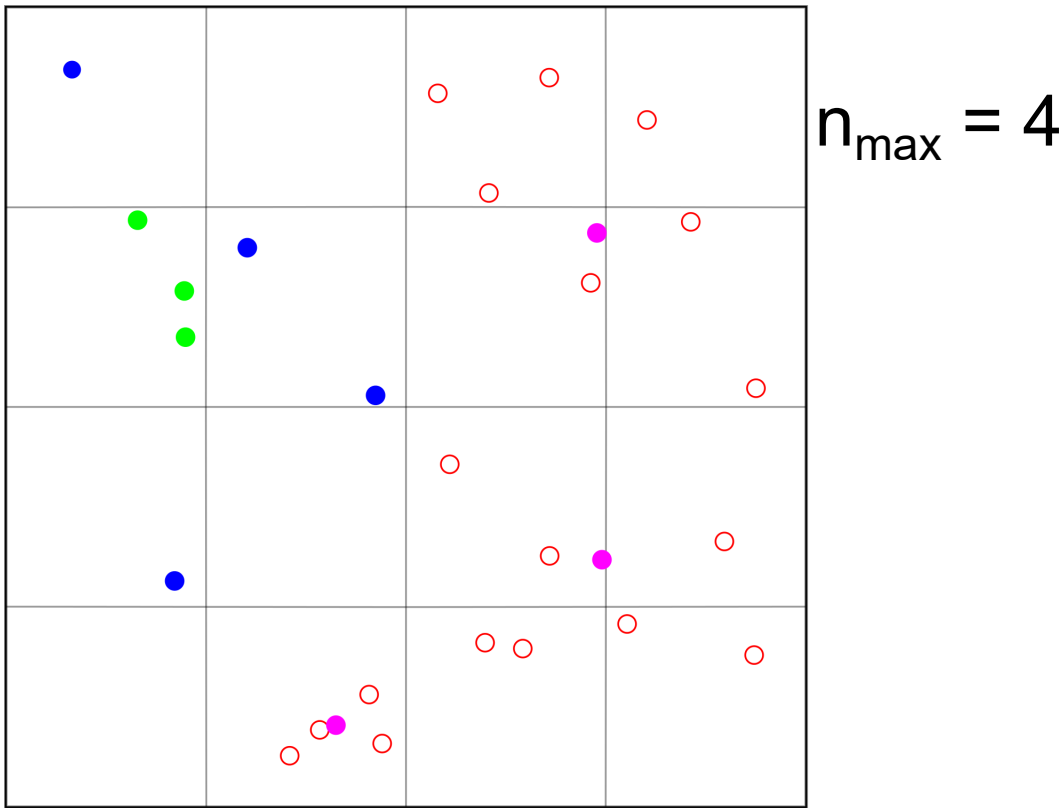


Figure 5.1: The green particles interact with each other in the cell's self interaction task, and interact with the blue particles from neighbouring leaves during the pair interaction tasks. The remaining interactions are covered by interactions with the monopoles (magenta), with 1 task per monopole. Since none of the cells contain enough particles, this is done at the highest level of non-neighbouring cells.

Rather than making a single particle-cell task for each cell during cell creation, I modified the CreateTasks function to perform a treewalk over the children of $cell_i$. Provided $cell_i$ contains more than a predefined number of particles, the function then creates tasks to interact each child cell of $cell_i$ with the monopole of $cell_j$, and each child cell of $cell_j$ with the monopole for $cell_i$. If $cell_i$ does not contain more than this number of particles, a task is created to interact $cell_i$ with the monopole of $cell_j$, and to interact $cell_j$ with the monopole of $cell_i$. This means the particle-cell tasks no longer need to recurse down the tree from the root to find non-neighbouring cells to interact, as this treewalk now takes place in the CreateTasks function.

An example of this breakdown into tasks is shown in Figure 5.1. The green particles interact with each other in the cell's self interaction task, and interact with the blue particles from neighbouring leaves during the pair interaction tasks. The remaining interactions are covered by interactions with the monopoles (magenta). Since none of the cells contain enough particles, this is done at the highest level of non-neighbouring cells.

The improved GPU algorithm performed significantly better, as shown in Table 5.2, and Table 5.3 shows results on larger problems.

Type	Runtime
New GPU algorithm, 128 parts per cell	5.641ms
Original GPU algorithm, 128 parts per cell	8.206ms
CPU, 1 thread, default setup	82.729ms
CPU, 4 threads, default setup	21.988ms

Table 5.2: Results with 15000 randomly distributed particles for the recursive task-based algorithm of the Barnes-Hut on one GPU of a NVIDIA GeForce GTX 690 and an Intel i7 (Sandy Bridge) CPU. The new algorithm performs significantly faster than the original algorithm on the GPU, even on this small test case which contains minimal recursion.

5.3 Adapting the Barnes-Hut Implementation for Hybrid Memory QuickSched

When implementing the algorithm on hybrid shared-distributed memory systems, the algorithmic changes discussed in section 5.2.1 did not perform well when using multiple MPI ranks, as shown in Table 5.4.

The algorithm tends to create particle-cell tasks that operate on cells close to the root of the tree, which makes it difficult to partition the work equally across the nodes. Furthermore, each of the particle-cell tasks potentially needs to *use* each resource

Simulation type	1M parts	3M parts	10M parts
1 CPU with Quicksched	15.9s	50.5s	174.5s
16 CPUs with Quicksched	1.217s	3.489s	12.0s
GTX690 GPU	0.239s	0.677s	2.636s
GTX690 GPU Single precision	0.116s	0.344s	1.414s
Tesla K40c GPU	0.099s	0.271s	2.025s

Table 5.3: Average time taken to compute the accelerations for a single timestep of a Barnes-Hut simulation. The CPU code run is our own test case[22] run on a single node of the DiRAC Data Centric system at Durham University, with up to 16 processors (2 Intel E5-2670 0 @ 2.60GHz per node). The GPU code uses mixed precision unless otherwise stated, and is primarily double precision.

Number of MPI ranks	Time per step
1	21.6s
4	23.6s

Table 5.4: Average time taken to compute the accelerations for a single timestep of a Barnes-Hut simulation with 1M parts. These results were run on a shared memory machine using only MPI parallelism (64-core AMD Opteron 6376 machine at 2.67 GHz.), so any loss in performance is primarily due to load imbalance/duplicated work, or synchronisation caused by the communication.

corresponding to any child cell of the root. For large simulations, this leads to too many resource uses in the system and worsens the performance of other sections of the scheduler. To reduce this issue, the algorithm was modified to create more particle cell tasks that operate nearer the leaves of the tree, which should help load balancing, and means each task only needs to use a single cell's resource. These tasks were created by removing the particle cell creation routine added to the GPU code, and instead adding a new function:

```
1: function CREATE_PCS(ci, cj)
2:     if cj = ∅ then
3:         for Each child of ci, cp do
4:             if cp is split then
5:                 create_pcs(cp, NULL)
6:             end if
7:             for Each sibling of cp, cps do
8:                 if cp and cps are both split then
9:                     create_pcs(cp, cps)
10:                end if
11:            end for
12:        end for
13:    else
14:        if ci and cj are neighbours then
15:            if ci and cj are both split then
16:                for Each child of ci, cp do
17:                    for Each child of cj, cps do
18:                        create_pcs(cp, cps)
19:                    end for
20:                end for
21:            end if
22:        else
23:            Create particle-cell tasks for ci and cj
```

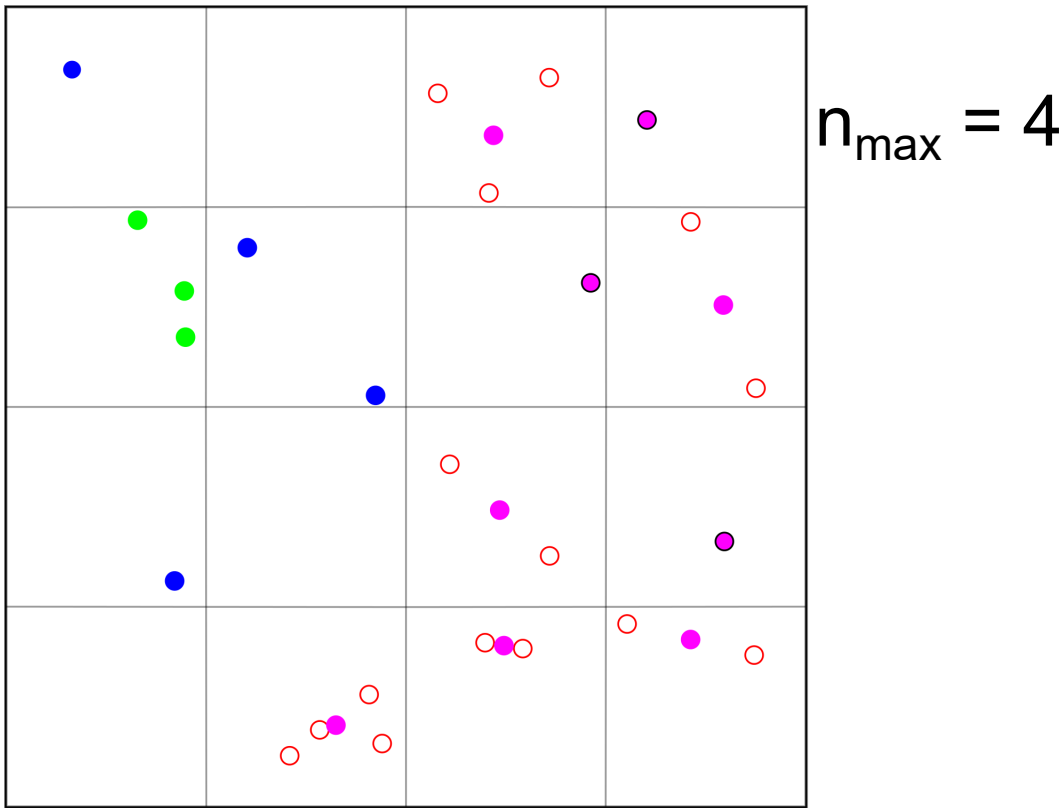


Figure 5.2: An example of the interactions created with this algorithm. The same self and pair tasks are created, however we create additional particle-cell tasks when compared to the GPU algorithm, as we create one per leaf.

24: **end if**
25: **end if**
26: **end function**

The function is initially provided the root cell as *ci*. The function recurses over all possible pairs of cells. For each pair of non-neighbouring cells, it creates a pair of particle cell tasks, rather than recursing to their children. Since this creates relatively few but large tasks, the implementation also used a minimum depth parameter, and would always recurse to that depth in the tree before creating any tasks.

An example of the interactions created with this algorithm is shown in Figure 5.2. The same self and pair tasks are created, however we create additional particle-cell tasks when compared to the GPU algorithm, as we create one per leaf.

One issue with this algorithm is that it can create a very large number of tasks, and this issue is apparent in the results section of Chapter 7.

The C implementations for QuickSched are given in Appendix D

Chapter 6

Task-Based Parallelism on GPUs

6.1 Implementing Task-Based Parallelism on CUDA GPUs

Although task-based methods are commonplace on shared memory systems, relatively few solutions exist for implementing them on GPUs. The methods that do exist tend to deploy single tasks on the entire GPU [43], meaning only problems whose individual tasks can parallelise to thousands of threads can use GPUs efficiently. Despite the vectorised programming model, the hardware itself can be viewed as a multithreaded multi-core computer, where every block is conceptually equivalent to a single core executing with a fixed number of threads in parallel. In principle we could use Task-Based Parallelism directly on the GPU by launching a set of blocks and letting each block dynamically select and execute tasks in parallel. The tasks themselves would be executed in SIMT parallelism using only the threads within each block. These tasks would only need to vectorise over a single block. Figure 6.1 shows a comparison of a task-based approach to treating the GPU as a large vector machine.

The usual strategy for using GPUs requires the user to manage the data movement to and from the GPU manually. This is a commonly discussed issue with GPU hardware and has led to the development of OpenACC [33], while OpenMP 4.5's offload model can also reduce the burden for the programmer. Requiring the user to manage their data transfers often leads to a synchronous "load, compute, unload" method, and these synchronisation points cause a loss in overall performance. As GPU technology has progressed they have gained the ability to directly access main memory (albeit slowly). I will show how using task-based methods on the GPU improves data transfer to the GPU as it helps to hide data movement behind actual work done, and thus can lead to overall performance improvement. By modeling the data transfers as tasks, we remove the requirement for the user to manually transfer data, which can reduce the difficulty

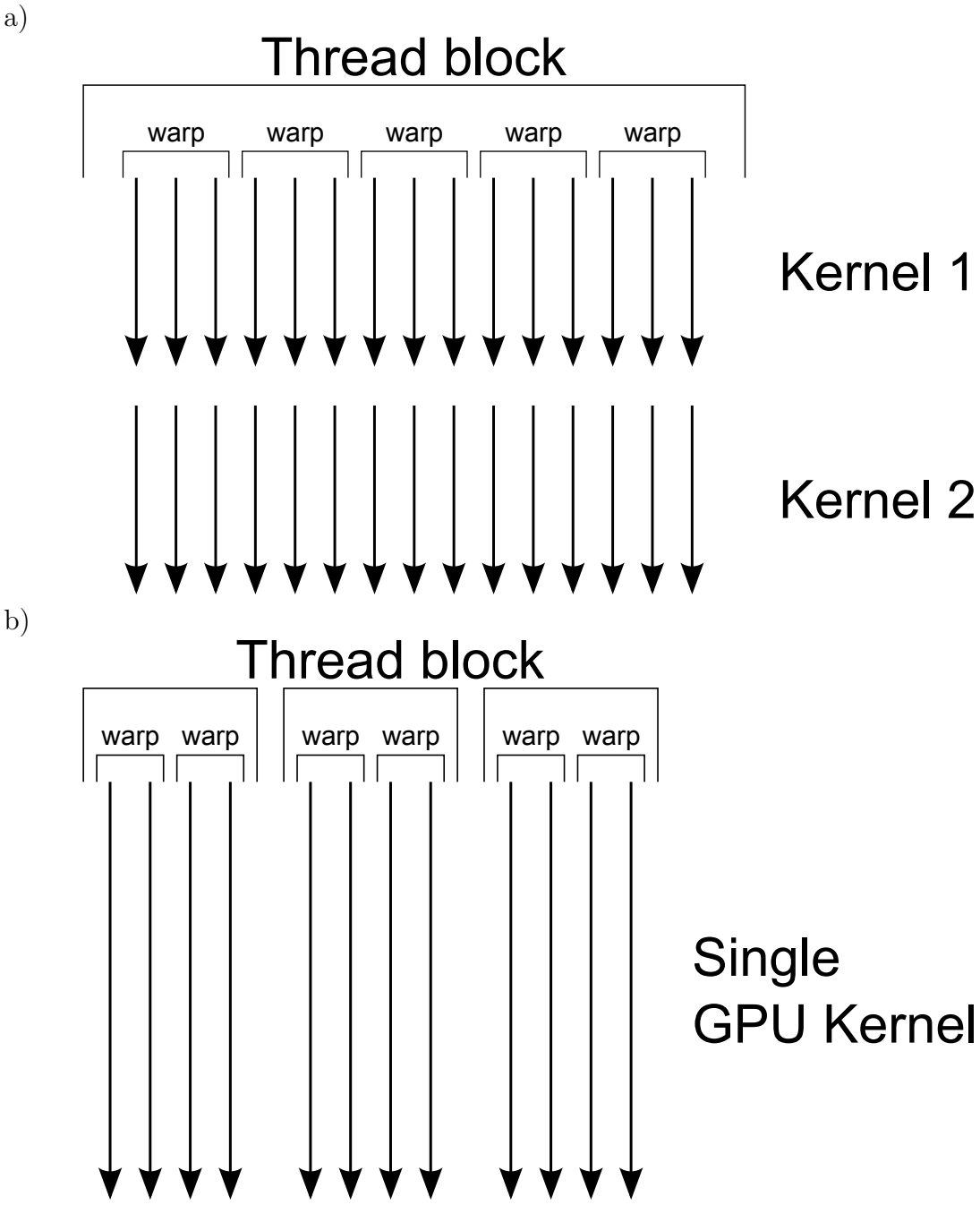


Figure 6.1: Figure a) shows the traditional approach to using the GPU, where a single kernel is executed by the entire GPU in parallel. Once the first kernel completes, another kernel is executed on the entire GPU. Figure b) shows our new approach, where small blocks of threads execute different work in parallel within a single large kernel.

of programming these devices. Automatic data transfer allows computation to start on the GPU without all of the memory required to have been copied to the GPU's memory, and data can be transferred back to the CPU as soon as the computation on that data is complete. Recent versions of CUDA already enable this with the use of streams and multiple kernels, however our task-based implementation allows even more fine-grained data transfers.

If a task-based methodology could be integrated with a CPU task-based scheme, it may enable load balancing across different types of devices. This is a concern on heterogeneous systems, however I do not address it in this work.

There have been previous projects looking at task-based parallelism on GPUs, with [10] and [47] introducing task-based schedulers for the GPU. [10] uses CUDA streams to launch many kernels, using the GPUs inbuilt ability to adaptively schedule the kernels to the device to provide good load balancing, and makes use of CUDA streams' asynchronous memory transfer tools, which allows data movement to the device in parallel with kernel execution. Their approach is less fine-grained than the approach I introduce in this chapter. [47] creates a task-parallel approach specifically designed for graphics processing, and it is unclear how suitable it is to more general problems.

In this chapter I will describe the initial scheduler used in `mdcore`, and the improvements to the scheduler. I then discuss some work done on improving the MPI load balance of `mdcore`, and how the methods used allowed the creation of a multi-GPU implementation of `mdcore`. Finally, I discuss the adaptation of the `mdcore` scheduler to fit the QuickSched task model, resulting in the creation of a general-purpose CUDA task scheduler for GPUs, including automated data transfer to the device.

6.2 Task-Based Parallelism for GPUs in `mdcore`.

Results in this section were run with CUDA 5.0 on the GTX480 and GTX690

This section details the development of the task-based scheduler on the GPU, which was originally developed as part of `mdcore`. It highlights the initial design created by Dr Pedro Gonnet, and my work to develop it and improve the performance of `mdcore`. These improvements were developed iteratively based on the results obtained, and these are also included.

6.2.1 Implementing Task Queues on the GPU

`mdcore` is a Molecular Dynamics (MD) library that implements MD on a variety of architectures, including shared and hybrid shared-distributed memory CPU systems, Intel Cell Broadband Engine Architecture and NVIDIA GPUs using CUDA. The neigh-

bour finding algorithms implemented are the cell list algorithm (section 4.1.2) and the pseudo-Verlet algorithm (section 4.1.2). The algorithm used to compute the van der Waals forces and the short-range electrostatics is discussed in section 4.1.3. `mdcore` does not implement long-range electrostatics.

Our task queues on the GPU primarily use two arrays, treated as cyclic buffers. The first array (known as `tids`) contains a list of task indices that still need to be executed in the current timestep, whilst the second array (called `done_tids`) contains all of the task IDs that have been executed (or are currently being executed) in this timestep. The only time the task IDs are not in one of these two arrays is when the task is actively being checked if they are ready to be computed. The task indices of completed tasks are kept, as they can be reused in later timesteps. At the end of each timestep, we can swap the `tids` and `done_tids` pointers. The `tids` array will then contain the task indices in the order in which they were executed in the previous timestep, which is likely to result in a better ordering and lead to improved performance.

Each task queue also stores four other values, used to compute the position of tasks in the queue, and to determine when the queue is empty. These are named `first`, `last`, `done_count`, and `count`. The first two values are used to find indicies of tasks in the queue. The `first` counter stores the position of the first task ID in the queue, while `last` stores the first empty position in the queue. `count` contains the total number of tasks in the task queue, while `done_count` stores how many tasks in this queue have been or are being executed in this timestep (equal to the number of tasks indices currently in the `done_tids` array).

The movement of a task through the queue structure is shown in Figure 6.2. To remove a task from the queue, the accessing thread atomically increments the `first` counter to retrieve the position of the head of the queue, and retrieves the task index stored at that position. If the task is ready to be executed, the index is then placed at the end (`done_count`) of the `done_tids` array, else the task index is added back to the queue. To add a task to the queue, the thread increments `last` (which finds the tail position of the queue) and places the task index in this position.

To check whether the task queue is empty, we check if `count == done_count`. Since tasks may not always be in either `tids` or `done_tids`, it is feasible for `first == last` before the computation is ready to terminate if one of the tasks being checked is not yet ready for computation.

The initial GPU task-based setup is based on the CPU implementation, but modified to have more lightweight queues. The outline of the main task-based kernel that is executed on the GPU by each threadblock in parallel is:

```
1 __global__ void kernel(...){
```

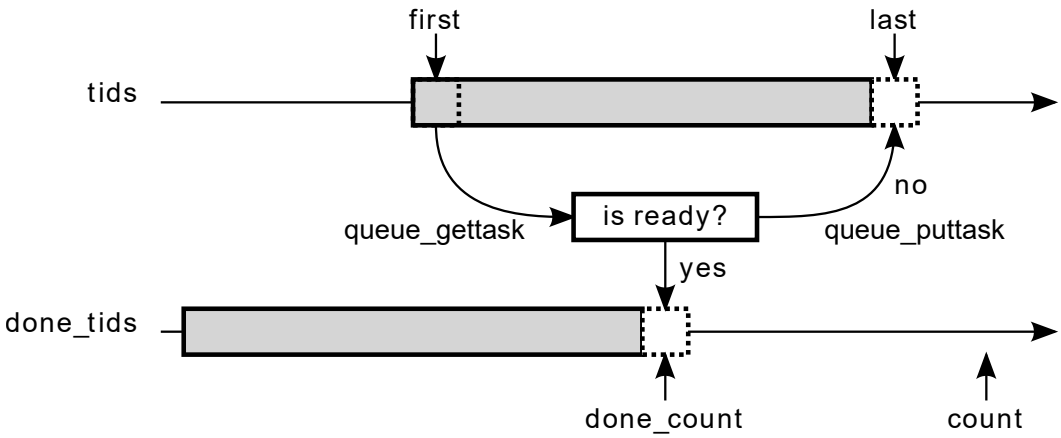


Figure 6.2: Schematic of the queue data structure. The indices of unexecuted tasks are stored in `tids`, between `first` and `last-1`. Task indices are removed from the head of the queue (`first`), and if they are ready to run, they are copied to `done_tids`. Otherwise, they are returned to the tail of the queue (`last`).

```
2   __shared__ int tid;
3   while ( there are tasks ){
4     if( threadIdx.x ==0 )
5       tid = get_task()
6     execute task using a single block
7   }}
```

where `tid` is a shared variable (i.e. visible to all of the tasks in the block), and stores the task index to be executed. In lines 4 and 5, the first thread in each thread block retrieves a task index from the queue, which is then implemented by the entire block in parallel in line 6. Since the rest of the threads in the block are idle while the task retrieval is taking place, we want to minimise the cost of queue access. To achieve this, the task queues are implemented as lock-free queues, i.e. no thread can cause any other thread to spin or wait, while guaranteeing system-wide progress. The queue data structure is as follows:

```
1 struct queue_cuda{
2   int* tids, done_tids;
3   volatile int first;
4   volatile int last;
5   volatile int done_count;
6   volatile int count;
7 };
```

The `tids` and `done_tids` arrays contain the task indices, and are usually of size greater than or equal to the number of tasks in the system. Initially the `tids` array contains

all of the task indices that need to be executed, and the done_tids array is empty. Once a task has been selected for execution, its index is placed in the done_tids array. Once a timestep is complete, the tids and done_tids pointers can be swapped.

The first variable stores the position of the first task index in the tids array, and the last variable stores the first empty index in the tids array. These values are always wrapped using the size of the array when modified, so the tids array is treated as a cyclic buffer. The done_count stores the first empty position in the done_tids array, and the count variable stores the total number of task indices stored in the queue. Once done_count equals count we know the tids array is empty.

The queue_gettask function which retrieves a task index from a queue is implemented as follows:

```
1  __device__ int queue_gettask(struct queue_cuda *q){
2    int ind, tid = -1;
3    if( q->done_count == q->count)
4      return -1;
5    ind = atomicAdd(&q->first, 1);
6    ind %= cuda_queue_size;
7    while(q->done_count < q->count && (tid = q->tids[ind]) < 0);
8    if(tid >= 0)
9      q->tids[ind] = -1;
10   return tid;
11 }
```

In line 3, the function checks if the queue is empty, and if so returns no index (-1). Otherwise, in lines 5 and 6 the first variable is incremented and wrapped around the length of the queue. Next, the function waits for a valid task index to be present at the position required in line 7. This check is necessary as when adding tasks to the queue, the q->last value is incremented before the index is added to the queue. In line 9 the task index is stored in tid and then the value in the q->tids array is set to be empty (-1). The value stored in tid is then returned.

The function to add a task to a queue (queue_puttask) is implemented as follows:

```
1  __device__ void queue_puttask(struct queue_cuda *q, int tid){
2    int ind;
3    ind = atomicAdd(&q->last, 1) % cuda_queue_size;
4    while(q->tids[ind] != -1 );
5    q->tids[ind] = tid;
6  }
```

The function computes the index of the next position in the queue in line 3, and will block in line 4 if the queue is full (if the queue is full then `first == last`, so none of the task indices in `q->tids` will be -1, which is only updated near the end (line 11) of `queue_gettask`). In line 5 the task index is written to the `q->data` array.

I also need to introduce the mutex operations implemented by CUDA atomic operations that are used on the GPU:

```
1  __device__ void cuda_mutex_lock (int *m){
2    while( atomicCAS( m, 0, 1 ) != 0);
3  }
4
5  __device__ int cuda_mutex_trylock (int *m){
6    if( *m == 0 ){
7      int res = atomicCAS( m, 0, 1 );
8      return res;
9    }
10   return 0;
11 }
12
13 __device__ void cuda_mutex_unlock (int *m){
14   atomicExch (m , 0 );
15 }
```

The mutexes are integer values, where a value of 0 represents the mutex being unlocked and a value of 1 represents the mutex being locked. To lock a mutex, we use the `atomicCAS` function which allows us to check if the mutex is currently unlocked, and lock it if so. This operation behaves as a spinlock. The `cuda_mutex_lock` function blocks until the mutex can be locked, while the `cuda_mutex_trylock` function attempts to lock the mutex, and returns the success of the operation. The `cuda_mutex_unlock` function takes a mutex and sets it to 0 using the `atomicExch` function. It is important that unlocking is only ever performed by the block that locked the mutex, otherwise write conflicts may occur.

The functions to lock and unlock the resources required to execute tasks are implemented as follows:

```
__device__ int lock_resources_for_task(int tid){
int cid, cjd;
cid = cuda_locks[tid].i;
cjd = cuda_locks[tid].j;
if(cuda_mutex_trylocks( &cell_lock[cid] ))
```

```
6   if(cid == cjd || cuda_mutex_trylock(&cell_lock[cjd]))
7     return 1;
8   else
9     cuda_mutex_unlock(&cell_lock[cid]);
10 return 0;
11 }
12
13 __device__ void unlock_resources_for_task(int tid){
14   int cid, cjd;
15   cid = cuda_locks[tid].i;
16   cjd = cuda_locks[tid].j;
17   cuda_mutex_unlock(cell_locks[cid]);
18   cuda_mutex_unlock(cell_locks[cjd]);
19 }
```

The `cuda_locks` variable is a device array that contains information required for the tasks, primarily the indices of the resources required to execute a task.

The `lock_resources_for_task` function checks whether a task is ready to be executed, and returns 1 if so or 0 otherwise. Tasks here are assumed to have either 1 or 2 required resources (as any task in `mdcore` will use either one or two cells), and if they only require a single resource, `cuda_locks[tid].i == cuda_locks[tid].j`. To avoid any potential deadlock due to the Dining Philosophers Dilemma, we enforce that `cuda_locks[tid].i <= cuda_locks[tid].j`. The function first locks `cuda_locks[tid].i`, and if successful, attempts to lock `cuda_locks[tid].j`. If both are successfully locked, then it returns success, else it reverses any locks that were obtained, and returns failure.

The `get_task` function is then specified as follows :

```
1  __device__ int get_task(struct queue_cuda *q, int steal){
2    int tid = -1, cid, cjd;
3    while( (tid = queue_gettask( q ) ) >= 0){
4      if(lock_resources_for_task(tid))
5        break;
6      cuda_queue_puttask( q, tid );
7    }
8  }
```

In lines 3 to 7, the function pulls a task index from the queue using the `queue_gettask` function. If the resources can't be locked, the task index is returned to the queue and the loop repeats.

Unlike on the CPU where mdcore uses 1 queue for each thread, on the GPU we only use 1 queue per SM. Since each SM is capable of executing up to 8 blocks concurrently on the NVIDIA GeForce GTX 480, this means a total of 15 queues, as the card contained 15 SMs, and each queue is shared by up to 8 blocks. Each block can compute which queue was its primary queue using an assembly function to retrieve the executing SM's ID. We use work stealing to improve load balancing between the queues.

At the end of each timestep, the done_tids and tids pointers are switched. The tids arrays then contain the tasks in the order in which they were executed, which is likely to be a better ordering and lead to improved performance in later timesteps.

6.2.2 The Initial GPU Setup in mdcore

To avoid any explicit synchronisation in the task-based kernel, each block was launched with only 32 threads (i.e. a single warp). Since warps are always executed in strict lock-step, all threads wait on data accessed by a single thread in the block (such as the task index to be executed). I chose to launch the minimum number of blocks to maximise occupancy at 25% of theoretical peak (limited by the number of threads in each block), which requires 8 blocks per SM on Fermi architectures, and a total of 120 blocks on the NVIDIA GeForce GTX 480. To attempt to compensate up for the low occupancy, the implementation of the task functions tries to use 4-way Instruction Level Parallelism (ILP), as [50] notes that high performance can still be achieved with low occupancy on GPUs when using ILP.

For example, the interactions are implemented as follows:

```
1  for(pjd = threadIdx.x; pjd < count_j; pjd += blockDim.x){
2    load particle pjd position
3    for(k = 0; k < 3; k++){
4      pjf[k] = 0.0f;
5    }
6    for( ind = 0; ind < wrap_i; i+= 4 ){
7      ilp_part[0] = particles[ind];
8      ilp_part[1] = particles[ind+1];
9      ilp_part[2] = particles[ind+2];
10     ilp_part[3] = particles[ind+3];
11     //compute 4 interactions
12     particles[ind] = ilp_part[0];
13     particles[ind+1] = ilp_part[1];
14     particles[ind+2] = ilp_part[2];
15     particles[ind+3] = ilp_part[3];
```

```
16    }
17    update particle pjd.
18  }
```

Where `count_j` is the number of particles in cell j, `wrap_i` is the number of particles in cell i, and `pjf` is a temporary store for the forces on particle `pjd`. The particle loads in lines 7-10 shift based upon the thread ID (not shown here), which avoids any threads accessing the same particle simultaneously and allowing non-atomic writes when updating the particles in cell i (due to the lock-step nature of the warps).

6.2.3 Sorting the Particles on the GPU

The neighbour-finding algorithm used in `mdcore` is the sorted cell algorithm from Section 4.1.2. To use this algorithm we need to be able to efficiently sort the particle indices on the GPU. We sorted the particles using normalized bitonic sort [38], which implements a parallel sorting network. The sorting network constructed by the algorithm ensures that no two threads access the same data simultaneously in each step, so there are no race conditions during each step of the algorithm. A small sorting network is shown in Figure 6.2.3 The algorithm requires almost no extra memory to perform.

The algorithm was implemented as follows:

```
1  __device__ inline void cuda_sort_descending ( unsigned int *a , int
       count ){
2    int i, j, k, threadID = threadIdx.x;
3    int hi, lo, ind, jnd;
4    unsigned int swap_i, swap_j;
5    for ( k = 1 ; k < count ; k *= 2 ){
6      /* First step. */
7      for ( i = threadID ; i < count ; i += blockDim.x ){
8        hi = i & ~(k-1); lo = i & (k-1);
9        ind = i + hi; jnd = 2*(hi+k) - lo - 1;
10       swap_i = ( jnd < count ) ? a[ind] : 0;
11       swap_j = ( jnd < count ) ? a[jnd] : 0;
12       if  ( ( swap_i & 0xffff ) < ( swap_j & 0xffff ) ){
13         a[ind] = swap_j;
14         a[jnd] = swap_i;
15       }
16     }
17     /* Let that last step sink in. */
18     __syncthreads();
```

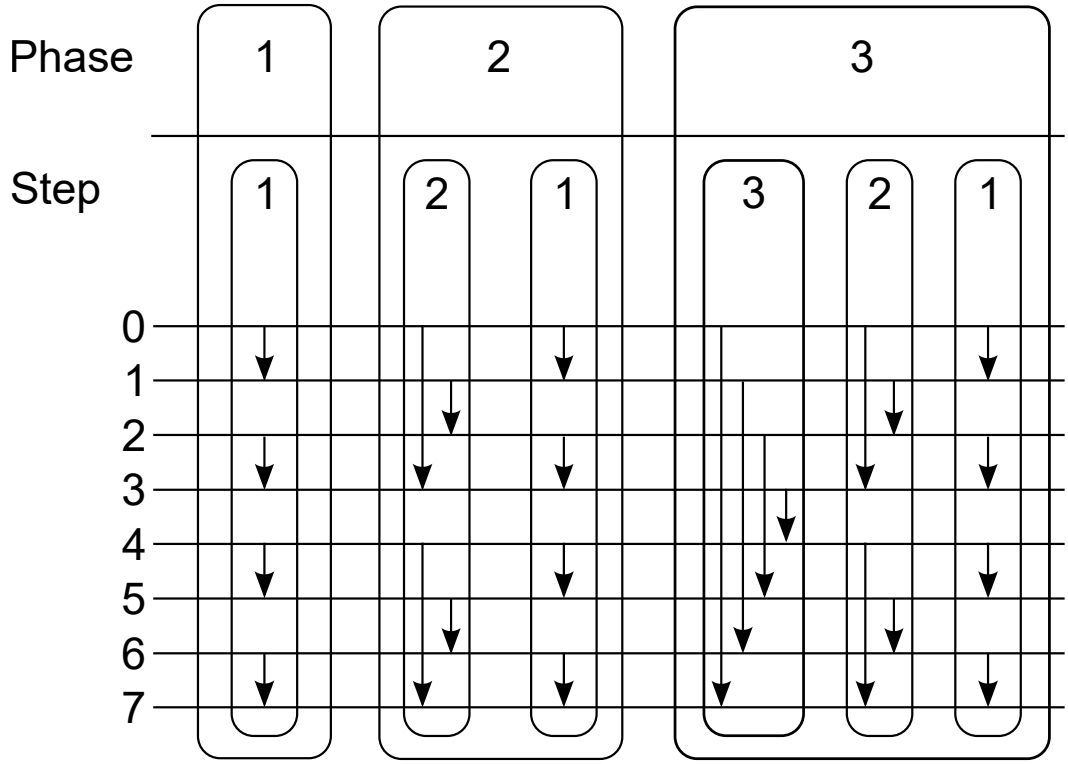


Figure 6.3: The sorting network for normalized bitonic sort on 8 elements. In phase k of the algorithm, steps k to 1 are performed. Each step can be performed in parallel, with synchronization required between steps of the algorithm. Adapted from [38].

```
19      /* Second step(s). */
20      for ( j = k/2 ; j > 0 ; j /= 2 ){
21        for ( i = threadID ; i < count ; i += blockDim.x ){
22          hi = i & ~(j-1);
23          ind = i + hi; jnd = ind + j;
24          swap_i = ( jnd < count ) ? a[ind] : 0;
25          swap_j = ( jnd < count ) ? a[jnd] : 0;
26          if  ( ( swap_i & 0xffff ) < ( swap_j & 0xffff ) ){
27            a[ind] = swap_j;
28            a[jnd] = swap_i;
29          }
30        }
31        __syncthreads();
32      }
33    }
34  }
```

The algorithm takes $\lceil log_2(\texttt{count}) \rceil$ iterations, each made up of two steps. In the first step, each thread takes pairs of values, and changes the order such that the larger value

Problem	Time per step
JAC	14.018ms
ApoA1	69.981ms

Table 6.1: Time to compute the nonbonded interactions for the JAC and ApoA1 MD test cases with the initial task-based setup, using a NVIDIA GeForce GTX 480.

is earlier in the array. In each iteration the two pairs of values are $2^{log(\mathtt{k})}$ entries apart, where k is the current iteration. The second step reorders the segments of the array to accomodate any changes made in step one (by performing each step one performed in previous iterations, in reverse order).

In lines 12 and 26 the code checks `swap_i & 0xffff` and `swap_j & 0xffff` as the array (`a`) being sorted stores two 16-bit values in each `int`, the particle index in the cell, followed by the particle position (along the cell axis).

6.2.4 Initial Results

I tested the setup described above on the JAC (Joint Amber-Charmm test case, 23558 particles, 1.03nm cutoff, $6 \times 6 \times 6$ cell grid) and ApoA1 (Apolipoprotein A1 in water, 92224 particles, 1.20nm cutoff, $8 \times 8 \times 6$ cell grid) benchmarks. The runtime with 120 blocks and 32 threads per block on a NVIDIA GeForce GTX 480 are shown in Table 6.1. I will later use the *STMV* (Satellite Tobacco Mosaic Virus [15]) in water test case, 1,066,628 particles, 1.20nm cutoff, $17 \times 17 \times 17$ cell grid).

As well as these initial results, I also looked at the scaling of the code as the number of blocks increased from 1 to 120, and examined the performance of the different sections of the code as the number of blocks increased. This allowed us to see if there are overheads due to a specific area of the code. The performance examination was done by coding timers into the simulations, using the CUDA `clock64` functions. When enabled, each function was wrapped in these clock functions, allowing us to measure the amount of time spent in certain functions, such as the queue, pair tasks, self tasks etc. These results are shown in Figures 6.4 and 6.5.

These results show the scheduling overhead (get task) reaching over 10% of the runtime with 120 blocks, which is too high of a cost for the task-based scheduler, so it needed to be improved. The required CPU time to run the computation does increase as the number of blocks increase, however the increase is offset by the increased number of active threads. This is common when looking at strong scaling, however the overheads in our case were higher than expected.

The poor performance of the scheduler for the JAC test case is likely due to the lock contention over the cells. In JAC, there are only a total of 216 cells. When running 120

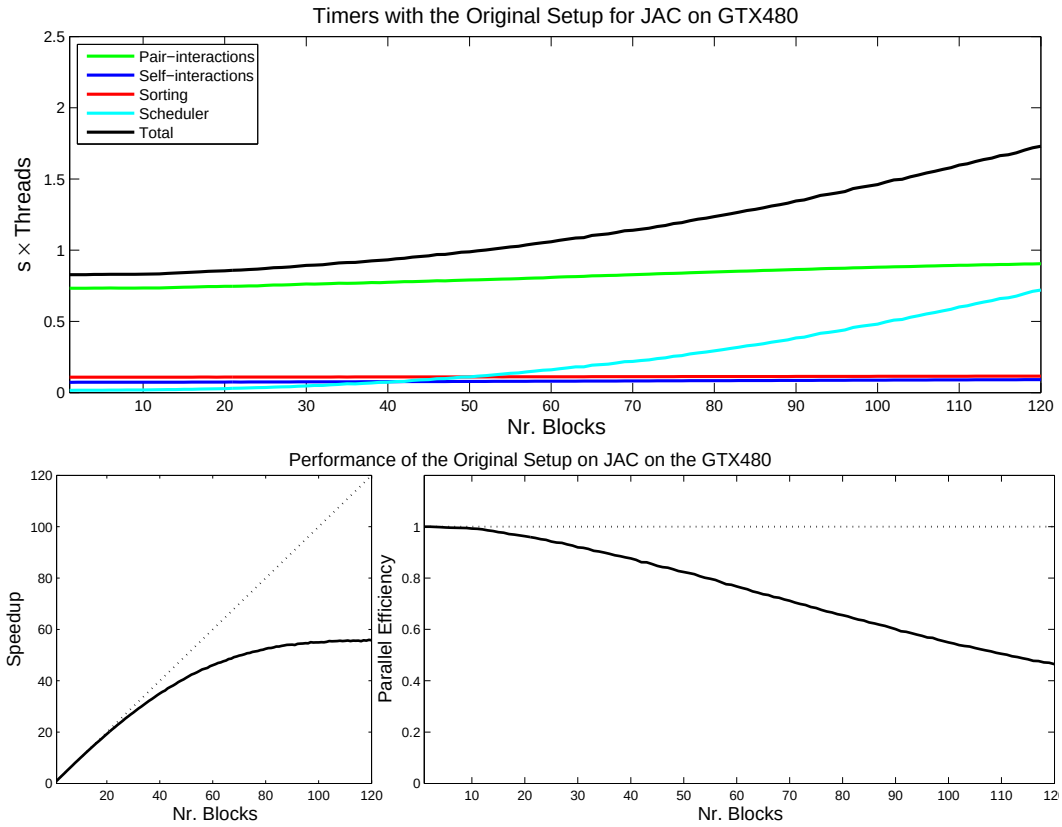


Figure 6.4: Code timing, scaling and efficiency plots on the GeForce GTX480 for the JAC test case. The JAC test case barely speeds up past 80 blocks, and the time spent in `get_task` dramatically grows. This is likely due to a lack of cell pairs available to be locked, leading to some blocks repeatedly accessing the queues and not doing any work.

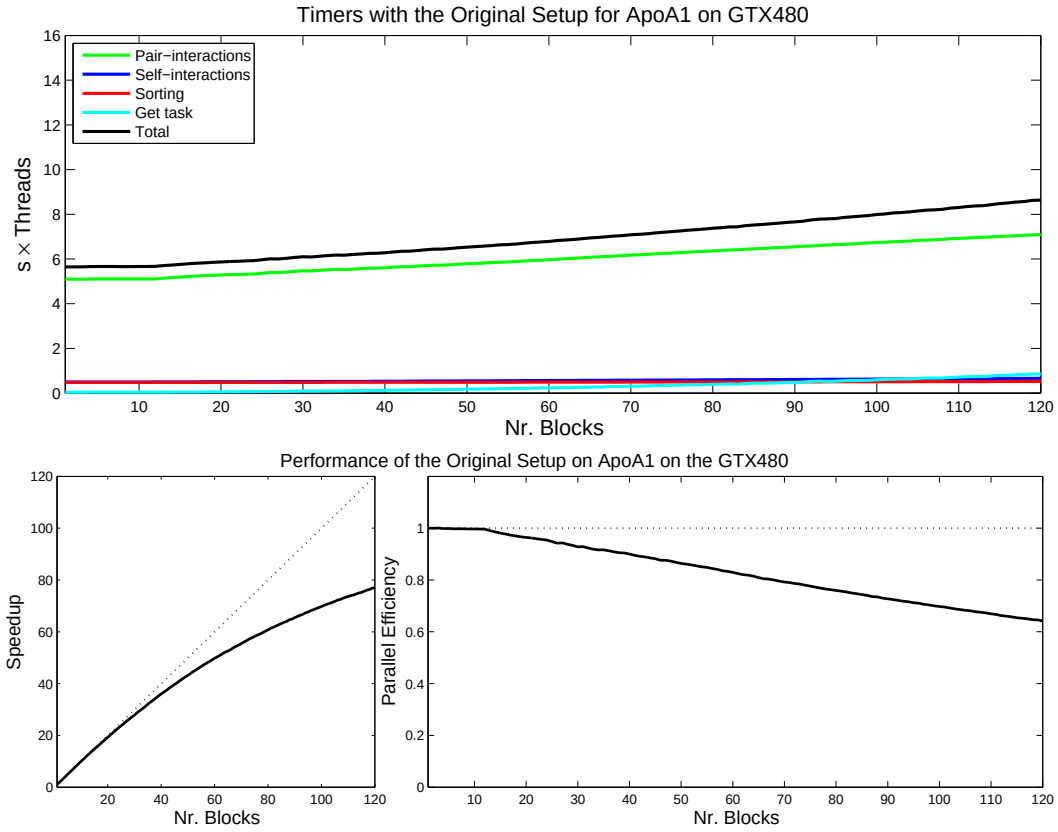


Figure 6.5: The measured performance of the initial setup on the GeForce GTX480 for the ApoA1 test case, as well as scaling. As the number of blocks increases, the cost of task retrieval dramatically grows from $< 1\%$ of runtime to approximately 10%.

blocks, it is already impossible for all blocks to lock a cell pair at the same time. Some blocks may therefore spend a large amount of time attempting to lock a cell/cell-pair before they can perform any computation.

6.2.5 Improving the Scheduler

When I investigated the cost of the scheduler, I found that the most expensive function was the `get_task` function, and not the `queue_gettask` function. This implied that attempting to lock a cell already in use resulted in high overheads with high block counts. I decided to test a variety of alternative approaches:

- Force blocking: The force blocking variant uses only a single queue, and modifies the `get_task` function (page 70) to not lock the cells, but simply break out of the loop in line 7 of the function. The particle interaction functions lock the cells before updating particles using the mutex operations. This variant will block if it cannot obtain the lock inside the interaction function, so if two blocks try to write to the same cell at the same time, one will perform no work and stay in the spinlock until the other has finished.

- Atomic updates: This variant uses the same modifications to the queue as for the force blocking variant. All of the updates on particle forces are replaced by the in built `atomicAdd` CUDA function instead. Atomic operations have higher overhead than normal operations, so updating the particle data will be more expensive with this method.

- Single queue (also called *blocking*): This variant is the same as the original setup. It only uses one queue shared by all of the SMs.

- No work stealing: This variant was the same as the original, except work stealing was disabled after the first timestep. The work stealing was enabled in the first timestep, as with less blocks than streaming multiprocessors the code would otherwise deadlock.

The first two variants were created to test the performance of alternative strategies to avoid race conditions on the GPU with locks or atomic operations inside the interaction functions (rather than using conflicts), while the latter two examined the advantages/overheads of work stealing and multiple queues with the previous setup. The timing plots of these variants are shown in Figures 6.6 and 6.7. The time taken to calculate the short-range electrostatics is shown in Table 6.2.

The force blocking method requires significantly less time to retrieve tasks from the queues with any number of blocks, and there is a slight improvement in runtime.

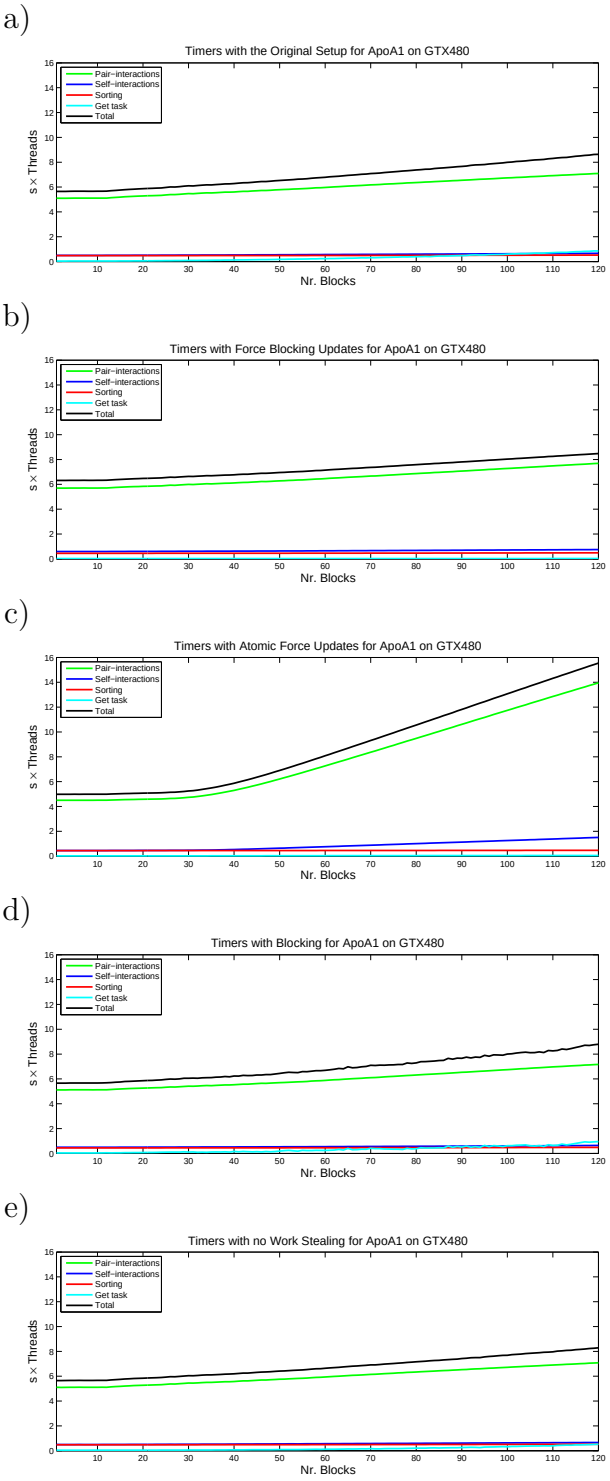


Figure 6.6: Timing plots for the five variants (including the original setup, a) on a GeForce GTX 480 with the ApoA1 test case. b) is the Force blocking variant, which shows improved scheduler performance at high block counts, though the pair interaction cost is slightly higher. c) is the atomic force update variant, which performs by far the best at low block counts. As the number of blocks increases, the performance drops off dramatically, as the overhead of the atomic operations outweighs the computation. d) shows the setup with no work-stealing, which performs similarly to the original setup, however the cost of retrieving tasks is slightly reduced at high block numbers. e) shows the single queue setup, which performs similarly to the original setup, however the cost of the queue accesses fluctuates wildly as the number of blocks is increased.

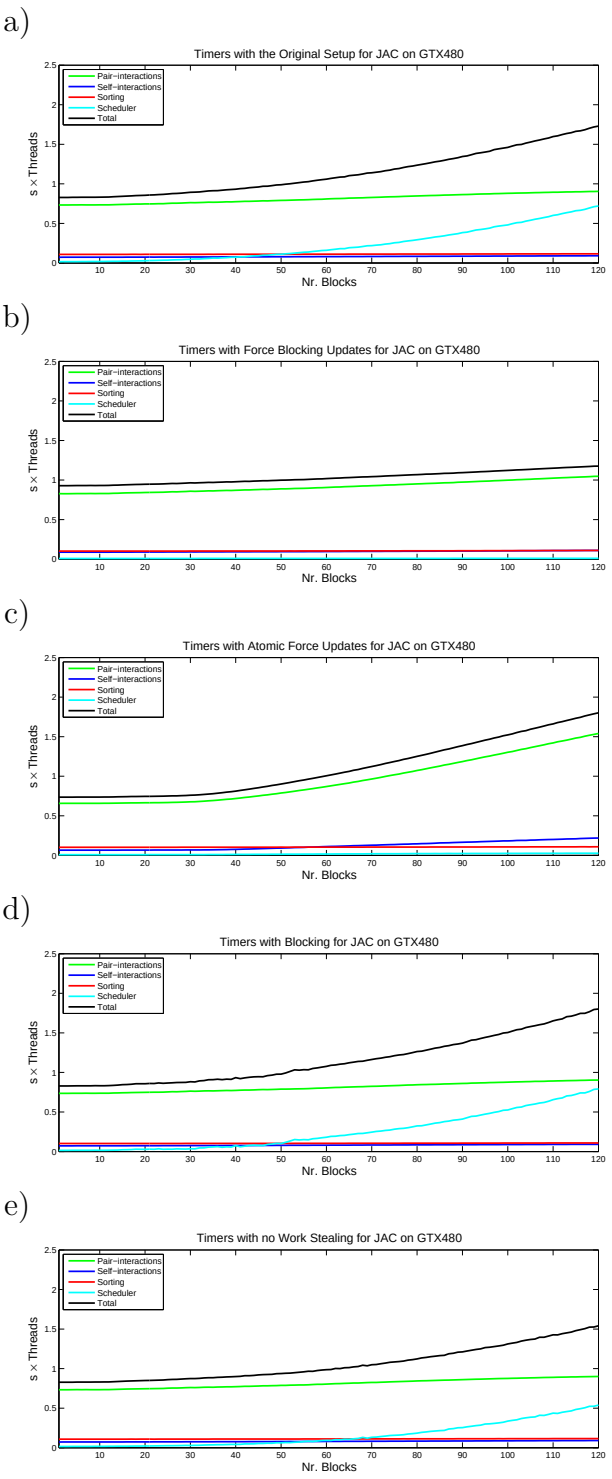


Figure 6.7: Timing plots for the 5 variants (including the original setup, a) on a GeForce GTX 480 with the JAC test case.

Variant	JAC	ApoA1
Original	14.018ms	69.981ms
Force blocking	9.425ms	68.084ms
Atomic forces	14.613ms	129.227ms
Single queue	14.631ms	70.105ms
Nosteal	13.683ms	69.384ms

Table 6.2: Time to compute the nonbonded interactions for the JAC and ApoA1 MD test cases with each of the variants on the GTX480. The timings are averaged over 1000 timesteps for JAC and 500 timesteps for ApoA1. Most of the variants are similar in speed, with minor advantages for the forceblocking interactions and no work-stealing setups. The atomic force updates perform badly on ApoA1, however with only 60 blocks this variant takes 135ms, suggesting the access to/latency of the atomic units is the major limiting factor.
For the JAC results, it is clear that not locking the cells during task retrieval is beneficial (hence the better performance with Force blocking), as there aren't enough cells to let each thread lock an entire cell pair, and the forceblocking setup is significantly faster.

However, the `dopair` and `doself` sections perform 5-10% worse than the original setup. The performance loss can be attributed to the calls to `cuda_mutex_lock`, which will block until the lock is available. With JAC, since all blocks can do work until needing to perform particle updates to main memory, the runtime improves significantly.

The atomic update method works significantly better than the original setup at low block counts (< 30). Not only does it have the same improvement as the force blocking method with respect to the queue access, the `dopair` and `doself` functions are significantly faster when using under 30 blocks. However, once this variant reaches 40 blocks, these functions start linearly slowing down with every additional block launched. Since the GeForce GTX 480 card has Compute Capability 2.0, the atomic units cannot handle so many atomic operations occurring simultaneously. Using atomic operations inside the innermost loop causes a large loss of performance.

The results with the force blocking and atomic update methods suggest that locking cells while in the queue is detrimental to performance, and we should avoid addressing race conditions inside the scheduler.

The single queue and no work stealing variants, which only modified the queue setups, barely differed in runtime from the original setup. Removing work stealing did show a reduction in time spent in the queues by around 50%, whilst using only a single queue had no overall net benefit, though the time spent in the queues fluctuated wildly as the number of blocks changed (as shown in Figure 6.6). This implies the assumed benefits of work stealing do not outweigh the additional cost of queue access.

One of the major issues I noticed was the cost of updating the particle forces in the innermost loops. As we loop through the particles, we need to avoid race conditions,

meaning either locking and unlocking the cell repeatedly in the innermost loop, or using atomic operations in the innermost loop which can cause the loop to slow down dramatically. A possible solution to this issue would be non-symmetric force computations, i.e. computing the forces between a cell pair twice, but only storing the forces for the particles in one cell each time. While non-symmetric force computations result in more computation, it reduces the number of memory writes during the innermost loop, which reduces the cost of the innermost loop.

6.2.6 One-Sided Force Interactions

Future sections also show results with an NVIDIA GeForce GTX690, a dual-GPU card with Compute Capability 3.0. The new generation of card features overhauled atomic operations, and these upgrades result in significantly better performance for atomic operations [36]. To make use of the improved atomic operations, I changed the interaction routines to be one-sided, as I wanted to avoid atomic operations within the innermost loop.

As well as implementing these one-sided force interactions, I created a statically scheduled version using these one-sided interactions. The statically scheduled variant launches two kernels: The first kernel loops through the cells and performs all the sorting required for the sorted interactions; The second kernel loops through the cells and performs all the interactions involving that cell. Each block loops over the cells starting on the cell with index `blockIdx.x`, and increments the index by `gridDim.x` after each iteration. The cell pairs are stored in a global array. Since no blocks would ever modify the particles at the same time, no mutexes or atomic operations are required to safely update particle forces.

6.2.7 Adding Dependencies

The sorting on the GPU is done using bitonic sort, and takes place in each of the pair interaction tasks. This results in sorting 26 times for each cell, whereas if we can perform all the sorts for a cell before any of the interactions involving that cell, we only need to sort 13 times, and reverse the indices (or loop over them in reverse) for the other 13 axes.

To avoid oversorting, I added *sort* tasks, which take a single cell and sort the particle indices along each of the 26 cell axes (by sorting in 13 axes and reversing the indices for the remaining axes). These can be stored and accessed when necessary in the pair interaction tasks. The pair interaction tasks depend on the sort tasks. To enable these dependencies I needed to modify the scheduler.

First the task representation needs to be modified. Initially, all of the tasks are

represented as cell pairs, and store the indices of the cells that are involved in the interaction. With the introduction of sort tasks, we need to be able to differentiate between different types of task, so create a `task_cuda` structure that contains this information:

```
1 struct task_cuda{
2   short int type, subtype;
3   volatile int wait;
4   int flags;
5   int i, j;
6   int nr_unlock;
7   int unlock[task_max_unlock];
8 }
```

The `type` field stores the type of task represented by this structure, whilst the `subtype` field is used to store further information on the task, for example if an interaction only needs data storing on one of the cells. The `wait` field stores the number of unexecuted tasks that unlock this task, i.e. the dependency counter. The `flags` field stores extra information required for the task (for example the shift vector between a cell pair in molecular dynamics), and `i` and `j` fields stores the indices of the cells involved. The `nr_unlock` field stores how many tasks are unlocked by this task and the `unlock` array stores the task indices of all of the tasks unlocked by this task. `task_max_unlock` is a compile time variable which limits how large the `unlock` array can be.

The `get_task` function has to be modified to check all of the dependencies on a task have been correctly resolved before it can be retrieved and executed. I modified the loop in lines 3-7 to:

```
1 while(( tid = queue_gettask( q ) ) > 0 ){
2   if(!cuda_tasks[tid].wait ){
3     if(cuda_tasks[tid].type == task_type_sort)
4       break;
5     if(cuda_tasks[tid].type == task_type_self)
6       if(lock_resources_for_task(tid)) break;
7     if(cuda_tasks[tid].type == task_type_pair)
8       if(lock_resources_for_task(tid)) break;
9   }
10  queue_puttask(q, tid);
11 }
```

Line 2 of the loop checks whether the task's dependencies are all satisfied, and if not, returns the task to the queue. If the task is a self or pair task, the function attempts to lock the cells required. If it is a sort task, it doesn't need to lock the cell as it doesn't directly update the particle data.

When a task is completed, its dependent tasks need to be unlocked, and is done by the first thread in the block, as follows:

```
1 if ( threadID == 0 )
2   for ( k = 0 ; k < cuda_tasks[tid].nr_unlock ; k++ )
3     atomicSub( (int *)&cuda_tasks[ cuda_tasks[tid].unlock[k] ].wait , 1);
```

The wait counters are initially set by looping through the tasks, and incrementing the wait counter of each task which depends on them. The wait counters are set in serial on the CPU before the first kernel execution:

```
1 for( i = 0 ; i < nr_tasks ; i++ )
2   for ( k = 0 ; k < cuda_tasks[tid].nr_unlock ; k++ )
3     cuda_tasks[cuda_tasks[tid].unlocks[k]].wait++;
```

In later timesteps, this is performed in parallel on the GPU. The threads in a single block are executed in parallel over the loop in line 1, and perform line 3 using an `atomicAdd` operation.

6.2.8 Moving Towards 100% Occupancy

Up until now, the kernels all used blocks of 32 threads (i.e. a single warp). Using only a single warp in each block avoids the need for explicit synchronisation within each block, as all the threads in a warp are executed in lock-step. The potential downside of this setup is that the maximum possible GPU occupancy when using blocks of 32 threads is 25% [34], as the GPU requires at least four warps per block to be able to reach 100% occupancy. With only one warp per block, the occupancy is limited by the maximum number of blocks per multiprocessor, capping occupancy at 25%.

In an attempt to reduce the effects of low occupancy, the initial implementation tried using 4-way Instruction Level Parallelism (*ILP*) in the innermost loops. We attempt to make use of ILP by manually unrolling the innermost loop n times (for n-way ILP) to perform multiple particle interactions inside one loop iteration. During compilation, the compiler may be able to reorder the instructions to better utilise the machine's resources (e.g. starting a memory read operation while doing computation on other data). This can also improve instruction pipelining, allowing more operations to

Variant	ApoA1	STMV
With ILP	83.5ms	825ms
Without ILP	84.0ms	821ms

Table 6.3: Time required to compute the nonbonded interactions for the ApoA1 and STMV test cases on the GTX480 with and without ILP.

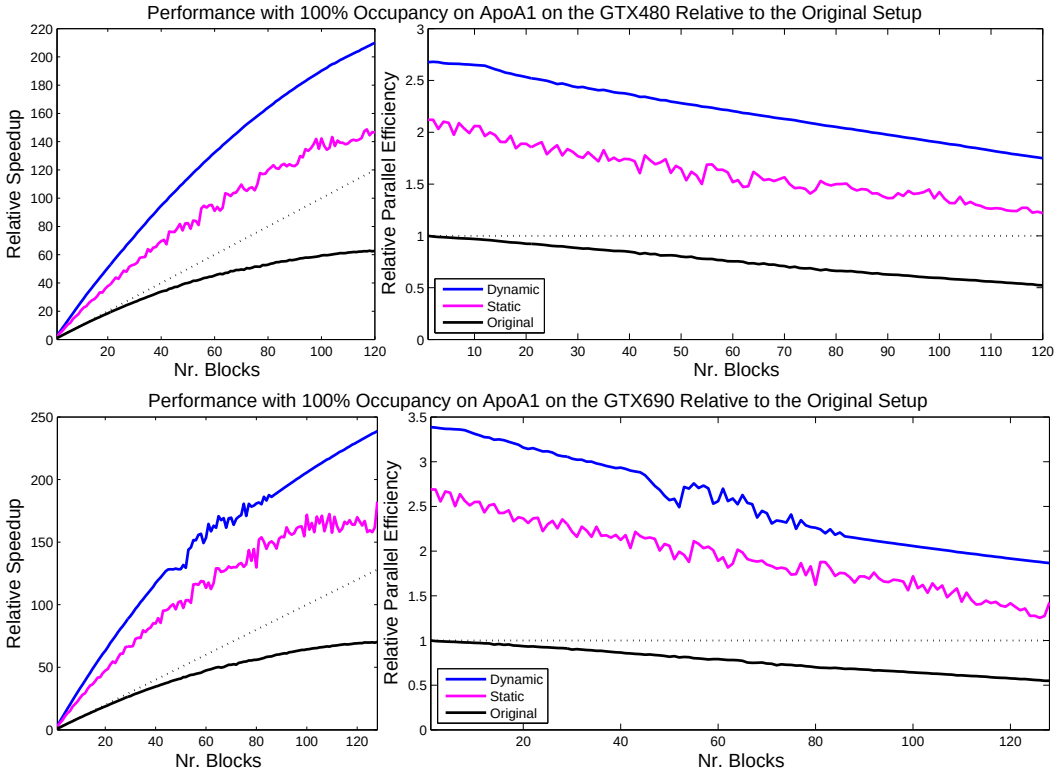


Figure 6.8: Timing plots for the ApoA1 test case with 100% occupancy. All the plots are relative to the original setup. The dynamic variant is approximately $3-3.5\times$ faster on the GTX480 and the GTX690 at the maximum block count on each GPU.

be overlapped. Often, the compiler will manually unroll the loops at high optimisation levels. When I tested the kernels where we attempted to use ILP against an identical kernel without ILP, the benefit is minimal, as shown in Table 6.3 for the NVIDIA GeForce GTX480.

Our results suggest that our attempt at using ILP provides no benefit, and likely meant that using more threads per block would improve performance. This requires modification of the code to explicitly synchronise the warps within each block where necessary using `__syncthreads()`.

As well as moving to 128 threads per block, I decided to use one-sided interaction functions in the kernels, as these allowed us to most effectively exploit the atomic force update method, which I felt was the most efficient method of updating the particles.

The results of increased occupancy are shown in Figures 6.8 and 6.9.

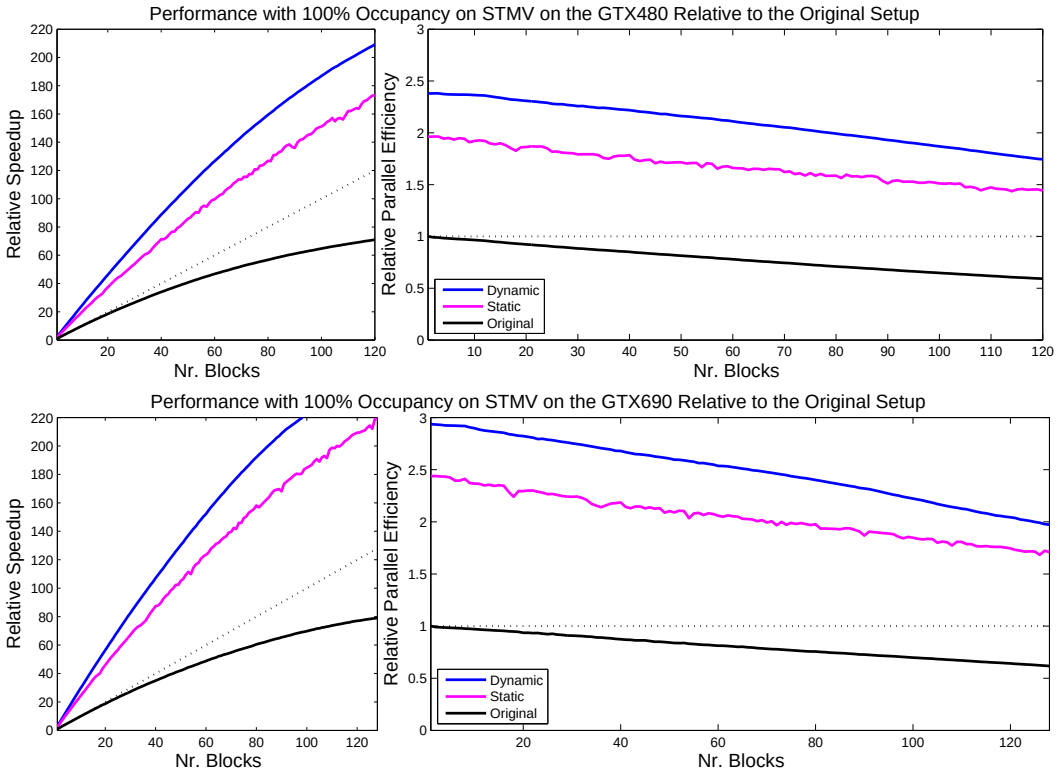


Figure 6.9: Timing plots for the STMV test case with 100% occupancy. All the plots are relative to the original setup. The dynamic variant is approximately $3-3.5\times$ faster on the GTX480 and the GTX690 at the maximum block count on each GPU.

Variant	ApoA1	STMV
Original	104.282ms	809.784ms
Dynamic	27.959ms	253.484ms
Static	36.925ms	292.319ms

Table 6.4: Results for the ApoA1 and STMV simulations on the GTX690 GPU. The Dynamic and Static variants both use 100%, i.e. 128 threads per block and 128 blocks, whilst the Original setup only uses 32 threads per block with 128 blocks.

Variant	JAC	ApoA1	STMV
Serial CPU bonded interactions	14.540ms	84.882ms	781.347ms
Parallel GPU bonded interactions	13.861ms	64.968ms	652.274ms

Table 6.5: Time required to compute a timestep for the all three test cases on the GTX480 with the GPU parallelised bonded interactions or the serial CPU bonded interactions.

Table 6.4 and Figures 6.8 and 6.9 show that increasing the occupancy leads to significant performance gains, despite the requirement to explicitly synchronise the blocks. The increased number of threads executing on the device helps offset the memory latency, which improves the performance.

6.2.9 Running the Bonded Interactions on the GPU

Having already added dependencies to the GPU, it is easy to extend the task setup to allow more types of task, such as the tasks for the bonded interactions. Bonded interactions are a list of interactions that need to be applied to a fixed set of particles. They vectorise relatively straightforwardly. The four types of bonded interactions (bonds, angles, dihedrals and exclusions) all use the same task `type`, however they each have their own `subtype`. The `subtype` is used to decide which function to execute.

Each type of bonded interaction has its own list of particles involved in the interaction, and the task's `i` value stores the index in the bonded interaction list to start iterating from. The `j` value stores the number of bonded interactions that needed to be executed by the task. Each bonded interaction is computed separately, and the forces are applied to the particles using atomic operations.

Results with the parallel bonded interactions are shown in Tables 6.5 and 6.6. The parallel bonded interactions scheme show a significant improvement over the serial bonded interactions executed on the CPU on the larger test cases, but minimal improvement on the JAC test case. The JAC testcase has relatively few bonded interactions, so the increased data transfer requirements probably outweigh the benefits from the parallel bonded interactions. These results show that our scheme can be extended to support more types of task without impacting performance.

6.2.10 Conclusions

The fastest version of the GPU code uses the sorted cell algorithm for neighbour finding inside the self and pair tasks (with an option to switch to pseudo-Verlet lists) and bitonic sort for the sort tasks. The kernel can be executed by any number of threads, though I found best performance was achieved using 128 threads per block, and enough

Variant	JAC	ApoA1	STMV
Serial CPU bonded interactions	9.750ms	80.503ms	656.002ms
Parallel GPU bonded interactions	9.768ms	43.855ms	449.312ms

Table 6.6: Time required to compute a timestep for the all three test cases on the GTX690 with and without the parallel bonded interactions.

blocks to reach 100% GPU occupancy. This makes best use of the GPU architecture, as the oversubscription of threads to physical cores helps hide the memory access latency.

The self and pair tasks use atomic operations to update the particle position, and are one-sided. I chose this scheme as atomic operations on modern GPU hardware are efficient enough to perform inside the interaction function, however I anticipated them being too expensive to perform repeatedly within the innermost loop. The schedule-based strategies for avoiding race conditions on particle updates performed much worse, which I believe is due to extending the amount of time spent in single-threaded regions.

There is also an option to compute the bonded interactions on the GPU in parallel, and this performs significantly better than serial CPU execution of these interactions.

6.3 Using Multiple GPUs with `mdcore`

`mdcore` implements both a hybrid shared-distributed memory algorithm as well as the GPU version discussed in Section 6.2. Many of the issues with distributed memory (or hybrid shared-distributed memory) architectures are mirrored in multi-GPU setups. Both require good load balancing and minimisation of the amount of duplicated computation, whilst distributed and hybrid memory architectures also want to minimise the amount of communication in the system.

6.3.1 Improving MPI use in `mdcore`

`mdcore` uses a data decomposition to parallelise over distributed memory systems, i.e. each cell is assigned to a node and each node runs a task-based computation on all of the cells assigned to the node. `mdcore` uses a synchronous halo exchange to deal with computation near the edge of each subdomain. The original setup in `mdcore` used a bisection method to divide the cells amongst the processors. This recursive function divides the space in whichever dimension has the largest number of cells, and recurses on each of the bisectors until the correct number of partitions have been created.

One issue with the bisection method is that if a dimension has an odd number of cells, the two partitions in that dimension are unlikely have the same number of particles associated with them. Even if the particle distribution was perfectly uniform, this method would introduce load imbalance due to these unequal partitions. Another issue with this method is that it ignores any variance in the computational cost of each cell, e.g. for non-uniform particle distributions. In general, if a cell has more particles it will be more expensive to compute the interactions involving that cell. The bisection method does result in partitions that are contiguous areas of the domain, which is beneficial as it avoids repeating as much computation (as both if a cell pair is split across two nodes, both nodes will compute those interactions).

To improve on the bisection method, we would ideally want a setup that:

1. Is not dependent on the number of cells in each dimension.
2. Takes into account the varying cost of computation associated with each of the cells.
3. Minimises the amount of repeated computation by keeping contiguous partitions when appropriate.

One common approach to load balancing for distributed memory systems is to use graph partitioning methods. To use a graph partitioning approach, we need to find a way to represent the system as a graph. Fortunately, the cell and task-based approach make this relatively straight forward. We create a node in the graph for each cell in the system, and create an edge between each cell pair that interact (i.e. are within r_c of each other). The edges can be added by looping through the `pair` tasks. When a partition breaks an edge in the graph, it means the task represented by the edge is duplicated. If we can minimise the amount of duplicated work, we may get better performance.

Without node and edge weights, the partitioning algorithm will not perform significantly better than the bisection method, so it is important to give reasonable estimates of these. One important factor when estimating these is that cell pairs that share a face will usually be more expensive to compute than cell pairs that share an edge or corner, as more particles will be within r_c.

Using a random, uniformly distributed particle distribution with cells of size r_c we can compute the rough percentage of particles that will be within r_c along the cell axis. I used a simple Monte-Carlo simulation to compute these values, and they are shown in Table 6.7.

From these values, we can compute an estimate of the cost of each task. For self interaction tasks, we use c_i^2, where c_i is the number of particles in the cell. For pair

Position	Percentage of particles within r_c
Face	50.0%
Edge	16.2%
Corner	3.62%

Table 6.7: Percentage of particles within r_c along the cell axis between a cell pair that share a face, edge or corner.

tasks, we use $c_i \cdot c_j \cdot K$, where c_i and c_j are the number of particles in the two cells, and K is the percentage of particles within r_c as shown in Table 6.7. Each node's weight is equal to the sum of all of the tasks' costs that involve the cell represented by the node, and each edge was weighted as the cost of the task it represents. The resulting graph is partitioned using the METIS graph partitioning library [29].

A scaling plot of ApoA1 on the Cosma 4[1] machine is shown in Figure 6.10. The METIS partitioning approach performs better than the bisection method on more than two nodes. The bisection method performs poorly on three nodes, as it cannot partition the space into three segments effectively.

Figure 6.11 shows a scaling plot of ApoA1 with only MPI. The METIS partitioning method works much better overall, achieving 71% parallel efficiency at 8 nodes. The bisection method achieves 61% parallel efficiency at 8 nodes, however it only performs well when using a power of 2 number of MPI ranks. This is due to uneven partitions when trying to divide an $8 \times 8 \times 6$ cell grid into an odd number of partitions(the largest partition is likely the same as for the previous power of 2, and the largest partition will limit the performance).

6.3.2 Using the Graph-Partitioning Approach to Enable Multiple GPU Setups

We can use the same partitioning models (bisection and graph partition-based) as above to partition the work for multi-GPU systems. In addition to the data partitioning, we also use CUDA streams to enable asynchronous data transfer between the host and the GPU. The use of these streams and asynchronous data transfers allows data transfer to one GPU while the other is doing computation, meaning the first GPU can start on its workload while the data is still moving to the second GPU in the system. The partitioning is still performed on the CPU.

[1] 12 core nodes with 2x Intel X5650 @ 2.67GHz

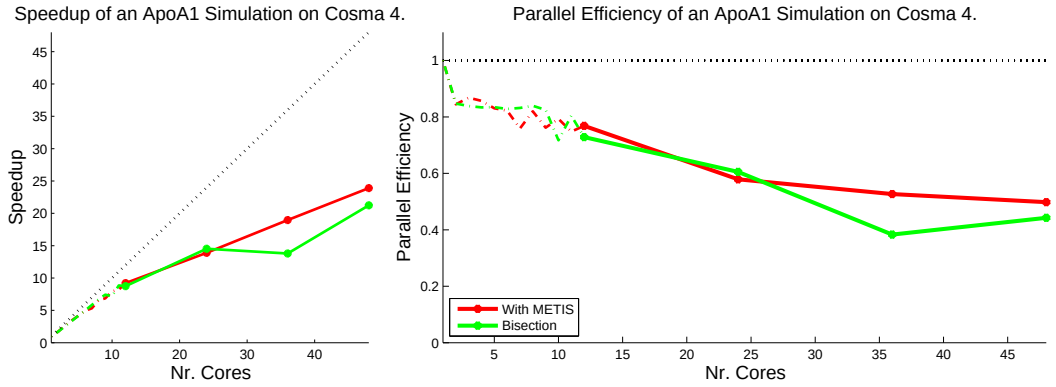


Figure 6.10: Results of an ApoA1 simulation on the Cosma 4 machine, with 12 cores per node. Scaling within nodes is shown with a dashed line, while scaling with multiple nodes is shown with a solid line. The code performs better when using METIS to perform the partitioning when executed on more than 2 nodes, notably the performance on 3 nodes with the bisection method is poor, due to an uneven partitioning.

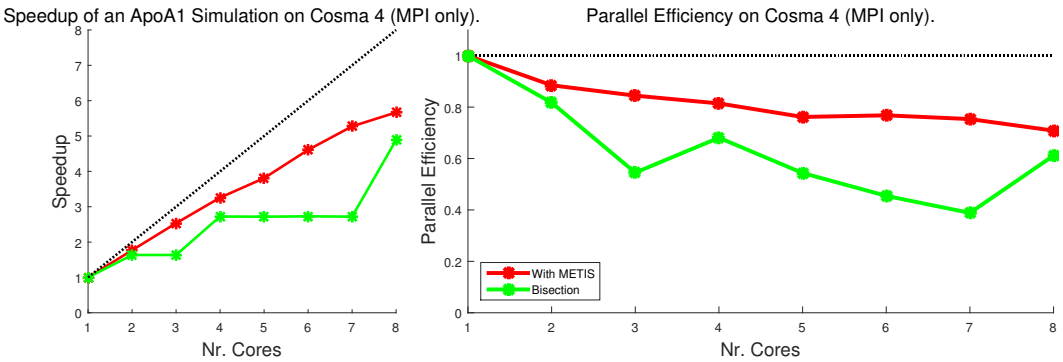


Figure 6.11: Results of an ApoA1 simulation on the Cosma 4 machine, with 1 thread per node. Each MPI rank is executed on a separate node. The bisection method does not improve except when using a power of 2 number of ranks, while the METIS partitioning approach scales with any number of MPI ranks.

The average time taken per timestep with the METIS graph partitioning setup is shown in Tables 6.8 and 6.10 using the GTX480 and GTX690 respectively. Comparing these results to the results with the parallel bonded interactions in Tables 6.5 and 6.6 shows there is a larger benefit from using only the parallel bonded interaction scheme than from solely using multiple GPUs. This is due to the bonded interactions taking a large percentage of the runtime of each timestep (36.879ms for ApoA1 on the GTX690 machine). Once we combine the parallel bonded scheme with the utilisation of multiple GPUs the code performs approximately twice as fast on the GTX690, and 1.5× faster on the GTX480 machine.

The main performance limiters are the load balancing of the GPUs (for example on the ApoA1 test case with the GTX690, the two cards average 12.975ms and 15.275ms respectively), and the movement of data to and from the CPU (whilst the kernel with higher execution time averages 15.275ms per timestep, the overall runtime for the GPU section of a timestep is 26.153ms, meaning over 11ms of the runtime of each timestep is used on data transfer).

I also ran the code using the bisection method, and these results are shown in Tables 6.9 and 6.11. On the JAC and ApoA1 test cases, the bisection method tends to perform slightly better than the METIS partitioning, however the METIS method is significantly better on the STMV test case. The STMV test case is the only one of the three which has an odd number of cells in each dimension. If the particles are reasonably evenly distributed, I would expect this to lead to a larger amount of load imbalance than the ApoA1 and JAC test cases. Overall, the METIS partitioning method doesn't appear to perform better than the bisection method when the number of cells can be split evenly using bisection, but performs better when the bisection method is expected to perform poorly.

6.3.3 Issues with the Multiple GPU Setup

While the cost to compute the nonbonded interactions with multiple GPUs is reasonably load balanced, the overall performance of the code does not improve significantly due to the data packing and data transfer methods used in `mdcore`. Additionally, the computation time on each GPU is lower, but the overall data transfer time does not change, resulting in poor scaling overall, as the data transfer begins to use a large percentage of the runtime in each timestep.

It is clear from these results that to use multiple GPUs effectively we need to solve the issue with data transfer in the system, most likely by performing the all of the molecular dynamics on the GPU, which means only the data from the halo regions needs transferring in every timestep. This was not investigated in this work.

Variant	JAC	ApoA1	STMV
Single GPU version	14.540ms	84.882ms	781.347ms
Multi GPU version	12.490ms	73.709ms	688.339ms
Multi GPU version with parallel bonded interactions	**12.476**ms	**52.830**ms	**540.700**ms
Multi GPU version with psuedo-Verlet Lists	11.024ms	73.165ms	696.859ms

Table 6.8: Average time required to compute a timestep for the all three test cases on the GTX480 using the METIS partitioning method.

Variant	JAC	ApoA1	STMV
Single GPU version	14.540ms	84.882ms	781.347ms
Multi GPU version	12.143ms	75.255ms	705.829ms
Multi GPU version with parallel bonded interactions	12.130ms	51.934ms	557.677ms
Multi GPU version with psuedo-Verlet Lists	10.760ms	74.613ms	701.990ms

Table 6.9: Average time required to compute a timestep for the all three test cases on the GTX480 using the bisection partitioning method.

Variant	JAC	ApoA1	STMV
Single GPU version	9.750ms	80.503ms	656.002ms
Multi GPU version	8.739ms	70.312ms	546.436ms
Multi GPU version with parallel bonded interactions	8.733ms	37.312ms	**331.697**ms
Multi GPU version with psuedo-Verlet Lists	7.610ms	69.225ms	563.387ms

Table 6.10: Average time required to compute a timestep for the all three test cases on the GTX690 using the METIS partitioning method.

Variant	JAC	ApoA1	STMV
Single GPU version	9.750ms	80.503ms	656.002ms
Multi GPU version	8.385ms	69.887ms	558.907ms
Multi GPU version with parallel bonded interactions	8.405ms	**35.741**ms	344.495ms
Multi GPU version with psuedo-Verlet Lists	**7.297**ms	68.876ms	554.445ms

Table 6.11: Average time required to compute a timestep for the all three test cases on the GTX690 using the bisection partitioning method.

6.4 Task-Based SPH on GPUs

As part of the SWIFT[24] project, I began working on an SPH implementation using CUDA. As discussed in Section 4.2.2, SWIFT uses a task-based algorithm to compute SPH on the CPU, so we felt using the same task scheduler used for `mdcore` might yield similar runtime improvements on the GPU.

I implemented the same task-based model as for `mdcore` in SWIFT, and began to work on SIMT parallelising the task functions to be executed on the GPU. I chose to work on the sort tasks, density tasks, ghost tasks, and force tasks

SPH is more difficult to implement on the GPU as the particles have dynamic smoothing lengths, which means more branch divergence when finding particle neighbours. Furthermore, to speed up SPH simulations, not all particles need to directly interact in every time step (particles that interact in a given timestep are called *active*). This is very different to `mdcore`, where all particles interact in every timestep.

The sort functions were implemented in a similar manner as in `mdcore`, utilising bitonic sort and parallelising the sorting network between the threads in a block.

It was difficult to SIMT parallelise the ghost tasks on the GPU. In these tasks the algorithm checks whether the all of the values calculated in the density tasks are correct. If not, these values are recalculated for the subset of particles which have incorrect values. These are commonly only incorrect for a tiny percentage of the particles in any given cell, resulting in the majority of the threads in a warp idling while they wait on other threads to calculate these values. The performance of these tasks was further worsened due to the recursion present in the tasks, which avoids performing this computation on large cells.

The GPU tasks to compute the densities and forces are a SIMT parallelised version of the one-sided CPU routines, with atomic updates to particles to ensure no race conditions. The force tasks require the maximum value of a variable computed for any particle (called the signal velocity, `v_sig` or `v_sig_stor`) in each cell to be found. The reduction can be done within each warp using the CUDA `__shfl` operations on newer architectures:

```
1 for( k = 16 ; k > 0 ; k >>= 1 ){
2   v_sig_stor = fmaxf(v_sig_stor , __shfl_xor( v_sig_stor , k ));
3 }
4 if( threadIdx.x % 32 == 0 )
5   while( v_sig_stor > ( v_sig = v_sig_shared ) )
6     atomicCAS( (int *)&v_sig_shared , __float_as_int(v_sig) ,
          __float_as_int(v_sig_stor) );
7 __syncthreads();
```

Task type	GPU Runtime	CPU Runtime
Density	22%	48%
Force	52%	38%
Ghost	16%	1%

Table 6.12: Runtime breakdown of the GPU and CPU SPH kernels into the main work sections. Any remaining runtime is taken up by the particle sorting and time integration scheme (CPU only), as well as any other overheads.

```
8  v_sig_stor = v_sig_shared;
9  if( threadIdx.x == 0 )
10   while(v_sig_stor > ( v_sig = ci->vsig ) )
11     atomicCAS( (int *) &ci->vsig , __float_as_int(v_sig ) ,
           __float_as_int(v_sig_stor) );
```

Lines 1-3 synchronise the maximum value of the signal velocity within a warp. Lines 4-6 force synchronisation of the value across the entire block, as `v_sig_shared` is a shared memory value. The final lines of the code segment updates the signal velocity in the cell structure. Since the CUDA inbuilt `atomicCAS` function only works on integers, the `__float_as_int` function is needed, which lets you pass floats to the function.

6.4.1 Results

I tested the code on a Sedov Blast[41] test case with 1 million particles, and a breakdown of the GPU kernel's runtime is shown in Table 6.12. The ghost tasks perform much worse relatively on the GPU.

The ghost tasks on the GPU were split into 2 sections, which follow recursion to the leaves of the tree. The first section calculated which particles needed to be recalculated, and zeroed their computed values. For all other particles, some additional computation takes place, which ensures all the values required for the force interaction are ready. This section of the task has branch divergence which worsens the SIMT parallelism of the code on the GPU.

The second section of the ghost tasks recomputed the values for the necessary particles. This involved interacting a small subset of particles (sometimes even a single particle) against each neighbouring cell.

We decided to halt work on the GPU SPH code, as the CPU implementation was still rapidly developing, and felt it was more important to work on other areas until the CPU code was more mature.

6.4.2 Future work for GPU SWIFT

The main area to focus on is the implementation of the ghost tasks on the GPU.

Firstly, the recursion through the octree to the leaves should be removed, or ghost tasks should be created directly on the leaf cells. Each of these possible solutions has disadvantages. The former leads to large ghost tasks, and makes finding the correct interactions for the particles more difficult. The latter involves potentially large numbers of tasks and dependencies.

Secondly, the amount of work done in branches while looping over the particles should be minimised (to reduce the cost of branch divergence), and the values that are needed for the force interaction should be computed only once all of the particles in the cell have corrected density values.

A further consideration that I did not investigate was how the implementation would deal with situations where not all particles were active in a given timestep.

6.5 Extending QuickSched to GPGPUs

This work was published in Parco 2015 - Minisymposia: Is the Programming Environment ready for Hybrid Supercomputers?
An early version of this work was presented at UKMAC 2014.
Results in this section were run with CUDA 7.0 on the NVIDIA GeForce GTX690.

Our results to this point with Task-Based Parallelism on GPUs showed the method had merit on these devices. However, requiring the user to implement the data movement as well as the task infrastructure requires a lot of work from the user to get efficient code, which makes the approach unappealing to use. To ease data movement between the GPU and the CPU, we decided to integrate data transfer into the scheduler by extending the resources already defined in QuickSched.

CUDA allows the user to allocate page-locked memory that is accessible to the device directly (known as *pinned* memory). Using pinned memory and the Unified Virtual Address Space in CUDA makes accessing data in the host CPU's memory straightforward from CUDA kernels, and allows data transfer to take place during the kernel, instead of before kernel execution. This is discussed in detail in Section 6.5.6.

The method we decided to use to integrate the data transfer was to create *load* and *unload* tasks which deal with the data movement in the system. Ideally, we would like these data transfer tasks to be created automatically by the scheduler, and be executed as part of the main task loop in parallel with work whenever possible.

6.5.1 The QuickSched Model and Using it with GPUs

The QuickSched programming model introduced in Section 3 defines resources as a structure that virtually represents data (or other resources in the system). To allow the utilisation of the resources to implement the data movement during the GPU kernel, I needed to extend the resource structures as follows:

```
1  struct res{
2    lock_type lock;
3    volatile int hold;
4    int owner;
5    int parent;
6    /* Pointer to data on the CPU */
7    void* data;
8    /*Size of the data (in bytes) associated with this resource */
9    int size;
10 #ifdef WITH_CUDA
11   /* Pointer to the data on the GPU */
12   void* gpu_data;
13   /* Index of the load task for this resource, if required.*/
14   int task;
15   /* Index of the unload task for this resouce, if required.*/
16   int utask;
17 #endif
18 };
```

I added a `data` pointer, as well as the `size` of the data associated with the resource. I also added three GPU-specific fields, the `gpu_data` pointer and two integers which store the task IDs of the load and unload tasks associated with the resource.

The `data`, `size` and `gpu_data` are provided to the scheduler when the resources are created, and in the case of hierarchical resources the `data` of a child resource needs to be fully enclosed by the `data` associated with its parent. This can be easily verified when the child is created by comparing the pointers:

```
1  if( s->res[ id ].parent != qsched_res_none && data != NULL){
2    char* parentdata = (char*)s->res[ parent ].data;
3    char* childdata = (char*) data;
4    int pos = childdata - parentdata;
5    if(pos < 0 || pos + s->res[id].size > s->res[parent].size){
6      error("Data for a child resource must be contained in the data of
           its parent");
```

```
  }
}
```

I also needed to define three reserved task types for the *load*, *unload* and *ghost* tasks. An alternate queue data structure is needed for GPU QuickSched, so I copied the queue setup as is used in GPU `mdcore`. The scheduler object remains unchanged from the CPU version of QuickSched, as the scheduler is never used on the device, but only for the setup performed on the CPU.

6.5.2 Modifying the Dependency Implementation from `mdcore`

One inconsistency between our CPU and GPU implementations of Task-Based Parallelism is that in the CPU implementation, tasks are not enqueued until they had no active dependencies, whereas all of the tasks were enqueued initially on the GPU. The GPU implementation of the task queue needed to perform additional checks when retrieving tasks to ensure the task was ready to be executed, i.e. the task's wait counter is 0.

I modified the GPU implementation of the task scheduler to mirror the CPU, and only enqueue a task once all of its dependencies are satisifed. Since enqueuing a task only costs two atomic operations, it is cheaper than having to retrieve another task when an unready task is retrieved.

I changed the CPU setup code to only place tasks with an initial `wait` of 0 into the GPU task queue. I also modified the code that is executed when a task is completed to enqueue tasks that are ready:

```
for(i = threadIdx.x; i < tasks_cuda[tid].nr_unlocks; i+= blockDim.x){
  if( atomicSub( &tasks_cuda[tasks_cuda[tid].unlocks[i]].wait, 1) == 1 ){
    cuda_queue_puttask( cuda_queue, tasks_cuda[tid].unlocks[i]);
  }
}
```

where `tid` is the task index of the completed task, and `tasks_cuda` is the task array on the GPU. This can also be parallelised, as the queue is completely thread-safe.

Finally I needed to modify the `get_task` function. Since any task retrieved by `queue_gettask` is guaranteed to be ready to execute, we no longer need to check the tasks `wait` counter, and can just return the retrieved task index:

```
__device__ int get_task ( struct queue_cuda *q ){
  int tid = -1;
```

```
  while ( ( tid = queue_gettask( q ) ) >= 0 ){
    break;
  }
  if ( tid >= 0 ){
    q->rec_data[ atomicAdd( (int *)&q->rec_count , 1 ) ] = tid;
  }
  return tid;
}
```

6.5.3 Implementing Conflicts with the new Task Setup

QuickSched defines conflicts to occur between pairs of tasks that lock one or more of the same resources. Since this definition is central to the CPU performance, I decided to implement conflicts in the GPU version of QuickSched too. Due to the previous experiences in `mdcore`, they are disabled by default (they can be enabled using a define). The conflicts mirror the implementation on the CPU i.e. whenever a task is retrieved from the queue, the resources associated with it are locked, and if the resources can all be locked then the task is executed. If they cannot be retrieved then the task is returned to the queue and the next task in the queue is tested.

On the GPU, the resource locks are implemented using the same mutex strategies as in `mdcore`. The locking functions (`cuda_trylock`, `cuda_lock` and `cuda_unlock`) are implemented identically to the mutexes used in early versions of `mdcore`. To lock a task, the scheduler has to loop through all of the resources locked by the task and attempt to hierarchically lock them. The function is implemented as follows:

```
__device__ int cuda_locktask ( int tid ){
  int k;
  struct task *t;
  t = &tasks_cuda[tid];
  for ( k = 0 ; k < t->nr_locks ; k++ )
    if ( cuda_lockres( t->locks[k] ) == 0 )
      break;
  if ( k < t->nr_locks ){
    for ( k -= 1 ; k >= 0 ; k-- )
      cuda_unlockres( t->locks[k] );
    return 0;
  }else{
    return 1;
  }
}
```

The `cuda_lockres` and `cuda_unlockres` functions implement the hierarchical locking and unlocking of the resources, and are discussed below. In line 5 we loop over the locks and attempt to lock them. If any of the locks are unsuccessful, we quit the loop early. In line 8 we check if all of the resources were locked successfully. If so then the function returns that it was successful (line 13), and the task can be executed. If not, it loops back through the resources that were locked successfully and releases them (line 9 and 10), then returns that the locking failed (line 11).

Since the resources can be hierarchical, successfully locking a resource requires both atomic compare-and-swapping the resource's lock, and holding all of the resources' parents. To hold the parents we have to temporarily lock the parent, and if successful, atomically increment the `hold` counter. The `cuda_lockres` function is then implemented as follows:

```
1  __device__ int cuda_lockres ( int rid ){
2    int finger, finger2;
3    if ( res_cuda[rid].hold || cuda_trylock( &res_cuda[rid].lock ) )
4      return 0;
5    if ( res_cuda[rid].hold ){
6      cuda_unlock( &res_cuda[rid].lock );
7      return 0;
8    }
9    for ( finger = res_cuda[rid].parent ; finger != qsched_res_none ;
         finger = res_cuda[finger].parent ){
10     if ( cuda_trylock( &res_cuda[finger].lock ) )
11       break;
12     atomicAdd((int *) &res_cuda[finger].hold , 1);
13     cuda_unlock( &res_cuda[finger].lock );
14   }
15   if ( finger != qsched_res_none ){
16     cuda_unlock( &res_cuda[rid].lock );
17     for ( finger2 = res_cuda[rid].parent ; finger2 != finger ; finger2 =
           res_cuda[finger2].parent )
18       atomicAdd((int *) &res_cuda[finger2].hold, -1 );
19     return 0;
20   }else
21     return 1;
22 }
```

In lines 3-4, the function checks if this resource is held (i.e. one of its child tasks is currently locked), and if not attempts to obtain the lock on the resource. If this is

unsuccessful, then the function failed to lock the resource. If successful, it again checks if the resource has been held in line 5. This needs to be checked again as it is possible for the resource to have been held while the function was waiting to execute the `trylock` function. If it has been held, the lock is released (line 6) and the function returns that it failed. The function then loops up the resource tree, and attempts to increment the hold counter of each of the parents (lines 9-14). If it can increment the hold counter of all the parents, then the function completes and terminates successfully. If any of the parents are locked, then it terminates the loop early, and executes the if statement in line 15. In the if statement, the resource is unlocked in line 16, and then the hold counter of all of the held parents are decremented in lines 17 and 18. The function then terminates unsuccessfully.

The `cuda_unlockres` function is implemented as follows:

```
1 __device__ void cuda_unlockres ( int rid ){
2   int finger;
3   cuda_unlock( &res_cuda[rid].lock );
4   for ( finger = res_cuda[rid].parent ; finger != qsched_res_none ;
        finger = res_cuda[finger].parent )
5     atomicAdd( (int *) &res_cuda[finger].hold, -1 );
6 }
```

In line 3, the function unlocks the resource. In lines 4-5 the function loops over the parent resources and decrements their hold counters.

We need to release all of the locks after a task is executed. This is done through an additional function, called `cuda_done`, which loops over all of the resources locked by the task and calls the `cuda_unlockres` function on them.

The `get_task` function (defined in section 6.2.1) was also modified to use the `cuda_locktask` function, lines 4-6 were changed to:

```
1 while ( ( tid = cuda_queue_gettask( q ) ) >= 0 ){
2   if( cuda_locktask(tid) == 1 )
3     break;
4   atomicAdd((int*)&q->nr_avail_tasks, -1);
5   cuda_queue_puttask ( q , tid );
6 }
```

Rather than breaking out of the loop immediately, it tries to lock the resources associated with the tasks in line 2, and only breaks if it is successful in locking the resources. If it is unsuccessful, it places the task back into the queue in line 5. The

decrement of `q->nr_avail_tasks` in line 4 is necessary as the `cuda_queue_puttask` function increments this value, but in this case the number of tasks in the queue does not increase.

6.5.4 Creating the Load and Unload Tasks

The *load* and *unload* tasks manage the movement of data between the CPU and GPU. The load tasks are scheduler-created tasks that copy data from CPU memory to GPU memory, while the unload tasks copy data from GPU memory to CPU memory, and are also created by the scheduler.

When creating the load and unload tasks we want to ensure the following:

1. The dependencies are added in the correct place in the dependency array. The dependencies are sorted by the unlocking task, and resorting the dependencies is computationally expensive.

2. The dependencies added enforce a correct computation, i.e. no task executes before the data required is copied to the device, and no data is copied back to the host until all tasks that write to that data are complete.

3. In the case of hierarchical resources, we transfer small amounts of data (i.e. resources closer to the leaves) unless the data associated with the sub-resources is too small for the transfer to be efficient. If the data transferred is too small, then the overheads of copying the data will outweight the benefits of performing the data transfer as part of the computational kernel.

4. Ideally we want to add as few dependencies as possible as there is some overhead to adding large numbers of dependencies, as every dependency in the system costs additional atomic operations.

The first requirement is relatively straightforward to implement. For each task, the scheduler already stores `nr_unlocks`, `nr_locks` and `nr_uses`. For each locked and used resource the implementation will create one unload task that is unlocked by the locking/using task, so we need to add `nr_locks` + `nr_uses` gaps between each task's dependencies when creating the new dependency list.

As well as enforcing that resources must be fully contained within their parents, it is necessary to ensure the `data` represented by any pair of resources never overlap. As we already check the first criteria when resources are created, to check the second we only need to check:

1. The highest level resources (i.e. resources with no parents) never overlap.

2. Any two resources that share a parent don't overlap.

To check these requirements, we use bucket sort to sort the tasks by ascending parent ID (where resources without a parent have a parent ID -1). We then use bucket sort on each of the subarrays (containing tasks that share parents) to sort the resources by the memory address pointed to by each resources' `data` pointer. Once we have this array, first sorted by parent ID, then memory address, we can loop over the array and check if task i and task $i+1$ share a parent ID. If so, we can then use pointer arithmetic to check `res_data[i] + s->res[res[i]].size <= res_data[i+1]`, where `res_data` contains the sorted data pointers, and `res` contains the IDs of the corresponding resources. If this is true, the resources don't overlap. If any of the resources overlap, the algorithm halts and returns an error message.

Once the extension of the dependency array and sorting of the resources is complete, we can create the load and unload tasks. The function that creates the load and unload tasks is a recursive function:

```
1  void qsched_create_loads(struct qsched *s, int ID, int size, int
       numChildren, int parent, int *res, int *sorted){
2    int i,j;
3    int task, utask;
4    if(numChildren > 0 && size/numChildren > 128*sizeof(int)){
5      task = qsched_addtask( s, type_ghost, task_flag_none, NULL, 0 , 0 );
6      qsched_adduse(s, task, ID);
7      s->res[ID].task = task;
8      utask = qsched_addtask( s , type_ghost, task_flag_none, NULL, 0 , 0);
9      qsched_adduse(s, task, ID);
10     s->res[ID].utask = utask;
11     for(i = sorted[ID]; i < sorted[ID+1]; i++){
12       qsched_create_loads(s, res[i], s->res[res[i]].size,
             sorted[res[i]+1]-sorted[res[i]], ID, res, sorted);
13     }
14   }else{
15     task = qsched_addtask( s , type_load , task_flag_none, &ID,
         sizeof(int), 0 );
16     s->res[ID].task = task;
17     utask = qsched_addtask( s , type_unload, task_flag_none, &ID,
         sizeof(int), 0 );
18     s->res[ID].utask = utask;
19     for( j = sorted[ID]; j < sorted[ID+1]; j++ ){
20       s->res[res[j]].task = task;
```

```
21      }
22    }
23  }
```

where `res` is the sorted array of resource IDs, and `sorted` stores the number of children that each resource has. The `sorted` array is computed during the sorting process. The function is called for each parentless resource. If the resource has children, and the size of the data associated with the current resource is deemed to be large enough, i.e. the average size of a child resource is $>$ 128 * `sizeof(int))`, then the scheduler creates two ghost tasks (lines 4-10). These ghost tasks are used to avoid large numbers of dependencies in systems with many levels of hierarchical resources. If the tasks directly depending on the load tasks were created for each of the leaf resources, there would be 8^l dependencies required, where l is the number of levels in the hierarchy. Using the ghost tasks means each task only depends on at most 8 unload tasks or 8 ghost tasks. Once the ghost tasks are created, the function recurses to each child resource (lines 11 to 13).

If the resource has no children, or the data associated with them is too small, then the scheduler creates a load and unload task for the resource directly, and sets the load task reference stored by all of its children to be the load task created for the parent resource (lines 15-20).

After creating all the load and unload tasks, we loop over the tasks and check that `task` (load task reference) and `utask` (unload task reference) are set for all resources. If not, we recurse up the resource hierarchy until we find a parent with the relevant task set, and set the `task` and `utask` variables.

Our initial strategy for computing the dependencies was to loop through every non-load, non-unload, non-ghost task t and add dependencies from every load task l to t, where l is the load task of a resource locked or used by t, and from t to all of the corresponding unload tasks. These dependencies were added to the end of the dependency array in an unsorted order, so the dependency array needed to be resorted before computation could begin. The additional sorting overhead was too large, and resulted in the pre-processing taking too long to be feasible.

Instead, we break the process into two stages. We first compute a usage list for each resource. To compute it, we loop through all of the tasks, and store their ID in the usage list for each resource they lock or use. If any of the resources they lock or use don't have their own load or unload task (i.e. `resource->utask == parent->utask`), the ID is instead stored in the usage list for the parent resource who the load and unload task belong to.

The second step is to compute the dependencies between the load and unload tasks, and the computation tasks. Since we want to avoid resorting the dependencies, this

step needs to create an in-place sorted list of the dependencies. To do this, we loop over the resources and create all of the dependencies for the load and unload tasks that correspond to each resource.

Since they are the most straightforward, we first add the dependencies for the unload task (`utask`). For any resource, the `utask` variable will either point to an unload task or a ghost task. In the case that `utask` points to an unload task, every child of the resource will point to the same unload task with its own `utask` variable. In this case the task will not unlock any further tasks. If `utask` points to a ghost task, then the task needs to unlock all of the child resources' `utask`. This can be done by looping through the children and adding all of these dependencies as a contiguous block.

Once this is completed, we can add the dependencies for the load task, and is only done if `res->task != parent->task`. First, if the resource has a parent, we need to add a dependency from `res->task` to `parent->task`, as `parent->task` is a ghost task that depends on data transfer done either by this task (if a load task) or one of this resource's children's load task (if a ghost). We then loop over the usage list for this resource, and add a dependency from `res->task` to each of the tasks in the usage list.

6.5.5 Unneccessary Dependencies

I tested two strategies (executed on the CPU) to minimise the number of dependencies added. The first strategy was a recursive strategy that was called before adding a dependency from a load task to a work task (or from a work task to an unload task). This strategy used two functions, `transitive_use_unlocks` and `transitive_use_locks`. Both functions relied on the task array to be sorted into a topological ordering, i.e. no task appears in the task array until all of its parent tasks have already appeared. The former function searched for any child task that also used or locked the specified resource:

```
1 int transitive_use_unlocks(struct qsched *s, struct task *t, int res,
      int depth){
2   int i;
3   for(i = 0; i < t->nr_uses; i++){
4     if(t->uses[i] == res)
5       return 1;
6   }
7   for(i = 0; i < t->nr_locks; i++){
8     if(t->locks[i] == res)
9       return 1;
```

```
10    }
11    if(depth >= MAX_DEPTH){
12      return 0;
13    }
14    for(i = 0; i < t->nr_unlocks; i++){
15      if(transitive_use_unlocks(s, &s->tasks[t->unlocks[i]], res, depth +
            1))
16        return 1;
17    }
18    return 0;
19  }
```

In lines 2-9, the function checks if the resource being search for (`res`) is used or locked by the current task. In lines 11-13, if the exit criteria is met then the function exits. Otherwise the function recurses to all tasks unlocked by `t`.

The `MAX_DEPTH` variable can be set at compile time, and is used to control how deep the search goes. If `MAX_DEPTH` is 1, the function will only check the tasks that directly depend on `t`, whereas if `MAX_DEPTH` is large enough, the algorithm will search the entire subgraph from `t`. This function was called before adding dependencies to unload tasks, and the dependency would only be added if the function returned 0.

The `transitive_use_locks` function is similar, but since the parents of a task aren't stored, it has to make use of the topological ordering to find them:

```
1   int transitive_use_locks(struct qsched *s, int tid, int res, int depth){
2     int i,j;
3     struct task *new_t;
4     struct task *t = &s->tasks[tid];
5     for(i = 0; i < t->nr_uses; i++){
6       if(t->uses[i] == res)
7         return 1;
8     }
9     for(i = 0; i < t->nr_locks; i++){
10      if(t->locks[i] == res)
11        return 1;
12    }
13    if(depth >= MAX_DEPTH){
14      return 0;
15    }
16    for(i = tid-1; i >= 0; i--){
17      new_t = &s->tasks[i];
```

```
18      for(j = 0; j < new_t->nr_unlocks; j++){
19        if(new_t->unlocks[j] == tid){
20          if(transitive_use_locks(s, i, res , depth + 1))
21            return 1;
22          break;
23        }
24      }
25    }
26    return 0;
27  }
```

In lines 16-22, the code searches backwards through the topological ordering to find all of the tasks that unlock the task with index `tid`. Again, `MAX_DEPTH` is used to limit how many levels of dependencies are searched to find another resource that locks or uses the resource. This function was used before adding dependencies from load tasks to the task with index `tid`, and the dependency was only added if the function returned 0.

This method worked reasonably well for small problems. When used for the QR decomposition on a 9×9 tile matrix, it only added 366 dependencies for the load and unload tasks (with `MAX_DEPTH` effectively set to infinite), rather than the 1530 added without it. However even on a 9×9 matrix, the algorithm took 2.5s and scaled poorly with problem size; on a 12×12 matrix the algorithm already took over a minute. With `MAX_DEPTH` set to 4, it still found the same result for the 9×9 case, and took only 15ms. However, no matter what I set `MAX_DEPTH` to (as low as 1), the overall algorithm was infeasible on larger test cases.

The second method I tested attempts to simulate an execution of the task tree. Rather than recursing through the task graph to see if any subtree contains a reference to the resources used by each task, we loop through the tasks in a topological ordering. For each task, we check if the data is already available for each resource used or locked by this task. If not, then we mark it as available and create the relevant dependency from the load task (or to the unload task). Once we have checked each of this tasks used or locked resources, we copy its accesses to its children before continuing to the next task.

The method uses a bit array, with each bit was initially set to 0. The first step involves looping forward through the topological ordering:

```
1  for(i = 0; i < s->count; i++){
2    if(s->tasks[i].type == type_load || s->tasks[i].type == type_unload)
3      continue;
```

```
4    for(k = 0; k < s->tasks[i].nr_uses; k++){
5      use = s->tasks[i].uses[k];
6      usek = use >> 5; // use / 32;
7      usem = use & 31; // use % 32.
8      if((is_loaded[i][usek] & (1 << (31-usem))) == 0 ){
9        qsched_addunlock(s, s->res[use].task , i ) ;
10       is_loaded[i][usek] |= (1 <<(31-usem));
11     }
12   }
13   /* Repeat process for locks*/
14
15   for(k = 0; k < s->tasks[i].nr_unlocks; k++){
16     if(s->tasks[s->tasks[i].unlocks[k]].type == type_load ||
17     s->tasks[s->tasks[i].unlocks[k]].type == type_unload )
18       continue;
19     for(j = 0; j < s->count_res/32 +1; j++){
20       is_loaded[s->tasks[i].unlocks[k]][j] |= is_loaded[i][j];
21     }
22     parents[s->tasks[i].unlocks[k]][num_parents[s->tasks[i].unlocks[k]]]
           = i;
23     num_parents[s->tasks[i].unlocks[k]] =
           num_parents[s->tasks[i].unlocks[k]] + 1;
24   }
25 }
```

Lines 2-3 ensure the function always skips any load and unload tasks that had been generated. The loop in lines 4-12 loops over each of the resources used by the task. Lines 5-7 compute the index corresponding to the task in the bit array. The if statement in line 8 tests to see if the resource has already been loaded by a parent of this task. If not, a dependency is added from the load task to this task, and the associated bit is set in this tasks' array (`is_loaded[i]`). The loop is repeated for the resources locked by this task as well (which is not shown in the above code segment).

Finally, the loop in lines 15-23 loops over each task unlocked by the current task, and updates their bit arrays so that any resources that will be loaded prior to this tasks execution have the corresponding bit set in the dependent tasks' bit arrays. This loop also sets up the parent array that is neccessary to compute the unload tasks' dependencies.

One this process is completed, the bit arrays are reset, and the same process is performed in reverse to compute the dependencies for the unload tasks.

This method was much quicker than the recursive method on small QR decompo-

Array Size	1e6	1e7	1e8
Non-pinned `cudaMemcpy`	6.01ms	54.86ms	543.5ms
Pinned `cudaMemcpy`	5.48ms	54.27ms	538.6ms
Kernel memory transfers	5.48ms	54.22ms	538.6ms

Table 6.13: Time required to move data to and from the device plus a simple computation kernel.

sition test cases, however on large test cases this method used too much memory to be feasible (as the memory requirement scales with the number of tasks and number of resources, and for the QR decompostion there are $O(n^3)$ tasks and $O(n^2)$ resources).

It is clear that this problem is at least as difficult as transitive reduction, however discussion with Prof. Daniel Paulusma suggested that our specific problem may be **NP**-complete to compute (Prof. Daniel Paulusma, personal communications). Since I noticed little change in the runtime of the GPU kernel when the system contained fewer dependencies, I decided to abandon this area of research.

6.5.6 Load and Unload Task Implementation

The load and unload tasks implementations were tested in a stand-alone program (discussed in more detail in Appendix A), and compared to `cudaMemcpy`. The implementation used the same block/thread structure as is used in `mdcore`, i.e. 128 blocks of 128 threads each. I ran and timed three variants:

1. `cudaMemcpy` with memory allocated using `malloc` on the CPU.
2. `cudaMemcpy` with memory allocated using `cudaMallocHost` (i.e. pinned memory) on the CPU.
3. A kernel moving the memory to and from the GPU before and after all of the computation.

The computation kernel was a simple array-squaring kernel with varying array sizes. The runtimes of these three methods is shown in Table 6.13.

These results showed that we could achieve the same performance as `cudaMemcpy` when using kernels to do the data transfer. With these results, I implemented the memory transfer technique into QuickSched:

Matrix size	With tasks	With `cudaMemcpy`
1024×1024	82.486ms	82.417ms
1536×1536	220.026ms	216.236ms
2048×2048	516.695ms	517.692ms

Table 6.14: Time taken to compute the tiled QR decomposition of various matrices. The times include the loading and unloading of data to the device as well as the QuickSched setup times.

Problem size	Kernel Runtime
1M particles	135.220ms
10M particles	2084.827ms

Table 6.15: Time taken to compute the accelerations on particles from a cosmological volume using the Barnes-Hut method. The times only include the runtime of the GPU kernel, with a maximum of 128 particles per cell.

```
1 __device__ __inline__ void cuda_memcpy_tasks ( void *dest , void
    *source , int count , int tid ){
2   int k;
3   int *idest = (int *)dest, *isource = (int *)source;
4   for ( k = threadIdx.x ; k < count/sizeof(int) ; k += blockDim.x ){
5     idest[k] = isource[k];
6   }
7 }
```

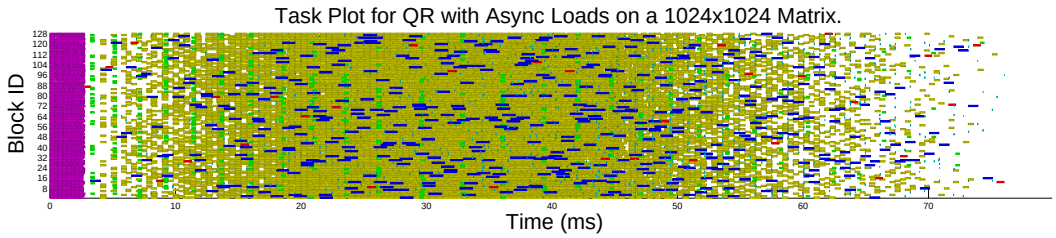
One limitation of our data transfer method is that all of the resources' data needs to be a multiple of `sizeof(int)`, however this is not a problem for the testcases implemented, and can easily be checked and enforced in the scheduler.

6.5.7 Results with the Initial Setup

I ran the initial setup with both the tiled QR decomposition (introduced in Section 4.3) and a Barnes-Hut implementation (described in Chapter 5).

In this thesis I use *task plots* to help visualise how the computations progress. When collecting the data for task plots, each task structure has stores a few additional values: The start and end time of the task (stored in clock ticks), and the block ID of the block that executed a task. Before executing a task, start time and block ID are stored in the task struct, and after a task completes the end time is stored. These values are output when the computation is completed, and task plots can be generated using MATLAB (or other tools).

a)



b)

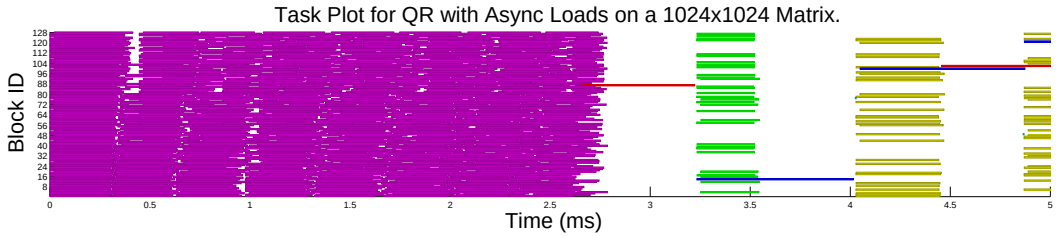


Figure 6.12: Task plots for the QR decomposition on a 1024×1024 matrix. a) shows the entire computation while b) shows a zoom into the start of the computation. The purple tasks show the load tasks. The computation can begin once the data arrives on the GPU, and every block is working for the majority of the computation. At the start and end of the computation, some of the blocks are idle, which is due to the dependency structure of the QR decomposition.
The magenta tasks are the load tasks, and the teal tasks are the unload tasks. The remaining colours represent the work tasks of the QR decomposition.

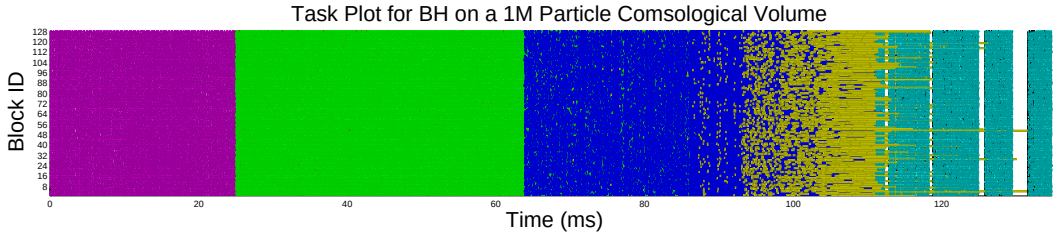


Figure 6.13: Task plot for the BH simulation of a 1M particle cosmological volume. Since there are no dependencies, the tasks are executed in the order they are placed into the queue. Since some of the `task_type_pc_split` tasks use large cells, these are executed last and have a long runtime. This results in the unload tasks being delayed until these tasks complete, leading to poor runtime.
The magenta tasks are the load tasks and the teal tasks are the unload tasks. The remaining colours represent the work tasks of the BH simulation.

Figure 6.12 shows how the QR decomposition computation progresses on the GPU. Initially, all the blocks copy data to the GPU before any computation begins, which can be seen in more detail in the zoomed Figure 6.12(b). Since all the data transfer happens before any computation, a lot of the blocks are idle early on the computation when there is relatively little work that can be done due to the dependencies. The unload tasks are almost completely hidden behind the computation. Table 6.14 shows that there is no benefit from the task-based data transfer with the initial scheduler for the QR decomposition.

Figure 6.13 shows the progression of the Barnes-Hut computation on the GPU. As with the QR, no computation occurs until all of the data has been moved to the GPU, and the unload tasks need to wait on the large `task_type_pc_split` tasks before they can be executing, resulting in almost no difference from the standard load-compute-unload methodology.

These results imply the scheduler needs to be modified to increase the amount of compute and data transfer performed in parallel.

6.5.8 Improving the GPU Setup

The main flaw with the single queue setup is that no computation will begin until almost all of the data movement tasks are complete, as the queue has no concept of priority on the GPU. This means relatively little computation will occur in parallel to data transfer, which means the kernel is almost no different to using `cudaMemcpy` or `cudaMemcpyAsync` to move the data to and from the GPU.

Since it is important that the GPU task queues are lock-free and as lightweight as possible, I decided it was not feasible to create priority queues (or any directly priority-aware data structure) on the GPU. Instead, I felt having rough estimates of priorities would be enough to improve the GPU performance. We prioritised the tasks based on their task type alone, with the load tasks being the lowest priority, and the unload tasks being highest priority, and ghost and user-defined tasks being medium priority. This decision was made as we want to prioritise work over loading data (to avoid all blocks initially just loading data until all the data has been transferred).

To implement the levels of priority, I created multiple task queues, and only allowed specific task types to be placed in specific queues, i.e. one queue only containing load tasks, one queue only containing unload tasks and one queue containing any other type of task. When attempting to retrieve a task, each block would first try the queue containing the unload tasks, and if unsuccessful in obtaining a task, try the other queues in the priority order.

Since the `queue_gettask` function blocks until either all of the tasks that belong to the queue are executed, or until a valid task index is retrieved, I modified the task queue

to maintain a counter of how many task indicies they contain at any time. This value can easily be checked directly, and avoid accessing empty queues when attempting to retrieve tasks.

The `queue_cuda` structure (which was previously identical to the queue structure used in `mdcore`) was modified, and a new field was added:
`volatile int nr_avail_tasks`. This value is initially set to the number of tasks initially placed in the queue, usually 0 for the unload and work task queues, and equal to the number of load tasks for the load task queue.

When adding this idea of multiple queues to be checked in priority order, it also became possible for a block to fail to retrieve a task, despite not all of the tasks in the system being completed. Since the exit criteria checked only if the system was able to retrieve a task, blocks could exit the main loop earlier than they should. To fix this, I added a new `__device__` variable, named `tot_num_tasks`, which stores the number of tasks that still needed to be executed.

The queues' `nr_avail_tasks` field is queried to check which queue to retrieve tasks from, resulting in the start of the main kernel being modified:

```
1  if(threadIdx.x == 0){
2    tid = -1;
3    if(unload_queue.nr_avail_tasks > 0)
4      tid = get_task(&unload_queue);
5    if(tid < 0 && cuda_queues[0].nr_avail_tasks > 0)
6      tid = get_task(&cuda_queues[0]);
7    if(tid < 0 && load_queue.nr_avail_tasks > 0)
8      tid = get_task(&load_queue);
9  }
10 __syncthreads();
11 if(tid < 0 && tot_num_tasks == 0)
12   break;
```

I also needed to modify the `get_task` and `puttask` routines to use these new variables:

```
1  __device__ int get_task ( struct queue_cuda *q ){
2    int tid = -1;
3    if( atomicAdd((int*)&q->nr_avail_tasks, -1) <= 0){
4      atomicAdd((int*)&q->nr_avail_tasks, 1);
5      return -1;
6    }
7    while ( ( tid = cuda_queue_gettask( q ) ) >= 0 ){
```

```
8       break;
9     }
10    if ( tid >= 0 ){
11      q->rec_data[ atomicAdd( (int *)&q->rec_count , 1 ) ] = tid;
12    }
13    return tid;
14  }
```

The only change to `get_task` is the addition of lines 3-6. Before progressing further, we decrement the `nr_avail_tasks` variable, and check if the previously stored value was ≤ 0. If so, then the queue is empty, so we reverse the change and exit the function. If not, then the queue contains tasks, and this access will remove one of them, so the counter is already updated.

The `puttask` function was also modified slightly:

```
1   __device__ void puttask ( struct queue_cuda *q , int tid ){
2     int ind;
3     ind = atomicAdd( &q->last , 1 ) % cuda_queue_size;
4     while ( q->data[ind] != -1 );
5     q->data[ind] = tid;
6     atomicAdd((int*)&q->nr_avail_tasks, 1);
7   }
```

The only required change to `puttask` is to increment the counter for the task queue in line 6. Finally, the `queue_gettask` function was modified slightly to decrement `tot_num_tasks`.

I also had to change the dependency count update to ensure that newly available tasks were placed into the correct task queue:

```
1   for(i = threadIdx.x; i < tasks_cuda[tid].nr_unlocks; i += blockDim.x ){
2     if( atomicSub( &tasks_cuda[tasks_cuda[tid].unlocks[i]].wait , 1 ) == 1
          ){
3       if(tasks_cuda[tasks_cuda[tid].unlocks[i]].type != type_unload)
4         cuda_queue_puttask( &cuda_queues[0] , tasks_cuda[tid].unlocks[i] );
5       else
6         cuda_queue_puttask( &unload_queue , tasks_cuda[tid].unlocks[i] );
7     }
8   }
```

Matrix size	Initial setup	New setup
1024×1024	82.486ms	79.928ms
1536×1536	220.026ms	214.919ms
2048×2048	516.695ms	509.654ms

Table 6.16: Time taken to compute the tiled QR decomposition of various matrices. The times include the loading and unloading of data to the device as well as the QuickSched setup times. The new setup performs slightly faster than the initial setup overall.

Problem size	Initial Runtime	New Runtime
1M particles	135.220ms	121.000 ms
10M particles	2084.827ms	1771.151 ms

Table 6.17: Time taken to compute the accelerations on particles from a cosmological volume using the Barnes-Hut method. The times only include the runtime of the GPU kernel, with a maximum of 128 particles per cell. We can see the benefit of starting the computation immediately, with a speedup of around 7-10%.

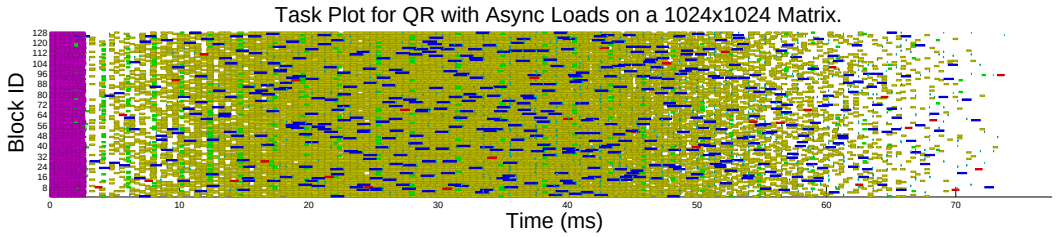
In lines 3-6, instead of just placing a task into the task queue, it checks if the task is an unload task. If it is, it is placed into the `unload_queue`, else it is placed into the standard task queue.

6.5.9 Results with the Multi-Queue Priority

The results for the QR decomposition are shown in Table 6.16 and task plots are shown in Figure 6.14. The QR speeds up by 3% on the smallest testcase, and around 1% on the largest. The benefit is small as the computation is limited primarily due to the dependency structure. The first work task that must be executed requires the top-left tile to be loaded to the GPU, and there is no mechanism in place to enforce that this is the first tile loaded to the GPU.

The results for the Barnes-Hut with this setup are shown in Table 6.17 and a task plot is shown in Figure 6.15. The Barnes-Hut benefits more from adding multiple queues, as the data is not all loaded simultaneously. Instead, some threads perform computation while other threads are performing data transfer, and this appears beneficial. The task plot still shows some near-synchronous regions of data transfer from the GPU to the host throughout the computation, so improvements are still required. These near-synchronous regions of data transfer occur as the unload tasks are the highest priority tasks, so when a task that operates on a large cell completes, many new unload tasks become available. This results in all of the threads unloading data rather than performing computation until there are no more unload tasks to be executed.

a)



b)

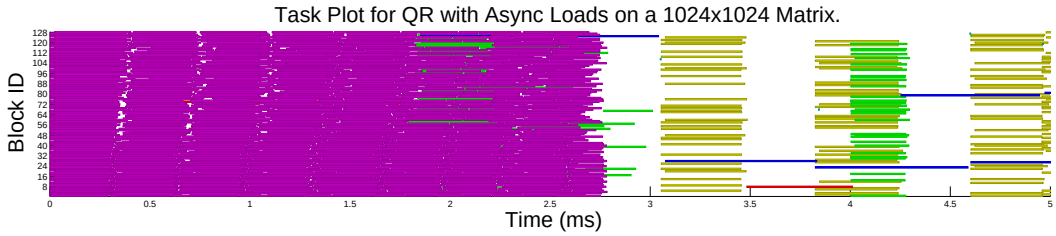


Figure 6.14: Task plots for the QR decomposition on a 1024×1024 matrix. a) shows the entire computation while b) shows a zoom in at the start of the computation. The purple tasks show the load tasks. The computation can start earlier than with the initial setup, however there is still some delay before it begins.
The magenta tasks are the load tasks, and the teal tasks are the unload tasks. The remaining colours represent the work tasks of the QR decomposition.

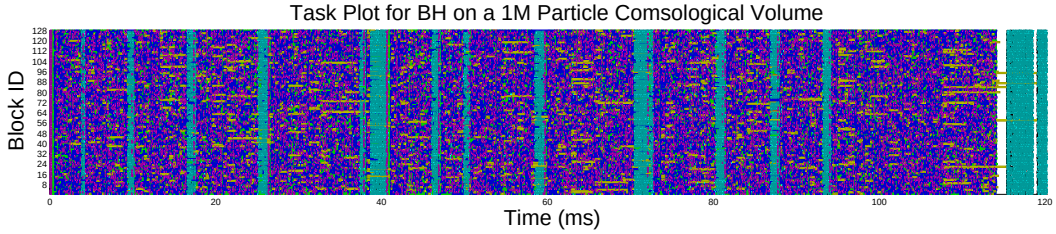


Figure 6.15: Task plot for the BH simulation of a 1M particle cosmological volume. The data transfer is now spread out throughout the computation, though there are still regular blocks of unload tasks being executed where no work is taking place.
The magneta tasks are the load tasks and the teal tasks are the unload tasks. The remaining colours represent the work tasks of the BH simulation.

6.5.10 Using Instruction Level Parallelism to Achieve High Data Transfer Rates with Few Threadblocks

The initial results showed that using the entire GPU to perform data transfer before any computation began lead to not as much improvement as hoped. One strategy to improve the performance was to try using fewer threadblocks to execute the load tasks, as then computation would occur in parallel to data transfer. This could only be successful if we could keep the data transfer rate high.

I extended the previous program used to test memory transfer by adding kernels that performed memory transfer with ILP. I implemented this with 2, 3, and 4-way ILP. The kernel to perform it with 4-way ILP is shown below:

```
1  __global__ void runner_run_copyTo_ILP4(int *d_a, int *h_a, int N){
2    int i;
3    int val1, val2, val4, val3;
4    for(i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i+=
         4*blockDim.x*gridDim.x){
5      val1 = h_a[i];
6      val2 = h_a[i + blockDim.x*gridDim.x];
7      val3 = h_a[i + 2*blockDim.x*gridDim.x];
8      val4 = h_a[i + 3*blockDim.x*gridDim.x];
9      d_a[i] = val1;
10     d_a[i + blockDim.x*gridDim.x] = val2;
11     d_a[i + 2*blockDim.x*gridDim.x] = val3;
12     d_a[i + 3*blockDim.x*gridDim.x] = val4;
13   }
14 }
```

This implementation is naive, so only works if `N` (the size of the array) is an exact multiple of `4 * blockDim.x * gridDim.x`.

I ran these variants on an array of size $15,360,000$, with varying block counts. For each increase in ILP (e.g. going from 1 to 2-way, 2-way to 3-way or 3-way to 4-way) the number of threadblocks required to transfer the data can be decreased by a factor of two without any significant loss in performance, and is shown in more detail in Appendix A.

This allows us to use ILP in QuickSched to reduce the number of blocks required for efficient data transfer, allowing concurrent data transfer and computation to occur without enforcing that some of the load tasks were delayed (as before).

The code that implements the load and unload tasks in QuickSched was modified

to use 4-way ILP, and to perform the last few copies with no ILP in the case that the size of the data was not a multiple of `4*blockDim.x`.

I then modified the task retrieval process to use a new queue ordering:

```
if(threadIdx.x == 0){
  tid = -1;
  if(unload_queue.nr_avail_tasks > 0)
    tid = get_task(&unload_queue);
  if(tid < 0 && load_queue.nr_avail_tasks > 0 && blockIdx.x < 12)
    tid = get_task(&load_queue);
  if(tid < 0 && cuda_queues[0].nr_avail_tasks > 0)
    tid = get_task(&cuda_queues[0]);
}
```

The load queue is now only accessed by blocks with `blockIdx.x < 12` (this choice was made by tuning the parameter on the GTX690), but those blocks prioritise it over the work tasks. The load tasks should take the same amount of execution time as before, however they will be done earlier on in the computation, so important tasks that rely on large amounts of data can occur earlier in the computation. Other blocks are free to perform computation as soon as any enough data has been copied to the GPU.

Finally, the load tasks were sorted by their weights (see page 21), meaning load tasks which unlock tasks with large dependent subgraphs are likely to occur earlier in the kernel.

6.5.11 Results with the ILP Data Transfer

The results for the QR decomposition with the new data transfer setup are shown in Table 6.18, and task plots are shown in Figure 6.16. The results are improved, with the computation starting almost immediately when the first tile is loaded to the GPU.

The results for the BH simulation are shown in Table 6.19, and a task plot is shown in Figure 6.17. The computation performs similarly to the previous setup.

Overall, this setup seems to performs slightly better (1% improvement for QR, up to 5% worse for BH), and the sorting of the load tasks leads to better ordering of data transfer when there is a rich dependency structure, as shown in the QR decomposition.

We still don't get ideal performance for the Barnes-Hut. As Figure 6.17 shows, many of the large tasks still rely on all the data being copied to the GPU which means they are executed last, and all of the unloading of all of the data waits for them to complete. If these tasks could be executed earlier, we could concurrently unload data while doing computation, as in the QR decomposition.

Matrix size	Initial setup	New setup
1024×1024	82.486ms	78.381ms
1536×1536	220.026 ms	211.481 ms
2048×2048	516.695ms	502.782ms

Table 6.18: Time taken to compute the tiled QR decomposition of various matrices. The times include the loading and unloading of data to the device as well as the QuickSched setup times. The new setup performs 2.5-5% faster than the initial setup overall.

a)

b)

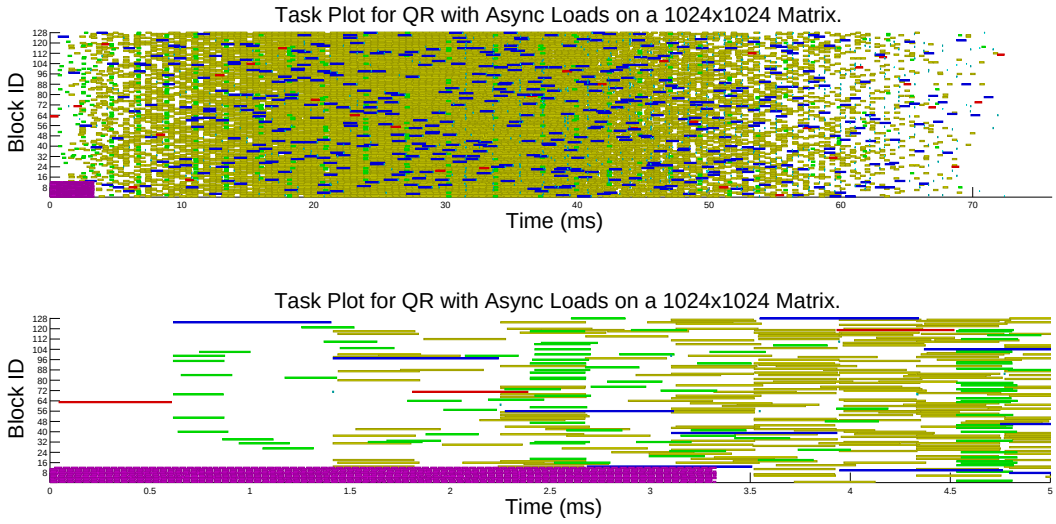


Figure 6.16: Task plots for the QR decomposition on a 1024×1024 matrix. a) shows the entire computation while b) shows a zoom in at the start of the computation. The purple tasks show the load tasks. The computation can begin almost immediately when the first tile's data arrives on the GPU, and every block is working for the majority of the computation.
The magenta tasks are the load tasks, and the teal tasks are the unload tasks. The remaining colours represent the work tasks of the QR decomposition.

Problem size	Initial Runtime	New Runtime
1M particles	135.220ms	120.400 ms
10M particles	2084.827ms	1847.997 ms

Table 6.19: Time taken to compute the accelerations on particles from a cosmological volume using the Barnes-Hut method. The times only include the runtime of the GPU kernel, with a maximum of 128 particles per cell. We can clearly see the advantage gained from fully asynchronous data transfer for the Barnes-Hut, with a speedup of around 7%.

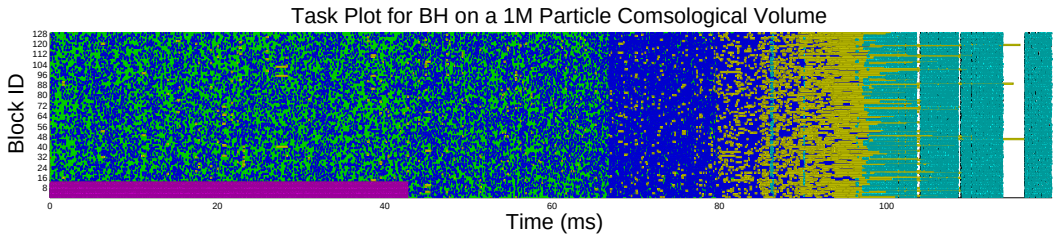


Figure 6.17: Task plot for the BH simulation of a 1M particle cosmological volume. The work tasks no longer need to wait on all of the data transfer to the GPU before they can begin, improving the runtime of the kernel despite the data transfer to the GPU taking 20ms longer. However, the ordering of the tasks is still poor, resulting in near-synchronous unloading of the data.
The magneta tasks are the load tasks and the teal tasks are the unload tasks. The remaining colours represent the work tasks of the BH simulation.

6.5.12 Adding Priorities to the Work Tasks

To fix the issue highlighted by the Barnes-Hut test case, I decided to add a priority setup to the user-defined work tasks. Since we can't use priority-aware data structures on the GPU, I added an additional work queue to the setup. The new queue was the last queue to be searched for tasks, i.e. the lowest priority queue.

Each task has a `cost` associated with it, which is provided by the user. Rather than using the `weight` of tasks to determine whether a task is important (as a task's weight is strongly dependent on its dependencies), I felt the `cost` was a better way to determine if a task was heavy enough to disrupt the execution of the kernel.

Using Quickselect [26], the CPU computes a value to use as a minimum task cost for the higher priority queue. With two queues, I used the 80th percentile, i.e. the top 20% of work tasks by cost are placed in the higher priority queue, whilst the remainder are placed in the normal priority queue.

This value is then copied to the GPU and stored in the variable `median_cost`. The task unlock section was also modified to make use of the `median_cost`:

```
1 for(i = threadIdx.x; i < tasks_cuda[tid].nr_unlocks; i += blockDim.x ){
2   if( atomicSub( &tasks_cuda[tasks_cuda[tid].unlocks[i]].wait , 1 ) == 1
        && !( tasks_cuda[tasks_cuda[tid].unlocks[i]].flags &
        task_flag_skip )){
3     if(tasks_cuda[tasks_cuda[tid].unlocks[i]].type != type_unload){
4       if(tasks_cuda[tasks_cuda[tid].unlocks[i]].cost > median_cost ||
            tasks_cuda[tasks_cuda[tid].unlocks[i]].type == type_ghost){
5         cuda_queue_puttask( &cuda_queues[0] , tasks_cuda[tid].unlocks[i]
              );
6       }else{
```

Matrix size	Initial setup	Final setup
1024×1024	82.486ms	79.290ms
1536×1536	220.026 ms	214.496 ms
2048×2048	516.695ms	506.294ms

Table 6.20: Time taken to compute the tiled QR decomposition of various matrices. The times include the loading and unloading of data to the device as well as the QuickSched setup times. The final setup performs slightly worse than the previous setup (Table 6.18), but still significantly better than the initial setup. The loss of performance is due to slightly more scheduling overhead during computation. The benefit of having more priority-aware scheduling for the QR does not make up for these costs.

```
7           cuda_queue_puttask( &cuda_queues[1] , tasks_cuda[tid].unlocks[i]
                ); 
8         }
9       }else
10        cuda_queue_puttask( &unload_queue , tasks_cuda[tid].unlocks[i] );
11    }
12  }
```

As before, the unload tasks are placed into the `unload_queue`. The high priority work tasks (i.e. those whose cost is larger than `median_cost`) and ghost tasks are placed into the high priority queue (`cuda_queues[0]`) and all other work tasks are placed into the normal priority queue (`cuda_queues[1]`).

6.5.13 Final Results

The runtimes (Table 6.20) for the QR decomposition again show only a small improvement over the original setup, and performs worse than some of the previous setups. However, the performance loss is likely due to the increased cost of scheduling in the final version.

The Barnes-Hut simulation gains significantly from additional priority awareness. It is clear from the task plot (Figure 6.18) that the unload tasks are now almost entirely hidden in the computation. The large tasks that previously limited the performance are prioritised, meaning large numbers of unload tasks are not waiting on a single task to complete before they can be executed. Table 6.21 shows significant improvements in the runtime (up to 56% over the initial variant).

Figure 6.19 shows a comparison of the original setup which uses `cudaMemcpy`, to the version with no priority queue and load/unload tasks, and to the final version with a priority queue setup and load/unload tasks. These results are for the Barnes-

Problem size	Runtime without priorities	Runtime with priorities
1M particles	120.400ms	102.756 ms
10M particles	2084.827ms	1330.728 ms

Table 6.21: Time taken to compute the accelerations on particles from a cosmological volume using the Barnes-Hut method. The times only include the runtime of the GPU kernel, with a maximum of 128 particles per cell. We can clearly see the advantage gained from the new priorities, with a speedup of 56% for the larger test case.

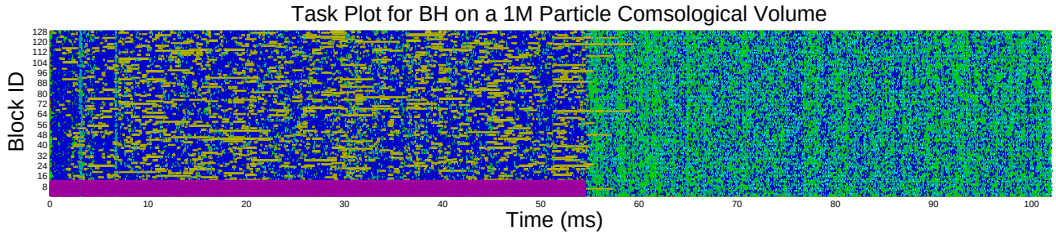


Figure 6.18: Task plot for the BH simulation of a 1M particle cosmological volume. The particle-cell tasks that operate on large cells are now prioritised, meaning the unload tasks can occur almost entirely during the computation. This leads to better load balancing and performance. Again the time taken to move data to the GPU rises by 10ms, however the improved ordering of tasks more than makes up for this.
The magneta tasks are the load tasks and the teal tasks are the unload tasks. The remaining colours represent the work tasks of the BH simulation.

Hut simulation. The improvements to the task-based scheme shows a speedup of approximately 15%, coming solely from performing the data transfer and computation concurrently, and prioritising the most computationally intensive tasks.

6.6 Porting the Load and Unload Tasks back to `mdcore`

Results in this section ran with CUDA 5.0 on the GTX690 for direct comparison with previous results.

The final work I undertook on Task-Based Parallelism for GPUs was to transfer the improvements made to the scheduler as part of QuickSched back into `mdcore`. I ported the load and unload tasks back into `mdcore`, with one load and unload task per cell. Dependencies were added from the load tasks to the sort tasks, and from all of the self and pair tasks to the unload tasks. `mdcore` only uses one queue for work tasks, as well as the load and unload queues as used in QuickSched. One other change I made was to place the sort tasks into the unload queue. These tasks are especially important, as they unlock all of the pair interaction tasks. Allowing priority-aware scheduling may improve this, however this led to better performance than placing the sort tasks into

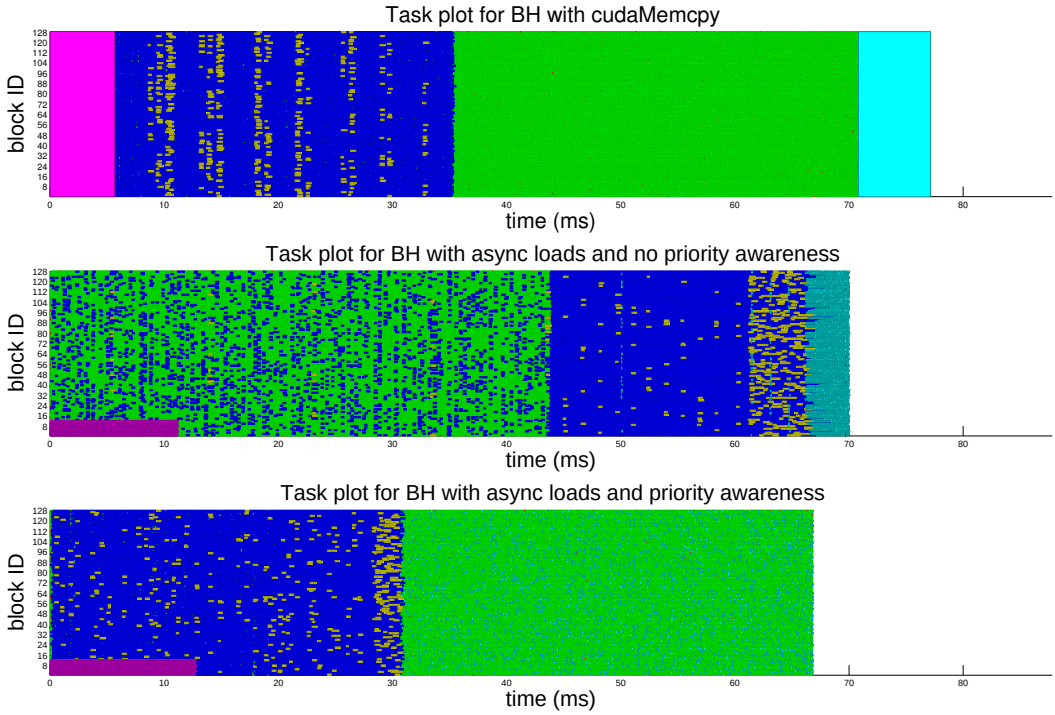


Figure 6.19: Task plots with and without load and unload tasks, and with/without priority awareness on 150k particles. Each rectangle represents a single task that is executed. The full task-based setup performs better than the versions using `cudaMemcpy` due to starting the computation sooner. Without priority awareness of user-specified tasks, we don't get as much of an advantage due to the apparent synchronisation of the unload tasks. This occurs as data is being written to large resources at the end of the computation, meaning any children of those resources have to wait until this is completed before they can be unloaded. Adding further priority awareness allows us to specify that these tasks should be executed first, so the unload costs become fully integrated within the computation.

Variant	JAC	ApoA1	STMV
Serial CPU bonded interactions with load and unload tasks	9.934ms	70.186ms	645.875ms
Parallel GPU bonded interactions with load and unload tasks	9.925ms	38.936ms	446.646ms

Table 6.22: Time required to compute a timestep for the all three test cases on the GTX690 with and without the parallel bonded interactions, and with load and unload tasks.

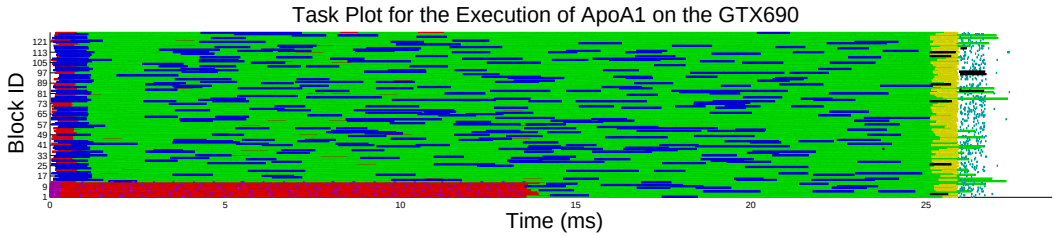


Figure 6.20: Task plot for an ApoA1 simulation on the NVIDIA GeForce GTX690, with parallel bonded interactions enabled. The black tasks are ghost tasks, which handle the dependencies between the work tasks and the parallel bonded tasks. You can see the load (magenta) and sort (red) tasks are interspersed as sort tasks are prioritised over other load tasks. There is some load imbalance at the end of the simulation, as some pair interaction tasks take longer than expected due to running in parallel with bonded interaction tasks.
The teal tasks are the unload tasks, and the blue, green and yellow tasks are the non-sort MD work tasks.

the work queue.

The results with these changes are shown in Table 6.22 and Figure 6.20. Overall, the time taken to compute a timestep is improved slightly compared to the final version created in Section 6.2.

6.7 Conclusions

In this section I have shown the evolution of our task-based approach on GPUs. Our early task-based scheduler showed promise for MD, and further work as part of a standalone library has shown it is effective on a problems with rich dependency graphs (such as the tiled QR decomposition) or no dependencies (Barnes-Hut Simulation). I also managed to integrate data transfer as part of the task-based framework, which allows fine-grained data transfer and computation to occur in parallel, and this can allow for significant speedup over solely using `cudaMemcpy`.

One difficulty I did not address during this work is parallelising the GPU computation with CPU computation. While QuickSched enables both CPU and GPU Task-Based Parallelism, I have not worked on a scheme for load balancing Task-Based Parallelism on both devices, and due to this a single scheduler cannot perform Task-Based Parallelism on the GPU and CPU simultaneously. I did check if the GPU data transfer method affects CPU performance on memory-bound computation, and it appears to make no difference (Appendix B), so I believe there should be no loss in CPU performance due to any of the GPU task-based parallel computation.

The load and unload scheme used in QuickSched also makes multi-GPU Task-Based Parallelism on the same node difficult. As there is currently no atomic access to CPU RAM on the GPU, any data writes need to not conflict when using multiple GPUs. Due to the new hardware (which has since been released, specifically NVLINK) for CUDA + Power8 I decided not to investigate how to overcome this difficulty at this time. This new hardware should increase the speed of data transfer to the GPU, and may change other ways in which the GPU accesses CPU RAM.

The task-based approach introduced here is aimed towards problems that cannot use the entire GPU solely by SIMT parallelising the algorithm, or problems that have complex data dependencies. The scheduler allows individual tasks to only exploit small amounts of SIMT parallelisation, as it can parallelise these tasks across the entire GPU.

The downside of our task-based approach is that is currently requires the data to be copied to and from the GPU after every kernel execution (through the load and unload tasks). For applications that perform the entire computation on the GPU this would hurt the performance, as such computations leave most of the data on the GPU, and only copy data received from other nodes back to the GPU. It would be possible to extend our work to avoid data transfer after every kernel.

I did not investigate load balancing CPU and GPU QuickSched inside the same code. To be able to do this well, the library could test the relative performance of each task type on the CPU and GPU, and then create a weighted partition of the work based upon the relative speed of the CPU and GPU tasks. This may require a custom partitioning algorithm to create unbalanced partitions.

Chapter 7

Task-Based Parallelism for Hybrid Homogeneous Architectures with Automated MPI

7.1 Introduction

Task-Based Parallelism is accepted as an effective shared-memory paradigm for parallel computing. Its ability to automatically load balance computation is one of its major advantages on these shared-memory systems. However, all massive-scale computations cannot be performed on solely shared-memory computers, requiring either distributed or hybrid shared-distributed memory setups. On these systems, the automatic load balancing no longer works, 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. This has resulted in little work being done on Task-Based Parallelism for hybrid memory systems, though projects such as DAGuE [8] and our SWIFT [40] project use Task-Based Parallelism for large scale computations and have shown its efficacy, but use data decomposition methods to perform load balancing between nodes.

When performing computations on large hybrid shared-distributed memory machines, there are two new main difficulties that do not arise on shared memory systems:

1. Load balancing between nodes.

2. Data transfer between nodes.

Load balancing on hybrid or distributed memory machines is usually based upon data decomposition. Naively, this involves dividing the data into N equal sized chunks (where N is the number of nodes the computation is being performed upon) and assigning each chunk to a separate node. Often codes use some data overlap between

neighbouring regions (often known as *halos*). This decomposition method relies on the assumption that the amount of work associated with each chunk of data is known and homogeneous, which is often not the case. Furthermore, as the number of nodes increases, the amount of data (and therefore the amount of computation) on each node decreases, although the amount of communication does not, and may even increase. This can result in communication dominating the computation. Other strategies for load balancing exist, based on graph partitioning approaches or space filling curves, however the data decomposition is still the most common approach.

Data transfer between nodes is usually performed by message passing (MPI) communication protocols, though other techniques such as Distributed Global Address Space (DGAS) methods do exist. Most applications perform communication and computation separately, iteratively using single-threaded synchronous MPI steps followed by bursts of computation across all of the nodes, though additions to MPI-2 and MPI-3 have provided alternatives. This approach is used in the DL_POLY_4 [49] and Gadget-2 [45] scientific codes (among many others), which are still commonly in use. This approach means most of the cores are idle waiting on data transfer for large chunks of the overall computation time, and this effect can further worsen the performance loss due to load imbalances.

To be able to use Task-Based Parallelism on hybrid memory systems, we would ideally need to be able to find an improved strategy to load balance the computation, as well as improve data transfer. This should all be done as part of the task-based scheme. Our solutions to both of these problems are based upon the work used to build our GPU task-based framework.

During the work on `mdcore` (Section 6.3), I attempted to use a graph partitioning approach to decompose the data between ranks based the expected amount of computation required for each segment of data. This seemed to be reasonably effective, and is applied in this chapter to a more general task-based framework.

I also introduced the concept of load and unload tasks as part of QuickSched on the GPU (Section 6.5). We build on this approach and use *send* and *receive* (shortened to *recv*) tasks to perform the communication in a hybrid memory task-based framework. Together with MPI's asynchronous communication functions (`Isend` and `Irecv`), these allow multi-threaded communication in parallel with computation.

One difficulty using Task-Based Parallelism on hybrid shared-distributed memory systems is correctly enforcing conflicts and dependencies between tasks on different nodes. Dependencies can be enforced using MPI communication to send a message to another node when a task completes. Conflicts could just be modelled as dependencies, however this has been shown to lead to poor performance for some task graphs as it can limit the amount of parallelism available, e.g. in Molecular Dynamics if we treat the

conflicts between pair tasks as dependencies, we would have an artificial ordering for some of the tasks which reduces the amount of parallelism available. This was found to be an issue in [31] and [2]. One way to avoid conflicts occurring between nodes would be to duplicate the conflicting tasks on each node, though this obviously can lead to a loss of parallel efficiency due to work duplication. We use the latter approach in this chapter.

The remainder of this chapter provides a proof-of-concept implementation for this approach as part of QuickSched, and the results of this implementation for the tiled QR decomposition and a Barnes-Hut simulation.

7.1.1 Extending the QuickSched Model for MPI

When deciding how to use MPI automatically as part of QuickSched, I made decisions to simplify some problems expected to arise during the implementation. The primary simplification was to let every rank have a global view of the computation, i.e. every rank knows all of the tasks and resources in the system, but doesn't neccessarily have all of the data associated with them. This makes some of the preprocessing steps easier to compute. The obvious downside is that this is expensive with respect to the amount of memory required.

In the standard QuickSched library, the tasks' and resources' indices are simply their position in the task or resource array, and this ordering is fixed. In MPI QuickSched, we do not enforce that the tasks and resources are all created on a single node. When the synchronisation of these objects occurs between ranks, there is no easy way to keep the original indices. Instead, I added `ID` fields to both the task and resource structures. These `ID` fields are stored as `long long int`, with the first 16 bits being the ID of the rank that created them, and the remaining 48 bits being a unique index on that node, though in practice this is much larger than necessary. Every task and resource can be uniquely identified by this ID. If this approach were to be used on over 2^{16} nodes the number of bits required to store the rank ID would need to be increased, while the number of bits used to store the task index within the rank can be reduced, as an individual rank is unlikely to create 2^{48} tasks.

When deciding how to schedule tasks and load balance the computation across multiple MPI ranks, I decided to no longer allow users to dynamically create tasks during execution. The `qsched_addtask_dynamic` function specified in QuickSched was renamed `qsched_addtask_dynamic_local`, and was only enabled if MPI was not enabled. This decision was made as it invalidates the global view of the computation. Furthermore, the default QuickSched `qsched_addtask` and `qsched_addres` were renamed similarly and can only be used if MPI is disabled. This means MPI QuickSched still supports all of the standard non-MPI QuickSched routines.

To utilise QuickSched with MPI, QuickSched needs to know where the data associated with each resource is stored. I created a new `qsched_addres` function, which requires the user to specify a data `size` and a `void**` pointer. The function then directly allocates memory using `malloc` of the size specified by the user, and returns the pointer to the user. If a task writes to data stored in a resource, then the task must lock the resource. If a task only reads from data stored in a resource, then the task must use the resource.

For hierarchical resources, a separate `qsched_addchildres` function was created, which required the user to specify a `parent` ID, a data `size`, a data `offset` (in bytes), and a `void**` pointer. The pointer was then set to point to the `offset` specified in the parents data array.

Since the scheduler/user should only use the IDs of tasks (as opposed to array indices), I needed to implement functions to be able to retrieve the array indices of the resources from their IDs, `getindex` (for resources) and `gettaskindex` (for tasks). The scheduler stores two arrays, `res_ranks` and `task_ranks`, which store the first array index of a resource/task that was created on each rank. All of the resources and tasks created on a single rank are stored in a contiguous segment of the resource/task array, in ascending order by ID. This means the index of any resource (or task) is `res_ranks[id>>48] + (id & 0xFFFFFFFFFFFFF)` (or `task_ranks` in the case of tasks).

7.1.2 Synchronising the Computation over MPI.

The model I decided to use for the MPI QuickSched setup requires two synchronisation steps. The first step synchronises the resource structures across the ranks, and the second synchronises the tasks, dependencies, uses, and locks across the ranks. Since task creation requires knowledge of the resources, this two-step synchronisation process is required to enable all ranks to create tasks.

To synchronise the resources, I created a `qsched_sync_resources` function:

```
1 void qsched_sync_resources(struct qsched *s){
2   int errors, i;
3   lock_lock(&s->lock);
4   s->res_ranks[s->rank+1] = s->count_res;
5   errors = MPI_Allreduce(MPI_IN_PLACE, s->res_ranks, s->count_ranks+1,
        MPI_INT, MPI_SUM, s->comm);
6   //Block to check for MPI errors.
7   s->res_ranks[0] = 0;
8   for(i = 1; i < s->count_ranks+1; i++){
9     s->res_ranks[i] += s->res_ranks[i-1];
```

```
10    }
11    struct res *res_new = (struct res*)
          calloc(s->res_ranks[s->count_ranks] , sizeof(struct res));
12    struct res *res_local = &res_new[s->res_ranks[s->rank]];
13    for(i = 0; i < s->count_res; i++){
14      res_local[i] = s->res[i];
15    }
16    int number = sizeof(struct res) * s->res_ranks[s->count_ranks];
17    number = number / sizeof(int);
18    errors = MPI_Allreduce(MPI_IN_PLACE, res_new, number, MPI_INT,
          MPI_SUM, s->comm);
19    //Block to check for MPI errors.
20    free(s->res);
21    s->res = res_new;
22    for(i = 0; i < s->res_ranks[s->count_ranks]; i++){
23      if(s->res[i].ID>>48 != s->rank)
24        s->res[i].data = NULL;
25    }
26  }
```

where the `s->res_ranks` variable was introduced as part of MPI QuickSched, and `s->res_ranks[i]` stores the index of the first resource created on rank `i`, and `s->res_ranks[num_ranks]` is the total number of resources in the system. Lines 4-10 initialise this array. Once this array has been synchronised, the new resource array (`res_new`) is created, and the resources created on this rank are placed into the corresponding section of that array in lines 12-14. This array is synchronised across the ranks in lines 16-20. Finally, in lines 21-24 each rank loops through the resources, and clears the `data` pointer if the resource wasn't created on the rank.

I chose to use `MPI_Allreduce` to synchronise these arrays as having synchronisation in this section is not a substantial part of the overall computation (as it is performed once and it not performance critical), and these routines are heavily optimised for performance in MPI implementations.

Synchronising the tasks, dependencies, uses, and locks (as well as two additional arrays required later, called `users` and `lockers`) is performed in a similar manner to the synchronisation of resources. Once this synchronisation has been completed, the data array that stores the data belonging to each task need to be synchronised. This array is synchronised in the same way, however the tasks no longer have offsets to the correct data in the `s->data` array. When synchronising the `s->data` array, the offset of each rank's data resource is stored in a temporary array, and these offsets are used

to reconstruct the tasks' data offsets in the new `s->data` array.

The MPI extensions to QuickSched needs more data than the GPU extensions, as each rank creates its own QuickSched instance. Each of these QuickSched instances needs to be able to be created independently, but must be able to be easily synchronised with the other objects in the MPI computation. This results in the additional of global IDs and resource owners for MPI QuickSched, while both MPI and GPU QuickSched have additional values stored to use the specific architectures. As MPI QuickSched manages data transfer for the user, the user needs to use QuickSched's functions to be able to find their data, while the GPU and CPU pointers in GPU QuickSched are created by the user.

7.2 Partitioning the Task/Resource Graph

To be able to use Task-Based Parallelism on hybrid memory machines, we need to address one of the primary issues of parallel computing, load balancing. Our strategy for this extends on the ideas used for `mdcore` (Section 6.3), and is somewhat similar to a data decomposition strategy.

In QuickSched, we have a sequence of dependent tasks, and each task uses and/or locks a number of resources which represent the data in the system. We want to spread the resources such that the amount of work that each node has to do is roughly the same. As each task has a `cost` associated with them (which can be either user-specified or based on runtime in previous iterations), we can work out a rough estimate of how computationally expensive each resource is.

One further complexity is that we aim to avoid communicating data back and forth during the computation - i.e. don't send data for a resource x from node A to node B, and from node B to node A in a single execution of the task graph. To enforce this, I decided that every task that locks (i.e. writes to the data owned by) a resource must be executed on the node that the resource is allocated to. This means any task that locks multiple resources that are allocated to different nodes must be executed on each node that owns one of the resources, i.e. reduce the amount of communication by replicating work.

When performing the load balancing, we therefore want to minimise two values:

1. The amount of work that has to be repeated, i.e. minimise the number of tasks executed on multiple nodes.

2. The load imbalance, i.e. minimise the difference in work executed across all nodes.

When working with no hierarchical resources, we can create a graph with one vertex for each resource in the system, where each vertex's weight is equal to the sum of the

costs of all tasks that lock the corresponding resource. We can create edges between all vertices where a task exists that locks both vertices, and the edge weight be equal to the sum of the costs of all tasks that lock both of the corresponding resources.

We can then partition this graph using traditional graph partitioning approaches, as these try to minimise both the difference in vertex cost between each partition, and minimise the edge cut between the partitions. These two values represent the load imbalance and duplicated work in the system, so these partitions should give a reasonable result.

When working with non-hierarchical resources, building the task graph is slightly more complex. The resource hierarchy can be a forest, where some of the resources are used or locked and some may not be. Rather than creating one vertex for each resource in the system, we instead create one vertex for each resource that is used or locked, but has no ancestors that are used or locked (these resources are referred to as *upper-level resources*). If a task locks a resource that isn't directly represented by a vertex, its costs are added to the upper-level resource that is an ancestor of the resource locked by the task.

This is a prime example of the difficulty of building a single scheduler to work on all possible problems, however this method should be effective for most problems.

7.3 Automated Task-Based Data Transfer

One main goal of MPI QuickSched was to create a task-based system where communication does not need to be handled directly by the user. Since the user specifies a detailed system of tasks, uses, locks and dependencies, we can work out the data requirements of the system automatically once we have an initial partitioning of the data, such as the one described in the previous section.

As we enforce that a rank executes every task that locks a local resource (a resource owned by the rank), we only need to transfer data between dependent tasks that are executed by different nodes.

If a task is executed on a different rank to any of its dependencies, then the task must wait on the data to arrive before it can be executed. Naively, we could create a send/receive pair for every such task, and force the send and receive tasks to depend on all of the tasks that unlock this task. However, this may involve sending the same data multiple times, which may be unnecessary. Instead, if we loop through the tasks in topological order, we can keep track of whether our local copy of a resource is up-to-date. For each task t that locks a non-local resource, we only create a send/receive pair if the local copy of each locked resource is not up-to-date. If we don't create a new send/receive pair, we ensure the previously created send/receive pair is unlocked by all

tasks that unlock the task t and lock the resource communicated by the send/receive pair, and the task t is unlocked by the most recently created receive task. Since all of the data dependencies need to be explicitly stated by the programmer, this is sufficient to reduce data transfer and ensure correct computation.

An example of this is shown in Figure 7.1. The figure shows two potential allocations of the 3 resources (rectangles) to nodes (coloured red and green). The 4 tasks (circles) are all dependent, and lock the resources they are joined to by a solid line, and use any resource they are joined to by a dashed line. In a), data from resource A needs to be sent from the red node to the green node after task 3 is executed, so the dependency between tasks 3 and 4 is effectively replaced by a send and receive pair. Figure b) shows an alternate allocation, where the data from resource A is sent from the red node to the green node after task 2, and the data from resource B needs to be sent from the green node to the red node before task 3. The data from resource A is never sent back to the red node, as the red node will execute all 4 tasks.

7.4 Load Balancing and Work Partitioning

Partitioning the work and data is a three step process:

1. Build the graph from the resources and tasks.
2. Partition the graph using METIS.
3. Move the resources and data associated with them around the system.

As briefly mentioned earlier in this chapter, I added a `lockers` and `users` array to the `qsched` structure. These arrays keep track of which tasks lock and use each resource, and each resource knows how many tasks use or lock it. Storing this relationship in both directions helps us to compute the resource graph efficiently.

7.4.1 Building the Resource Graph.

The METIS library takes an adjacency list input format, consisting of an edge list, vertex weight array and edge weight array. The first step of building this graph is to compute the cost associated with each resource. This is done on each rank by calling the `qsched_partition_compute_costs` function.

```
1 void qsched_partition_compute_costs( struct qsched *s, idx_t
     *res_costs){
2   struct task *t;
3   int i, j;
```

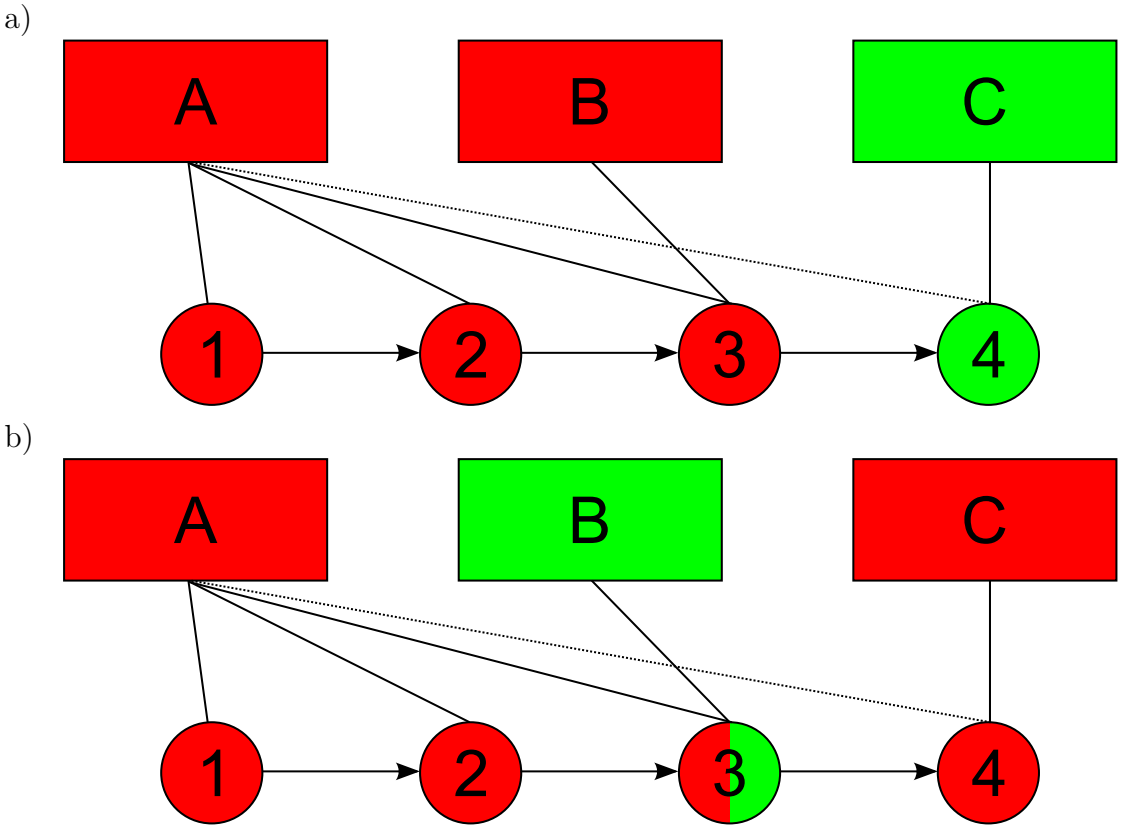


Figure 7.1: The figure shows two potential allocations of the 3 resources (rectangles) to nodes (coloured red and green). The 4 tasks (circles) are all dependent, and lock the resources they are joined to by a solid line, and use any resource they are joined to by a dashed line.
In a), data from resource A needs to be sent from the red node to the green node after task 3 is executed, so the dependency between tasks 3 and 4 is effectively replaced by a send and receive pair.
Figure b) shows an alternate allocation, where the data from resource A is sent from the red node to the green node after task 2, and the data from resource B needs to be sent from the green node to the red node before task 3. The data from resource A is never sent back to the red node, as the red node will execute all 4 tasks.

```
4    for(i = s->task_ranks[s->rank]; i < s->task_ranks[s->rank+1]; i++){
5      t = &s->tasks[i];
6      for(j = 0; j < t->nr_locks; j++){
7        res_costs[getindex(t->locks[j],s)] += t->cost;
8      }
9      for(j = 0; j < t->nr_uses; j++){
10       res_costs[getindex(t->uses[j],s)] += 1;
11     }
12   }
13 }
```

The loop in line 4 loops over all of the tasks created on the rank, and adds the task's cost to the `res_cost` of each resource that the task locks. It also increases the `res_cost` of all of the resources used by each task by 1. While tasks that only use a resource don't have a large effect on the runtime associated with a resource, there is some cost associated with having tasks executed on ranks that don't contain all the resources required for execution. I tested this by increasing `res_cost` by `t->cost` when a task used a resource, or ignoring the uses completely, but didn't see any benefit.

This array is then synchronised across the ranks using `MPI_Allreduce` with the `MPI_SUM` operator.

Once the resource costs are computed, we can build the vertex weight array. When performing this operation, we also need to keep track of which vertex corresponds to which resource in QuickSched. The function that implements this loops through all of the resources that are used or locked, and recurses through the resource hierarchy above the resource. If the resource has no parents that are locked or used, then a vertex is added to the graph to represent this rank, with the vertex's cost being equal to the `res_cost` calculated for this resource.

The `res_cost` values are also computed recursively, i.e. the `res_cost` of a resource is equal to the sum of the costs of all tasks that lock the resource plus the `res_cost` of all tasks that lock any children of that resource. This also occurs in the `qsched_partition_compute_costs` function, however it is not shown in the code snippet above.

The final step of building the graph is to create the edge and edge weight arrays. These are initialised as an array of arrays, which are merged into a single list once the arrays have been populated. To build the edgelists, we loop through the tasks, and find all pairs of locked resources for each task. For each pair, if the vertices representing the resources (or resources' parents) in the graph are distinct, an edge is added between the vertices if one does not already exist. The task's cost is then added to the weight of that edge. Since the edge cut represents the amount of repeated work, uses are ignored in this calculation, as using a resource does not enforce duplication of a task's

computation.

Once the graph has been constructed, it is partitioned using `METIS_PartGraphKway`, which outputs an array containing the partition number that each vertex belongs to.

7.4.2 Moving the Resources and Data

Once the partition is obtained, each rank loops through the partition:

```
1  for(i = 0; i < node_count; i++){
2    if(nodeIDs[i] == s->rank){
3      if(s->res[noderef[i]].node != s->rank){
4        s->res[noderef[i]].data = malloc(s->res[noderef[i]].size);
5        MPI_Irecv(s->res[noderef[i]].data, s->res[noderef[i]].size,
             MPI_BYTE, s->res[noderef[i]].node, i, s->comm, &reqs[reqnr] );
6        reqnr++;
7        s->res[noderef[i]].node = s->rank;
8      }
9    }
10   if(nodeIDs[i] != s->rank){
11     if(s->res[noderef[i]].node == s->rank){
12       MPI_Isend(s->res[noderef[i]].data, s->res[noderef[i]].size,
             MPI_BYTE, nodeIDs[i], i, s->comm, &reqs[reqnr] );
13       reqnr++;
14     }
15     s->res[noderef[i]].node = nodeIDs[i];
16   }
17 }
```

where `nodeIDs` is the partition created by METIS. For each resource that is allocated to a new node (line 3), the owning rank allocates memory to store the resource's data in line 4, and emits the `MPI_Irecv` to receive the data in line 5. The `reqs` array stores `MPI_Request` objects created by the asynchronous MPI routines. Lines 10-13 emit the `MPI_Isend` calls to send resources to other ranks.

Once all of the MPI calls have been emitted, each rank calls `MPI_Waitall` to wait for the communication to complete. Once this is complete, we loop through the resources, and update any child resources accordingly:

```
1  for(j = 0; j < s->res_ranks[s->count_ranks]; j++){
2    struct res *temp = &s->res[j];
3    struct res *parent;
```

```
4    int offset = temp->offset;
5    while(temp->parent != -1){
6      parent = &s->res[getindex(temp->parent, s)];
7      if(parent->num_lockers > 0 || parent->num_users > 0){
8        s->res[j].node = s->res[getindex(parent->ID, s)].node;
9        s->res[j].data =
             &(((char*)s->res[getindex(parent->ID,s)].data)[offset /
             sizeof(char)]);
10       break;
11     }
12     temp = parent;
13     offset += temp->offset;
14   }
15 }
```

For each resource, we recurse through the tree to find the first ancestor that was used or locked (lines 5-7). Once this is found, we update the `node` and `data` pointer and quit recursing. For each ancestor that was not used or locked, we continue to recurse, and update the data `offset` in line 13. Since all resources occur after any of their ancestors in the resource array (since any ancestors must be specified before `qsched_addchildres` can be called during setup), this data will always be up-to-date.

Once this is updated, we loop over the resources, and nullify the pointer of any resource that is not local. Finally, we loop through the tasks and set the `task_flag_skip` flag on each task that doesn't lock any local resource.

7.5 Creating the Send and Recv Tasks.

The final setup step is to create the send and receive tasks required for the computation. At this stage, the `qsched` object is completely synchronised across the ranks, and the inbuilt `qsched_addtask` and `qsched_addunlock` functions will no longer function correctly on this object, as any new tasks will be added to the wrong location in the arrays, and the object can no longer be easily synchronised. To avoid any issues occuring at this point, we create a temporary `tsched` object (which is essentially a cutdown version of a `qsched` object that doesn't contain any queues or other sections not required for this section of code), which allows us to add new tasks and dependencies. Once the send and receive tasks have been created, this is merged back into the `qsched` and deleted.

Each send and recv task has five integers associated with it (stored in the `task->data` array). These five values are necessary to be able to execute the task

during the computation, and are:

1. The rank ID of the rank sending the data.
2. The rank ID of the rank receiving the data.
3. The first 32-bits of the sent resource's ID.
4. The second 32-bits of the sent resource's ID.
5. The communication tag (required for MPI's asynchronous communication functionality).

The MPI specification requires that any two communications that occur simultaneously and share the same send rank, receive rank, and tag, must be received in the same order as they are sent. This adds an ordering to data transfer, which is unnecessary for our method. Instead, we use a different tag for every receive task on each node. The method used to create our send and receive tasks means the task pair is always created by the receiving rank, so it is sufficient to use a tag value of i when creating the i^{th} send/receive pair on each rank.

Before we can create the send and receive tasks, we initialise the `wait` counters for all of the tasks, and topologically order the tasks. We also create `first_recv` and `data_pos` arrays, which store the topological index of the first task that needs to receive a resource, and the topological index of the last task that had an up-to-date copy of a resource respectively. The values in both of these arrays are initially set to -1.

The final preprocessing step is to compute each task's parents. This can be done while setting the `wait` counters.

Once the preprocessing has been completed, each rank loops through all of the tasks in the topological ordering that need to be executed locally. These tasks have different operations applied depending on whether their `wait` counter is 0 (i.e. are not unlocked by any tasks) or not.

For every task with a `wait` counter of 0, we loop through all of the resources used and locked by the task. If a resource is not local, we check whether it has already been received on this rank (i.e. `first_recv` $>= 0$). If not, we create a send/receive pair, make the task dependent on the receive task, and set `first_recv` equal to the receive task's ID. If the data has previously been received, then we make the task dependent on the `first_recv` task for that resource instead. In this case, we do not update `data_pos`, as we only know the data will be up-to-date as of the start of task execution.

For the tasks with a nonzero `wait` counter, we want to only send the data if it is not up-to-date on this rank. As before, we loop through all of the resources used and

locked by the task. For each of these resources, we find all of the parent (unlocking) tasks that lock that resource, or any child or parent resource. If none of the parent tasks lock the resource, then we perform the same operation as if the task had a `wait` counter of 0 for that resource, as this task is not dependent on the result of any other task for this resource. Otherwise, if we didn't execute all of the parents on this rank, we check if the resource was last received before any of the parents were executed (i.e. `data_pos[res_index] < last_parent`). If so, then create a send/receive pair, make the send task depend on each of the parent tasks, and make the receive task depend on all of the locally executed parent tasks. If not, then ensure the most recent send and receive tasks for the resource depend on the parent tasks, and the receive task unlocks the work task. If we execute all of the parents, then we already have up-to-date data, so we don't need to add anything extra to the task graph. Finally, we update the `data_pos` array to reflect that we have correct data after all of this tasks parents, so set `data_pos[res_index] = last_parent`.

Once this process is complete, then each rank has a `tsched` structure that contains a number of send and receive tasks, an array of `unlocks` which contains all the task IDs that the send and receive tasks unlock, and an array of `unlockers` which contain all the task IDs that unlock send and receive tasks. These `tsched` objects are now synchronised across all of the ranks in a similar manner to the `qsched` synchronisation. At this point, every rank has all of the task information required for the computation. The final stage of setup before computation can take place, is to integrate the data from the `tsched` structure back into `qsched` structure. First, append the `tsched` task data array to the `qsched->data` array, and update the pointers for each task in the `tsched`. The next step is to place the new send and receive tasks into the task array. Since each rank makes their own send/receive pairs, the tasks cannot just be appended to the task array, as all tasks created on the same rank need to be concurrent in the task array (as otherwise the task structures cannot be quickly found from their ID during execution). We can count how many new tasks were created by each rank and create a new task array to contain all of the tasks. Each rank can then copy in the tasks from the `qsched` followed by the tasks from the `tsched` to give our new task array.

Adding the `locks` and `unlocks` to the task array requires appending the `tsched`'s array to the `qsched` array, and append the `unlockers` array to the `qsched`'s `unlock` array. Since these aren't guaranteed to be ordered correctly in the `tsched`, these arrays are sorted using bucket sort.

Finally, we loop over all of the send and receive tasks and set the `qsched_flag_skip` flag on each task that is not executed locally.

Once this is complete, the usual QuickSched setup routines are conducted and the

computation can begin.

7.6 Implementing the Send and Receive Tasks

I used Intel MPI (5.1.0, 5.1.1 and 5.1.2) throughout this chapter.

The send and receive tasks differ slightly from the other tasks in that the work associated with them should be done in the background,[1] as we use asynchronous MPI routines. This means we want to call the MPI routines as early as possible, and check the communication status as often as is feasible, as this gives the MPI runtime the maximum flexibility.

The `MPI_Isend` and `MPI_Irecv` calls require seven parameters. The first two parameters are the `data` pointer and the `size` of the data, given in multiples of the third parameter, which specifies the size of a single piece of data (we use `MPI_BYTE` for all of our routines). The fourth parameter specifies the rank ID of the receiving rank for `MPI_Isend`, or sending rank in the case of `MPI_Irecv`. The fifth parameter is the MPI message's `tag`, which is stored in the task's `data`. The sixth parameter is the `MPI_communicator` object and the final parameter is the `MPI_Request` object that is used to track of the status of the communcation.

When executing a send task, the task's `data` stores the ID of the receiving rank, the ID of the resource being sent, and the tag associated with the task. We know that the data pointer in the resource will always be initialised before computation, so it is safe to call `MPI_Isend` on the resource's `data`.

When executing a receive task, there is no guarantee that the resource's `data` pointer will have been set correctly. Before we can execute the `MPI_Irecv` call, we either need to allocate memory for the resource, or find its parent's `data` pointer if it exists. To do this, we loop upwards through the resource hierarchy to find a parent whose data pointer is not `NULL`, and set the `data` pointer to the correct offset in its parent's `data` array. This check is done for every resource, regardless of whether the resource's `data` is `NULL`. This is necessary as we could receive a child's data early on in the computation, and later receive its parent's data. Since we don't update the child pointers when we receive the parent, we need to correct the child's pointer if we later update the child resource again.

This execution all takes place at the start of the `qsched_enqueue` function, which means that as soon as a send or receive task is unlocked, the communication can begin. Once these calls have been executed, the task is placed into the first thread's

[1] what this means is MPI implementation dependent. For Intel MPI it appears to use compute cycles when the main threads/cores are not active

task queue.

To identify when communication is complete (and potentially to start/complete the communication, the exact behaviour MPI implementation dependent) we need to call `MPI_Test` on the `MPI_Request`. Whenever a task would be removed from a queue, the scheduler attempts to lock the resources associated with the task by calling the `qsched_locktask` function. When the scheduler calls the `qsched_locktask` function on a send or receive task, it calls the `MPI_Test` function on the request associated with the task. If the `MPI_Test` is successful, the `qsched_locktask` returns that it is successful. The scheduler then performs any unlocks for the communication task.

7.6.1 Retreiving the Data Pointers

As QuickSched manages all the data flow in the system, the user cannot use any pointers created during initialisation when implementing task code. To allow the user to find the data required for tasks, I created a `qsched_getresdata` function, which requires the `qsched` object and a resource ID. Since the data pointer for a resource isn't guaranteed to have been set (if only the resource's parent has been received for example), it will recurse up the hierarchy to find the correct data pointer if required.

7.7 Results

I implemented the tiled QR decomposition (introduced in Section 4.3) and Barnes-Hut algorithm (discussed in Section 5.3) with MPI QuickSched.

7.7.1 Tiled-QR Decomposition

The QR decomposition uses no hierarchical resources - each tile of the matrix (and section of the τ matrix) was created using `qsched_addres`. The task data for each task are the IDs of the resources locked or used by the task. This is the only data required for the task to be able to find the matrix data.

I ran the Tiled-QR decomposition on a 8192×8192 matrix, with tiles of size 128×128, resulting in a matrix broken down into 64×64 tiles. This was used to perform two scaling tests, the first to demonstrate the strong scaling and parallel efficiency from 1-96 cores (12 cores per node), and the second to compare performance of shared memory performance vs MPI-only setups. For the latter, I ran the code with 1-12 cores, using 1/2/3/4/6/12 cores per rank and examined the difference in performance.

Figure 7.2 shows reasonable scaling on this test case, which takes 303.7s on a single core and 4.6s on the full 96 cores I tested it on. Furthermore, QR appears to still be scaling at 96 cores, suggesting that this problem can continue to scale to higher node

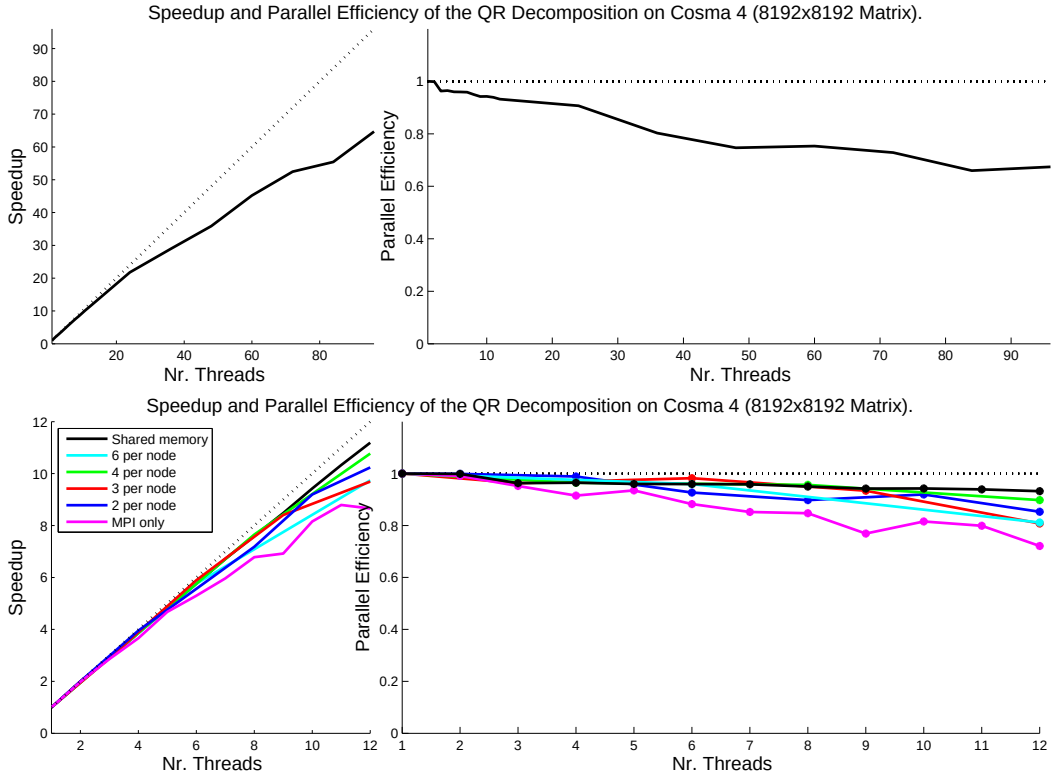


Figure 7.2: Speedup and efficiency plots for the QR decomposition on a 8192×8192 matrix, with tiles of size 128×128. The code scales up to 8 full nodes at an efficiency of 67%. When comparing MPI only performance to shared memory performance, the code drops from 93% efficiency (shared memory only) to 73% (MPI only). The code generally performs better with more threads per node (as expected).

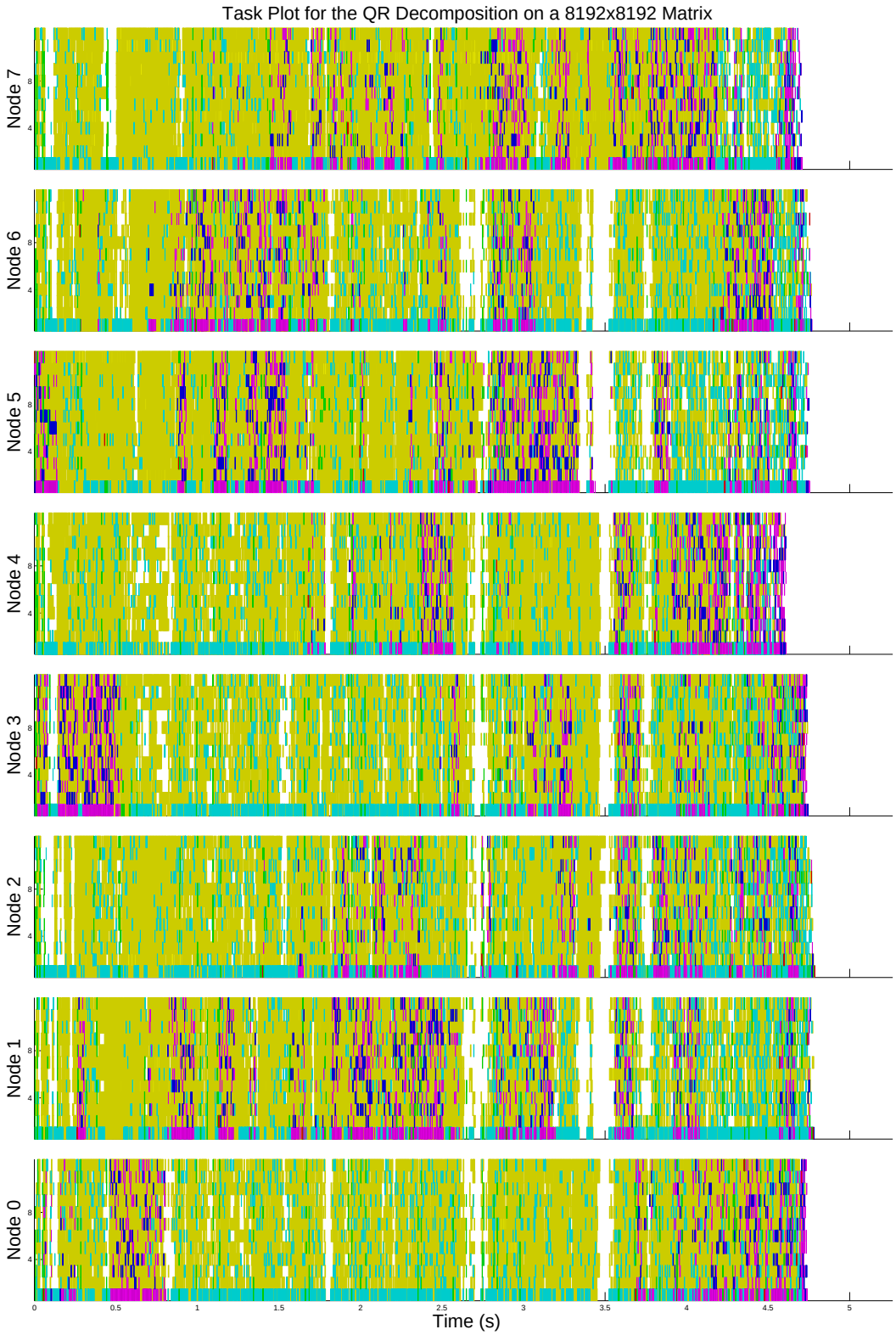


Figure 7.3: A task plot for the execution of the QR decomposition on a 8192×8192 matrix on 8 nodes. The resource allocation given by METIS results in no tasks being duplicated, which helps with the performance. However, some sections of the computation result in all but one of the nodes waiting on communication before they can continue. The send tasks are magents and the receive tasks are teal.

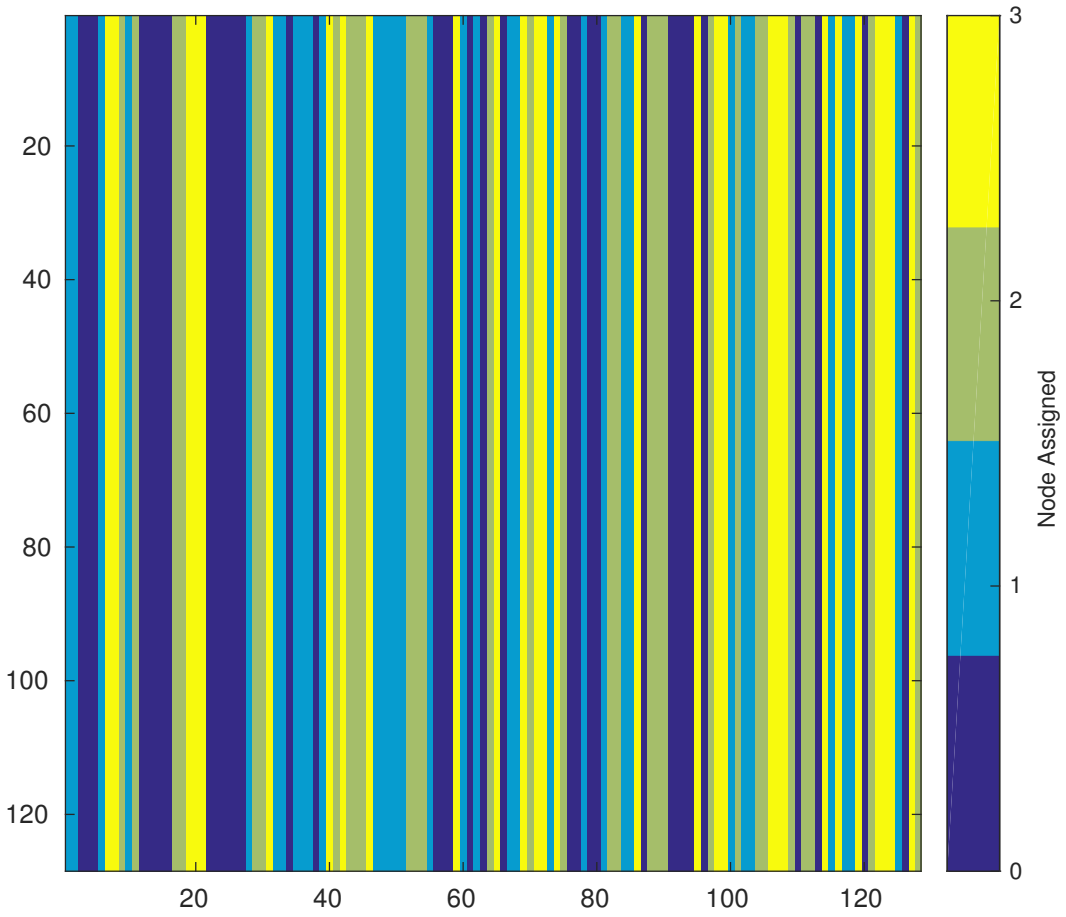


Figure 7.4: The matrix decomposition for 4 ranks. Each tiles is coloured according to the rank it is assigned to. The matrix is striped by the automatic decomposition tool.

counts if available. As the number of nodes increase, we get the expected fluctutations in parallel efficiency, as the partitioning will likely perform better at some node counts than others.

Figure 7.3 shows a task plot on 8 nodes for a 8192×8192 matrix (tiles of size 128×128). Overall, the computation is reasonably load balanced. However, there are several areas where multiple nodes are idle whilst waiting on data to be computed on other nodes. Due to the dependency structure of the QR, this can be difficult to avoid.

The matrix decomposition for 4 ranks is shown in Figure 7.4. Each of the tiles is coloured according to the rank it is assigned to. The automatic decomposition appears to stripe the matrix, which seems like a sensible way of balancing the tasks between the node, though towards the end of the computation some nodes will be idle. This is difficult to avoid with the QR decomposition, and load balancing the computation may also result in increased communication.

Many Linear Algebra problems can be easier to load balance than other simulation types (such as particle simulations) in that each matrix operation takes a fixed number of arithmetic operations. This allows the user to make accurate estimates of task

pc_depth	Runtime	Number of tasks
1	16.7s	10.5M
2	18.4s	10.5M
3	18.9s	10.7M
4	48.0s	26.5M

Table 7.1: Average time taken to compute the accelerations for a single timestep of a Barnes-Hut Simulation on a 3 million particle cosmological volume as the `pc_depth` parameter is varied. This setup used a maximum of 128 particles per leaf cell, and 12 threads on a single node. As `pc_depth` increases, the runtime increases. This is due to an increase in the number of tasks, leading to increased contention over resources and more scheduling overheads.

costs. In QuickSched I used costs of 200 for DGEQRF operations, 300 for DLARFT and DTSQRF operations, and 500 for DSSRFT operations (the more computationally expensive tasks have higher costs). It may be possible to improve performance slightly with more accurate cost estimates, however these values gave us reasonably good load balancing across nodes.

7.7.2 Barnes-Hut Simulation

The Barnes-Hut simulation is built on hierarchical resources. The space (single cell) is created using `qsched_addres`, and the particles are added to the space. When the particle sort and subcell creation is done, any subcells that contain particles are created using `qsched_addchildres`. Along with the child resource, each `cell` object (which contains data about the cell other than the particle data) required is created with `qsched_addres`, and if a task uses or locks particles in a cell, it needs to know the resource ID of the child resource and the `cell` object.

I performed the same scaling tests for the Barnes-Hut simulation as for the Tiled-QR decomposition. The particle distribution used was a 3 million particle cosmological volume, a small approximation of the type of particle distribution that would be used for large cosmological runs.

One other parameter that was added for the MPI Barnes-Hut is the `pc_depth`. This parameter can be used to tune how many times the `create_pcs` function recurses before creating the particle-cell tasks. This does not affect the accuracy of the simulation, but affects how many particles are involved in a single particle-cell task. With a higher value of `pc_depth`, the `create_pcs` function creates more tasks, which leads to more contention over resource locks, leading to worse shared memory scaling. Table 7.1 shows how the shared memory runtime is affected by the value of this variable.

One downside with the algorithm implemented for the Barnes-Hut as part of MPI QuickSched is the large number of particle-cell tasks it creates. In the shared memory

Max. parts per cell	Runtime (1 thread)	Runtime (12 threads)	Nr. tasks	Percentage of runtime spent in gettask (12 threads)
128	74.1s	16.8s	10.5M	70.2%
256	87.1s	12.6s	4.9M	58.2%
512	124.3s	12.9s	2.3M	32.5%
1024	202.9s	18.6s	1.1M	6.0%

Table 7.2: Average time taken to compute the accelerations for a single timestep of a Barnes-Hut Simulation on a 3 million particle cosmological volume as the maximum number of particles per cell is varier. This setup used a `leaf_depth` of 1, and a single node. If interactions are created on smaller cells, the tasks require less computation on average, resulting in an increase in scheduling overhead, and a loss of scaling.

CPU implementation, the execution time increases as the maximum number of particles in a cell increases, as the cell-pair interactions dominate the computation, and increasing the number of particles in a cell increases the number of particle-particle interactions. With the MPI QuickSched implementation, this is not necessarily the case, as shown in Table 7.2.

I also examined the scaling of the code with 1-12 threads in a variety of layouts when differing the maximum number of particles in each cell.

Figure 7.5 shows the scaling of the code with different layouts of up to 12 threads, with different numbers of particles per cell. With small numbers of particles per cell, shared memory scaling is poor. This is due to the scheduler overheads when locking large numbers of very small tasks, and can result in the scheduler taking nearly 50% of the overall runtime. MPI performance worsens as the number of particles per cell increases, however this is probably since the performance of a single thread is higher, so the overheads due to MPI is a larger percentage of the computation.

Figure 7.6 shows the scaling of the Barnes-Hut from 1-12 threads in a single node, followed by MPI scaling up to 8 nodes (96 total threads). The code continues to scale well up to 7 nodes, before the performance stops improving on 8 nodes. It is not clear from this whether it is just a poor work partition on 8 nodes, or if the implementation will no longer scale beyond 8 nodes. On 8 nodes, the simulation is only running 375k particles per node, so 31k particles per core, which is quite a small number for a simulation of this type.

Figure 7.7 shows a task plot for the Barnes-Hut on a 3 million particle run on 8 nodes of Cosma 4. This figure suggests that the load balancing of the simulation could be improved between nodes. All of the communication takes place at the beginning of the simulation, as these tasks are all ready immediately, so not as much as much work and communication is happening in parallel.

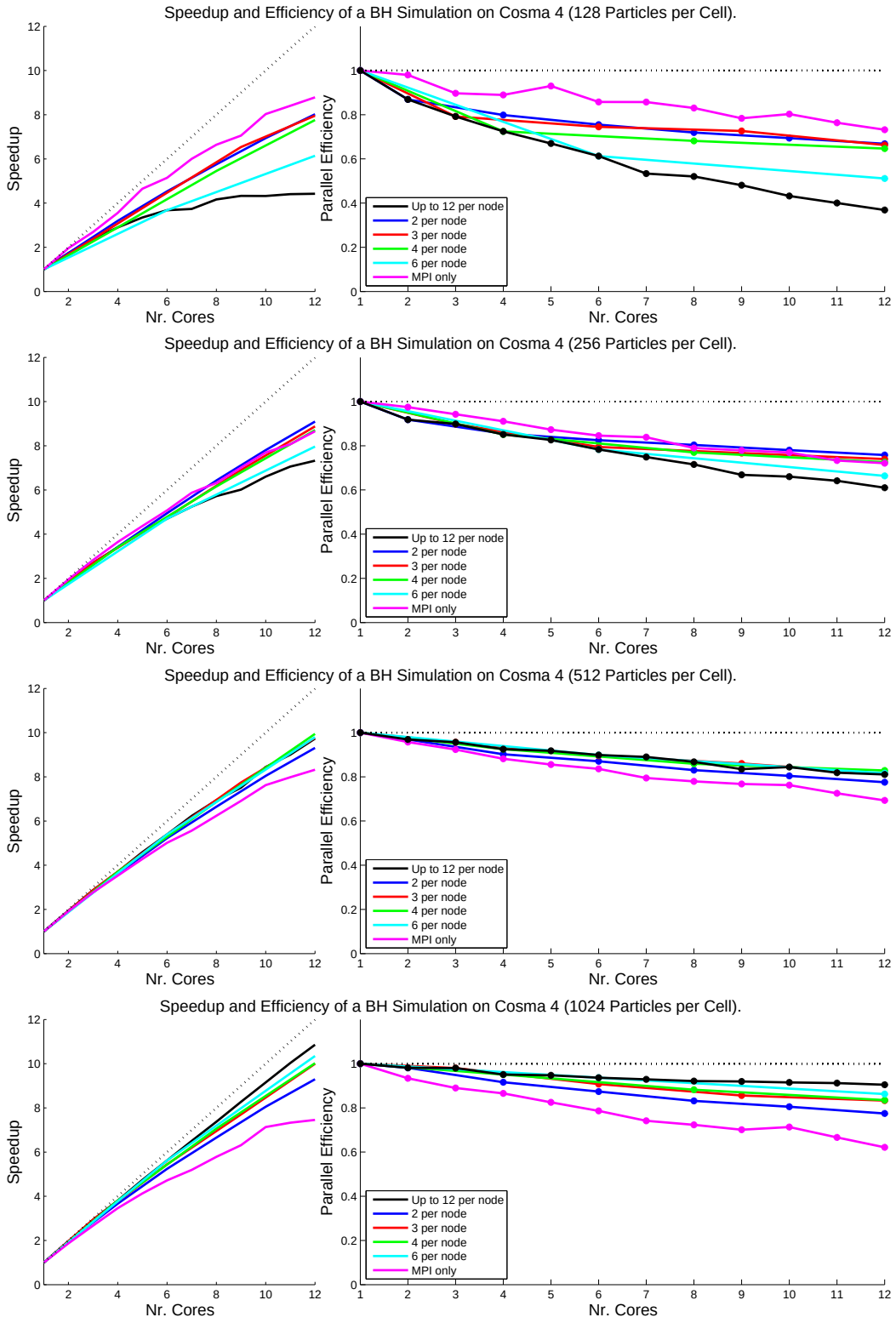


Figure 7.5: Speedup and efficiency plots for the Barnes-Hut simulation on a 3 million particle cosmological volume, as the number of particles per cell is changed.

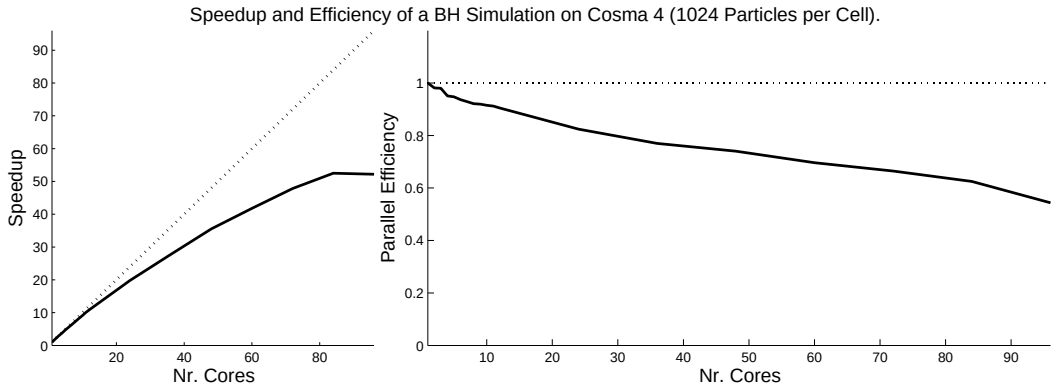


Figure 7.6: Speedup and efficiency plots for the Barnes-Hut simulation on a 3 million particle cosmological volume, with a maximum of 1024 particles per cell. The code scales up to 8 full nodes at an efficiency of 54%.

7.8 Conclusions and Future Work

In this chapter I have shown a proof-of-concept method for a hybrid shared-distributed memory task-based parallel library with automated data transfer. I use a graph partitioning based approach for load balancing between nodes, and have shown the methods efficacy on two problems on small scale computations.

While the simulations paralellise effectively, a lot of the work done to setup the scheduler is done in serial. In particular, the `qsched_partition` function can be slow with high task and resource counts, and more so in hierarchical cases. Since a lot of this is performed by a single thread on a single node, finding a way to parallelise the operations could help reduce the overheads introduced by this setup function.

One other area to improve could be the load balancing method. Currently a graph partitioning approach is used to partition the resources based on the estimated amount of work for each resource. It is possible that using a local search or genetic algorithm approach to partition the resources (either by itself or to improve the graph partitioning result) might be more effective than a graph partitioning approach, however I have not yet investigated this.

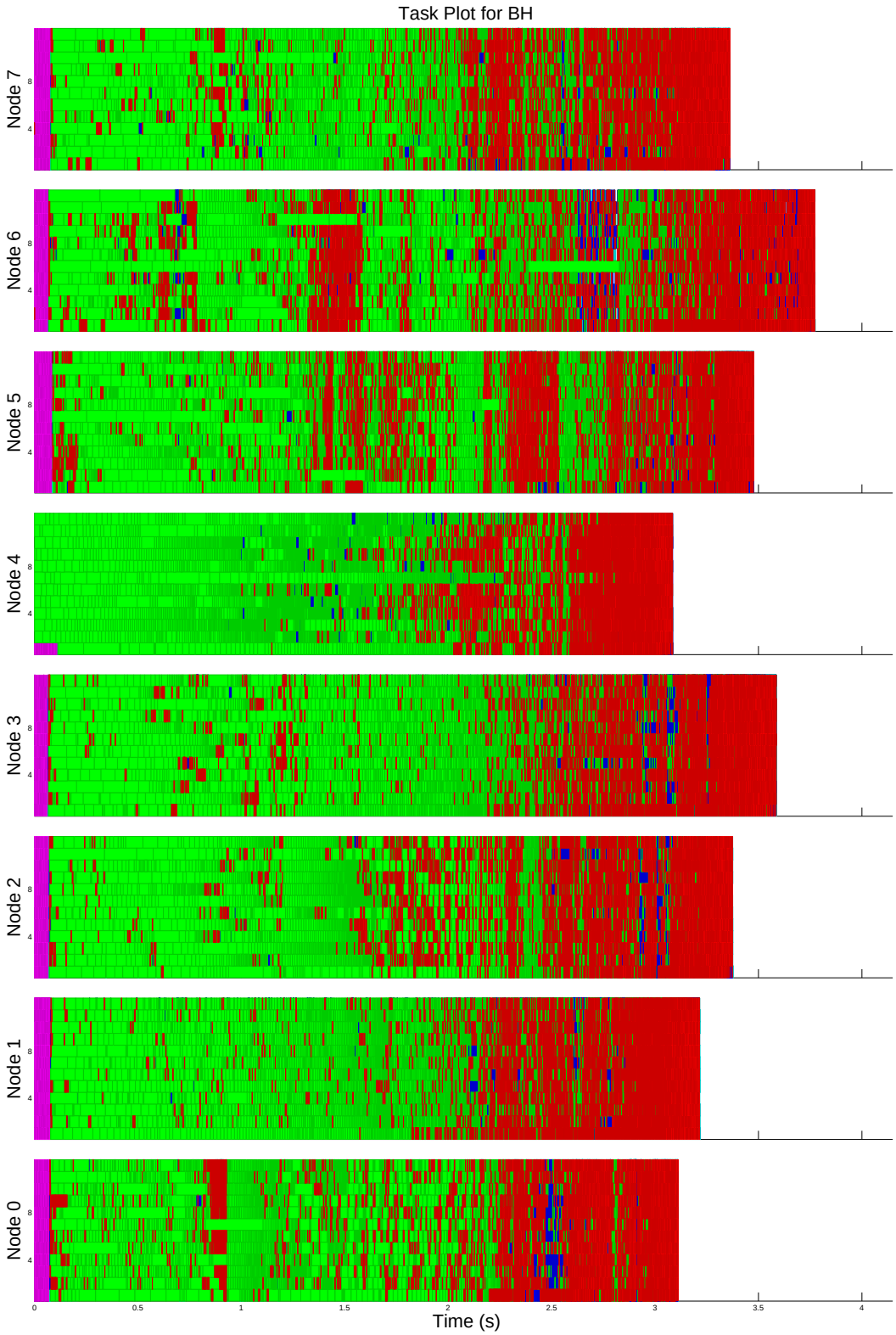


Figure 7.7: A task plot of the Barnes-Hut simulation on a 3 million cosmological volume run on 8 nodes. The load balancing could clearly be improved on this number of nodes, as the nodes have up to 15% load imbalance. The send tasks are magenta and the receive tasks are teal.

Chapter 8

Conclusions and Future Work

In this thesis, I have introduced two independent strategies which enable the use of Task-Based Parallelism on GPUs and on hybrid shared-distributed memory machines. The major contribution in both cases is the automated data management between each memory region, which aims to reduce the burden on the programmer.

The initial GPU work in `mdcore` led to many improvements. The biggest improvement to the runtime in `mdcore` came from moving from 32 threads in each block to 128. This change improved up the performance dramatically, despite the additional synchronisation required inside the blocks. The other main conclusion drawn from this work is that the cost of conflicts on the GPU was high, and that if these can be avoided by using atomics it may lead to better performance.

I also looked at using a graph partitioning approach for domain decomposition. The task tree is used to create the graph to be partitioned, and other information from the tasks (such as the task's expected cost) is used to weight the graph. This was shown to create better partitions than a simple bisection approach. This method has been extended in SWIFT [40], as SWIFT measures each task's execution time during the computation. In later timesteps, this information gives more precise information about each task's cost, allowing the graph partitioning library to create better partitions. Additionally, SWIFT has worked on a strategy to minimise the data transfer required for subsequent partitions, e.g. if METIS were to output identically partitions with different numberings, SWIFT is capable of mapping the new partition onto the old partition.

One further idea that can be considered is to use a genetic algorithm to improve upon the initial partition. The graph partition approach I used in this thesis tries to minimise the edge cut (which represents the amount of communication/repeated computation) while attempting to balance the sum of the weights of the vertices (amount of computation) in each partition. It may be better to minimise the maximum vertex weight of any partition. Brief experimentation with a simple genetic algorithm sug-

gested that if the genetic algorithm is started from a random initial partition it is slow to converge, however it could improve a partition generated by METIS in relatively few iterations. I feel that more experimentation with alternative partitioning strategies would be valuable, as it may improve the load balancing generated by the partitioning approaches discussed in this thesis.

Our work with the QuickSched extensions for the GPU focused primarily on merging the approach developed for `mdcore` with the idea of automatic data transfer as part of the task-based scheduler. I extended QuickSched to allow the user to provide data pointers for each resource. Using information about the task graph and the data requirements of each task, the scheduler creates tasks to transfer the data to and from the GPU, and sets up dependencies so tasks only execute once the data required for them has been transferred to the GPU, and to transfer data from the GPU as soon as the computation on the data is complete. More coarse-grain approaches to this are now standard in both OpenACC and the gcc OpenMP implementation (OpenMP Target directives). Our setup performs best when only a small number of blocks perform the data transfer to the GPU, while the rest begin computation as it becomes ready. Using only a few blocks (the required number is likely specific to each GPU) to transfer data with ILP allows the kernel to transfer data at the maximum speed through the PCI-express bus, enabling most of the device to be used for computation.

While I didn't test implementing priority queues on the GPU, the setup used multiple queues to prioritise certain tasks. Our implementation tested at most 4 queues, the highest priority queue containing unload and ghost tasks, the next highest the load tasks (however was only ever accessed by a small number of blocks), and the remaining contain the user-defined work tasks. Enabling the user-defined tasks to have different levels of priority is clearly beneficial for certain problems where some tasks may lock large numbers of resources, or have much higher runtimes that other tasks.

The QuickSched extensions for the GPU do not support multi-GPU usage inside a single process, though the programmer could use multiple MPI ranks to make use of multiple GPUs. Any data decomposition would have to be handled by the programmer before using QuickSched. I did not investigate multi-GPU usage at the time, as I do not have a good strategy for synchronising data between GPUs, nor for load balancing between GPUs. With the addition Compute Capability 6.0, CUDA GPUs now have access to the `atomicAdd_system` function (along with a range of other system-wide atomics) which enables synchronisation across the entire process. Additionally, NVLINK increases the speed of data transfer between devices, and may allow work stealing to occur between GPUs in the same process.

The final piece of work discussed in this thesis is the QuickSched extensions for hybrid shared-distributed memory CPUs, focused on automating the data transfer

between MPI ranks. In MPI QuickSched, the scheduler manages all of the memory required during the task-based computation, i.e. any allocations or deallocations of memory during the computation are managed by QuickSched. Similarly to the GPU extensions to QuickSched, the data movement is modeled using tasks (send tasks and receive tasks), which are automatically created by the scheduler once the tasks required for the computation have been specified. MPI QuickSched uses the same graph partitioning approach as tested for `mdcore` to automatically load balance the work between the MPI ranks.

The data transfer tasks made extensive use of the nonblocking MPI routines (`MPI_Isend`, `MPI_Irecv`, `MPI_Test`) to perform the communication in parallel to the computation. The library also requires full `MPI_THREAD_MULTIPLE` support to function correctly. Our results showed that this approach was reasonably successful for small testcases.

The MPI QuickSched implementation is only a proof of concept, and has a few major limitations that will prevent it performing well at large node counts. The main limitation is that every node requires a global view of the computation, i.e. all of the nodes need to know all of the tasks, dependencies, and resources that are in the computation (though not the data associated with the resources). This limits the size of the computation that can be performed with MPI QuickSched due to the memory overheads. The other limitation is that all of the scheduler setup is performed by only a single thread on each rank, which will limit the overall scalability of the method. It should be possible to avoid these issues, and for the approach to be used for large workloads these would need to be solved.

The two QuickSched extensions cannot currently be used in tandem. With the progression of High Performance Computers towards heterogeneous architectures, it is essential to be able to make use of all of the hardware available on each node to maximise performance. The main hurdle is dividing the computation between the CPU and the GPU, and being able to notify the GPU of the arrival of data from other ranks. A check function for the GPU could be created using the new system-wide atomics, where the GPU tests a part of the CPU memory to see if the data has arrived and can be copied to the GPU, however experimentation would be needed to see how often this variable should be checked.

This thesis attempts to create a general purpose task framework that performs well for any problem. The results show that using a single setup for all domains may not be optimal. Allowing the user to tweak the options to suit their specific problem can yield better performance. Having a large suite of options which need to be tweaked by the user to extract performance can be daunting, especially for non-expert users. This is a common complaint amongst domain scientists. Domain specificity is becoming popular

in HPC, with domain specific languages aiming to separate the domain scientist from the performance aspects of their code. To keep up with this trend, task frameworks need to both be able to perform well for the general case, but also allow fine-tuning for specific domains through different scheduling options or other tasking extensions. An example of this is conflicts (or commutative dependencies), which significantly improve the performance of certain particle-based algorithms.

Appendices

Appendix A

Kernel-based data transfer to the GPU

I created a small test program to test kernel-based data transfer to the GPU. The program consisted of 3 kernels:

1. A kernel to copy the data to the GPU.
2. A kernel to square every value in the array.
3. A kernel to copy the data back to the host.

I created multiple versions of this first kernel, some of which included instruction-level parallelism:

```
1  __global__ void runner_run_copyTo(int *d_a, int *h_a, int *h_b, int N){
2    int i;
3    for(i = blockIdx.x*blockDim.x + threadIdx.x; i < N ; i+=
         blockDim.x*gridDim.x){
4      d_a[i] = h_a[i];
5    }
6  }
7
8  __global__ void runner_run_copyTo_ILP2(int *d_a, int *h_a, int *h_b,
       int N){
9    int i;
10   int val1, val2;
11   for(i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i+=
         2*blockDim.x*gridDim.x){
12     val1 = h_a[i];
13     val2 = h_a[i + blockDim.x*gridDim.x];
```

```
14      d_a[i] = val1;
15      d_a[i + blockDim.x*gridDim.x] = val2;
16    }
17  }
18
19  __global__ void runner_run_copyTo_ILP3(int *d_a, int *h_a, int *h_b,
        int N){
20    int i;
21    int val1, val2, val3;
22    for(i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i+=
          3*blockDim.x*gridDim.x){
23      val1 = h_a[i];
24      val2 = h_a[i + blockDim.x*gridDim.x];
25      val3 = h_a[i + 2*blockDim.x*gridDim.x];
26      d_a[i] = val1;
27      d_a[i + blockDim.x*gridDim.x] = val2;
28      d_a[i + 2*blockDim.x*gridDim.x] = val3;
29    }
30  }
31
32  __global__ void runner_run_copyTo_ILP4(int *d_a, int *h_a, int *h_b,
        int N){
33    int i;
34    int val1, val2, val4, val3;
35    for(i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i+=
          4*blockDim.x*gridDim.x){
36      val1 = h_a[i];
37      val2 = h_a[i + blockDim.x*gridDim.x];
38      val3 = h_a[i + 2*blockDim.x*gridDim.x];
39      val4 = h_a[i + 3*blockDim.x*gridDim.x];
40      d_a[i] = val1;
41      d_a[i + blockDim.x*gridDim.x] = val2;
42      d_a[i + 2*blockDim.x*gridDim.x] = val3;
43      d_a[i + 3*blockDim.x*gridDim.x] = val4;
44    }
45  }
```

I then ran 6 variants of the code on an array of size $15,360,000$. The first variant used `malloc` to initialise the memory and `cudaMemcpy` functions to copy the data to the device. The second used `cudaMallocHost` to initialise the memory and `cudaMemcpy` functions to copy the data to the device. The remaining variants used `cudaMallocHost`

Variant	Runtime
`malloc` + `cudaMemcpy`	83.907394ms
`cudaMallocHost` + `cudaMemcpy`	83.210945ms
no ILP	83.176003ms
2-way ILP	83.198174ms
3-way ILP	83.139969ms
4-way ILP	83.149376ms

Table A.1: Runtimes of each method for data transfer plus the small kernel with the GPU. Kernel-based approaches seem to perform best, and `cudaMallocHost` leads to better performance than `malloc`

Variant	Runtime
no ILP	144.102692ms
2-way ILP	89.689377ms
3-way ILP	80.860771ms
4-way ILP	80.818077ms

Table A.2: Runtimes of each kernel-based method for data transfer to the GPU. Even with only 2 blocks, using 4-way ILP allows us to reach high data-transfer speeds.

to initialise the memory and a different `copyTo` kernel to copy data to the device. The final 4 variants all used the same kernel to copy data back from the host, and all variants used the same kernel to perform the array operation. The non-`cudaMemcpy` variants all used 128 threads per block and 128 blocks to perform data transfer. The methods were timed using the cuda Event API.

These results are shown in table A.1. These results suggest kernel-based data transfer methods lead to the best performance.

A.1 Reducing occupancy

I also wanted to reduce the number of blocks required to keep memory transfer speeds high. I tested with differing block counts, and found I could reach peak performance with as few as 2 blocks if I used 4-way ILP. Table A.2 shows the performance of the kernel-based methods with 2 blocks, each with 128 threads on an array of size 15,000,000 elements. This shows that using 4-way ILP can lead to high speed data transfer with small block counts.

Appendix B

Does GPU-based kernel transfer affect the speed of CPU code?

One potential performance issue I did not check was if GPU access to CPU memory affected CPU performance. To test this, I created a pair of memory bound kernels, one for the GPU and one for the CPU, and tested the CPU performance with and without the GPU active, while ensuring the GPU program took longer than the CPU program. The test program is shown below:

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <math.h>
7  #include <assert.h>
8  #include <omp.h>
9  #include <time.h>
10
11 inline
12 cudaError_t checkCuda(cudaError_t result){
13 #if defined(DEBUG) || defined(_DEBUG)
14   if (result != cudaSuccess){
15     printf("CUDA Runtime Error: %sn",
16           cudaGetErrorString(result));
17     assert(result == cudaSuccess);
18   }
19 #endif
20   return result;
```

```
21 }
22
23 __global__ void runner_run(int *d_a, volatile int *h_a, int *h_b, int
       N){
24   int i, j;
25   for(j = 0; j < 8; j++)
26     for(i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i+=
         blockDim.x*gridDim.x){
27       d_a[i] = h_a[i];
28       d_a[i] += d_a[i];
29       h_a[i] = d_a[i];
30     }
31 }
32
33 int main( int argc , char *argv[] ){
34   int blocks = 128;
35   int warps = 4;
36   float cuda_time=0.0f;
37   int c;
38   while ( ( c = getopt( argc , argv , "m:n:" ) ) != -1 )
39     switch( c ) {
40       case 'm':
41         if ( sscanf( optarg , "%i" , &blocks ) != 1 ){
42           printf( "Error parsing blocks." );
43           exit( EXIT_FAILURE);
44                                       }
45         break;
46       case 'n':
47         if ( sscanf( optarg , "%i" , &warps ) != 1 ){
48           printf( "Error parsing warps." );
49           exit( EXIT_FAILURE);
50                                       }
51         break;
52     }
53   const int N = blocks*32*warps*12 * (100000/blocks);
54   const int bytes = 4*N;
55   int i,j,k;
56   omp_set_num_threads(4);
57   cudaEvent_t startEvent, stopEvent;
58   cudaSetDevice(0);
59   double timespent1 =0.0, timespent2 =0.0;
```

```
60    for(k = 0; k < 10; k++){
61      cudaEventCreate(&startEvent);
62      cudaEventCreate(&stopEvent);
63      volatile int *h_a;
64      int *h_b;
65      int *d_a;
66      cudaMallocHost((void**) &h_a, bytes);
67      cudaMallocHost((void**) &h_b, bytes);
68      cudaMalloc((int**)&d_a, bytes);
69      for(i = 0; i < N; i++)
70        h_a[i] = i;
71      const int host_N = N / 16;
72      int *host = (int*) malloc(bytes/16);
73      #pragma omp parallel for
74      for(i = 0; i < host_N; i++)
75        host[i] = i;
76      cudaEventRecord(startEvent,0);
77      runner_run<<<blocks, 4*warps>>>(d_a, h_a, h_b, N);
78      cudaEventRecord(stopEvent,0);
79
80      clock_t begin = clock();
81      for(j = 0; j < 80; j++){
82        #pragma omp parallel for
83        for(i = 0; i < host_N; i++){
84          host[i] += host[i];
85        }
86      }
87      clock_t end = clock();
88      timespent1 += ((double)(end-begin) / CLOCKS_PER_SEC);
89      cudaEventSynchronize(stopEvent);
90      checkCuda(cudaPeekAtLastError());
91      float time;
92      cudaEventElapsedTime(&time, startEvent, stopEvent);
93      cudaDeviceSynchronize();
94      printf("CUDA runtime: %.3f\n",
            (/*(float)bytes*1000.0)*/time/1000.0));
95      #pragma omp parallel for
96      for(i = 0; i < host_N; i++)
97        host[i] = i;
98      begin = clock();
99      for(j = 0; j < 80; j++){
```

```
100       #pragma omp parallel for
101       for(i = 0; i < host_N; i++){
102         host[i] += host[i];
103       }
104     }
105     end = clock();
106     timespent2 += ((double)(end-begin) / CLOCKS_PER_SEC);
107     for(i = 0; i < N; i++)
108       h_a[i] = i;
109     cudaEventRecord(startEvent,0);
110     runner_run<<<blocks, 4*warps>>>(d_a, h_a, h_b, N);
111     cudaEventRecord(stopEvent,0);
112     cudaEventSynchronize(stopEvent);
113     checkCuda(cudaPeekAtLastError());
114     cudaEventElapsedTime(&time, startEvent, stopEvent);
115     cuda_time += time;
116     cudaFree(d_a);
117     cudaFreeHost((void*)h_a);
118     cudaFreeHost(h_b);
119     free(host);
120   }
121   timespent1 = timespent1 / 10.0;
122   timespent2 = timespent2 / 10.0;
123   cuda_time = cuda_time / 10.0 / 1000.0;
124   printf("Average time to run CPU in parallel = %f\n", timespent1);
125   printf("Average time to run CPU alone = %f\n", timespent2);
126   printf("Average time to run GPU alone = %f\n", cuda_time);
127 }
```

Running the program resulted in the following output:

```
1 Average time to run CPU in parallel = 2.424000
2 Average time to run CPU alone = 2.421000
3 Average time to run GPU alone = 5.252724
```

It is possible there is a small amount of cost to running the CPU and GPU in parallel, however this was $< 0.1\%$ of the runtime, so it is unlikely to cause any major issues.

Appendix C

Implementation details for the Tiled QR decomposition

C.1 Reduction function using shared memory

On old CUDA architectures, warp reductions need to be implemented with shared memory as follows:

```
__shared__ float vec[32*WARPS];
int tid = threadIdx.x;
vec[tid] = value;
int group = threadIdx.x / 32;
char n = 32 >> 1;
while(n > 0 ){
  if(tid < n ) vec[tid] = vec[tid] + vec[tid + n];
  n = n >> 1;
}
__threadfence();
*value = vec[group * 32];
```

C.2 SLARFT implementations

The CUDA implementations of the SLARFT function are shown in this section:

```
void __device__ SLARFT(volatile float* cornerTile, volatile float*
    rowTile,
       int tilesize, int jj, int kk, volatile float* tauMatrix, int
          tauNum){
```

```
3   int i, j;
4   float z;
5   float w;
6   int TID = threadIdx.x % 32;
7   int set = threadIdx.x / 32;
8   for(i = 0; i < tilesize; i++){
9     if(TID > i)
10      w = cornerTile[i*tilesize + TID];
11    if(TID == i)
12      w = 1.0;
13    if(TID < i)
14      w = 0.0;
15    for(j = set; j < tilesize; j+=(blockDim.x/32)){
16      z=0.0;
17      z = w * rowTile[j*tilesize+TID];
18      reduceSumMultiWarp(&z);
19      if(TID >= i)
20        rowTile[j*tilesize + TID] = rowTile[j*tilesize + TID] -
21      tauMatrix[(kk*tilesize+i)*tauNum + kk] * w * z;
22    }
23  }
24 }
```

Each warp individually loops over the rows in the corner tile (line 8), and reads in `w` (which is always stored in the lower diagonal of `cornerTile`). The warps then applies the transformation to the column tile (lines 16-23) in strides of `blockDim.x/32`. As the same thread always updates the same values in `rowTile`, there is no need for any explicit synchronization during the routine. The SSSRFT function can be implemented similarly. This implementation could easily be extended to use `K = blockDim.x` as follows:

```
1  void __device__ SLARFT(volatile float* cornerTile, volatile float*
       rowTile,
2          int tilesize, int jj, int kk, volatile float* tauMatrix, int
               tauNum){
3   int i, j;
4   float z;
5   float w;
6   int TID = threadIdx.x;
7   int set = threadIdx.x;
```

```
8    __shared__ int reduction;
9    for(i = 0; i < tilesize; i++){
10     if(TID > i)
11       w = cornerTile[i*tilesize + TID];
12     if(TID == i)
13       w = 1.0;
14     if(TID < i)
15       w = 0.0;
16     if(TID == 0)
17       reduction = 0;
18     __syncthreads();
19     for(j = 0; j < tilesize; j++){
20       z=0.0;
21       z = w * rowTile[j*tilesize+TID];
22       reduceSumMultiWarp(&z);
23       if(TID % 32 == 0){
24         atomicAdd(&reduction, z);
25       }
26       __syncthreads();
27       if(TID >= i)
28         rowTile[j*tilesize + TID] = rowTile[j*tilesize + TID] -
29       tauMatrix[(kk*tilesize+i)*tauNum + kk] * w * reduction;
30     }
31   }
32 }
```

where we have to use shared memory to perform a reduction over the multiple warps in the block. This would allow the operations to use tiles where $K == blockDim.x$. It is possible that using larger tiles could lead to faster calculations for larger matrices as we would reduce the number of idle threads during SGEQRF and STSQRF functions, however I did not test this during my thesis.

Appendix D

C code for the Barnes-Hut algorithm with QuickSched

The task to create the tasks is implemented as follows:

```
1  void create_tasks(struct qsched *s, struct cell *ci, struct cell *cj){
2    qsched_task_t tid;
3    struct cell *data[2], *cp, *cps;
4    if (cj == NULL){
5      if (ci->split && ci->count > task_limit / ci->count){
6        for (cp = ci->firstchild; cp != ci->sibling; cp = cp->sibling){
7          create_tasks(s, cp, NULL);
8          for (cps = cp->sibling; cps != ci->sibling; cps = cps->sibling)
9            create_tasks(s, cp, cps);
10       }
11     }else{
12       data[0] = ci;
13       data[1] = NULL;
14       tid = qsched_addtask(s, task_type_self, task_flag_none, data,
15                            sizeof(struct cell *) * 2, ci->count * ci->count /
                                 2);
16       qsched_addlock(s, tid, ci->res);
17     }
18   }else{
19     if (are_neighbours(ci, cj)){
20       if (ci->split && cj->split && ci->count > task_limit / cj->count){
21         for (cp = ci->firstchild; cp != ci->sibling; cp = cp->sibling){
22           for (cps = cj->firstchild; cps != cj->sibling; cps =
                   cps->sibling){
23             create_tasks(s, cp, cps);
```

```
24          }
25        }
26      }else{
27        data[0] = ci;
28        data[1] = cj;
29        tid = qsched_addtask(s, task_type_pair, task_flag_none, data,
30                             sizeof(struct cell *) * 2, ci->count *
                                 cj->count);
31        qsched_addlock(s, tid, ci->res);
32        qsched_addlock(s, tid, cj->res);
33      }
34    }
35  }
36 }
```

where `task_limit` is a predefined variable that controls how small individual tasks can get. If the algorithm is supplied a single cell `ci`, then it checks whether to recurse in line 7. If so, it recurses on each child of `ci` (line 11) and each pair of children of `ci` (line 14). If `ci` is not split (i.e. doesn't have children) then it creates a self interaction task for `ci`. If the algorithm is supplied with a pair of cells `ci` and `cj`, it checks if they are neighbouring cells in line 31. If so, and both cells are split, then the algorithm recurses on all pairs of children from `ci` and `cj` in line 37. If either cell is not split, then a pair direct interaction task is constructed between cells `ci` and `cj`. The locks added in lines 26, 48 and 49 tell the scheduler which data is written to by the tasks. When building the octree, a particle-cell task is created for every leaf task.

When tasks are created on non-leaf cells (i.e. when `task_limit` causes early termination of the recursion), the execution of the direct interaction tasks recurses before doing the interaction on leaf cells.

```
1  static inline void iact_self_pc(struct cell *c, struct cell *leaf){
2    struct cell *cp, *cps;
3    for (cp = c->firstchild; cp != c->sibling; cp = cp->sibling){
4      if (is_inside(leaf, cp)) break;
5    }
6    if (cp->split){
7      iact_self_pc(cp, leaf);
8      for (cps = c->firstchild; cps != c->sibling; cps = cps->sibling){
9        if (cp != cps && cps->split) iact_pair_pc(cp, cps, leaf);
10     }
11   }
```

```
12 }
```

```
1  static inline void iact_pair_pc(struct cell *ci, struct cell *cj,
       struct cell *leaf){
2    struct cell *cp, *cps;
3    for (cp = ci->firstchild; cp != ci->sibling; cp = cp->sibling){
4      if (is_inside(leaf, cp)) break;
5    }
6    if (are_neighbours_different_size(cp, cj)){
7      for (cps = cj->firstchild; cps != cj->sibling; cps = cps->sibling){
8        if (are_neighbours(cp, cps)){
9          if (cp->split && cps->split){
10           iact_pair_pc(cp, cps, leaf);
11         }
12       }else{
13         make_interact_pc(leaf, cps);
14       }
15     }
16   }else{
17     for (cps = cj->firstchild; cps != cj->sibling; cps = cps->sibling){
18       make_interact_pc(leaf, cps);
19     }
20   }
21 }
```

In lines 5-7 it finds the child (`cp`) of `ci` that contains the leaf cell. If `cp` and `cj` are neighbours, then it has to recurse (lines 8-14) as any neighbours of the leaf that are contained in `cj` will have direct interaction tasks associated with them, and it must not compute these interactions. The leaf then interacts with the monopoles of any children of `cj` that are not neighbours of `cp`. If `cp` and `cj` are not neighbours, then the leaf interacts with the 8 monopoles of the children of `cj` individually.

I modified `create_tasks` for the GPU as follows:

```
1  if( ci->count > 64*cell_maxparts){
2    for(cp = &cell_pool[ci->firstchild]; cp != &cell_pool[ci->sibling]; cp
       = &cell_pool[cp->sibling]){
3      data[0] = cp-cell_pool;
4      data[1] = cj-cell_pool;
5      tid = qsched_addtask(s, task_type_pair_pc_split, task_flag_none,
         data,
```

```
6                         sizeof(int) * 2, cp->count * cj->count);
7     qsched_addlock(s, tid, cp->res);
8   }
9   for(cp = &cell_pool[cj->firstchild]; cp != &cell_pool[cj->sibling]; cp
        = &cell_pool[cp->sibling]){
10    data[0] = cp-cell_pool;
11    data[1] = ci-cell_pool;
12    tid = qsched_addtask(s, task_type_pair_pc_split, task_flag_none,
          data,
13                         sizeof(int) * 2, cp->count * cj->count);
14    qsched_addlock(s, tid, cp->res);
15  }
16 }else{
17  data[0] = ci -cell_pool;
18  data[1] = cj - cell_pool;
19  tid = qsched_addtask(s, task_type_pair_pc, task_flag_none, data,
        sizeof(int) * 2, ci->count * cj->count);
20  qsched_addlock(s, tid, ci->res);
21  data[0] = cj - cell_pool;
22  data[1] = ci - cell_pool;
23  tid = qsched_addtask(s, task_type_pair_pc, task_flag_none, data,
        sizeof(int) * 2, ci->count * cj->count);
24  qsched_addlock(s, tid, cj->res);
25 }
```

The use of pointer arithmetic (such as `ci - cell_pool`) is necessary as the GPU implementation uses integer indexes to cells rather than pointers.

For MPI QuickSched I created a new function to create the particle-cell tasks:

```
1  void create_pcs(struct qsched *s, struct cell *ci, struct cell *cj, int
       depth, int leaf_depth){
2   qsched_task_t tid;
3   qsched_task_t data[2];
4   qsched_task_t cp, cps;
5   struct cell *cp1, *cp2;
6   if(cj == NULL){
7     for (cp = ci->firstchild; cp != ci->sibling; cp = cp1->sibling){
8       cp1 = (struct cell*) qsched_getresdata(s, cp);
9       if(cp1->split)
10        create_pcs(s, cp1, NULL, depth+1, leaf_depth);
```

```
11        for (cps = cp1->sibling; cps != ci->sibling; cps = cp2->sibling){
12          cp2 = (struct cell*) qsched_getresdata(s, cps);
13          if(cp1->split && cp2->split)
14            create_pcs(s, cp1, cp2, depth+1, leaf_depth);
15        }
16      }
17    }else{
18      if(depth < leaf_depth){
19        for (cp = ci->firstchild; cp != ci->sibling; cp = cp1->sibling){
20          cp1 = (struct cell*) qsched_getresdata(s, cp);
21          for (cps = cj->firstchild; cps != cj->sibling; cps =
                cp2->sibling){
22            cp2 = (struct cell*) qsched_getresdata(s, cps);
23            create_pcs(s, cp1, cp2, depth+1, leaf_depth);
24          }
25        }
26      }else{
27        if(are_neighbours(ci, cj)){
28          if(ci->split && cj->split){
29            for (cp = ci->firstchild; cp != ci->sibling; cp = cp1->sibling)
                  {
30              cp1 = (struct cell*) qsched_getresdata(s, cp);
31              for (cps = cj->firstchild; cps != cj->sibling; cps =
                    cp2->sibling){
32                cp2 = (struct cell*) qsched_getresdata(s, cps);
33                create_pcs(s, cp1, cp2, depth+1, leaf_depth);
34              }
35            }
36          }
37        }else{
38          data[0] = ci->res;
39          data[1] = cj->res;
40          tid = qsched_addtask(s, task_type_pair_pc, task_flag_none, data,
41                sizeof(qsched_task_t) * 2, ci->count * 8 );
42          qsched_addlock(s, tid, ci->res_parts);
43          qsched_adduse(s, tid, ci->res);
44          qsched_adduse(s, tid, cj->res);
45          data[0] = cj->res;
46          data[1] = ci->res;
47          tid = qsched_addtask(s, task_type_pair_pc, task_flag_none, data,
48                sizeof(qsched_task_t) * 2, cj->count * 8 );
```

```
49          qsched_addlock(s, tid, cj->res_parts);
50          qsched_adduse(s, tid, cj->res);
51          qsched_adduse(s, tid, ci->res);
52        }
53      }
54    }
55  }
```

The function is called by providing the root cell as `ci` and a `leaf_depth`. The `leaf_depth` parameter can be tuned to choose at what level in the tree tasks are created, but it must be less than or equal to the depth of all leafs in the tree. The function recurses over all possible pairs of cells until it reaches `leaf_depth`. Once it reaches this depth, for every pair of non-neighbouring cells it creates a pair of particle cell tasks. For neighbouring cells, it keeps recursing until it reaches the leaves.

Bibliography

[1] Top500 list - june 2016.

[2] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. Task-based fmm for multicore architectures. *SIAM Journal on Scientific Computing*, 36(1):C66–C93, 2014.

[3] M. P. Allen and D. J. Tildesley. *Computer simulation of liquids*. Oxford university press, 1989.

[4] C. Augonnet and R. Namyst. A unified runtime system for heterogeneous multicore architectures. In *Proceedings of the International Euro-Par Workshops 2008, HPPC'08*, volume 5415 of *LNCS*, 2008.

[5] J. Barnes and P. Hut. A hierarchical o (n log n) force-calculation algorithm. 1986.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995.).

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[8] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.

[9] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[10] S. Chatterjee, M. Grossman, A. Sbîrlea, and V. Sarkar. Dynamic task parallelism with a gpu work-stealing runtime system. In *Languages and Compilers for Parallel Computing*, pages 203–217. Springer, 2011.

[11] T. Darden, D. York, and L. Pedersen. Particle mesh Ewald: An N log (N) method for Ewald sums in large systems. *The Journal of Chemical Physics*, 98(12):10089–10092, 1993.

[12] J. Demouth and NVIDIA. Shuffle: Tips and tricks. GTC 2013, 2013.

[13] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[14] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, and L. G. Pedersen. A smooth particle mesh Ewald method. *The Journal of Chemical Physics*, 103(19):8577–8593, 1995.

[15] P. L. Freddolino, A. S. Arkhipov, S. B. Larson, A. McPherson, and K. Schulten. Molecular dynamics simulations of the complete Satellite Tobacco Mosaic Virus. *Structure*, 14(3):437–449, 2006.

[16] T. Gautier, X. Besseron, and L. Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, pages 15–23. ACM, 2007.

[17] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181(3):375–389, 1977.

[18] P. Gonnet. A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. *Journal of Computational Chemistry*, 28(2):570–573, 2007.

[19] P. Gonnet. Pseudo-Verlet lists: a new, compact neighbour list representation. *Molecular Simulation*, 39(9):721–727, 2013.

[20] P. Gonnet. Quicksched: Task-based parallelism with dependencies and conflicts. Technical report, Technical Report ECS-TR 2013/06, School of Engineering and Computing Sciences, Durham University, South Road, DH1 3LE Durham, United Kingdom, 2013.

[21] P. Gonnet. Efficient and scalable algorithms for smoothed particle hydrodynamics on hybrid shared/distributed-memory architectures. *SIAM Journal on Scientific Computing*, 37(1):C95–C121, 2015.

[22] P. Gonnet, A. B. Chalk, and M. Schaller. Quicksched: Task-based parallelism with dependencies and conflicts. *arXiv preprint arXiv:1601.05384*, 2016.

[23] P. Gonnet and A. B. G. Chalk. `mdcore`, http://mdcore.sourceforge.net, 2013.

[24] P. Gonnet, M. Schaller, T. Theuns, and A. B. Chalk. Swift: Fast algorithms for multi-resolution SPH on multi-core architectures. *arXiv preprint arXiv:1309.3783*, 2013.

[25] F. G. Gustavson. *New Generalized Matrix Data Structures Lead to a Variety of High-Performance Algorithms*, pages 211–234. Springer US, Boston, MA, 2001.

[26] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961.

[27] R. W. Hockney and J. W. Eastwood. *Computer simulation using particles*. CRC Press, 1988.

[28] A. S. Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM*, 5(4):339–342, 1958.

[29] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[30] V. Loup. Computer experiments on classical fluids. *Physical Review*, 159(1):98–103, 1967.

[31] H. Ltaief and R. Yokota. Data-driven execution of fast multipole methods. *Concurrency and Computation: Practice and Experience*, 26(11):1935–1946, 2014.

[32] NVIDIA Corporation. Dynamic parallelism in CUDA.

[33] NVIDIA Corporation. OpenACC toolkit.

[34] NVIDIA Corporation. CUDA C programming guide, 2012.

[35] NVIDIA Corporation, Santa Clara, CA 95050, USA. *NVIDIA CUDA C Programming Guide 4.2*, 2012.

[36] NVIDIA Corporation. Kepler GK110 whitepaper. 2013.

[37] OpenMP ARB. OpenMP 4.0 specification, 2013.

[38] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. Fast in-place, comparison-based sorting with CUDA: a study with bitonic sort. *Concurrency and Computation: Practice and Experience*, 23(7):681–693, 2011.

[39] J. Reinders. *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2010.

[40] M. Schaller, P. Gonnet, A. B. 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*, page 2. ACM, 2016.

[41] L. I. Sedov. Propagation of strong shock waves. *Journal of Applied Mathematics and Mechanics*, 10:241–250, 1946.

[42] R. D. Skeel, I. Tezcan, and D. J. Hardy. Multiple grid methods for classical molecular dynamics. *Journal of Computational Chemistry*, 23(6):673–684, 2002.

[43] SMP Superscalar. Users manual, July 2007.

[44] M. Snir. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[45] V. Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

[46] V. Springel. Smoothed particle hydrodynamics in astrophysics. *Annual Review of Astronomy and Astrophysics*, 48:391–430, 2010.

[47] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the GPU. *ACM Transactions on Graphics*, 33(6):228–239, 2014.

[48] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[49] I. Todorov, W. Smith, and U. Cheshire. The dl poly 4 user manual. *STFC, STFC Daresbury Laboratory, Daresbury, Warrington, Cheshire, WA4 4AD, United Kingdom, version*, 4(0), 2011.

[50] V. Volkov. Better performance at lower occupancy. 2010.

[51] Z. Yao, J.-S. Wang, G.-R. Liu, and M. Cheng. Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. *Computer Physics Communications*, 161(1):27–35, 2004.

[52] A. Yarkhan, J. Kurzak, and J. Dongarra. QUARK users guide. Technical report, Technical Report April, Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tenessee, 2011.