5-2013

# A study of possible optimizations for the task scheduler 'QUARK' on the shared memory architecture

Vijay Gopal Joshi
vjoshi1@utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Vijay Gopal Joshi entitled "A study of possible optimizations for the task scheduler 'QUARK' on the shared memory architecture." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Jack J. Dongarra, Major Professor

We have read this thesis and recommend its acceptance:

Lynne E. Parker, Stanimire Tomov

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

# A study of possible optimizations for the task scheduler 'QUARK' on the shared memory architecture

A Thesis Presented for

The Master of Science

Degree

The University of Tennessee, Knoxville

Vijay Gopal Joshi

May 2013

*To my beloved family*

# Acknowledgements

I would like to thank professor Dr. Jack Dongarra for providing me the excellent opportunity to work in the Innovative Computing Lab (ICL), University of Tennessee, Knoxville as a graduate research assistant. I am thankful to my mentors Jakub Kurzak and Asim YarKhan for all their support and guidance. This work would not have been possible without the encouragement, freedom to try different ideas at ICL, and all the brainstorming sessions I had with my mentors. I am also thankful to Dr. Lynne Parker and Dr. Stan Tomov for being on my committee and providing me with feedback.

*"Stay hungry. Stay foolish." Steve Jobs, Stanford University, 2005*

# Abstract

Multicore processors are replacing most of the single core processors nowadays. Current trends show that there will be increasing numbers of cores on a single chip in the coming future. However, programming multicore processors remains bug prone and less productive. Thus, making use of a runtime to schedule tasks on multicore processor hides most of the complexities of parallel programming to improve productivity. QUARK is one of the runtimes available for the multicore processors. This work looks at identifying and solving performance bottlenecks for QUARK on the shared memory architecture. The problem of finding bottlenecks is divided into two parts, low level details and high level details. Low level details deal with issues like length of the critical section and locking mechanisms. High level details involve use of a suitable scheduling algorithm and better load balancing. We discuss the possible solutions of the bottlenecks and its impact on the overall performance.

# Contents

**Chapter 1 Introduction And Motivation** . . . . . . . . . . . . . . . . . . **1**

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**Chapter 2 Background** . . . . . . . . . . . . . . . . . . . . . . . . . **3**

2.1 Dynamic Task Schedulers . . . . . . . . . . . . . . . . . . . . 7

2.1.1 SMPSs . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.1.2 StarPU . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.1.3 QUARK . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.2 Task Graph . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.3 Data Structures . . . . . . . . . . . . . . . . . . . . . . . . . 11

**Chapter 3 Low Level Optimizations** . . . . . . . . . . . . . . . . . . . **16**

3.1 Unlocking New Tasks . . . . . . . . . . . . . . . . . . . . . . 16

3.2 Parallel DAG Update And Spinlocks . . . . . . . . . . . . . . . 17

3.3 EZTrace Tracing . . . . . . . . . . . . . . . . . . . . . . . . 20

3.4 Master Lock Contentions . . . . . . . . . . . . . . . . . . . . 21

**Chapter 4 Scheduling Optimizations** . . . . . . . . . . . . . . . . . . . **28**

4.1 Scheduling Algorithms . . . . . . . . . . . . . . . . . . . . . . 28

4.1.1 Greedy Heuristic . . . . . . . . . . . . . . . . . . . . . 30

4.1.2 Experimental Results . . . . . . . . . . . . . . . . . . . 32

**Chapter 5 Conclusion and Future Work** . . . . . . . . . . . . . . . . . **38**

5.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

5.2 Future Work . . . . . . . . . . 39

**Bibliography** . . . . . . . . . . **40**

**Vita** . . . . . . . . . . **43**

# Chapter 1

# Introduction And Motivation

## 1.1 Motivation

Implementing parallel or multi-threaded programs from scratch using basic threading APIs such as Pthreads can be time consuming and error prone. Issues like load balancing and synchronization are critical from performance point of view. These issues are often a distraction for the programmer, compared to spending time on the main algorithm. Thus it is better to make use of a layer of abstraction. The motivation for a task scheduler is that it provides programmers with a layer of abstraction over these details and allows them to concentrate on structuring programs to expose parallelism and exploit locality. It is the scheduler's responsibility to execute programs in an efficient way on the given architecture. Hence, it is critical to ensure that the scheduler has the lowest overhead and maximum performance possible. QUARK [1]is one such scheduler developed at Innovative Computing Lab, University of Tennessee, Knoxville. QUARK is a dynamic runtime optimized for linear algebra libraries which have high performance requirements. In this work we look at exploring different optimization opportunities in order for QUARK to have minimum overhead and maximum performance possible. Removing bottlenecks and improving performance further will make QUARK even more attractive option for application developers.

## 1.2 Introduction

Multicore architectures require parallel programs in order to exploit maximum performance. There are different programming abstraction options available for different kinds of parallel constructs like for the fork-join parallel construct we have OpenMP, for recursive parallelism Cilk[2] is more suitable. Other options are Intel's Thread Building Blocks[3], Microsoft's Parallel Pattern Library etc. QUARK is more suitable for data driven task execution. In the background section we describe more about how QUARK works.

As part of this work, we explore optimization options for QUARK scheduler. The goal of the QUARK is to schedule tasks on the multicore CPU with shared memory architecture in the most efficient way. We use the dense linear algebra workload 'tiled QR algorithm' for performance analysis[4]. We also compare QUARK performance to other similar schedulers. We approach the task of optimization in two parts; first is the low level details and second is high level details. Low level details involve optimizations related to use of data structures, locking mechanisms, length of the critical section etc. The important outcome of this work is improving performance of the QUARK more than 87% for certain cases. This work involves making use of a lightweight tracing library EZTrace [5] for QUARK to make the profiling process fast and productive. In the following sections we explain how lightweight tracing has helped us to zero on the performance bottleneck part of the QUARK implementation. The second part involves optimizations related to algorithmic decisions made while scheduling tasks. Scheduling involves making decisions about which task should be executed at a certain time in order to have the minimum finish time and maximum performance.

# Chapter 2

# Background

We started this work with the goal of overall improvement in the performance of QUARK. QUARK is a dynamic task scheduler that extracts dependencies amongst tasks and makes the scheduling decisions about which task should run on which core, at runtime. In contrast, some linear algebra libraries such as PLASMA[6] make manual scheduling decisions known as static scheduling. However these manual schedules for various algorithms are difficult to generate, and can limit the ability of PLASMA to implement new and innovative algorithms. As our first step to improve performance, we compare the performance of QUARK with statically scheduled PLASMA routines as well as with some similar schedulers developed by other research groups. The motivation behind this comparison is to get an idea about which scheduler is better and the reasons behind the same. Through this understading we improve QUARK implementation to get more performance. Such scheduler comparison work has been done before by Kurzak et. all using static routines, Cilk runtime, and SMPSs scheduler[7]. This work concludes that for linear algebra routines, SMPSs performs better than Cilk scheduler. However, SMPSs does not performs better than static routines. Thus, for our comparison work we select PLASMA static routines, SMPSs scheduler and StarPU scheduler. We describe the SMPSS and StarPU schedulers in more detail in Section 2.1, but before we explain

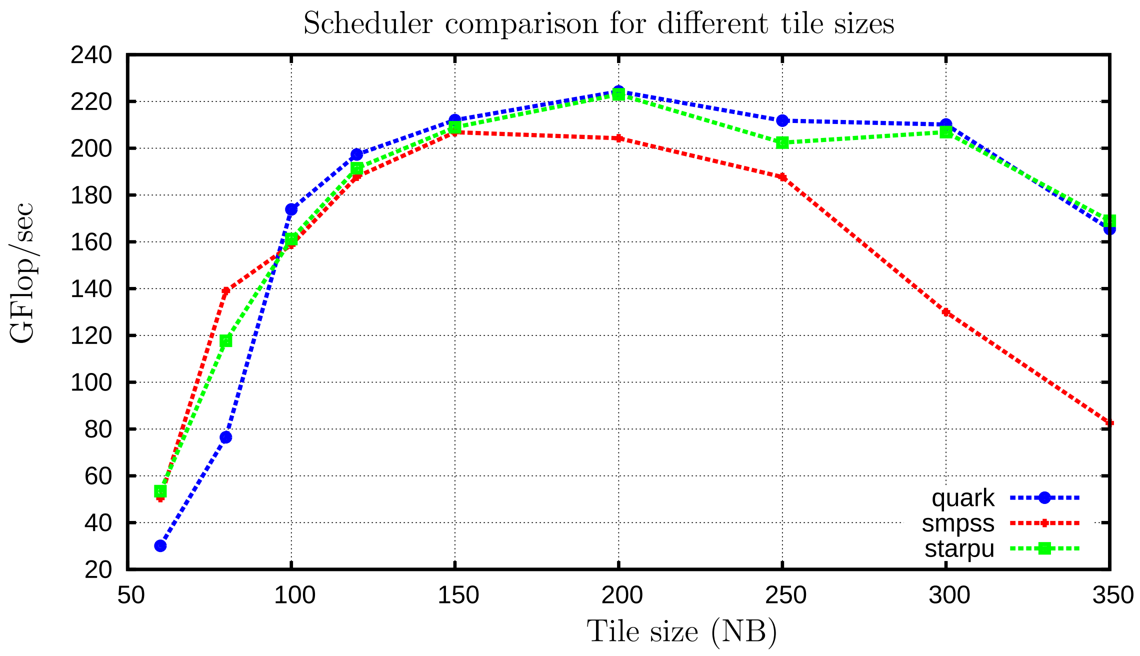

**Figure 2.1:** Scheduler comparison for different tile sizes using Tiled QR factorization for a fixed problem of size N=8000, double precision on a 2.4 GHz 12-socket quad-core (48 cores total) AMD Opteron(tm) processor. A tile size corresponds to the length of the average task. Hence this figure shows effect of the scheduling overhead on the overall performance compared to the average task length.

how these schedulers work, we discuss the concept of the overhead associated with these schedulers and its effect on the performance.

Figure 2.1 shows the performance of each tested scheduler on the 'tiled QR factorization' [4] workload using different tile sizes. A given tile size is directly proportional to how long a unit task will run on an average. Thus the Figure 2.1 shows effect of change in unit task length on scheduler performance. It can be seen that for smaller tile sizes, SMPSs is performing better than both StarPU and QUARK. This behaviour can be explained in following terms. A dynamic task scheduler does multiple things in order to execute a task at runtime. It needs to extract and maintain information about different task dependencies for logical correctness. This extra work

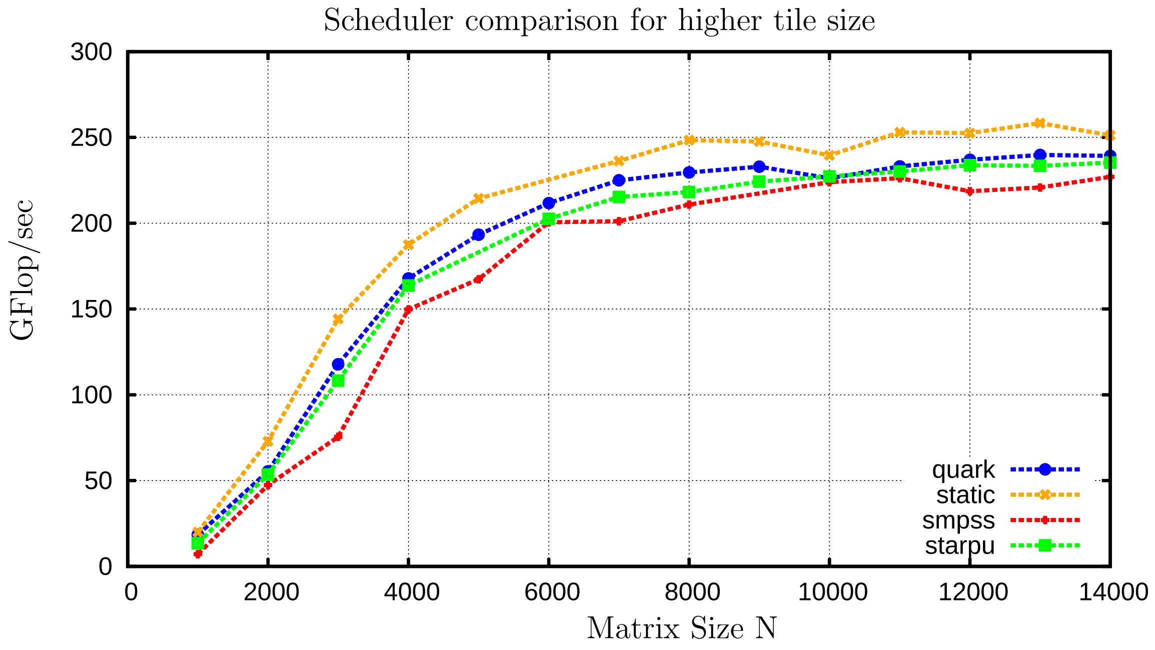

**Figure 2.2:** Scheduler comparison for tile size 200 using Tiled QR factorization, double precision on a 2.4 GHz 12-socket quad-core (48 cores total) AMD Opteron(tm) processor. A tile size corresponds to the length of the average task. Hence this figure shows that for tile size around 200 all scheduler are having similar performance.

is the overhead involved in scheduling tasks. This overhead can be ignored if task length is long relative to the time required for the overhead processing. As we decrease the tile size that is average task length, effect of overhead becomes significant. In Figure 2.1, for tile size below 200 performance for all three schedulers is dropping. As shown in Figure 2.2 it can be seen that QUARK performance is better than SMPSs and StarPU for tile size 200. But at smaller tile size, the performance of QUARK decreases very fast, so we focus our attention on what is happening at tile size 80. Figure 2.3 shows QUARK performance is lower than both SMPSs and StarPU. Thus, as part of this work we try to find what are the performance bottlenecks for QUARK causing drop in the performance for lower tile size like tile size 80. We discuss how these dynamic task schedulers work in general and give a brief description for each

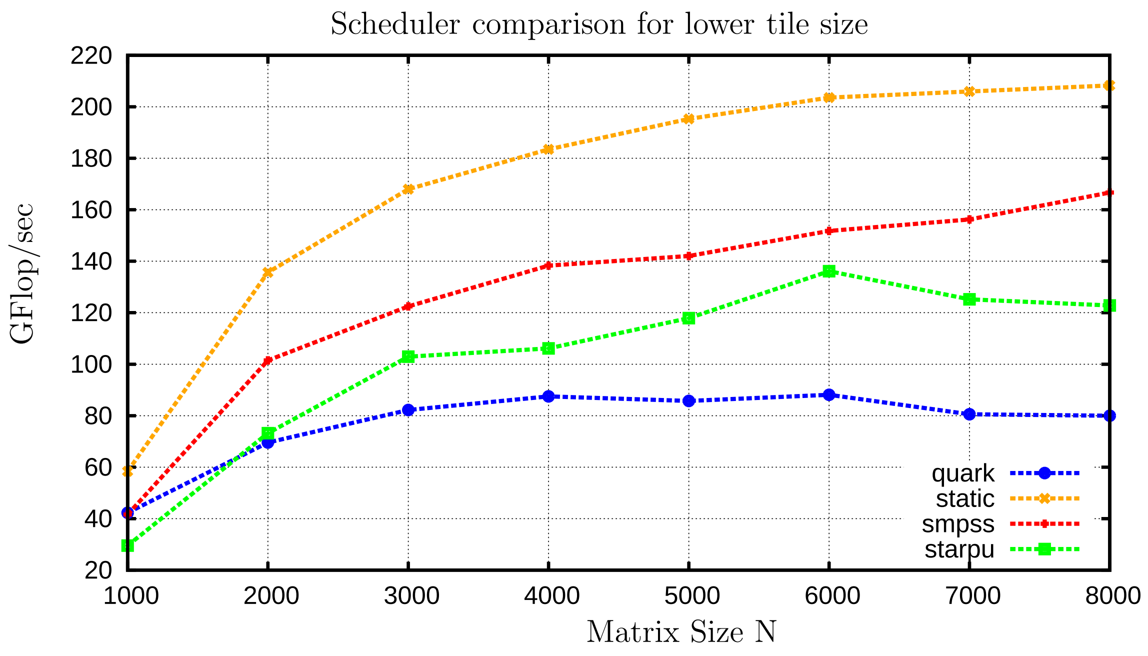

**Figure 2.3:** Scheduler comparison for tile size 80 using Tiled QR factorization, double precision on a 2.4 GHz 12-socket quad-core (48 cores total) AMD Opteron(tm) processor. A tile size corresponds to the length of the average task. Hence this figure shows that for tile size around 80 all scheduler are having different performance indicating scope for performance improvement for QUARK.

scheduler. After that we discuss in further detail about QUARK and in subsequent sections we discuss about performance improvements.

## 2.1 Dynamic Task Schedulers

A dynamic task scheduler makes scheduling decisions at the runtime. Compared to static scheduling, dynamic task schedulers have advantage that they can do better load balancing based on the runtime conditions. Any dynamic task scheduler requires user to provide certain information for each task such as arguments for each task, the size of the memory used by the arguments and the direction of the data movement like read or write for the given memory region. Some schedulers have an additional source to source compiler phase to make it easy for users to specify task details using some compact notations such as '# pragma' in C programming language. Some schedulers ask user to provide all task details in verbose fashion to avoid extra compilation phase. A library is provided which collects the information for all tasks and makes scheduling decisions at runtime. In following subsections we give a brief description for each scheduler being used in this work.

### 2.1.1 SMPSs

SMP Superscalar(SMPSs)[8] is a dynamic scheduler developed at the Barcelona Supercomputer Center (Centro Nacional de Supercomputacion). SMPSs supports CPU only scheduling and does not provide support for any accelerator or GPUs. Hence it provides a good comparison option for QUARK, as QUARK also supports CPU only scheduling. As described above for SMPSs, programmers need to specify size of the memory being used by each argument of the task and also direction of the data movement (input, output, inout). SMPSs provides compiler support to collect these details so programmer can use simple pragma annotations to provide these details making code changes less verbose.

### 2.1.2 StarPU

StarPU[9] dynamic scheduler is being developed at INRIA Bordeaux, LaBRI, Universit de Bordeaux, France. StarPU is developed for hybrid environment i.e. CPU plus accelerators such as GPU. However in this study we are only looking at CPU scheduling. StarPU, similar to SMPSs, requires user to provide certain information for each task such as all the arguments, the size of the memory used by the arguments and the direction of the data movement for the given memory region. However there is no compiler support for StarPU so this information becomes verbose from programmer point of view.

StarPU provides many scheduling strategies i.e. the different criteria using which a core should select a task for execution. A scheduling strategy named 'prio' uses a central task queue, but sorts tasks by priority. The ws (work stealing) strategy schedules tasks on the local worker by default. When a worker becomes idle, it steals a task from the most loaded worker. Most of the scheduling strategies provided are useful in hybrid (CPU + GPU) architecture environment. In this work we have used a scheduling strategy called 'eager' which uses a central task queue, from which workers draw tasks to work on. This strategy provides good load balancing.

### 2.1.3 QUARK

QUARK is a dynamic runtime being developed by Innovative computing lab at University of Tennessee, Knoxville. QUARK is similar to both SMPSs and StarPU as it also extracts and build dependency information at runtime. It also makes scheduling decisions at runtime so it involves certain overhead to schedule tasks. QUARK is a CPU only scheduler and there is no extra compiler support to specify task argument details. In the next section we specify how QUARK extracts and build dependency information for tasks.

## 2.2 Task Graph

Figure 2.4 shows a serial toy program which performes some simple mathematical operations. In Figure 2.5, each line of the program in Figure 2.4 is expressed using the QUARK API so that QUARK will understand input and output arguements of each task to be executed. The function QUARK_Insert_Task adds a task to be executed to a given QUARK runtime instance. Let us look at one example line 'ADD(D2, D3, D4)' from the toy program, where D2, D3, D4 are the data memory location used by function ADD. Following function,

```
ADD(D2, D3, D4)
```

is converted to

```
QUARK_Insert_Task(quark, QUARK_ADD,
                    address_of(D3), size_of(D3), INPUT,
                    address_of(D2), size_of(D2), INPUT,
                    address_of(D4), size_of(D4), OUTPUT).
```

The first argument to QUARK_Insert_Task is the QUARK instance to which tasks are to be added. The second argument is the function to be called in order to execute a given task. For each data item in original ADD function QUARK needs address of the memory, size of the memory that would be accessed from the specified address location, and direction of the data i.e. OUTPUT, INPUT, and INOUT. For all the lines in our toy program the corresponding QUARK code will looks similar to that shown in Figure 2.5.

QUARK works similar to a superscalar processor trying for the maxium instruction level parallelism. Each memory location being handled by the QUARK could be related to a given register in a superscalar processor. QUARK will try to maintain logical correctness similar to superscalar processor but try to change the execution sequence to get maximum performance. QUARK gets details about

| | | | |
|---|---|---|---|
| D1 | D2 | D3 | D4 |

| | | |
|---|---|---|
| SQRT(D1) | Read D1 | D1 = $\sqrt{D1}$ |
| MULT(D1, D2) | Read D1, D2 | D2 = D1 * D2 |
| MULT(D1, D3) | Read D1, D3 | D3 = D1 * D3 |
| ADD(D2, D3, D4) | Read D2, D3 | D4 = D2 + D3 |

**Figure 2.4:** A serial toy program

| | | |
|---|---|---|
| Task 1(T1) | QUARK_Insert_Task | (quark, QUARK_SQRT,<br>address_of(D1), size_of(D1), INOUT) |
| Task 2(T2) | QUARK_Insert_Task | (quark, QUARK_MULT,<br>address_of(D1), size_of(D1), INPUT,<br>address_of(D2), size_of(D2), INOUT) |
| Taks 3(T3) | QUARK_Insert_Task | (quark, QUARK_MULT,<br>address_of(D1), size_of(D1), INPUT,<br>address_of(D3), size_of(D3), INOUT) |
| Task 4(T4) | QUARK_Insert_Task | (quark, QUARK_ADD,<br>address_of(D3), size_of(D3), INPUT,<br>address_of(D2), size_of(D2), INPUT,<br>address_of(D4), size_of(D4), OUT) |

**Figure 2.5:** A serial toy program to QUARK tasks

the memory addresses and their size and the direction of the data movement for the memory addresses as well as the sequence of tasks insertion. Using these details QUARK builds a Directed Acyclic Graph (DAG) which helps to maintain and update information regarding different data hazards such as Read After Write(RAW), Write After Read (WAR). The following description illustrates how the DAG is built for the toy program. First task T1 is reading and writing to memory location D1. There are no other tasks inserted before T1 hence T1 becomes the first node of the DAG as shown in Figure 2.6. Task T2 reads data from memory location D1, however D1 would be written by T1 hence T2 has to wait till T1 is over. Therefore, there is an edge from T1 to T2 and T2 becomes child of T1. Similarly Task T3 reads data from memory location D1, thus there is an edge from T1 to T3. However QUARK finds that T2 and T3 are not writing to any common memory location, hence T2 and T3 can run in parallel. Task T4 reads memory location D2 and D3 written by task T2 and T3 respectively. Hence there are two edges into task T4, one from T2 and other from T3. In the next section we explain how QUARK represents the DAG information using different data structures.

## 2.3 Data Structures

Figure 2.7 shows a snapshot of the QUARK data structures after three tasks have been inserted into system. Each task inserted into QUARK will be executed later as a function call. In Figure 2.7, we have three function calls. For each function call we have a list of parameters. If a function parameter is a memory location passed by pointer, QUARK will identify that as a dependency. Hence for function call Task_1(geqrt, A00, T00) two dependencies named as $t\_1\_A00^{RW}$ and $t\_1\_T00^{W}$ are created. Note that QUARK requires users to provide INOUT(RW), OUTPUT(W), and INPUT(R) data movement direction for each memory argument. QUARK also allocates a task data structure, Task_1, to refer to this list of dependencies. Each dependency refers to a memory location and a task. In this example, dependency

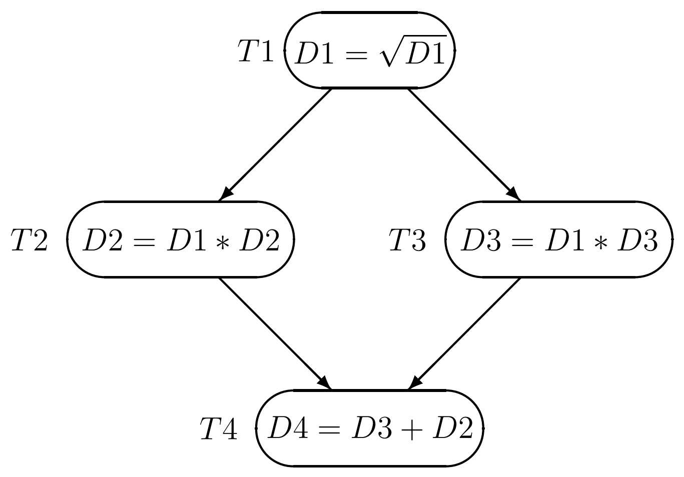

**Figure 2.6:** DAG for the toy program

Task_1(geqrt, A00, T00)
Task_2(unmqr, A00, T00, A01)
Task_3(unmqr, A00, T00, A02)

| Dependency list | | | | |
|---|---|---|---|---|
| | Task_1 | $t\_1\_A00^{RW}$ | $t\_1\_T00^{W}$ | |
| | Task_2 | $t\_2\_A00^{R}$ | $t\_2\_T00^{R}$ | $t\_2\_A01^{RW}$ |
| | Task_3 | $t\_3\_A00^{R}$ | $t\_3\_T00^{R}$ | $t\_2\_A02^{RW}$ |

| Task set hash |
|---|
| Task_1 |
| Task_2 |
| Task_3 |

| address set hash | Dependency waiting | | |
|---|---|---|---|
| A00 | $t\_1\_A00^{RW}$ | $t\_2\_A00^{R}$ | $t\_3\_A00^{R}$ |
| T00 | $t\_1\_T00^{W}$ | $t\_2\_T00^{R}$ | $t\_3\_T00^{R}$ |
| A01 | $t\_2\_A01^{RW}$ | | |
| A02 | $t\_3\_A02^{RW}$ | | |

**Figure 2.7:** QUARK Bookkeeping showing Task_1 ready to go and Task_2 and Task_3 are blocked waiting for access to various data items.

$t\_1\_A00^{RW}$ refers Task_1 and memory location A00. QUARK maintains a list of dependencies for each memory location. QUARK keeps track of which all tasks are waiting for a given memory location in address_set_hash data structure. In this case, memory location A00 has three tasks waiting i.e. Task_1 ($t\_1\_A00^{RW}$), Task_2 ($t\_2\_A00^{R}$), Task_3 ($t\_3\_A00^{R}$). For a memory location, the first dependency is always marked as ready. This indicates there are no other tasks using the memory location apart from the first dependency task. Thus we see, dependency $t\_1\_A00^{RW}$ is marked as ready (highlighed with green color). Any given task is ready to run if all its dependencies are ready. Task_1 has all the dependencies ready (highlighted with green color) hence QUARK will execute Task_1 using one of the worker threads. Once a task is executed, we remove the dependencies for that task from the bookkeeping data structures. This results in unlocking the next tasks which can be executed. Figure 2.8 shows the snapshot of data structures after Task_1 dependencies are removed. Note

Task_2(unmqr, A00, T00, A01)
Task_3(unmqr, A00, T00, A02)

| Dependency list | | | | |
|---|---|---|---|---|
| | Task_2 | $t\_2\_A00^{R}$ | $t\_2\_T00^{R}$ | $t\_2\_A01^{RW}$ |
| | Task_3 | $t\_3\_A00^{R}$ | $t\_3\_T00^{R}$ | $t\_2\_A02^{RW}$ |

| Task set hash |
|---|
| Task_2 |
| Task_3 |

| address set hash | Dependency waiting | |
|---|---|---|
| A00 | $t\_2\_A00^{R}$ | $t\_3\_A00^{R}$ |
| T00 | $t\_2\_T00^{R}$ | $t\_3\_T00^{R}$ |
| A01 | $t\_2\_A01^{RW}$ | |
| A02 | $t\_3\_A02^{RW}$ | |

Process_finish_tasks    Task_2    Task_3

**Figure 2.8:** QUARK Bookkeeping showing Task_2 and Task_3 are over but not yet processed as finished tasks and waiting in ‘process finished task’ queue.

that Task_2 and Task_3 are marked ready to run (highlighted with green color). When Task_2 and Task_3 execution is completed they would be put into a 'process finished tasks queue' for the DAG update step (refer figure 2.8). A single thread would acquire a lock on this queue and unlock next tasks. We call this implementation a serial DAG update. In the next chapter we discuss we show how serial DAG updates can cause performance issue.

# Chapter 3

# Low Level Optimizations

In the background chapter we explained performance problem for QUARK for lower tile sizes as shown by performance plot in Figure 2.3. Following sections explain different optimizations helping to improve performance of QUARK.

## 3.1 Unlocking New Tasks

The Figure 2.3 showed QUARK performance is below StarPU and SMPSs for smaller tile size. We have described how QUARK maintains a 'process finished tasks queue' to keep track of finished tasks. Whenever a task is finished it is added to this global queue. One thread periodically goes through all the latest finished tasks and decides which all new tasks are ready to run. Instead of a single thread checking finished tasks, we made each thread to check this queue after finishing the task. This change ensured all the DAG updates are processed as soon as possible and new tasks are unlocked without any delay. As shown in Figure 3.1, after this change we observed that QUARK is giving more performance than both SMPSs and StarPU for lower tile size and lower matrix size. However this performance was dropping for bigger matrix sizes. One possible explanation for this issue was many threads trying to acquire lock on the 'process finished tasks queue'. Consider the scenario in which 5 tasks have finished executing a task around the same time. All of them will compete

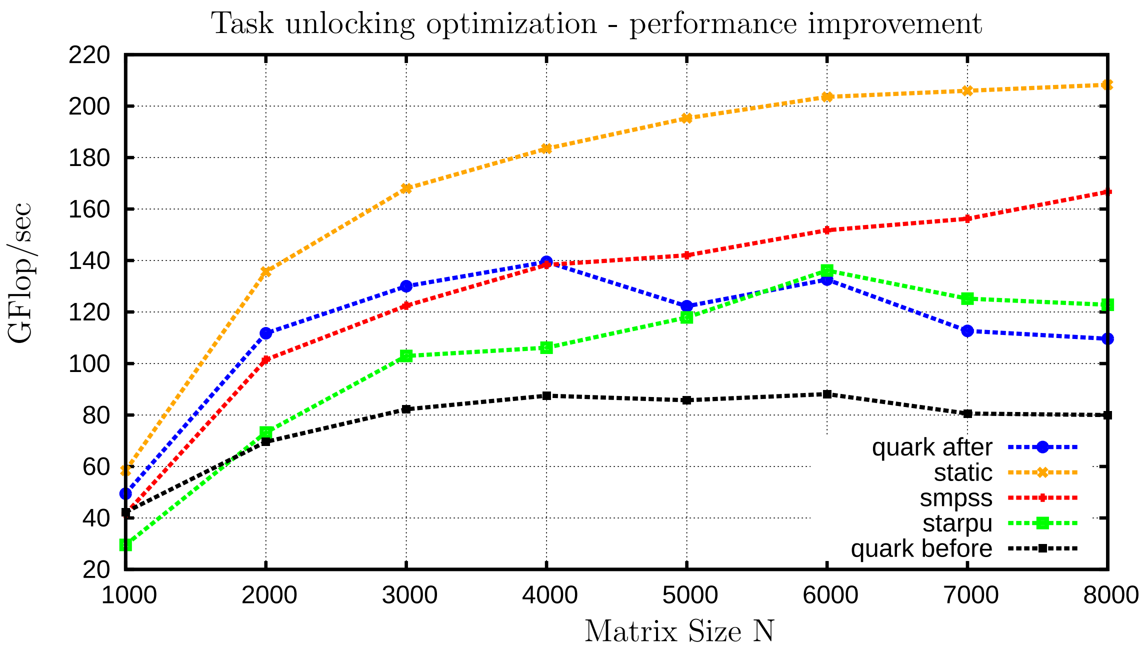

**Figure 3.1:** Performance comparison for original QUARK (quark before) and improved QUARK (quark after). Improved QUARK result shows that default QUARK performance could be improved if DAG is processed more frequently. Tiled QR factorization, Tile size NB = 80, inner block size IB = 32, double precision on a 2.4 GHz 12-socket quad-core (48 cores total) AMD Opteron(tm) processor.

against each other to acquire lock on the 'process finished task queue'. This problem of threads competing with each other for acquiring a common resource is called as lock contention. However, there is no such global queue present in case of SMPSs which reduces the lock contentions. The next section discusses efforts involved in removing this global queue and observing its impact on the overall performance.

## 3.2 Parallel DAG Update And Spinlocks

We have explained how QUARK would put each finished task in the queue for further processing as in figure 2.8. Even if we processed items in this queue as soon as they were inserted, it was still an avoidable bottleneck. Instead of putting tasks in a queue, we modified the logic of the QUARK runtime implementation to process the finished task as soon as its execution was over. Doing this means that we do not have to acquire lock on common big data structure to update the task status and unlock new tasks. This can be seen as in a big DAG, two different regions can be updated in parallel. This was not the case with original QUARK. We call this improved implementation 'parallel DAG'. However, we still had other common locks for which there were contentions. For example, we had a lock to keep track of total number of tasks in the system. We were using mutex lock and we did not get significant performance improvement just after using parallel DAG. The problem with mutex lock is if there is a lock contention, threads can be put into sleep state. Moving a thread into sleep state and waking it up is a heavy operation. This overhead is worthwhile when the length of the critical section is long enough to ignore this state change overhead. After using parallel DAG improvement, the critical section of the code was reduced to very small operations such as incrementing a common counter of total number of tasks in the system. In this case, using spinlocks would be a better choice. Spinlocks are different from the mutex locks from the implementation point of view. Spinlock keeps checking if a lock can be acquired in a busy loop. For a mutex locks if it can not acquire a lock, language specific scheduler puts it into the sleep state. Whenever lock becomes available, language specific scheduler wakes up the sleeping thread and it can acquire the lock. As spinlocks are never put to sleep state and they keep spinning and take less time to acquire the lock relative to mutex locks. Thus, if critical section of the code is small, spinlocks are faster compared to mutex locks. Hence after implementing the code changes for spinlocks, we got the performance improvement as shown in Figure 3.2. If we compare QUARK performance in Figure 3.2 to Figure 3.1, we see that performance improves till matrix size 9000. So use of parallel DAG and spinlock as per expectation helped to improve the performance. However, performance starts to drop after matrix size 9000. We discuss the analysis of this problem in the next section.

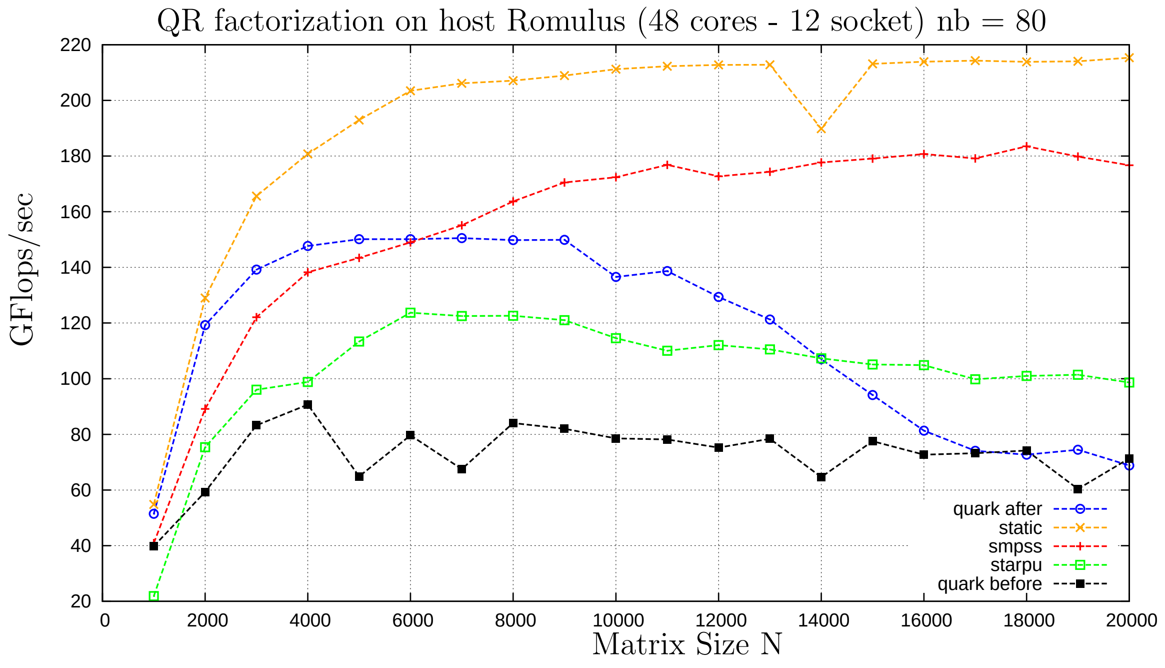

**Figure 3.2:** Performance improvement for 'Parallel DAG' with spinlock for tiled QR factorization, tile size NB = 80, inner block size IB = 32, double precision on a 2.4 GHz 12-socket quad-core (48 cores total) AMD Opteron(tm) processor.

## 3.3 EZTrace Tracing

After using spinlock and parallel DAG optimizations there was no obvious optimization option for performance improvement. QUARK has one master thread to insert the user tasks and many worker threads to execute the tasks inserted by master thread. To keep logical correctness at different parts of the code, synchromization mechanisms are used. Thus whenever multiple worker threads are waiting for the same resource, generally known as lock contentation, they don't do any useful work and remain idle. Idle worker threads cause performance drop as system is not used completely. Thus lock contention could be one generic answer for performance drop after matrix size 9000 in Figure 3.2. Problem with optimization to reduce lock contention is some locks take very short time and some don't. We decided we would profile the program to get more details. However, there were many tools to profile the code. Some were emulator based such as valgrind profiler. However many of these tools profile entire code thus they have a big overhead and end up giving distorted picture of performance impact of the part of code. QUARK is a parallel program and highly performance sensitive. Thus we needed a lightweight tracing tool which should have little or no impact on overall performance. Hence, we used EZTrace [5] profiler so that we can trace any specific part of the code as shown in Figure 3.3. This manual approach of profiling has two advantages. First, it helps to reduce the noise in terms of the profiler output. Second, it does not impact overall performance as it is really light weight compared to other profilers.

We started with tracing QUARK functions to see where it was spending all the time. (Refer to Figure 3.4.) Black spaces means we do not know the details about what thread is doing at that time or the thread is doing nothing. We try to fill that black space by adding more trace information with corresponding part of the code to know where threads are spending their time when they are not doing any useful work. This helps to zoom down to the exact part of the code which is causing performance

```
EZTRACE_EVENT0(FUT_QUARK(LOCK_ADDRESS_MUTEX));
/* Lock the address_set_node and manipulate it */
if ( pthread_mutex_lock_wrap( &address_set_node->asn_mutex ) == 0 )
    {
    EZTRACE_EVENT0(FUT_QUARK(STOP));
    ...
}
```

**Figure 3.3:** EZTrace for manually tracing QUARK

bottlenecks. In our next section we explain how tracing helps to find out performance bottlenecks.

## 3.4 Master Lock Contentions

As explained in the previous section, we profiled QUARK routines and started focusing on part of the traces affecting overall performance. In Figure 3.4 we see the master thread (at top of the Figure 3.4 cpu trace line with yellow color tiles) is inserting tasks. Each small yellow tile represents time required to insert the task. Each green tile in the following rows show time required to execute a unit task by the worker. If we count the number tasks inserted into the system while a unit task is running i.e. for the length of a green tile how many yellow tiles are present, the count is approximately 26. In the given Figure 3.4 there are 47 worker threads. Thus while inserting 26 tasks, first worker is not idle. However, consider the scenario when 27 th task is being inserted by master thread. For the 27th task, first worker thread has completed executing first task and we still have not assigned any tasks to worker number 27 to 47 i.e. 21 worker threads plus first worker thread i.e. 22 worker threads are waiting for a task to execute and only one task is available. This shows master is not pumping enough tasks into the system. Thus worker threads

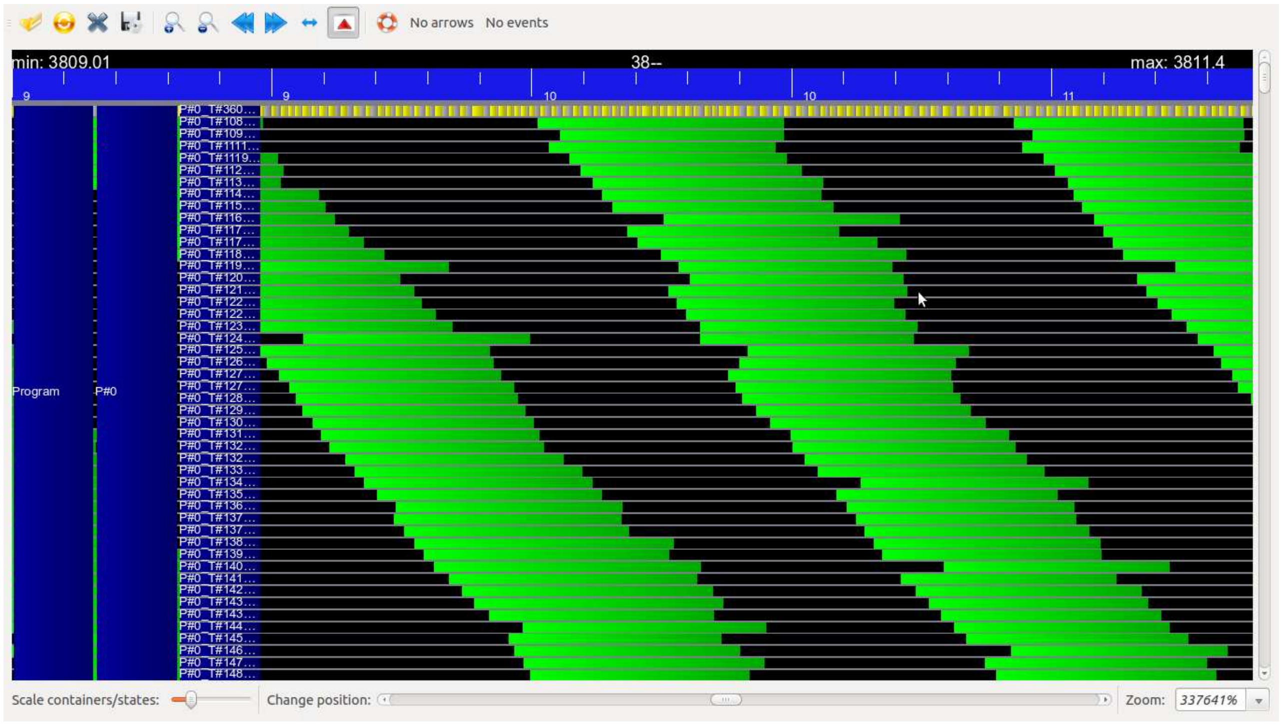

**Figure 3.4:** CPU trace generated by EZTrace for a dummy program for which each task is not dependent on any other task on a 2.4 GHz 12-socket quad-core (48 cores total) AMD Opteron(tm) processor. The motivation to use dummy program is check for a perfectly parallel program what are the implementation bottleneck.

are idle. We want to keep worker threads as busy as possible. Thus we decided to get a more detailed profile of inserting task into the QUARK. In Figure 3.5, we see that sometimes master thread is taking very long time to acquire lock on the address_set_node data structure of QUARK. The time to acquire the lock is shown by relatively long white colored tile in the first row of the trace. Note that around the same time many worker threads show orange color tile. Orange color tile corresponds to part of the QUARK code where worker thread finishes the task and unlocks the next task in the DAG as explained in previous sections. This behaviour can be explained graphically in Figure 3.6. The master thread wants to append dependency (colored in red) to the address_set_node A. However, thread 1, 2, and 3 are already waiting for address_set_node lock to update the status of previous tasks. Thus master thread will keep on trying to acquire lock. If master thread, instead of competing with other threads, can append this task to the specific address_set_node (in this case address_set_node A) and move on to add other tasks to be inserted, we can get a better rate of task insertion. Whereas in this case master thread is blocked and it cannot push enough tasks into the QUARK resulting in sharp drop in the performance.

We used a simple solution to overcome this problem. (As shown in Figure 3.7), in this solution we keep one additional auxilary list to main dependency list for each address_set_node. Whenever the master thread cannot acquire a lock on the main dependency list, it will append the task to an auxillary pointer and keep pumping task into the QUARK. Solving this bottleneck helped us to get more performance improvement without reaching a sharp drop in the performance at matrix size 9000 as shown by the performance plot in Figure 3.8.

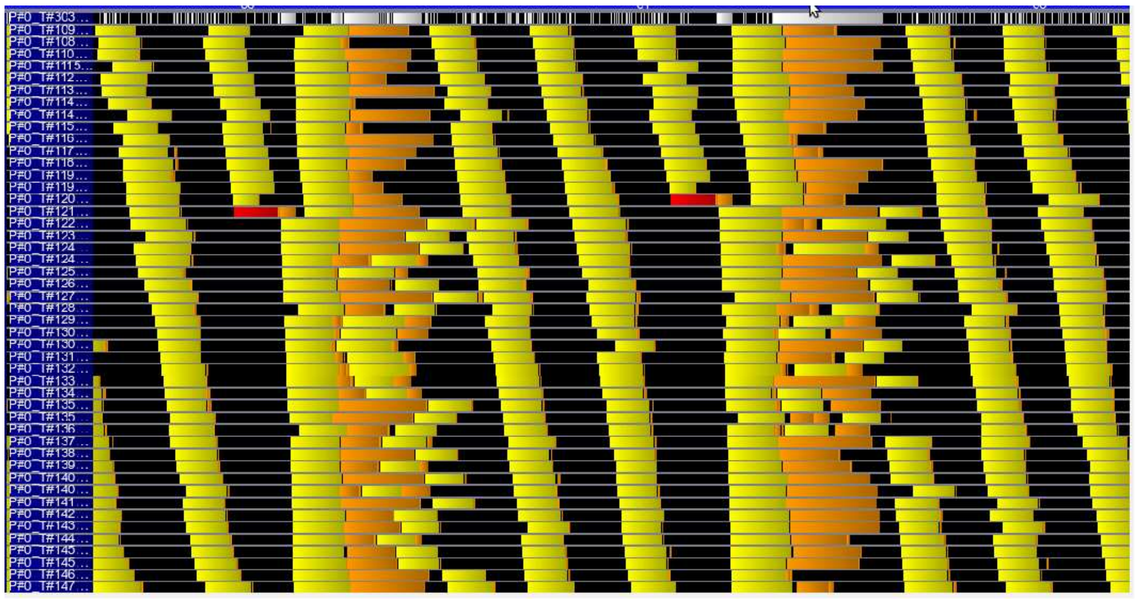

**Figure 3.5:** EZTrace showing lock contention problem for master while inserting task into the system. Trace for tiled QR factorization, tile size NB = 80, inner block size IB = 32, double precision on a 2.4 GHz 12-socket quad-core (48 cores total) AMD Opteron(tm) processor.

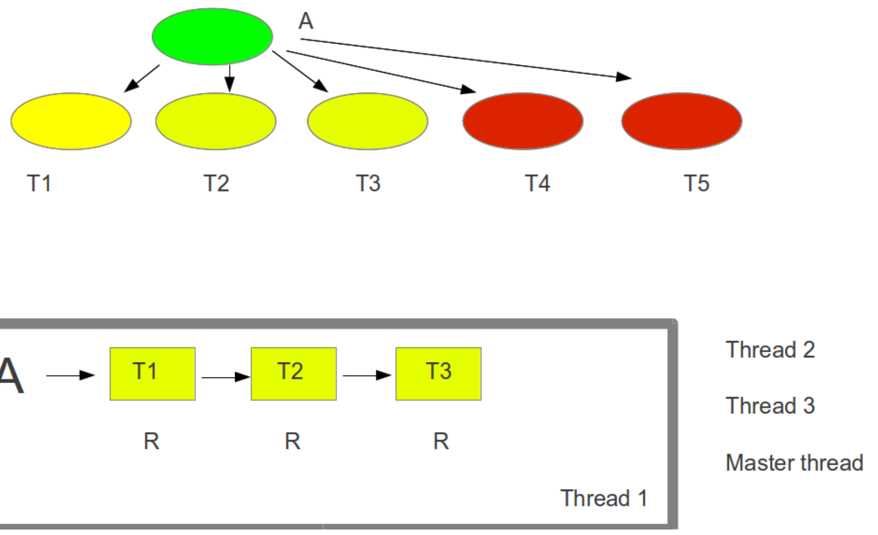

**Figure 3.6:** Graphical illustration of how master is blocked while appending a task to a single dependency. Here master is competing with other worker threads to acquire the lock. In this processes master end up not inserting enough tasks into the system.

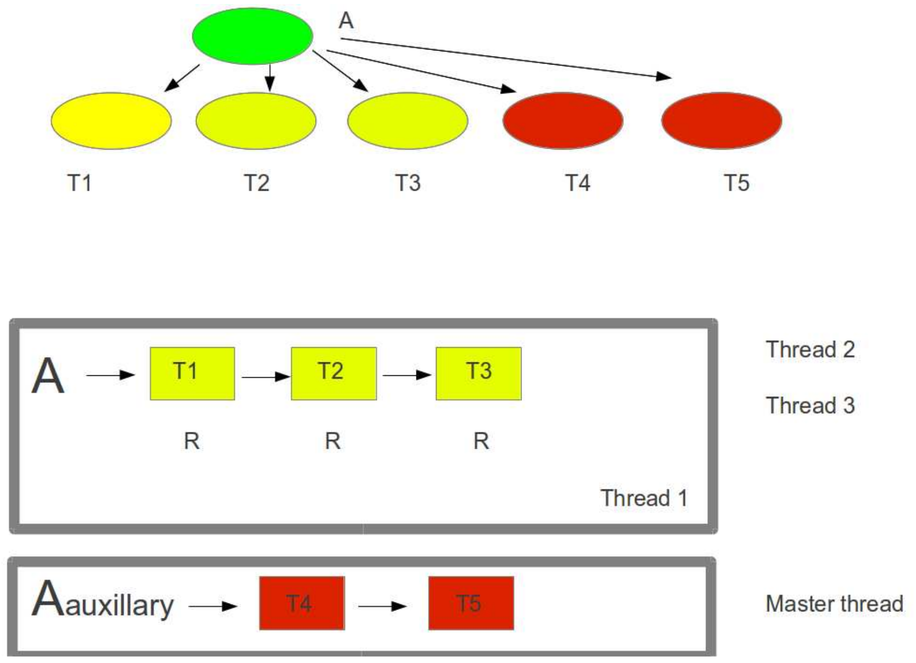

**Figure 3.7:** Graphical illustration of a how masater need not block while inserting a task to an address_set_node. This would improve rate of task insertion.

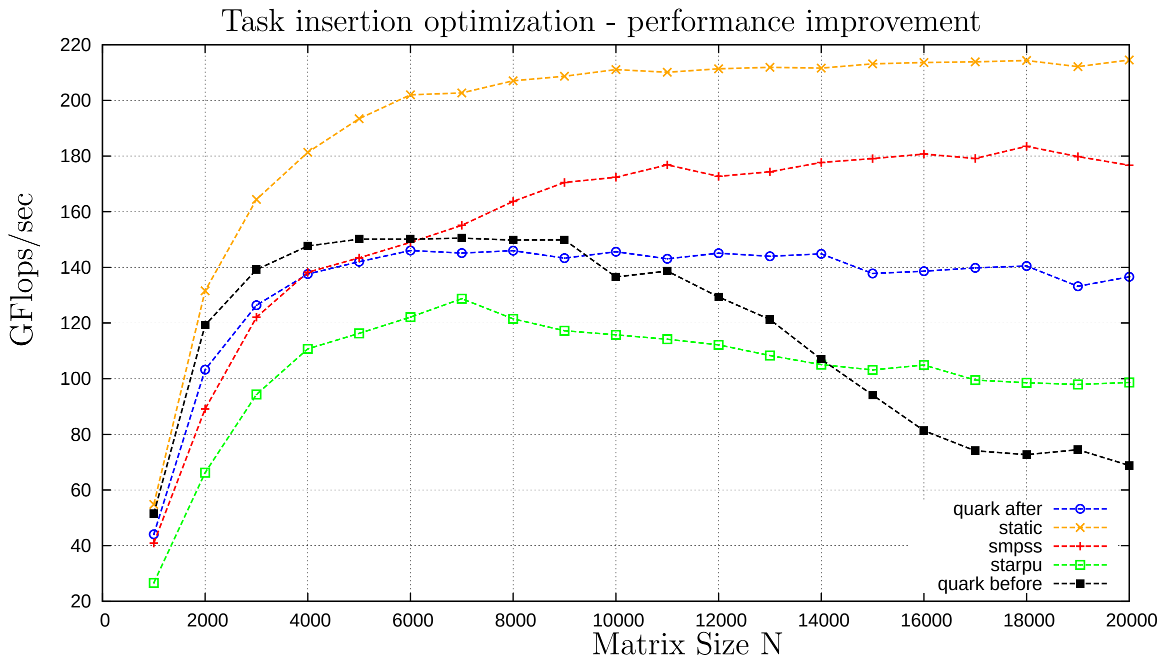

**Figure 3.8:** Experment result for task insertion improvements for tiled QR factorization, tile size NB = 80, inner block size IB = 32, double precision on a 2.4 GHz 12-socket quad-core (48 cores total) AMD Opteron(tm) processor.

# Chapter 4

# Scheduling Optimizations

This section explains optimization possibilities with QUARK at the logical level that is which scheduling algorithm can be used. We present variations of scheduling that can improve CPU utilization.

## 4.1 Scheduling Algorithms

Scheduling tasks specified by a DAG on multiple cores is an extensively studied problem(see [10] for a review). It has been shown that scheduling a set of tasks specified by a DAG on a finite number of cores is a NP complete problem[11]. Thus different heuristics are being used for all practical applications and there are many different heuristics described in the literature[10]. We focus our attention on list scheduling algorithms[12][13][14]. List scheduling algorithms assign each task a priority and create a list of such tasks with decreasing order of priority. Each task is executed with descending order of priority i.e. most important task is executed first. One list scheduling algorithm is 'Highest Level First with Estimated Times (HLFET)' which gives a near optimal scheduling solution in most cases[12]. We will illustrate how HLFET works for a simple DAG shown in Figure 4.1.

For the purpose of illustration we will assume all tasks take the same amount of time. We calculate the bottom level for each node in the DAG. Bottom level is

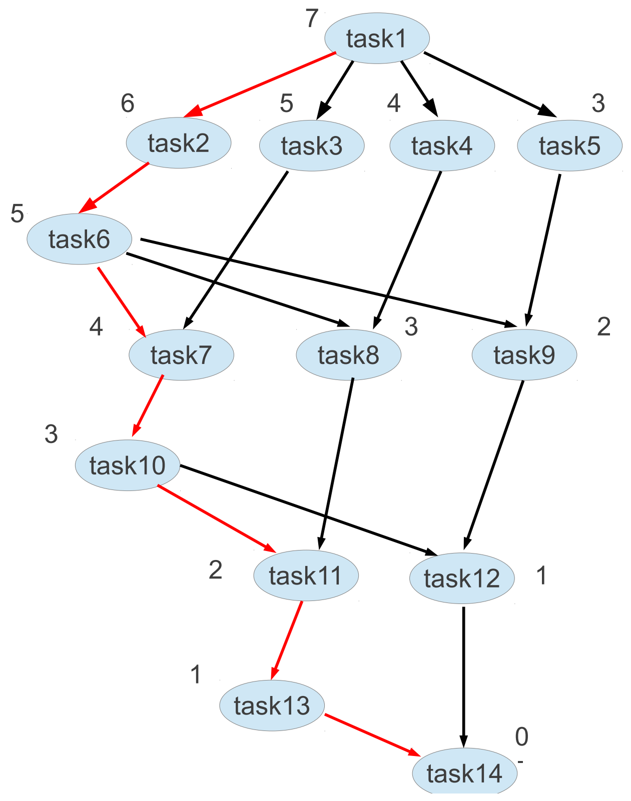

**Figure 4.1:** A simple DAG to illustrate HLFET

defined as the longest path between a node in the DAG and the end node. Thus for the DAG in Figure 4.1, the end node is ‘task14’. For ‘task10’ there are two paths to reach the end node (‘task14’), the first path is through node ‘task12’ and the second path is through node ‘task11’. Out of these two paths, the longest path for ‘task10’ is though node ‘task11’ which is of length 3. For each task we assign priority equal to the bottom level. In the Figure 4.1, for each node we show what is the corresponding priority. After we assign the priority for each task, we create a list of tasks in descending order of priority.

```
list = (task number, priority)
list = (1,7)->(2,6)->(3,5)->(6,5)->(4,4)->(7,4)->(5,3)->(8,3)
       ->(10,3)->(9,2)->(11,2)->(12,1)->(13,1)->(14,0)
```

Initially, the first task in the list will be executed and after that whichever tasks become ready for execution are executed as per the priority provided by the list. While calculating bottom level HLFET also takes duration for each task into consideration. Thus HLFET gives a good scheduling order. However, the problem with HLFET heuristic is that it requires the entire DAG before assigning priority. However, the common workloads for QUARK are linear algebra algorithms with big matrix sizes, where unrolling the entire DAG will result in poor performance because of the size of the problem. In order to handle large linear algebra problems QUARK maintains a moving window of the tasks where new tasks are added into system only when old tasks are executed. This way, the total number of tasks handled at a given time in QUARK remains constant. This kind of approach requires variation of the HLFET algorithm. In the following section we present a heuristic for which one does not require the entire DAG in order to make scheduling decisions, making it a suitable option for QUARK.

### 4.1.1 Greedy Heuristic

Figure 4.2 shows the partial DAG for Cholesky factorization[7], a linear algebra

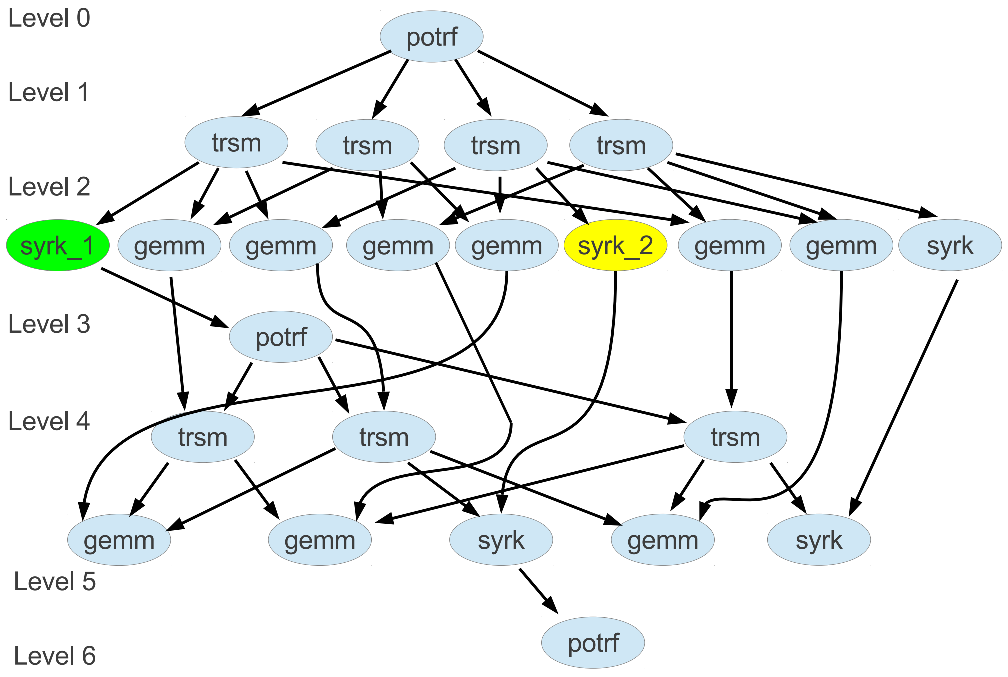

**Figure 4.2:** DAG for Cholesky factorization

routine, showing the first 6 levels of the DAG. In this DAG for each task we assign the priority equal to the lowest top level of its immediate children. The top level for a node is defined as the longest path from the start node. In Figure 4.2 at level 2 there are two syrk task nodes, first syrk_1 (highlighed with green color) and second syrk_2 (highlighted with yellow color). It can be seen that output of syrk_1 is immediately required at level 3 by task potrf, however, the syrk_2 task's output is not required until level 5 task is not pushed into the system. So the heuristic should make syrk_1 more important than syrk_2. Thus we make priority of the task equal to the level of the task wherever the output of the task is first required. Thus syrk_1 tasks would get priority as 3 and syrk_2 would get priority as 5 thus making syrk_1 more important than syrk_2. The advantage of this heuristic is that one need not process the entire DAG as only immediate children are needed to make the priority decisions. This makes the particular scheduling logic a greedy heuristic. In the next subsection we show experimental analysis for this heuristic.

### 4.1.2 Experimental Results

We implement the heuristic mentioned in the previous subsection for the linear algebra Cholesky factorization routine. The DAG for the Cholesky factorization routine is shown in Figure 4.3. For each node, the figure shows the priority of the task calculated by the greedy heuristic. Also, each node in the DAG is labeled as name of the routine followed by a '_' and the task number. Hence, the first task in the DAG is named as 'potrf_1'. In this Figure at level two, there are three tasks, syrk_4, gemm_5, and syrk_6; which can run in parallel. If we consider the problem of scheduling these tasks on two cores, then there are three ways in which these three tasks can be scheduled on two cores. We run this problem using two cores with the default QUARK scheduling logic. Corresponding CPU trace for the problem are shown in the Figure 4.4. In this figure, the width of each tile represents the proportion of time required by a given task in the DAG. Each tile is also labeled with the corresponding task number in

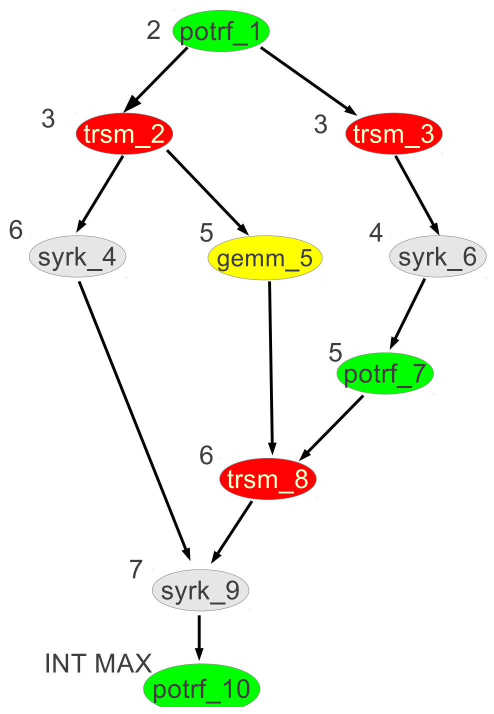

**Figure 4.3:** DAG for cholesky factorization

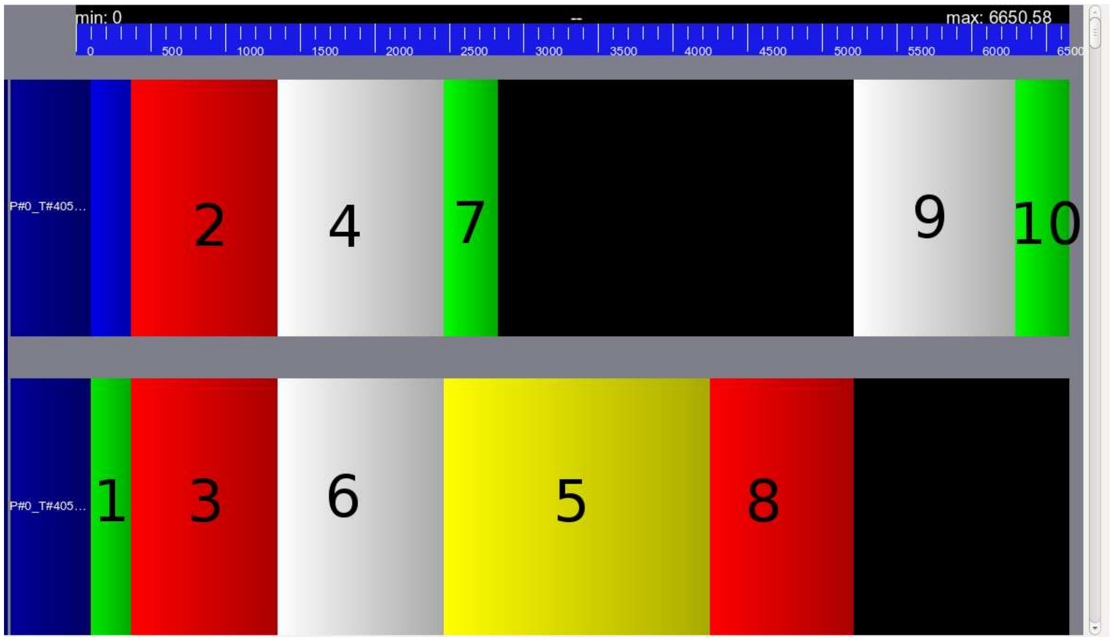

**Figure 4.4:** CPU trace for default QUARK scheduling

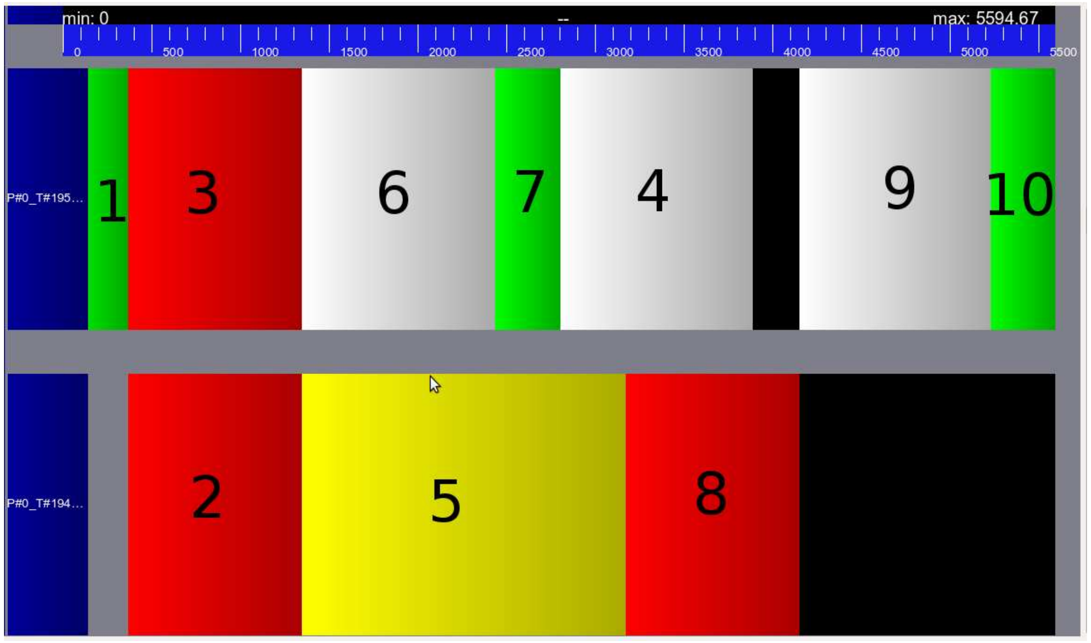

**Figure 4.5:** CPU trace for greedy heuristic scheduling

the DAG. It can be seen that after executing task number 7 (potrf_7), the first core remains idle till the time task 8 is over. After that, the first core executes task number 9 i.e. 'syrk_9'. This CPU trace shows that there is scope for improvement using the default QUARK scheduling.

After that we run the same problem using the same two cores with the greedy heuristic specified in the previous subsection. The trace of this execution is in the Figure 4.5. Similar to the previous CPU trace, in this Figure, each tile represents proportionate time required by a given task in the DAG. Each tile is also labeled with the corresponding task number in the DAG. It can be seen that there is no idle first core as it was with the default QUARK scheduling. Thus showing QUARK performance could be improved using the greedy heuristic to assign task priorities.

We run this problem for the larger matrix size and observe that the performance improvement is not as big as that of the low level optimizations. Figure 4.6 shows the

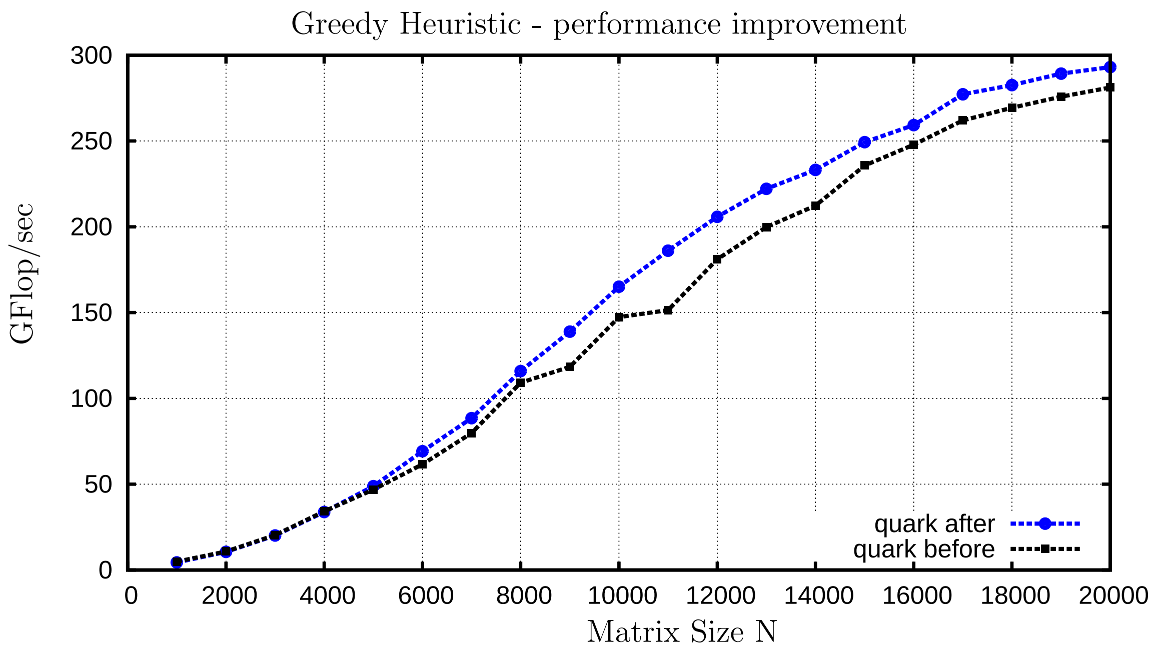

**Figure 4.6:** Performance improvement for greedy heuristic for tiled Cholesky factorization, tile size NB = 700, double precision on a 2.4 GHz 12-socket quad-core (48 cores total) AMD Opteron(tm) processor.

for bigger matrix size and bigger tile size as well, the greedy heuristic shows marginal improvements over default QUARK.

# Chapter 5

# Conclusion and Future Work

## 5.1 Conclusion

Task schedulers or runtime systems enable programmers to use multicore machines in a productive way. Thus runtime should ensure the best possible performance on the given architecture. In this work, we looked at the QUARK runtime scheduler, and showed that the performance could be increased with careful selection of locking mechanisms and data structures. It is difficult to find the synchonization locks causing performance bottlenecks just by qualitative analysis. We show how a lightweight tracing library can be used to analyze the performance issues of a scheduler. For the improvements achieved as part of this work, we observe that it is critical to ensure that the task insertion process is given maximum priority when compared to other threads. Any delay while inserting the tasks into the system has a significant impact on the overall performance. We also observed scheduling algorithms does not make big impact on the performance for the selected workloads. Bottom level based scheduling algorithms give near optimal scheduling solutions, but they require the entire DAG to calculate priorities of the task. To have a fast scheduling algorithm we suggest a simple greedy heuristic based on the earliest time at which the results of a task are needed. However, through our experiments we did not observe a great performance improvement using this heuristic. This underlines the fact that low level details involving overheads is the domain that one should focus first while making performance improvement to runtime systems like QUARK.

## 5.2 Future Work

As part of the future work one can get more tracing information for the schedulers and try to eliminate as much lock contentions as possible. The other possible approach to improve scheduler could be trying to minimize data transfers amongst different threads.

# Bibliography

[1] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK User's Guide: QUeueing And Runtime for Kernels. Technical report, Innovative computing lab, University of Tennessee, Knoxville, 2007. 1

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, pages 207–216, 1995. 2

[3] James Reinders. *Intel threading building blocks.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. 2

[4] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. In *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, PPAM'07, pages 639–648, Berlin, Heidelberg, 2008. Springer-Verlag. 2, 4

[5] F. Trahay, Y. Ishikawa, F. Rue, R. Namyst, M. Faverge, and J. Dongarra. EZTrace: A Generic Framework for Performance Analysis. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 618 –619, may 2011. 2, 20

[6] Emmanuel Agullo, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julie Langou, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Asim YarKhan. PLASMA user's guide. Technical report. 3

[7] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurr. Comput. : Pract. Exper.*, 22(1):15–44, January 2010. 3, 30

[8] Rosa M. Badia, Jos R. Herrero, Jesus Labarta, and Josep M. Perez. Parallelizing Dense and Banded Linear Algebra Libraries using SMPSs, 2008. 7

[9] Cdric Augonnet, Raymond Namyst, and Inria Bordeaux. StarPU: A Unified Runtime System for Heterogeneous Multi-core Architectures. 8

[10] I. Ahmad, Yu-Kwong Kwok, and Min-You Wu. Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors. In *Parallel Architectures, Algorithms, and Networks, 1996. Proceedings., Second International Symposium on*, pages 207 –213, jun 1996. 28

[11] W.H. Kohler. A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems. *Computers, IEEE Transactions on*, C-24(12):1235 – 1238, dec. 1975. 28

[12] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, December 1974. 28

[13] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, April 1989. 28

[14] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 4(2):175 –187, feb 1993. 28

# Vita

Vijay Gopal Joshi was born in India and raised in Pune, a city located in the western part of India. He graduated from Modern junior college at Pune, India. He enrolled in the Computer Engineering undergraduate program of the Govt. College of Engineering, Pune (COEP), India. He received his undergraduate degree in Computer Engineering in the year 2009. He worked for two years as a software developer in D. E. Shaw India software pvt. ltd. till 2011. In 2011, he enrolled in the Master of Science with major in computer science program in EECS department in the University of Tennessee, Knoxville. He expects to graduate in May 2013. Between years 2011 and 2013, he worked as a Graduate Research Assistant in the Innovative Computing Laboratory UTK. He is planning to join Amazon.com at Seattle, WA as a full time software developer.