NLAFET
Parallel Numerical Linear Algebra for Future Extreme Scale Systems

H2020–FETHPC–2014: GA 671633

D6.1

Prototypes for runtime systems exhibiting novel types of scheduling

April 2017

Document information

Scheduled delivery	2017-04-30
Actual delivery	2017-04-28
Version	1.3
Responsible partner	UMU

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2017-03-29	Lars Karlsson	Draft	1.0	First draft
2017-04-02	Carl Christian Kjelgaard Mikkelsen	Draft	1.1	Minor additions and formatting
2017-04-03	Lars Karlsson	Review	1.2	Minor changes
2017-04-27	Lars Karlsson	Final	1.3	Minor changes

Author(s)

Lars Karlsson, Angelika Schwarz, Carl Christian Kjelgaard Mikkelsen (UMU)
Samuel Relton (UNIMAN)

Internal reviewers

Samuel Relton (UNIMAN)
Florent Lopez (STFC)

Contributors

Bo Kågström (UMU)

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

List of Figures

List of Tables

1 Introduction

The *Description of Action* document states for deliverable D6.1:

> "*D6.1: Prototypes for runtime systems exhibiting novel types of scheduling*
>
> Prototype for runtime systems capable of scheduling at a varying level of granularity/abstraction (i.e., basic kernels, BLAS, LAPACK, etc), and runtime systems that can execute the tasks along the critical path with an adaptive level of parallelism."

This deliverable is in the context of Task 6.1 (Scheduling and Runtime Systems).

Extreme-scale systems are projected to be hierarchical and heterogeneous in nature. To match such architectures one needs schedulers for multi-level parallelism—essentially a hierarchy of schedulers with different scopes (socket, accelerator, node, rack, etc). In Section 2, we determine the demands which a run-time system must satisfy in order to be useful for NLAFET. We make a brief comparison of the existing run-time systems. StarPU emerges as a clear winner and is compared against OpenMP in the case of Cholesky factorization.

One special case of multi-level scheduling consists of parallelizing the tasks along the critical path to reduce the parallel execution time near the limit of strong scalability, i.e., where the critical path (rather than, say, the number of cores) limits/bounds the execution time. In Section 3, we develop a prototype runtime system based on the principle of parallelizing the critical path and evaluate its effectiveness.

Together, these recent developments contribute towards fulfilling the aims of Task 6.1. Planned future work (e.g., Deliverable D6.3) will focus on applying these new tools to codes developed as a part of other NLAFET work packages.

The work presented in Section 2 and Section 3 has mainly been done by UNIMAN (Manchester) and UMU (Umeå), respectively.

2 Runtime systems for HPC

2.1 NLAFET requirements from runtime systems

Since task-based programming is a rather new model of programming, there is no general consensus from the wider computing community on how tasks should be described and scheduled. As such, there are a variety of different runtimes implemented, each supporting a variety of different features and approaches, which are not compatible with one another.

The NLAFET project aims to provide a software library capable of performing linear algebra operations on large-scale, distributed, heterogeneous machines. As such there are a number of desired features that the NLAFET consortium would like to see.

First, it is essential that the runtime system supports distributed memory computation in order to scale to large problems. Related to this, it is desirable for the runtime system to handle all MPI calls itself, so the library developers need not make explicit MPI calls. This improves the ease of programming for developers and allows the software to schedule tasks dynamically (and therefore more efficiently and adaptively than a developer could).

Second, it has been shown repeatedly that the strategy used to assign tasks to cores (or GPUs etc.) can have a dramatic impact on the overall performance of the routine. Therefore it is important that the runtime system supports a variety of task scheduling

Table 1: Summary of runtime systems against the requirements of the NLAFET project. Each entry of the table is marked for yes, or left blank for no.

Requirement	HPX	Stapl	Quark(-D)	Pulsar	ParSec	StarPU	OpenMP
Dist. memory	×	×	×	×	×	×	
Task scheduling			×		×	×	
Custom scheduling					×	×	
GPU aware				×	×	×	
Good documentation						×	×

strategies, and it is desirable that there is a framework in place to add custom scheduling strategies too.

Third, to run on heterogeneous clusters it is imperative that the runtime supports offloading computation to GPUs and other accelerators (such as older models of the Intel Xeon Phi). Ideally the runtime would automatically use any such devices found on each node, instead of the library developer having to handle the hardware devices themselves.

Finally, from a pragmatic perspective, it would be sensible to use a runtime system that is relatively mature, has good documentation, and has supportive developers who are happy to answer questions and interact with users looking to push their runtimes to the limit. If the runtime is fairly simple to use this will also be a benefit, as it can save a lot of developer time.

Therefore our first task is to summarise how well each of the major runtime systems fits the requirements above. We will be considering the following runtime systems.

- HPX – Stellar group from Louisiana State University.
- Stapl – Parasol group from Texas A&M University.
- Quark/Quark-D – ICL group from The University of Tennessee.
- Pulsar – ICL group from the University of Tennessee.
- ParSec – ICL group from The University of Tennessee.
- StarPU – STORM group from Inria Bordeaux.
- OpenMP – Standardised API for shared memory systems.

2.2 Comparison of existing runtime systems

A brief summary of how each runtime system fits the NLAFET requirements is given in Table 1. We give a more detailed analysis of each runtime in the remainder of this subsection.

First we consider the HPX runtime system. HPX is primarily focused upon an Active Global Address Space, which is essentially an alternative model for distributed memory that negates the need for MPI calls. It seems that they exhibit good scaling in shared memory architectures but, in a distributed setting, are not currently matching state-of-the-art performance. Furthermore HPX does not support GPU computation, though does support Intel Xeon Phi, and the task scheduling process they use is neither described in the documentation nor easily interchangeable.

The Stapl runtime uses Intel TBB (Threading Building Blocks) for parallelism[1]. The runtime itself does not support accelerators and is not readily available to download and try. As far as we can tell, there is no available documentation so it is unclear how the runtime system works internally.

Quark and Quark-D (the distributed version) support distributed memory operation and various task scheduling strategies can be used (though it appears difficult to add custom strategies). GPUs and other accelerators are not supported and there is very little documentation. Neither runtime is currently under active development so there is little chance of getting help from the developers to extend the system further.

The Pulsar runtime does not support multiple task scheduling strategies and it is not currently possible to add any. Furthermore the documentation is not sufficiently descriptive to fully understand the internal mechanisms of the runtime. Pulsar is also not under active development at present.

ParSec fits a number of the NLAFET requirements: it supports multiple task scheduling strategies, it is possible to add customised ones, and the runtime can make use of GPUs and other accelerators. The main issue with ParSec is that the system is rather difficult to use since there is very little documentation and the method by which tasks are created is somewhat complex: ParSec uses a "parameterised task graph" approach as opposed to the "sequential task flow" model used by StarPU and other runtime systems. Essentially, this requires the developer to describe the entire DAG (Directed Acyclic Graph) and specify the data dependencies in a ParSec-specific format before compiling the code. The advantage of this approach is that the tasks can be scheduled very efficiently with a small memory footprint. By contrast, in StarPU tasks can be added dynamically at runtime by calling the relevant function within a loop. The additional flexibility in the StarPU approach is helpful for sparse matrix algorithms, where the DAG is typically not known at compile time. On the other hand, there is a large team of developers currently working on ParSec who are happy to give advice to users.

The StarPU runtime from Inria Bordeaux also satisfies the NLAFET requirements. It supports all the features that we are looking for, is fairly easy to use, and has extremely good documentation along with a helpful development team. Another benefit of StarPU is that it allows the use of OpenMP within tasks and has a source-to-source compiler called KStar which converts OpenMP task-based code into StarPU code for easy comparison. The only issue with StarPU when compared to ParSec is that StarPU is cannot scale as efficiently when thousands of nodes are used simultaneously; this is something that the StarPU development team are currently working on.

Finally, although it does not support distributed memory computation, we mention OpenMP because it is standardised and very efficient on a single node of CPUs (including the Intel Xeon Phi). One possibility is to use another runtime system at the distributed level make calls to OpenMP from within a node.

To conclude this analysis, it seems clear that either StarPU or ParSec provide a suitable runtime system for the NLAFET project and that, due to the ease of use, StarPU may be preferred. However, it is important to compare against OpenMP at the node level since it has shown extremely promising performance on NUMA nodes and the Intel Xeon Phi.

[1]Available from `https://github.com/wjakob/tbb`

2.3 Comparing OpenMP and StarPU

As part of NLAFET Deliverable D2.1, we have been comparing the performance of various runtime systems on matrix factorizations including Cholesky, LU, QR, and LDL^T. Since the full details of this comparison can be found in Deliverable D2.1, we will describe how to install the relevant software to run our software using both OpenMP and StarPU.

All the comparisons were made using the PLASMA software library for dense linear algebra. This software was recently rewritten into OpenMP in a collaboration between The University of Manchester and The University of Tennessee. A snapshot of the PLASMA version used to produce Deliverable D2.1 can be found in the NLAFET github directory[2] whilst the current development version can be found at Bitbucket.[3] (See also NLAFET Deliverable D7.5.)

Once either version has been downloaded it is simple to compile the software with OpenMP. We merely need to rename the `make.inc.mkl-gcc` file to `make.inc` and ensure that the Intel MKL libraries have been loaded onto the path: typically this is done via typing `source /opt/intel/.../bin/compilervars.sh intel64` on the command line. Note that the development version of PLASMA needs to use `gcc` version 6 or later in order to make use of task priorities, which are not present in earlier versions. After this is complete simply type `make -j` to compile the software in parallel.

After compilation is complete there will be a binary called test, found inside the test folder, which can be used for our comparisons. Before launching the binary it is necessary to set the number of OpenMP cores using the environment variable `OMP_NUM_THREADS` and to set the environment variable `PLASMA_TUNING_FILENAME` to point to the default tuning file (found in tuning/default.lua). The binary can then be used as follows: `./test routine --dim=n --test=c`, where

- `routine` is a function name e.g. `dgetrf` for a double precision LU decomposition,
- `n` is an integer giving the size of the matrices, and,
- `c` is either `y` or `n` to determine whether or not the accuracy of the solution is compared against the Intel MKL implementation.

Note that there are multiple other advanced options that can be used but are not required here.

To install PLASMA using the StarPU runtime system, in order to test the advanced scheduling strategies implemented, one needs only change the compiler from `gcc` to `kstar --runtime=starpu` in the `make.inc` file. Clearly the KStar source-to-source compiler will need to be installed before it can be used. KStar can be downloaded from the Inria repositories.[4] After this, we can simply compile the software and use the testing routine as before.

Note that the scheduling strategy used by StarPU can be changed by modifying the environment variable `STARPU_SCHED`. In particular we have been experimenting with the eager, ws (work-stealing), and dmda (deque model data aware) strategies. Furthermore, since we are simply converting OpenMP code rather than writing a native StarPU version, we cannot take advantage of any GPUs attached to the node.

[2]`https://github.com/NLAFET/plasma`
[3]`https://bitbucket.org/icl/plasma`
[4]`https://scm.gforge.inria.fr/anonscm/git/kstar/kstar.git`

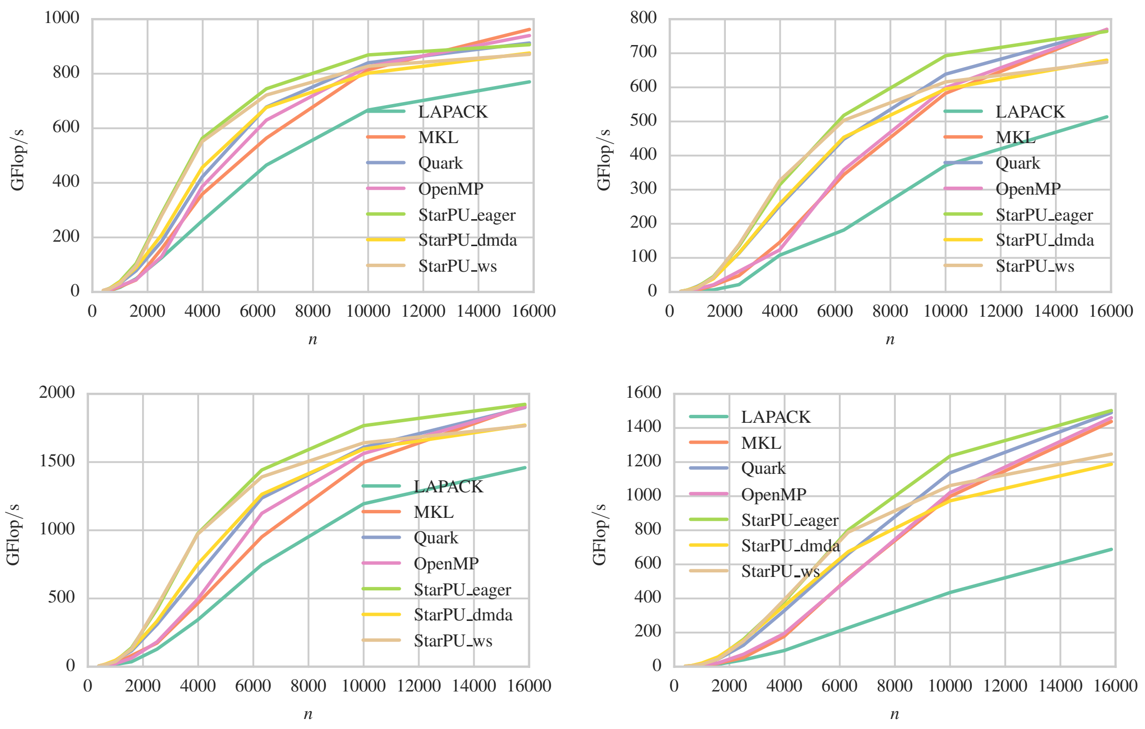


Figure 1: Performance of Cholesky factorization on NUMA node. The top row has double complex precision on the left and double precision on the right. The bottom row has complex precision on the left and single precision on the right.

Whilst the full details of our runtime system comparison can be found in Deliverable D2.1, we reproduce the results for the Cholesky decomposition here. In particular, we compare LAPACK, Intel MKL, and an older version of PLASMA using the Quark runtime against OpenMP and StarPU versions of PLASMA as the matrix size grows.

The experiments were performed on a NUMA node using Broadwell processors, in particular a Xeon E5-2690 v4 chip with 2 sockets containing 14 cores each. The software was compiled with gcc 5.4.0 and Intel MKL 17.0.1.

The results of the experiment can be found in Figure 1. On the top row we plot the results for double complex and double precision, whilst the bottom row contains results for complex and single precision computations. We see that the LAPACK implementation lags behind the others and that in most cases one of Quark, OpenMP, or the various StarPU versions perform better than MKL. In particular the eager scheduler within StarPU gives very good performance for this particular routine, and results for the outer routines can be found in NLAFET Deliverable D2.1.

In summary, we have found that NLAFET software should attempt to use the StarPU runtime where possible to maximize performance and ease the transition to heterogeneous and distributed computation.

3 The PCP runtime

A task-based parallel program modeled by a directed acyclic graph (DAG) and scheduled dynamically on a multi-core machine with shared memory often efficiently utilizes the cores by avoiding many synchronization points not mandated by the inherent data dependencies. Consider, for example, a tiled Cholesky factorization algorithm parallelized using a dynamically scheduled DAG. The size of a tile, b, is a tunable parameter that has a significant impact on performance. The tile size determines both the task granularity and the degree of concurrency. The task granularity in turn affects both the efficiency with which a task can be executed (coarser tasks are generally more efficient) and the scheduling overhead (finer granularity leads to more overhead due to more tasks). The degree of concurrency shrinks as the tile size increases, making it more difficult to keep all cores busy. These competing forces imply that an optimal tile size must be neither too small nor too large.

Suppose that the tile size is so large that the critical path of the DAG is limiting the overall execution time. For Cholesky factorization in particular, and many matrix algorithms in general, shrinking the tile size reduces the *weight* (i.e., the total number of flops) of the critical path. Shrinking the tile size will shorten the *length* (i.e., the execution time) of the critical path if the reduction in work along the critical path outweighs the resulting loss in task efficiency (of the critical tasks) and increased overhead (along the critical path) associated with a finer task granularity. Any reduction in the length of the critical path translates into a reduction in the overall parallel execution time if the critical path continues to be the limiting factor.

Now suppose instead that the tile size is far too small. Then the cores are almost never going to be idle since there are many tasks to execute. However, this comes at the expense of having very inefficient tasks with plenty of scheduling overhead. In this case, increasing the tile size would simultaneously improve the task efficiency and reduce the overhead. This ought to translate into a shorter overall parallel execution time.

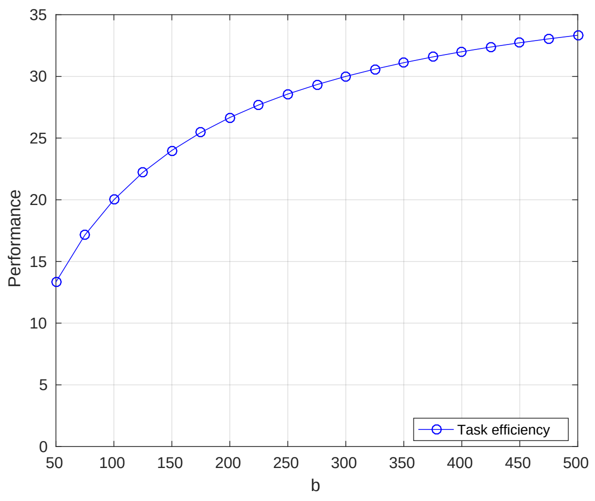


Figure 2: The speed profile (efficiency) $s(b)$ of a task.

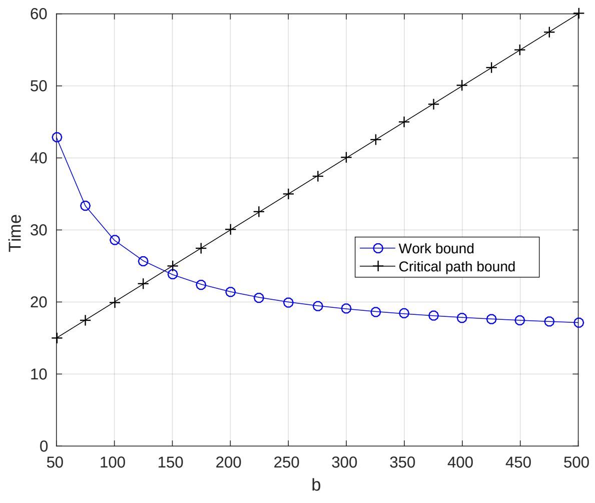
Let us illustrate this discussion with a simple model that takes the granularity-dependent task efficiency into account. Consider a computation with a total of n^2 flops distributed over tasks operating on tiles of size $b \times b$ and each involving b^2 flops. Suppose that the weight of the critical path is $(n/b)b^2 = nb$ flops. A task is assumed to execute more efficiently for larger values of b in accordance with the speed profile illustrated in Figure 2. When the task granularity is very fine, then the parallel execution time on p cores is bounded below by

$$T(n, b, p) \geq \frac{n^2/p}{s(b)}. \tag{1}$$

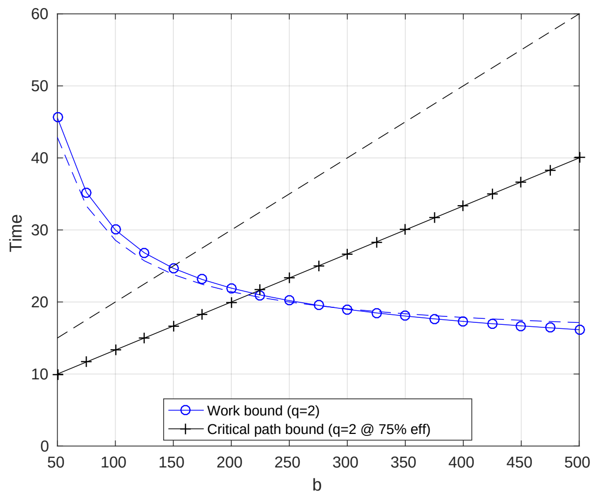
This inequality assumes that all n^2 flops are executed with the speed $s(b)$ and that there is perfect speedup when using all p cores in parallel. If instead the task granularity is very coarse, then the critical path will dominate the execution time and we get the following second lower bound:

$$T(n, b, p) \geq \frac{nb}{s(b)}. \tag{2}$$

This inequality assumes that all nb flops on the critical path are executed with the speed $s(b)$. The bounds (1) (“work bound”) and (2) (“critical path bound”) are illustrated in Figure 3a for $n = 4000$ and $p = 28$. According to this model, the optimal tile size occurs where the two curves intersect.



(a) The bounds (1) and (2).



(b) The bounds (3) and (4) for $q = 2$ with 75% parallel efficiency of the critical tasks. The curves from Figure 3a are shown as dashed lines for comparison.

Figure 3: The bounds on the parallel execution time illustrated for two different ways of executing the tasks when $n = 4000$ and $p = 28$.

Figure 3a suggests the following idea. What if one could scale down the “critical path bound” by a factor $\alpha \in (0, 1)$? The intersection point between the two curves would then shift to the right and the optimal execution time would therefore be reduced given that the work bound is monotonically decreasing. One way to accomplish this in practice is to reserve $q > 1$ cores for the sole purpose of executing (in parallel across all reserved cores) all critical tasks (and no other tasks). But parallelizing tiny tasks would be difficult to achieve with perfect speedup, so let $E(b, q) \in (0, 1)$ denote the parallel efficiency one can obtain by parallelizing a task that operates on a tile of size b across q cores. Under this extended model, the “critical path bound” changes to

$$T(n, b, p, q) \geq \frac{nb}{qE(b, q)s(b)}. \tag{3}$$

In addition, since reserving q cores also reduces the number of cores available to execute the non-critical tasks, the “work bound” changes to

$$T(n, b, p, q) \geq \frac{(n^2 - nb)/(p - q)}{s(b)}. \tag{4}$$

The numerator captures the fact that the non-critical tasks account for $n^2 - nb$ flops and are executed by $p - q$ cores at speed $s(b)$. These modified bounds are illustrated in Figure 3b. Note that the intersection point has shifted to the right and despite the slight increase in the work bound, the optimal execution time has been reduced.

In deliverables D6.1 and D6.3, we are investigating if and why this idea of *parallelizing the critical path* can improve the optimal execution time of various matrix computations. In this *demonstrator* Deliverable D6.1, we are presenting a prototype runtime system for shared-memory machines designed specifically for the purpose of investigating the merits of this idea. We include two simple examples to validate the implementation. In the upcoming Deliverable D6.3, we will apply the runtime system to challenging eigenvalue computations of particular interest to the NLAFET project.

The main purpose of the runtime system presented here is to evaluate the idea of parallelizing the critical path. If the idea turns out to be sufficiently advantageous, then we will pursue the development/enhancement of support in mature runtime systems such as StarPU.

3.1 Installation

Download the source code by cloning the `NLAFET/pcp-runtime.git` repository on GitHub. Follow the compilation instructions given in the file `1-COMPILE` to build the runtime system library and executables for the two examples. Follow the instructions in the file `2-TEST` to test the correctness and performance of the two examples. Follow the instructions in the file `3-ANALYZE` to analyze the results of the performance tests and see the effect of using the idea.

3.2 Runtime system design

Recall that the purpose of the runtime system is to test a particular idea, not to propose a mature runtime system design. Some of the design decisions, such as pre-generating and analyzing the entire DAG, are certainly not recommended in production code but they do make the prototype easier to implement without affecting our ability to evaluate the idea.

3.2.1 Threads, cores, and the reserved set

The runtime system uses p threads pinned to cores (one thread per core). We will therefore use the terms thread and core interchangeably. The threads/cores are ranked from 0 to $p - 1$. Thread/core 0 (i.e., the first thread) is referred to as the *master* thread/core and all other threads/cores are known as *slaves*. All threads participate in the execution of tasks.

At any given moment, $q \in \{0, 1, \ldots, p - 1\}$ of the threads are reserved for the sole purpose of executing critical tasks. This set of threads is referred to as the *reserved set* and always contains the first q threads.

3.2.2 Scheduling and the critical path

The tasks are scheduled by a list scheduling algorithm. Each task is assigned a priority and as soon as a thread becomes idle it selects a task with maximum priority from the set of ready tasks and executes it sequentially (or idles if there are no ready tasks). The priority of a task is set to the *height* of the task, which is defined as the length of the longest path (as measured by the number of tasks) starting at the task. The *critical path* is defined as (any one of) the longest path(s) in the DAG (again measured by the number of tasks). The tasks on the critical path are referred to as *critical* while all other tasks are referred to as *non-critical*.

The scheduling is dynamic and distributed. Associated with each task is a counter that keeps track of the current number of incoming edges from unfinished tasks. When a task's counter reaches zero it signals that the task is ready for execution. As soon as a thread completes a task, the counter of each successor task is decremented by one and any tasks that become ready as a result are added to a shared priority queue. The thread then distributes ready tasks (in the order specified by the task priorities) to all idle threads in rank order.

3.2.3 Execution modes

The runtime system has four different execution modes:

Regular mode All threads are treated the same (i.e., the reserved set is empty). The tasks are allowed to execute on any thread regardless if they are critical or not.

Fixed mode A fixed number of $q \geq 1$ threads are reserved for the (parallel) execution of critical tasks. All critical tasks (and no other tasks) are executed by the reserved set.

Prescribed mode The user prescribes for each critical task a desired number of threads to use for its execution. The runtime system attempts to honor these requests but will refrain from adding synchronization overhead when attempting to grow the reserved set (see Section 3.6).

Adaptive mode The runtime system tries to adapt the size of the reserved set during the execution in an attempt to optimize the parallel execution time.

Note that the *fixed mode* with $q = 1$ is very similar—but not identical—to the *regular mode*. The difference is that the *fixed mode* will reserve one thread for the execution of critical tasks, whereas the *regular mode* will not. Therefore, the core utilization is expected to be lower for the *fixed mode* than the *regular mode* whenever the critical path is not a limiting factor.

The *regular mode* is the baseline for our comparisons since it is identical to the *fixed mode* except that it neither parallelizes the critical path nor gives special treatment to critical tasks. This gives us a controlled environment in which the only variable that has changed is the application of the idea.

The *prescribed mode* can be used together with an external auto-tuner to find an optimal (possibly dynamic) size for the reserved set. This report does not contain results obtained from using this mode.

The *adaptive mode* is an example of online tuning. There are two major potential advantages of the *adaptive mode*. First, the *adaptive mode* may yield better performance than the *fixed mode* by allowing the reserved set to dynamically vary in size. Second, the *adaptive mode* may achieve performance close to the *prescribed mode* after auto-tuning but without the significant tuning cost. However, the control algorithm that determines when and how much to adapt the size of the reserved set has not yet been developed. This report does not contain results obtained from using this mode.

3.2.4 Statistics

The runtime system can report statistics on aspects of the execution that are relevant for understanding the benefits and limitations of the parallel critical path idea.

The *parallel execution time* measures the time from the start of the first task to the completion of the last task. The *cost* (processor time product) measures the amount of consumed CPU resources. The *resource utilization* reports the fraction of the cost that can be attributed to executing tasks.

The cost is further partitioned into what can be attributed to the reserved (and non-reserved) set of threads. Similarly, the cost is also partitioned into what can be attributed to the critical (and non-critical) tasks. From these partitioned cost metrics one can derive, for example, the resource utilization of the reserved (and non-reserved) set of threads.

The execution time of the critical path is also reported, along with the execution time of the longest path (which may be longer than the critical path; see Section 3.2.2).

3.2.5 Traces

The runtime system can generate basic traces in the form of a LaTeX document containing a TikZ picture. The trace distinguishes between different task types as well as sequential and parallel tasks and also shows the dynamic size of the reserved set. For an example of a trace see Figure 4 on page 16.

3.3 Usage of the runtime system

Recall that the runtime system has been designed to evaluate a specific idea. All design decisions which are irrelevant for this purpose have been chosen to simplify the implementation of the runtime system and the toy examples.

The following list provides an overview of a typical user work flow:

1. **Start the runtime system**
   The number of threads/cores, p, is specified and the runtime system creates $p - 1$ internally managed slave threads and pins all threads to separate cores. The execution mode is chosen and the size of the reserved set, q, is specified if the *fixed mode* is selected.

   Relevant API functions:

   - `pcp_start` Initialize the runtime system, create slave threads, and pin threads to cores. Takes the number of threads/cores as an argument.
   - `pcp_set_mode` Set the execution mode.
   - `pcp_set_reserved_set_size` Set the size of the reserved set (if the *fixed mode* is used).

2. **Register task types**
   All the task types along with their sequential and possibly also parallel implementations are registered with the runtime system prior to building the DAG.

   Relevant API function:

   - `pcp_register_task_type` Register a task type. Takes a structure containing pointers to a sequential and (optionally) a parallel implementation of a given task type as an argument and returns a handle to the task type.

3. **Build the DAG**
   Tasks and task dependencies are manually inserted in a topological ordering.

   Relevant API functions:

   - `pcp_begin_graph` Prepare to receive tasks and dependencies.
   - `pcp_insert_task` Insert a task by specifying the handle to its task type and a user-defined `void*` argument specifying the task parameters. Returns a handle to the inserted task.
   - `pcp_insert_dependence` Insert a dependence between two previously inserted tasks. Takes two task handles as arguments.
   - `pcp_end_graph` End the construction of the DAG.

4. **Execute the DAG**
   The DAG is executed and the user thread regains control once all tasks have completed.

   Relevant API function:

   - `pcp_execute_graph` Execute the graph in parallel and return after all tasks have completed.

5. **Analyze statistics and traces**
   Statistics and/or traces are extracted and manually analyzed.

   Relevant API functions:

   - `pcp_view_statistics_stdout` Print the internally gathered statistics (see Section 3.2.4).
   - `pcp_view_trace_tikz` Save the trace as a TikZ-based LaTeX document (see Section 3.2.5).

6. **Stop the runtime system**
   The slave threads are terminated and internal resources are freed.

   Relevant API functions:

   - `pcp_stop` Shut down the runtime system.

3.4 Example: Triangular solver

In this example, a lower triangular linear system

$$LX = B$$

with multiple right-hand sides is solved using a tiled task-based forward substitution algorithm. The matrix $L \in \mathbb{R}^{n \times n}$ is lower triangular and partitioned into an $N \times N$ tile matrix, where $N = \lceil n/b \rceil$, with tiles of size $b \times b$. The matrix $B \in \mathbb{R}^{n \times m}$ initially contains the m right-hand sides as columns and will be overwritten by the solution matrix X. The matrices B and X are partitioned into row tiles compatible with L. For example, if $N = 5$ we have the partitioned matrices

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} & 0 \\ L_{51} & L_{52} & L_{53} & L_{54} & L_{55} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \\ B_5 \end{bmatrix}.$$

There is one task for each (non-zero) tile of L. The L_{ii} tile on the diagonal is associated with a `trsm` task that computes $X_i \leftarrow L_{ii}^{-1} B_i$ by a call to the `dtrsm` BLAS routine. The L_{ij} tile, where $i > j$, is associated with a `gemm` task that computes $B_i \leftarrow B_i - L_{ij} X_j$ by a call to the `dgemm` BLAS routine. There are $i - 1$ `dgemm` tasks that update the same block B_i. We nevertheless allow for parallel execution of these tasks and avoid race conditions by safely computing $W \leftarrow L_{ij} X_j$ into local scratch space W and then updating the shared tile $B_i \leftarrow B_i - W$ while holding a lock associated with B_i. The overhead of protecting the matrix addition by a lock in this way is negligible considering that the preceding matrix multiplication is far more costly ($\mathcal{O}(b^3)$ flops compared to $\mathcal{O}(b^2)$ flops).

3.4.1 The critical path

The critical path (as defined in Section 3.2.2) consists of the tasks associated with the tiles

$$L_{1,1} \to L_{2,1} \to L_{2,2} \to \cdots \to L_{N,N}$$

and hence alternates between `trsm` tasks and `gemm` tasks. This implies that both of these task types must be parallelized.

3.4.2 Parallel kernels

Both the `trsm` tasks and the `gemm` tasks are trivially parallelized by partitioning the right-hand sides into column blocks with a uniform width. Each thread in the reserved set calls the `dtrsm` respectively `dgemm` BLAS routines using only its own block of the right-hand sides as an argument.

3.4.3 Usage

The source code for this example is located in the directory `src/examples/dtrsm/`. The resulting executable `test-dtrsm.x` takes five command line arguments:

1. n: the number of equations,
2. m: the number of right-hand sides,
3. b: the tile size,
4. p: the total number of threads,
5. q: the size of the reserved set.

For $q \in \{1, 2, \ldots, p-1\}$ the *fixed mode* (see Section 3.2.3) will be used. For $q = 0$ the *regular mode* will be used instead.

For example, to run the example with $n = 5000$, $m = 500$, $b = 200$, $p = 14$, and $q = 1$, use the command

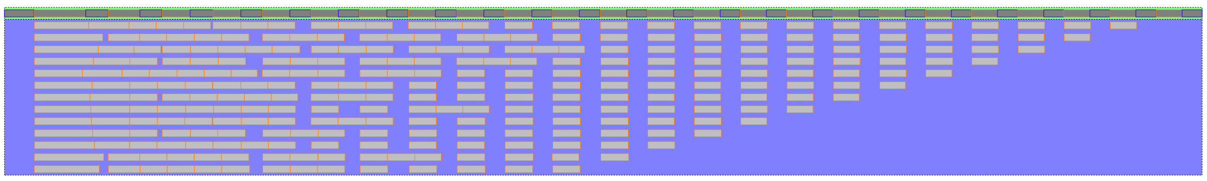


Figure 4: Trace obtained from the command `./test-dtrsm.x 5000 500 200 14 1`. (The exact cause of the above-average task execution times in the early stages of the execution is unknown but is possibly due to frequency scaling.)

```
./test-dtrsm.x 5000 500 200 14 1
```

Figure 4 illustrates a resulting trace.

3.4.4 Results

For a given problem of size (n, m) and a given number of cores p, we are interested in the question: How does the *fixed mode* $(q \geq 1)$ compare to the *regular mode* $(q = 0)$ under optimal parameter settings? In particular, will dedicating additional cores to the critical path, $q > 1$, yield any speedup? Table 2 provides answers to this question for $n \in \{2000, 4000, 6000, 8000, 10000\}$ and $p \in \{7, 14, 21, 28\}$ on the Kebnekaise[5] system at High Performance Computing Center North (HPC2N), Umeå University. Each node of Kebnekaise contains two sockets (Intel Broadwell processors) with 14 cores each for a total of 28 cores. The largest observed speedup is 1.625 (for $n = 6000$ and $p = 28$).

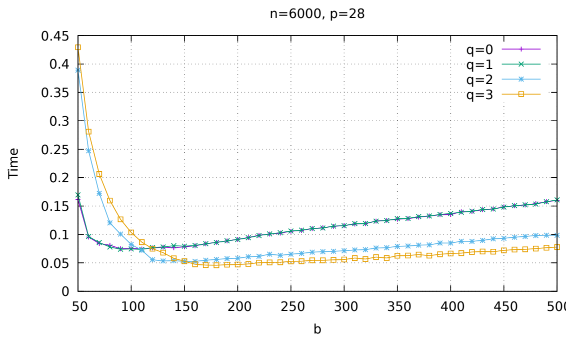
Table 2: Speedup by using (the best) $q \in \{1, 2, 3\}$ compared to using $q = 0$ for various combinations of n and p with $m = 500$. The notation $(x/y : z)$ means that x and y are the optimal tile sizes for $q = 0$ (*regular mode*) and $q \in \{1, 2, 3\}$ (*fixed mode*), respectively, and z is the optimal value for q.

	$p = 7$	$p = 14$	$p = 21$	$p = 28$
$n = 2000$	1.133 (100/190:2)	1.499 (50/110:3)	1.528 (80/70:3)	1.540 (60/130:3)
$n = 4000$	1.026 (180/360:2)	1.206 (90/170:2)	1.610 (100/160:3)	1.620 (100/130:3)
$n = 6000$	0.994 (280/280:1)	1.094 (130/240:2)	1.455 (110/230:3)	1.625 (110/180:3)
$n = 8000$	0.990 (390/410:1)	1.043 (180/360:2)	1.375 (130/300:3)	1.477 (130/230:3)
$n = 10000$	0.994 (460/490:1)	1.019 (220/380:2)	1.287 (170/380:3)	1.392 (130/280:3)

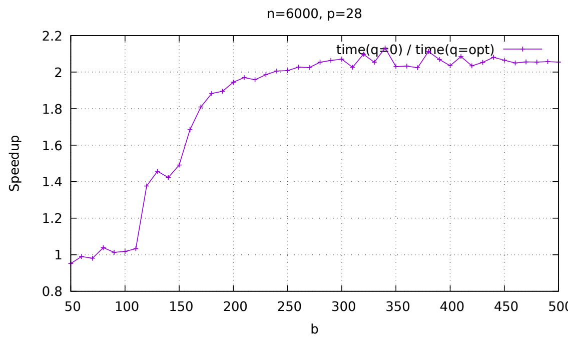
Let us take a closer look at the data behind the observation with the largest speedup ($n = 6000$, $p = 28$). Figure 5a illustrates the parallel execution times for $q = 0, 1, 2, 3$ as we vary the tile size from 50 to 500 in increments of 10. Figure 5b shows the speedup when using the best $q \in \{1, 2, 3\}$ compared to using the *regular mode* $(q = 0)$ for each tile size. According to Table 2, the optimal tile size for the *regular mode* is $b = 110$ whereas it is $b = 180$ for the *fixed mode* using the optimal $q = 3$.

The length of the longest path in the DAG can be computed after the execution based on measurements of the task execution times. These data are illustrated in Figure 6a. Figure 6b expresses the length of the longest path as a fraction of the parallel execution time (Figure 5a).

[5] `https://www.hpc2n.umu.se/resources/hardware/kebnekaise`

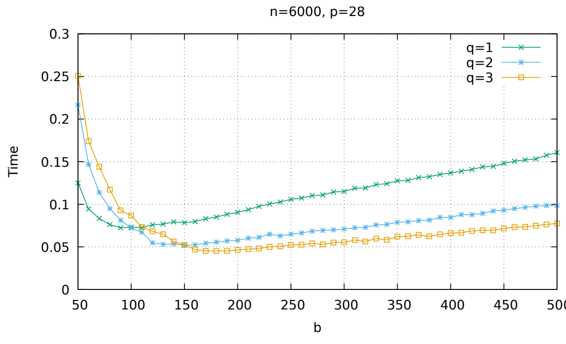


(a) Parallel execution time.

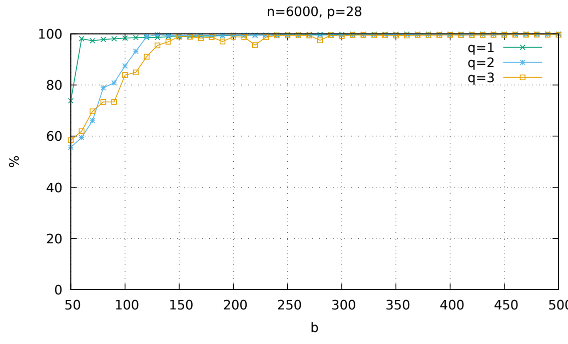


(b) Speedup by using (the best) $q \in \{1, 2, 3\}$ compared to using $q = 0$.

Figure 5: Illustration of the impact that the tile size has on the execution time of the triangular solver for various choices of q when $n = 6000$, $m = 500$, $p = 28$ (full node).



(a) The length of the longest path.



(b) The length of the longest path as a fraction of the parallel execution time.

Figure 6: Illustration of the impact that the tile size has on the length of the longest path for the triangular solver for various choices of q when $n = 6000$, $m = 500$, $p = 28$ (full node).

3.4.5 Conclusions

Figure 6a suggests that parallelizing the critical path can reduce the length of both the critical path and the longest path (as measured after the execution) provided that the tile size is sufficiently large for the parallelization to yield speedup.

The length of the critical path can limit the parallel execution time when the tile size is sufficiently large. The scheduler in the runtime system can realize an execution time that is close to this limit. This is evident from Figure 6b.

The optimal tile size can be larger than the smallest tile size for which the critical path is a limiting factor. According to Table 2, the optimal parameters for $n = 6000$ and $p = 28$ are $b = 180$ and $q = 3$. Comparing with Figure 6b, we see that for $q = 3$ the critical path becomes a limiting factor already at $b = 150$, yet the optimal tile size is larger than that.

The data in Table 2 suggests that the speedup achieved by the *fixed mode* is largest when p is large and the problem size is simultaneously neither too large nor too small. As the problem becomes larger, the *regular mode* can afford to use larger (and therefore more efficient) tile sizes without sacrificing concurrency. This can explain why the speedup drops for larger problems. As the problem becomes smaller, the *fixed mode* ends up with a smaller optimal tile size and the resulting tasks are therefore less efficiently parallelized. This can explain why the speedup drops for smaller problems.

The optimal tile sizes are almost always larger for the *fixed mode* compared to the *regular mode*, which is evident from Table 2.

3.5 Example: Cholesky factorization

Let $A \in \mathbb{R}^{n\times n}$ be a symmetric positive definite matrix. The aim is to compute its Cholesky factorization $A = LL^T$, where $L \in \mathbb{R}^{n\times n}$ is a lower triangular matrix. From the partitioned matrix equation

$$\begin{bmatrix} A_{11} & \star \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix}$$

we can construct the following recursive scheme:

1. compute the Cholesky factorization $A_{11} = L_{11}L_{11}^T$,
2. compute $L_{21} \leftarrow A_{21}L_{11}^{-T}$,
3. update $A_{22} \leftarrow A_{22} - L_{21}L_{21}^T$, and
4. continue recursively on A_{22} (i.e., repeat 1–4 on A_{22}).

With A_{11} chosen to be of a fixed size $b \times b$ and the tail recursion replaced by iteration we obtain the so-called *right-looking variant* of Cholesky factorization.

A tiled algorithm is obtained by partitioning A into tiles of size $b \times b$ and split the `trsm` (Step 2) and `syrk` (Step 3) updates into sets of independent tile updates. The corresponding DAG is constructed by identifying the data dependencies.

3.5.1 The critical path

The critical path, as defined in Section 3.2.2, consists of `potrf` on tile $A_{1,1}$, `trsm` on $A_{2,1}$, and `syrk` on $A_{2,2}$, all from the same (first) iteration. This pattern then repeats itself for all subsequent iterations (i.e., `potrf` on $A_{2,2}$, `trsm` on $A_{3,2}$, and so on). The affected tiles are those on the diagonal and first sub-diagonal.

3.5.2 Parallel kernels

The update step in the Cholesky algorithm is a symmetric rank-k update of the form $A \leftarrow LL^T$. This kernel is parallelized (see Algorithm 1) without synchronization by partitioning the columns of A into one block column per thread. Each thread independently updates its block of A. The number of flops is proportional to the number of updated elements, which means that non-uniform blocks are necessary to balance the load. For a lower triangular matrix of size $n \times n$, there are $w = n(n+1)/2$ entries to update. Given t threads, each thread should ideally process w/t entries. We can come close to the ideal partitioning by incrementally building blocks from left-to-right until the load associated with the current block reaches or exceeds w/t.

Algorithm 1: $A = \texttt{ParallelSYRK}(A, L)$

```
  **Data:** Diagonal block $A \in \mathbb{R}^{n \times n}$ to be updated using $t$ threads $\{p_1, \ldots, p_t\}$.
  **Result:** Updated diagonal block $A \leftarrow A - LL^T$.
  // Balance the load
1 Partition $A$ into block columns ▯, one per thread, such that the load per thread is
    approximately balanced;
  // Update in parallel
2 Let $j_1 : j_2$ denote the range of $m = j_2 - j_1 + 1$ columns of $A$ assigned to this thread;
3 Partition $A(j_1 : n, j_1 : j_2)$, ▯, into a triangular part, ◺, and a rectangular part, ▯;
4 Update the triangular part using dsyrk;
5 Update the rectangular part using dgemm;
6 **return** $A$;
```

The Cholesky factorization (`potrf`) kernel is parallelized with barrier synchronization and one level of look-ahead using the right-looking variant as outlined in Algorithm 2. The idea is to partition the matrix into column panels of width b (e.g., $b = 32$) and apply `potrf` and `trsm` to factor a panel and `syrk` to update the trailing matrix. The `trsm` and `syrk` computations are parallelized by partitioning the rows of the output matrix into one block per thread. The `trsm` is load-balanced by uniform blocks, whereas the `syrk` needs non-uniform blocks with similar numbers of non-zeros to be balanced. A barrier ensures that the data dependencies from `potrf` to `trsm` in the next iteration are respected. Another barrier ensures the same for the data dependencies from `trsm` to `syrk` in the same iteration. The data dependencies from `syrk` to `potrf` in the next iteration are respected without synchronization since thread p_1 performs both the `potrf` and the overlapping portion of the preceding `syrk`, so the dependencies are automatically respected by the program order.

3.5.3 Usage

The source code for this example is located in the directory `src/examples/dpotrf/`. The resulting executable `test-dpotrf.x` takes four command line arguments:

1. n: the size of the matrix,
2. b: the tile size,
3. p: the total number of threads,

Algorithm 2: $L = \texttt{ParallelPOTRF}(A)$

```
Data: Diagonal block $A \in \mathbb{R}^{n\times n}$ to be factorized using $t$ threads $\{p_1, \ldots, p_t\}$.
Result: Cholesky factor $L$ such that $A = LL^T$.
1  Choose a block size $b$ (e.g., $b = 32$);
2  $N \leftarrow \lceil n/b \rceil$ is the number of iterations;
3  Partition $A = \begin{bmatrix} A_{11} & \star \\ A_{21} & A_{22} \end{bmatrix}$ such that $A_{11} \in \mathbb{R}^{b\times b}$;
4  Apply POTRF to $A_{11}$ on thread $p_1$;
5  for $k \leftarrow 1, 2, \ldots, N-1$ do
6      Barrier;
7      Partition $A_{21} = \begin{bmatrix} A_{21}^{(1)} \\ \vdots \\ A_{21}^{(t)} \end{bmatrix}$ into uniform blocks;
8      Compute $A_{21}^{(i)} \leftarrow A_{21}^{(i)} A_{11}^{-T}$ using DTRSM on thread $p_i$ in parallel;
9      Barrier;
10     Partition $A_{22} = \begin{bmatrix} A_{22}^{(1)} \\ \vdots \\ A_{22}^{(t)} \end{bmatrix}$ into blocks with the same number of non-zeros with the
         constraint that $A_{22}^{(1)}$ must contain at least $b$ rows;
11     Re-partition $A_{21} = \begin{bmatrix} A_{21}^{(1)} \\ \vdots \\ A_{21}^{(t)} \end{bmatrix}$ conformally with $A_{22}$;
12     Compute $A_{22}^{(i)} \leftarrow A_{22}^{(i)} - A_{21}^{(i)} A_{21}^T$ using DSYRK and DGEMM on thread $p_i$ in parallel;
13     Apply POTRF to the leading $b \times b$ submatrix of $A_{22}^{(1)}$ on thread $p_1$;
14     Re-partition $A_{22} = \begin{bmatrix} A_{11} & \star \\ A_{21} & A_{22} \end{bmatrix}$ such that $A_{11} \in \mathbb{R}^{b\times b}$;
15 return $A$;
```

4. q: the size of the reserved set.

For $q \in \{1, 2, \dots, p-1\}$ the *fixed mode* (see Section 3.2.3) will be used. For $q = 0$ the *regular mode* will be used instead.

For example, to run the example with $n = 5000$, $b = 200$, $p = 14$, and $q = 1$, use the command

```
./test-dpotrf.x 5000 200 14 1
```

3.5.4 Results

For a given problem of size n and a given number of cores p, we are again interested in the question: How does the *fixed mode* ($q \geq 1$) compare to the *regular mode* ($q = 0$) under optimal parameter settings? In particular, will dedicating additional cores to the critical path, $q > 1$, yield any speedup? Table 3 provides answers to this question for $n \in \{1000, 2000, 3000, 4000, 5000, 6000\}$ and $p \in \{7, 14, 21, 28\}$. The largest observed speedup is 1.129 (for $n = 1000$ and $p = 21$). However, the optimal value $q = 1$ associated with all speedups > 1 shows that the speedups are actually not attributable to the effects of parallelizing the critical path (since all tasks run sequentially when $q = 1$).

Table 3: Speedup by using (the best) $q \in \{1, 2, 3, 4, 5\}$ compared to using $q = 0$ for various combinations of n and p. The notation $(x/y : z)$ means that x and y are the optimal tile sizes for $q = 0$ and $q \in \{1, 2, 3, 4, 5\}$, respectively, and z is the optimal value for q.

	$p = 7$	$p = 14$	$p = 21$	$p = 28$
$n = 1000$	0.990 (110/120:1)	1.071 (90/70:1)	1.129 (130/60:1)	1.026 (90/100:1)
$n = 2000$	0.910 (120/200:1)	1.003 (120/120:1)	1.022 (120/120:1)	1.022 (140/120:1)
$n = 3000$	0.899 (200/240:1)	0.971 (160/160:1)	0.988 (160/160:1)	0.986 (160/160:2)
$n = 4000$	0.897 (360/380:1)	0.952 (240/240:1)	0.995 (160/200:1)	0.997 (200/200:1)
$n = 5000$	0.867 (400/460:1)	0.954 (240/240:1)	0.983 (240/240:1)	0.986 (240/240:1)
$n = 6000$	0.856 (440/440:1)	0.956 (320/320:1)	0.965 (320/320:1)	0.980 (240/240:1)

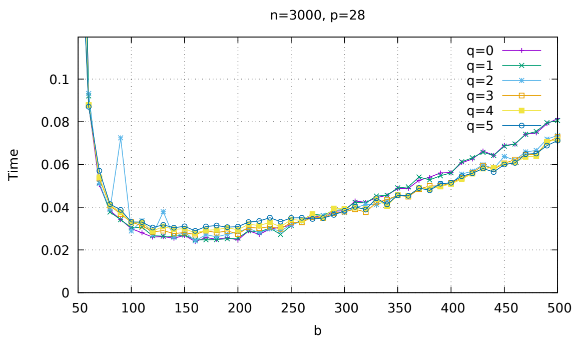
The case $n = 3000$ and $p = 28$ helps us gain insight into the lack of speedup. Figure 7a illustrates the parallel execution times for $q = 0, 1, \dots, 5$ as we vary the tile size from 50 to 500 in increments of 10. Figure 7b shows the length of the longest path as a fraction of the execution time. According to Table 3, the optimal tile sizes are $b = 160$ in both cases.

Figure 8a illustrates the length of the longest path, and Figure 8b illustrates the length of the critical path.

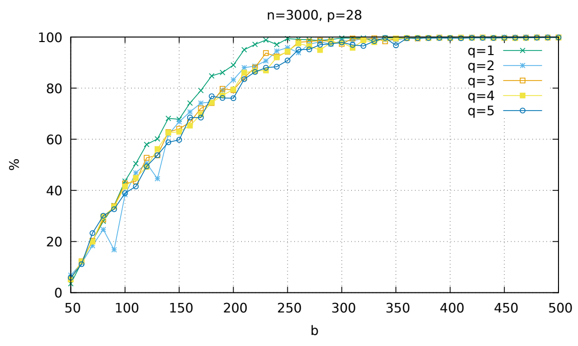
3.5.5 Conclusions

According to Table 3, no speedup from parallelizing the critical path could be observed. Small speedups are reported for some cases, but since the optimal value for q in these cases are $q = 1$, these results are actually not due to the effect of parallelizing the critical path.

Figure 8b shows a significant reduction in the length of the critical path when $q > 1$. However, according to Figure 8a this reduction has only a minor effect on the length of the longest path. Since the longest path is the factor that actually limits the performance (according to Figure 7b), there is no advantage by accelerating the critical path in this case.

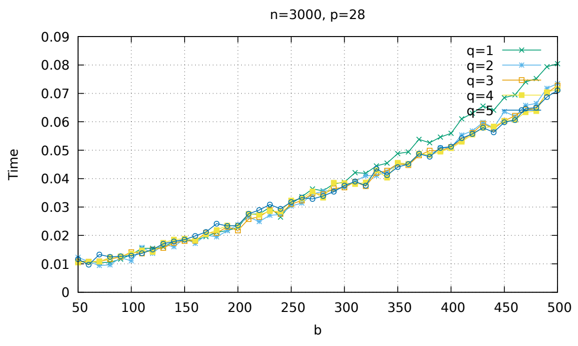


(a) Parallel execution time.

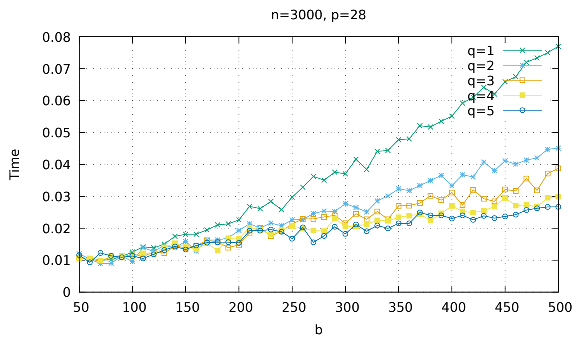


(b) The length of the longest path as a fraction of the execution time.

Figure 7: Illustration of the impact that the tile size has on the execution time of the Cholesky factorization algorithm for various choices of q when $n = 3000$ and $p = 28$ (full node).



(a) The length of the longest path.



(b) The length of the critical path.

Figure 8: Illustration of the impact that the tile size has on the lengths of the longest path and the critical path for the Cholesky factorization algorithm for various choices of q when $n = 3000$ and $p = 28$ (full node).

3.6 Implementation details

The runtime system is implemented using POSIX threads and assumes a Linux-based operating system running on a shared-memory multicore system with processors of the x86-64 architecture. Porting to other architectures and operating systems is estimated to be straightforward but not a priority since the runtime system is a tool for evaluating an idea and not meant for production codes.

The p threads are made up of one *master* thread and $p - 1$ *slave* threads. The master thread is borrowed from the user and the slave threads are managed internally by the runtime. All threads participate in the execution of tasks.

Each thread has its own thread-safe message queue consisting of a queue data structure and a condition (synchronization) variable which the thread waits on when there are no messages in its queue. An EXECUTE message is sent by any thread to demand that a particular thread executes a given ready task. Tasks are assigned to idle threads in rank order. A TERMINATE message is sent by the master to shut down a slave thread. A WARMUP message is sent by the master to demand a warmup to nominal clock frequency (to limit the negative effect of dynamic frequency scaling). A RECRUIT message is sent by the master to demand that a thread joins the reserved set. The reserved set always consists of the first q threads so that these tightly synchronized threads are physically close.

All non-critical tasks (and also all critical tasks in the *regular mode*) are scheduled for execution by being sent to an idle thread through an EXECUTE message. The critical tasks are scheduled differently. All critical tasks lie on the same path and therefore are executed one-by-one in sequence. All cores presently in the reserved set participate in the execution of the next critical task and synchronize by means of a (low-latency) barrier. To execute a task, the master writes information about the task to shared memory and waits on the barrier to signal the other threads. All reserved threads then partake in the parallel execution using the user-defined parallel implementation. Completion is signaled by waiting on the barrier a second time.

Adjusting the size of the reserved set at runtime requires an orderly replacement of the barrier to accommodate the changing number of threads. Shrinking the set is relatively easy since all the affected cores are idle at that point. The master sets up a new (smaller) barrier and waits on the (original) barrier to signal all reserved threads. The threads that remain in the reserved set continue using the new barrier whereas the other threads go back to waiting for messages. Since the reserved set consists of the first q threads, the threads with the highest ranks will leave the set first.

Growing the reserved set is more complicated since some of the threads that will be requested to join the set may be currently busy executing non-critical tasks. Waiting for these threads to finish would effectively stall the critical path for an unknown duration. Instead, the reserved set grows in an asynchronous fashion in two steps. First, the master sets up a new (larger) barrier and sends it via a RECRUIT message to all threads that should join the set. This is an immediate operation and the master continues without any delay. As soon as a thread receives (after some delay) a RECRUIT message, it signals the master that it is available and waits on the new (larger) barrier. When the master eventually detects that all recruits are available, it coordinates the replacement of the barrier with all previously reserved threads and all new recruits. More precisely, the master waits on the (old) barrier to signal the previously reserved threads and then waits on the (new) barrier to signal all recruits.

To reduce the measurement noise caused by dynamic frequency scaling, all cores are warmed up to (at least) the nominal clock frequency immediately prior to starting the execution of tasks.

3.7 Conclusions and future work

The idea of parallelizing the critical path works well for some types of matrix computations (e.g., triangular solves) while not at all for others (e.g., Cholesky factorization). In general, the idea has potential for a particular computation if shortening the critical path also shortens the longest path. Interestingly, this is more likely to happen for computations that are more difficult to parallelize. Indeed, for a perfectly parallel computation the idea would be useless since every path is a critical path. At the other extreme, the idea would work optimally for a linear graph since there is only one path.

Future work includes applying the idea to other types of matrix computations. Based on the findings presented here, the QR algorithm for Schur decomposition and eigenvalue reordering algorithms are candidates that are likely to benefit.

Developing an effective control logic for the *adaptive mode* is another direction for future work. To what extent mature runtime systems such as StarPU are capable of supporting the idea and what (if anything) needs to be done to improve the support will be investigated if the idea proves to be beneficial after more realistic examples have been studied.

Finally, extensions to distributed-memory can be considered. Linda Sandström has made a preliminary study of the idea in an MPI setting using triangular solve as an example in her Master's Thesis conducted at Umeå University. Her results and conclusions warrant further investigation of an MPI extension and the associated open questions she identified.