On Runtime Systems for Task-based Programming on Heterogeneous Platforms

Samuel Thibault

Mémoire

présenté à

L'Université Bordeaux

École Doctorale de Mathématiques et Informatique

par Samuel THIBAULT

Maître de conférence
Laboratoire Bordelais de Recherche en Informatique
Équipe Supports et AlgoriThmes pour les Applications Numériques hAutes performanceS
Thème Supports Exécutifs Haute Performance
Équipe-projet commune avec Inria: Static Optimizations - Runtime Methods

en vue d'obtention du diplôme d'

Habilitation à Diriger des recherches

Spécialité : Informatique

On Runtime Systems for Task-based Programming on Heterogeneous Platforms

Soutenance le : 12/12/2018

Après avis de :

M.	Julien	LANGOU	Professor and Chair	Rapporteur
Mme	Elisabeth	LARSSON	Senior Lecturer	Rapporteuse
M.	Denis	TRYSTRAM	Professeur des universités	Rapporteur

Devant la commission d'examen formée de :

M.	Albert	COHEN	Research Scientist chez Google AI	Examinateur
M.	Philippe	DUCHON	Professeur des universités	Examinateur
M.	Julien	LANGOU	Professor and Chair	Rapporteur
Mme	Elisabeth	LARSSON	Senior Lecturer	Rapporteuse
M.	Pierre	MANNEBACK	Professeur	Examinateur
M.	Christian	PEREZ	Directeur de Recherches	Examinateur
Mme	Isabelle	TERRASSE	Expert exécutif en mathématiques appliquées, AirBus Group Innovations	Examinatrice
M.	Denis	TRYSTRAM	Professeur des universités	Rapporteur

To whom we will pass the torch to,

Acknowledgements

It is complex to thank people over a 10-year time frame, but it's also a sincere pleasure to do it.

I would like to start with warmly thanking all the members of the committee, who accepted to assess my work, and in particular the reviewers. It is a honor to have you in the jury, I am glad that you all managed to be part of it despite your busy schedules. I also want to thank Lionel and Loris who happily proofread the scheduling parts.

A special thank goes to Raymond, who built the Runtime team in Bordeaux, attracted brilliant students and colleagues, and instilled a warm spirit in the team which kept up in the children teams STORM and TADaaM.

Many thanks go to Denis, Pierre-André and Raymond, who kept providing me with precious advices. "Advisor" seems to be a lifetime title/duty/burden/?

A specific thank goes to Nathalie, who manages to put up with my former student's oddly-written source code, and to bring StarPU to production-level quality. Big thanks to Olivier for all the help along the years, and who eventually joined the StarPU game. More generally, thanks to the numerous interns and engineers who dared to put their hands around or even inside StarPU. Your contribution credits are carefully recorded by Nathalie.

I want to give individual thanks to my former and current PhD students. A PhD is always a long journey, but it is a real pleasure to see you keep growing up to and after the defense. Thanks Cedric for your gem, you have left me with what could look like a lifetime TODO list. The people reading your thesis often say "wow, all the ideas were already in there", and I can't disagree. Thanks Paul-Antoine for daring trying to put dynamic scheduling in embedded environments. We had a hard time doing so, but we have learnt a lot and built interesting models. Thanks Corentin for playing with tiny tasks, this ended up as an interesting approach to compare with. Thanks Marc for your deep work with distributed execution, and I hope we have helped you with managing your stress :) Thanks Suraj for your incredible work at being a bridge between three very different domains, this bridge will keep florishing now that it is set up. Thanks Romain for joining us, the story is only starting, but I see that you already have good intuitions.

I send a warm thank to the research teams immediately involved in my work, notably SATANAS/STORM/TADaaM/HiePACS and REALOPT. It is a real pleasure to work in such a friendly environment, with people always ready for impromptu chating on research or non-research. I also want to give a special thank to our team assistants: we unfortunately never cite you in our research papers, but without your precious work no science would be achieved. A thank also goes to the people who maintain the hardware platforms and put up with our odd accelerators, network cards etc. and the corresponding crazy software stacks.

More generally, I broadcast a thank to LaBRI and INRIA colleagues. The general labs mood is very positive and full of events and stories. This is a ground for excellent science.

A big thank goes to all colleagues with whom I happily collaborate. Our ideas emerge from discussions, sometimes out of very different grounds and expertises, in the end we never remember who actually spoke the ideas, what's important is working on them and telling others about them together.

A sincere thank goes to `https://www.thesaurus.com/` . When I first met a thesaurus in middle school I wondered how it could be useful at all, I now have to say that it is basically bound to a corner of my desktop.

I must give a thank to libre software. It has provided me with a lot of material to study and learn from, play with, contribute to. I can't remember something I took at look at, as fortuitous as it may have seemed at the time, that didn't end up being useful at some point for my work, one way or the other. This common good is there to stay, for everybody's benefit, the *software heritage* slogan is really not just a catchword.

Un grand merci aussi aux différents orchestres et groupes dans lesquels j'ai pu jouer de très divers styles de musique sur divers instruments, histoire de faire autre chose que de taper sur un clavier (d'ordinateur). Une mention spéciale à Manu qui ne s'est jamais lassé de corriger mes défauts au trombone.

Un petit mot aussi pour mes différents amis, ici et là, que j'arrive à voir plus ou moins souvent, parfois cela se compte en années, mais c'est toujours avec autant de plaisir.

Et enfin merci à ma famille, qui m'a offert le cadre qui m'a lancé jusqu'ici, en m'apprenant la curiosité, la ténacité, la générosité, ...

Contents

Chapter 1

Introduction

SIMULATION has become pervasive in science. Real experimentation remains an essential step in scientific research, but simulation replaced a wide range of costly and lengthy or even dangerous experimentation. It however requires massive computation power, and scientists will always welcome bigger and faster computation platforms, to be able to keep simulating more and more accurately and extensively. The HPC field has kept providing such platforms but with various shifts along the decades, from vector computers to clusters. It seems that the past decade has seen such a shift, as shown by the top500 list of the fastest supercomputers. To be able to stay in the race, most of the largest platforms include accelerators such as GPGPUs or Xeon Phi, making them heterogeneous systems. Programming such systems is significantly more complex than programming the homogeneous platforms we were used to, as it now requires orchestrating asynchronous accelerator operations along usual computation execution and communications over the network, to the point that it does not seem reasonable to optimize execution by hand any more.

A deep trend which has emerged to cope with this new complexity is using task-based programming models. These are not new, but have really regained a lot of interest lately, showing up in a large variety of industrial platforms and research projects using this model with various programming interfaces and features. A key part here, that is however often forgotten, misunderstood, or just ignored, is the underlying runtime system which manages tasks. This is nonetheless where an extremely wide range of optimization and support can be provided, thanks to a task-based programming model. As we will see in this document, this model is indeed very appealing for runtime systems: since they get to know the set of tasks which will have to be executed, which data they will access, possibly an estimation of the duration of the tasks, etc., this opens up for extensive possibilities, which are not reachable by an Operating System with the current limited system interfaces. It is actually more and more heard in conference keynotes that runtime systems will be key for HPC's continued success.

Thanks to such rich information from applications, runtime systems can bring a lot of questions on the desk, in terms of task scheduling of course, but also transfer optimization, memory management, performance feedback, etc. When the PhD thesis of Cédric Augonnet started in 2008, we started addressing a few of these questions within a runtime system, StarPU. During the decade that followed, we have deepened the investigations, and opened new directions, which I will discuss in this document. We have chosen to keep focused on the runtime aspects, leaving a bit on the side for instance programming languages which can be introduced to make task-based programming easier. The runtime part itself indeed did keep providing various challenges which show up in task-based runtime systems in general. These challenges happen to be related to various other research topics, and collaboration with the respective research teams has then only become natural: task scheduling of course, but also network communication, statistics, performance visualization, etc. Conversely, the existence of the actual working runtime system

StarPU provided them with interesting test-cases for their respective approaches. It also allowed for various research projects to be conducted around it, without direct contribution to StarPU.

Structure of the document

This document discusses the contributions I have been working on in the past decade, hand in hand with PhD students and close colleagues, as well as with various research teams, thus the pervasive use of "we" in the text. It also presents some contributions, that I have not directly worked on, from close colleagues. In the bibliography, I have separated out the publications I am co-author of (marked with numerical references) from publications I am not co-author of but have revolved closely to StarPU.

In Chapter 2 we will discuss the task graph programming model itself. It will be the opportunity to introduce the context of its resurgence. We will present a programming paradigm which a lot of project have eventually converged to, which we call Sequential Task Flow (STF), and will be used throughout the document. We will then discuss how a notion of runtime system fits in the picture, actually taking an essential role, and briefly present a panel of runtime systems. The programming interface itself will be discussed: how it provides portability of performance, at the expense of making programmers change their habits (but for good reasons in general anyway), and whether we can hope for a common interface that could get consensus. We will consider a few interesting extensions before giving examples of application cases where task-based programming was really successful, and examples of application cases for which using it is questionable.

In Chapter 3 we will really dive into the runtime system itself. We will first discuss task scheduling of course, notably how schedulers can be implemented within a runtime, how well heuristics get confronted to reality, and how we could build a bridge from theoretical scheduling work. We will then discuss the management of the limited amount of memory embedded in GPUs, but also of the main memory itself. We will examine how commutative data access can be optimized for some application cases. We will then consider the concerns of the size of tasks, and how we can manage it by either aggregating tasks, dividing them on the contrary, or using parallel tasks. A completely new direction for general-purpose task-based runtimes will then be introduced: simulating the whole execution of the runtime, so that application performance can be very accurately predicted. We will describe a wide range of entailed potential, one of which is providing guarantees or verification. We will eventually discuss the potential for performance profiling and debugging tools.

Chapter 4 is dedicated to the case of distributed execution. Master/slave designs will first be discussed, and then how we can insist on keeping an STF design. We will then describe various important optimizations we have found essential to achieve proper performance: caching data, coping with the defects of MPI implementations, and avoiding submission overhead. We eventually discuss future work on leveraging the STF principle for distributed execution: dynamic redistribution, collective operations optimization, and resiliency to hardware faults.

The last Chapter 5 will focus on the implementation itself. We will explain how we managed to assemble various research contributions into a single software project. We will discuss a way (not) to design features and a few noteworthy implementation questions. We will eventually consider the general maintenance of a runtime system for research.

Chapter 2

Using a task-based programming model for HPC

Contents

In this first chapter, we will discuss the usage of the task-based programming model in general for HPC, with a point of view rather on the programmer side.

We will first introduce how and why this programming model has (at last!) become a trend in HPC with the introduction of programming interfaces and languages. We will then discuss the position of runtime systems in the picture, which at the same time are often somewhat forgotten, hidden behind programming languages, and also have a wild diversity of implementations. We will then question the programming model itself and how the runtime studied in this document, StarPU, fits in the picture. We will consider how it makes programming different but effective, whether a common programming interface is possible, and how flexible it should be. We will eventually consider successes and limits of the model.

2.1 The revival of a well-known paradigm

Task graphs have been used to model and optimize parallel execution since the very early history of computers [Cod60]. This was however rather applied to whole jobs rather than application-defined small tasks. It was proposed as a generic-purpose programming model for applications much later, for instance with the Jade [RSL93a] language (1993) which used a task-based model for programming heterogeneous clusters. The mainstream parallel programming interfaces at the time were still mostly following MIMD and SIMD paradigms (PVM, MPI, and the first versions of OpenMP). This was still showing up while I was taking a parallel algorithms course in early

2000's: while a fair part of the course was spent on task graph scheduling, practice sessions were limited to SIMD paradigms.

Task graph programming models have however recently become a real practical trend, as shown by the flurry of runtime systems proposed by the research community, but also the definition of industrial standards such as OpenMP which introduced tasks in version 3.0 in 2008, and task dependencies in version 4.0 in 2013.

2.1.1 Getting away from the hazards of shared-memory or message-passing

A tendency of HPC programmers is to take a tight control on the hardware through low-level programming layers, so as to get the best performance. With such perspective, shared-memory parallel programming with explicit synchronizations is a natural way of programming parallel systems. This approach is however inherently hard, due to having to deal with all concurrency issues. When D. Knuth was asked, at the Sorbonne Université colloquium in 2014, "Can we make parallel programming easy?" he answered a mere "Err... no.", and got an audience laughter. Concurrency indeed makes the number of potential scenarii explode, and programmers can not manage considering all of them without a lot of training (and even with a lot of training). OpenMP was proposed in late 90's to make parallel programming easier, but it does not really solve the concurrency issues. One can think of thread programming as the assembly language for parallelism, and early versions of OpenMP only as a macro system on top of it.

Message-passing interfaces such as MPI [CGH94] have been proposed and widely used. They really bring progress over threads, since they make people think and synchronize in terms of explicit messages rather than implicit shared-memory concurrency, making it way easier to get it right. Getting the best performance however comes at a price. Using the mere synchronous `MPI_Send/Recv` operations typically leads to a lot of idle time while waiting for messages from other nodes. Using the asynchronous `MPI_Isend/MPI_Irecv` primitives allows to make better use of both the communication network and computation units at the same time (overlapping communication with computation), but getting it right and efficient remains a challenge. The recent introduction of one-sided communications actually brings back the concurrency complexity of shared-memory programming.

Task graph programming actually proposes instead to get back to that pure message-passing programming style. Asynchronicity is completely handled automatically by the task-based runtime instead of being explicitly managed by programmers. More precisely, while with MPI programmers bind computations to MPI processes and thus processors, with task graphs programmers bind computations to tasks, which can be dynamically scheduled on processors. That alleviates the problem of tasks waiting for messages: they are simply marked as non-ready and other tasks can be scheduled.

In our previous work on thread scheduling [TNW07], one of the main issues was that the thread programming interface typically does not let the application tell the future of a thread to the runtime. A thread could for instance just terminate exactly when the scheduler tries to migrate it within the system to compensate an observed load imbalance. With task graphs, knowledge of the future is provided naturally to the runtime, which can then be proactive and wisely schedule tasks even before they start executing, instead of remaining mostly reactive.

Task-based programming thus seems like a promising parallel programming paradigm, and it has indeed recently been adopted more and more.

2.1.2 A real trend toward task-based programming

During the 2000's, the wide generalization of multicore systems has firmly raised the question of bringing parallel programming to the average programmer. As discussed above, shared-memory and message-passing programming as used by HPC do not seem to be a reasonable answer any more. The industry for instance has instead introduced various task-based programming interfaces, such as TBB[1] or Grand Central Dispatch[2], which are now widely used. An even simpler programming interface is MapReduce [DG08], which abstracts yet more the notion of task down to a mere pair of map function and reduce function. As mentioned previously, the OpenMP standard[3] which was originally essentially a shared-memory programming language, introduced tasks with dependencies, thus making it effectively a task-based language as of version 4.0.

Academia tends to agree that this is indeed the future of parallel programming, as explained by Jack Dongarra[4]. His original LINPACK [DBMS79] library was designed for sequential but vector-parallel systems. It was extended into LAPACK [ABD+90a] which used cache-aware panel computation for shared-memory parallel systems, and then extended further into ScaLAPACK [CDD+96] for distributed-memory parallel systems. Nowadays, this turned into PLASMA [BLKD09a] which migrated to using tasks scheduled dynamically by the Quark runtime system [Yar12] to get full parallelism and pipeline, and is now even being migrated to just use OpenMP tasks [YKLD17]. Similarly, the FLAME [GGHVDG01] environment relies on the SuperMatrix [CVZB+08a] runtime system.

For linear algebra, tasks are indeed a particularly natural way of programming: basically, each whole blocked BLAS operation [LHKK79b] can be simply made a task. More fine-grain operations such as pivoting pose more questions, but while previous work on using a thread-based runtime for sparse operations was tricky [FR09], our recent work on using a task-based runtime for the same application worked very well [LFB+14]. Beyond linear algebra application, we have also seen successes with a lot of application classes, detailed in Section 2.8, but also issues with other application classes, detailed in Section 2.9.

Overall, task-based programming thus seems to have gained a large attention. Programming interfaces themselves seem to have somehow converged to a style which we here call STF, decribed below.

2.1.3 A "submit-and-forget" paradigm: the Sequential Task Flow (STF)

How the task graph should be expressed is a first question. A straightforward approach would be to express it just like $G = (V, E)$: have the application express tasks, and then dependencies between tasks. This is actually the interface that our runtime system StarPU [ATNW11a] initially proposed. This is however very tedious for users, while task dependencies can actually be automatically inferred from data dependencies, by assuming the sequential consistency order [Lam79].

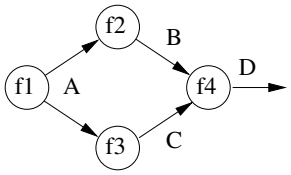
For instance, on Figure 2.1(a), *data access modes* of functions have been specified, and show that calling $f2$ and $f3$ has to be done after $f1$ has finished writing to data A, since they read A that $f1$ produces. Similarly, calling $f4$ has to happen after both $f2$ and $f3$ are finished, since it reads B and C that they produce. This automatically leads to the task graph shown on Figure 2.1(b).

The StarSs [PBAL09a] research team calls this Task SuperScalar[5]. The situation is indeed similar to the case of nowadays' super-scalar processors. With such processors, when programmers

[1] `https://www.threadingbuildingblocks.org/`
[2] `https://apple.github.io/swift-corelibs-libdispatch/`
[3] `https://www.openmp.org/`
[4] `https://www.hpcwire.com/2015/10/19/numerical-algorithms-and-libraries-at-exascale/`
[5] thus the name StarSs: Star SuperScalar

$f1(A_W)$
$f2(A_R, B_W)$
$f3(A_R, C_W)$
$f4(B_R, C_R, D_W)$

(a) STF sequential source code.



(b) Task graph generated from STF.

Figure 2.1: Example of task graph generated from sequential source code.

```
for (k = 0; k < NT; k++) do
  POTRF (A[k][k]);
  for (m = k+1; m < NT; m++) do
    TRSM (A[k][k], A[m][k]);
  end for
  for (n = k+1; n < NT; n++) do
    SYRK (A[n][k], A[n][n]);
    for (m = n+1; m < NT; m++) do
      GEMM (A[m][k], A[n][k], A[m][n]);
    end for
  end for
end for
```

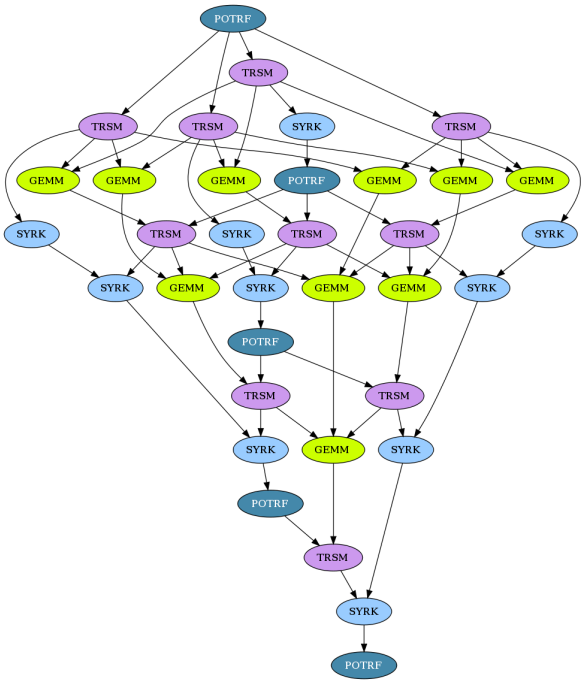


Figure 2.2: Tiled Cholesky factorization and the resulting task graph.

write a sequential-looking series of instructions which happens to contain instructions which can be run in parallel, the processor can detect this because it exactly knows register dependencies between instructions. With a task super-scalar programming interface, programmers write a sequential-looking series of task submissions with some data access annotations, and the runtime can properly detect the inherent task parallelism by inferring all implicit task dependencies from the data dependencies. In the end, the programmer writes sequential-looking code and the runtime manages to find parallelism. A lot of different task-based programming interfaces have actually converged to this kind of submit-and-forget programming style. We have called this the Sequential Task Flow (STF) [2], by analogy to DataFlow denominations and to emphasize the sequential-looking source code.

Figure 2.2 shows how the Cholesky factorization can be implemented with such a paradigm, and the resulting task graph on a small instance. This is actually looking exactly like a blocked sequential implementation of the algorithm, which is rather easy for an average programmer to write, and still exposes a lot of parallelism since the series of GEMM operations can be pipelined to very large extents.

Of course, programmers have to write code which happens to actually exhibit parallelism, but they do not have to express it explicitly, which makes a substantial difference in writing cor-

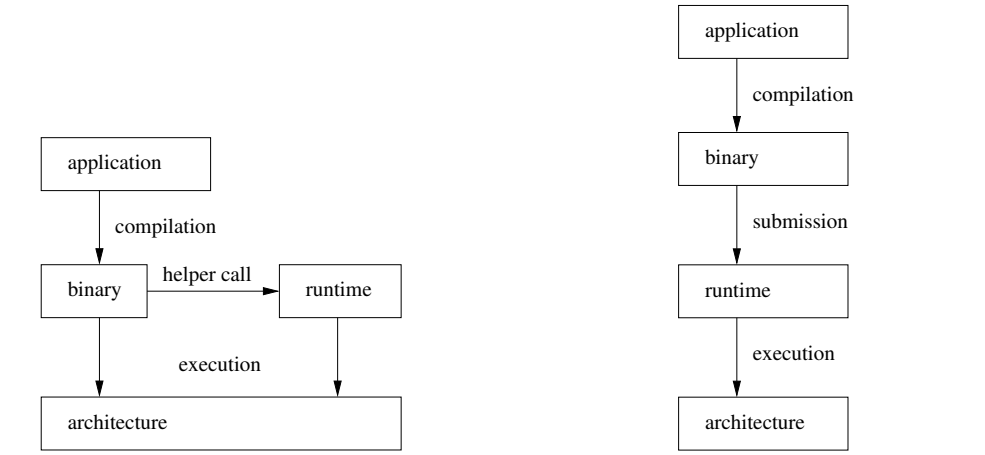
```
TRSM(k, m)

// Execution space
k = 0   .. NT-1
m = k+1 .. NT-1

// Task Mapping
: A[m][k]

// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? A[m][k]
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
        -> A[m][k]

BODY
  trsm( A /* A[k][k] */,
        C /* A[m][k] */ );
END
```

Figure 2.3: TRSM part of the PTG version of the tiled Cholesky factorization.

rect code: provided that the execution in sequential order yields the correct result, the inferred parallel execution will provide the correct result too, whatever the task scheduling. Application programmers can thus concentrate on writing correct algorithms, and just execute with a runtime to see how well it gets parallelized.

The other very interesting consequence of this programming interface is that the whole main program can be made to only asynchronously submit tasks. This means that the submission time and the execution time are completely decoupled, the application can for instance just submit all tasks and wait for the execution of the whole task graph, which is completely managed by the runtime. The time interval between task submission and actual task execution can be used to achieve various optimizations in-between, such as triggering data prefetches (discussed in Section 3.2.1), executing on a distributed-memory system (discussed in Section 4.2), or optimizing collective operations (discussed in Section 4.7). Submission can be also done progressively by submitting a set of tasks (e.g. a few applications iterations), and waiting for the execution of a part of this set before submitting more tasks, etc. This permits for instance to reduce scheduling costs by using incremental scheduling (discussed in Section 3.1.3.2), or to throttle memory use (discussed in Section 3.2.3), or to balance its load (discussed in Section 4.6),

A very different way to submit the task graph is using Parameterized Task Graphs (PTG) [CL95b, DBB+14], as used for instance by the PaRSEC runtime [BBD+13a]. The idea is to have the application express the task graph in a parameterized way. Figure 2.3 shows for instance a part of the Cholesky factorization, the TRSM tasks. We here express that TRSM tasks are indexed by k and m, and that they read the results of the POTRF tasks and GEMM tasks, and that they provide their result to SYRK and GEMM tasks. It hence means expressing dependencies explicitly, but this is done in a parameterized way: TRSM tasks are not specified one by one, but algebraically through indexes k and m. From such an algebraic representation of the task graph, the runtime can achieve interesting automatic optimizations, for instance decide at will which way to unroll the graph, determine which data will never be used again, communication patterns, good spots for checkpoint snapshots, etc. Whether these optimizations get significantly more performance than optimizations achievable with only the STF representation is however still an open question. Our comparison [22], which will be detailed in Section 4.2 and shown on Figure 4.5 page 77 does not exhibit a definite advantage of one over the other. Even optimized collective operations, which are definitely a must for very large-scale systems, and are easy to infer from the PTG, should also be discoverable from the STF between submission time and execution time through opportunistic cooperative sends, as will be discussed in Section 4.7.



(a) Software stack using a runtime on the side.

(b) Software stack fully based on a runtime.

Figure 2.4: Divergence on the position of a runtime in the software stack.

Writing such a PTG source code is relatively complex. PaRSEC has support for translating from STF source code to PTG source code [BBD+12a] by using a polyhedral analysis of the loop nest, with the inherent limitations of such kind of analysis.

Another popular way of expressing task graphs is using dividable tasks, which was notably introduced by the Cilk language [BJK+95]. Dividable tasks will be discussed in Section 3.4.2

In this document, we mainly use the STF programming paradigm, but more precisely we will discuss the underlying *runtime* layer.

2.2 An oft-forgotten software layer: the runtime system

Software stacks usually include a runtime layer, even if it is very often rather placed on the side of the picture, as shown on Figure 2.4(a). Even compilers usually stuff some runtime code along the generated machine code, to e.g. handle corner cases. The OpenMP language itself does require some runtime pieces to handle thread teams, even if the language was initially on purpose made so that the runtime can remain very trivial.

With a task graph programming model, the runtime part can not be so trivial, since it has to manage at least dependencies between tasks, and it is usually extended to handle data transfers, optimize tasks scheduling, etc. The situation ends up showing up rather like Figure 2.4(b) where the runtime takes a central role during the execution, separating clearly between task submission and task execution.

In this Section, we will describe how a runtime layer happens to take such place progressively more or less easily, but also why making it a really separate component, possibly taken from an external project, is so far not a common thing, resulting in the flurry of implementations described in next Section 2.3. Put another way, the formalization of what a task-based runtime can be, so that it may be externalized and thus shared, does not seem to have really happened in the community's minds yet.

2.2.1 Optimizing by hand brings performance (unfortunately)

Generally enough, optimizing application execution is first done by hand, at least to observe and understand the way optimizations should be done and the potential benefits they can have. This is true both for people who plan to implement automatic optimizations (to understand how much can be saved), and for people who will use the automatic optimizations (to understand what to express to make automatic optimizations possible). Unfortunately, the tendency to keep optimizing by hand is strong.

There is no denying that optimizing programs by hand does bring performance improvement. CUDA programming tutorials, for instance, start with teaching synchronous kernel execution, and then explain how to submit kernel execution and data transfers asynchronously, to improve performance. This step is essential for the programmer to understand how asynchronous programming makes better use of the hardware. But furthermore, the resulting benefits make the experience really enjoyable for the programmer.

This is however one of the reasons why programmers favor keeping control of the execution flow. Because they for instance know the hardware they are running on, they would keep a strict ordering of operation, or a more or less static split of work between heterogeneous processing units, or use Bulk Synchronous Programming (BSP) [Val90], or at best establish a pipeline, but constructed by hand. For instance, during our work with CEA (the French Atomic Energy Commission) on large-scale distributed Cholesky factorization (which will be described in Section 4), results obtained with a completely dynamic runtime have shown that the hand-made pipeline used by CEA was missing some performance because it was not aggressive enough. They have just modified their pipeline to get the extra performance, and keep the control with it, instead of migrating to using a completely dynamic runtime. The availability of the completely dynamic runtime was interesting to show which additional optimizations are possible, but in the end these were integrated by hand. It however allowed to determine how much was to be gained, which could then be compared to how much engineering time was needed to implement the optimization, and thus made that happen in the end.

That being said, the CEA implementation is actually based on an internal dynamic runtime. It is however a rather simple one, which allows to keep a lot of control on how it behaves (but also prevents from easily trying optimizations, as mentioned above). Actually, there seems to be a sort of tipping point between such kind of simple runtime, and a really dynamic runtime on which we do not really have control any more. For a lot of applications, their programmers have optimized the execution by hand, then abstracted optimizations enough to actually contain an internal runtime. Such runtime however usually provides a limited set of features, actually similar to what OpenACC proposes, which is still very directive, typically rather an offload engine rather than using a really task-based paradigm. Moving to a runtime, whose dynamism means losing most control over it, seems to be a decisive step.

Replacing a well-controlled execution engine with a fully dynamic engine is indeed questionable: will the latter obtain good-enough performance compared to the existing carefully-tuned implementation? Probably, a general-purpose runtime will not be able to catch all optimizations that an engineer could write by hand, but it would save the time spent on implementing what the runtime can achieve. Unfortunately, very often there already is an optimized implementation, whose author is not keen on throwing away, even if the general-purpose runtime happens to provide some improvement, since it was so much fun spending time on improving performance by hand. The general-purpose runtime needs to provide really striking performance improvement, in order to be adopted. This is basically *loss aversion* coming into play: had the implementation been based on a general-purpose dynamic runtime from the start, the deal would have been different. This can be seen for instance in some work on data analysis [POP16] which started from a very statically-scheduled behavior and introduced some dynamism, but did not really let it loose with real classical list scheduling.

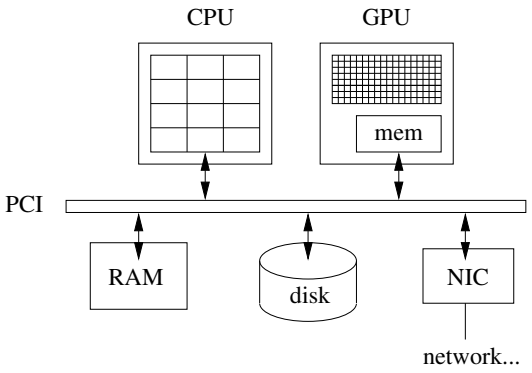


Figure 2.5: Not-so-uncommon system architecture, with a few CPU cores, a GPU, the main RAM, some disk, and a network card to other nodes of the network.

More generally, it seems like a question of trusting the dynamic runtime's abilities. Nowadays we do trust compilers and almost never write assembly code. Even further, Domain-Specific Languages (DSLs) are used to write applications for a specific domain in concise terms, and a compiler is trusted to generate appropriate source code, to be compiled by an underlying compiler. We do not seem to have reached this stage of trust with dynamic runtimes so far, even if optimizing by hand is more and more questionable.

2.2.2 Can we continue optimizing by hand in the long run?

As detailed in the previous Section, programmers tend to try to keep optimizing performance by hand. It is however questionable whether this is sustainable in the long run. The question at stake is *performance portability*: with new, very different hardware, how much of the software will have to be rewritten?

For instance, parallelizing with POSIX threads or OpenMP may have seemed appealing to fully control parallelization, but in the past decade, General-Purpose Graphics Processing Units (GPGPUs, also called just GPUs) have notably appeared and exhibit a very different low-level programming interface, as well as performance heterogeneity compared to the CPU. Optimizing then requires balancing the load between CPUs and GPUs, handling the limited amount of discrete memory of GPUs, overlapping computation with data transfers, etc.

Some programming environments allow to get such support achieved automatically, for instance the CUDA virtual memory (GMAC), CUDA unified address space, PGAS languages, etc. but optimizations achieved automatically are usually hindered by the lack of knowledge of the future, preventing for instance fully-optimized data prefetches. Put another way, such programming interfaces typically do not let applications fully give up control to an underlying runtime: the control flow remains coordinated by the application.

In addition to GPU support, it is very desirable to support parallelism over the network, and thus manage e.g. MPI transfers. Some use cases may also process more data than can fit the main memory, and thus Out-of-Core techniques must be introduced to store data on the disk. NUMA memory nodes, as seen in standard multicore systems, but also seen with the recent KNL [Sod15] processors' fast-memory MCDRAM, require careful distribution of data. Remote data storage may also be available with various levels of size and performance, and so on. We end up with a very complex situation such as shown on Figure 2.5, with various computation and data transfers to be carefully scheduled to avoid any idle time. Setting up all of this by hand is not reasonable any more.

This was for instance observed in the case of the Magma [TDB10] linear algebra library, which

was initially made to perform careful scheduling of computation and data transfers in a static way to fully exploit one GPU for massive parallel computation concurrently with the CPU for more complex computation [ADD+09a]. When multiple GPUs became widespread, and to support matrices beyond the size of the GPU, Magma migrated to using dynamic scheduling instead.

2.2.3 Optimizing through a runtime eventually emerges, though usually internally only

Introducing the notion of a dynamic runtime is thus a natural move when facing the complexities of nowadays systems (even if there often remains resistance, as was explained in Section 2.2.1).

Ideally, programmers would use an existing generic-purpose runtime system which already properly implements all optimizations that they need. The application can then boil down to two parts: the computation part, made of independent computation kernels, and the task submission part, which only expresses which kernels need to be run on which data. The whole execution control is then left to the external runtime, allowing all kinds of automatic optimizations. To make end-user programming simpler than with a generic-purpose interface, an intermediate domain-specific layer can be designed, to generate task submission details from the application's simpler high-level domain-specific stanzas.

The move to using a dynamic runtime is however usually rather initiated from existing dynamic support which was progressively added to applications. While introducing more and more dynamism, programmers typically abstract it more and more from the rest of the application, up to extracting a generic runtime layer which can actually be used independently. Only then the question of comparing it with existing runtime systems shows up. Replacing it with one of them is then typically rejected with scientifically-weak arguments, such as a feature missing from existing runtimes (even if it could just be contributed, we will discuss this in Section 5.6), or the complexity of generic-purpose interfaces (which could be hidden behind a domain-specific layer). Again, only very significant performance improvement over the programmer's own runtime can beat *loss aversion*, which leads to a wide variety of runtime implementations.

2.3 A flurry of home-made task-based runtime systems

We have seen in the past decade a huge flurry of task-based runtimes, a lot of which actually emerged from existing software which became more and more dynamic. There are actually so many of them that taxonomies have even appeared [PBF10, TDH+18].

It is questionable whether comparing them according to their respective features is really meaningful, since they usually keep evolving, and get extended with the features they would be previously lacking. It can however make sense for a given application to choose a given runtime for e.g. its efficiency even against very small tasks, because that characteristic is conflicting with e.g. genericity, which could conversely be sought after for other applications. It thus makes more sense in the long run to distinguish runtimes by their long-run goals which entail the compromises chosen for them: genericity vs fine-grain task efficiency vs customizability, etc.

In this Section, we will thus rather present the origins and goals of a few runtime systems, with domain-specific target, language target, or generic-purpose target. We will then present in more details the runtime system discussed in this document, StarPU.

2.3.1 Domain-specific runtime systems

As was explained in Section 2.2.3, dynamic runtime can grow naturally out of domain-specific software. We here only take a few instances from dense linear algebra.

TBLAS [SYD09b] and Quark [YKD] were designed at the University of Tennessee, Knoxville, to schedule tasks of dense linear algebra applications, so as to achieve for instance better task graph pipelining, but also to make writing these applications much simpler since it becomes a matter of submitting a graph of BLAS calls. Similarly, at the University of Texas, Austin, the SuperMatrix [CVZB+08a] runtime is used for the FLAME project [GGHVDG01]. The focus of these runtimes is thus to optimize execution for dense linear algebra, at the expense of genericity.

It is worth noting that the University of Tennessee has also designed PaRSEC [BBD+13a] (previously called DaGUE [BBD+10b]), whose programming principle (PTG [CL95b, DBB+14]) is very different from the commonly-found task-based interfaces, as was explained in Section 2.1.3. This allows for more potential for optimization, at the expense of more programming complexity and less genericity.

2.3.2 Compiler-provided runtime systems

Even when a task-based programming interface is proposed directly at the language level, the compiler for the language will embed into the resulting application a runtime that manages execution of tasks. Notable examples include the OpenMP standard[6], for which the libgomp and libiomp implementations actually represent runtimes, OpenACC [Ope13a] or HMPP [DBB07].

In the case of OpenMP, it is interesting to note how the first versions of the language were rather designed to *avoid* run-time decisions. The only piece of decision that was initially left to the runtime was the chunk size for parallel loops, and even that was questioned. When we tried to introduce dynamic scheduling for nested parallel loops [30], the main reaction was a hearted "but this means the runtime has to be really *smart*!" Since then, positions have largely changed, and the addition of task dependencies in OpenMP 4.0 opened the way to dynamic runtimes (we will discuss more on this in Section 2.6). We can still notice a lot of remnants of the static-control-flow position in the standard (such as the notion of tied vs untied tasks) which makes moving to really dynamic execution sometimes even convoluted.

The early-OpenMP example is symptomatic of an overall tendency of these runtimes towards still keeping the application mostly in charge of the control flow. The OpenACC standard for instance is mostly an offload interface, and does not let the application leave complete execution control to a runtime, which thus prevents implementations from achieving a lot of the optimizations we will detail in Section 3.

2.3.3 General-purpose runtime systems

A lot of general-purpose runtime systems have been proposed over the past decades. We here present a few instances which target very differing goals.

One of the first largely-known task-based programming interface was the Cilk [BJK+95, Lei09] programming language. It was initially mostly a proof of concept for how small tasks can be efficiently executed with quite trivial programming: function calls can be marked with the *spawn* keyword to express the potential for task creation, which translates to only some stack mangling at runtime. It is then work stealing which actually creates tasks by tinkering the stack to steal the execution of the code found after the function call. The set of features is however kept limited in order to keep the extreme efficiency even with tiny tasks.

[6] `https://www.openmp.org/`

Intel and Apple have respectively proposed TBB[7] [Rei07b] which provides loop parallelization and task-based programming, and GCD[8] which provides task-based programming. These runtimes actually provide only task queues and notably do not capture the management of data. They do use a real scheduler which governs the control flow, but it has a very deterministic, typically hybrid depth-first / breadth-first behavior on the task queues. The design decision not to capture the data means that it will be limited to shared-memory task parallelism and thus not target e.g. GPUs with discrete memory or seamless network support.

Charm++ [KK93a, KRSS94] is an example of programming environment which is not based on tasks, but rather on graphs of actors (called *chares*) which contain state and exchange explicit messages. The charm++ runtime can at will move chares between computation resources, manage network communications, make checkpoints etc. because it has made the complete move to a really runtime-driven execution. The programming model however does not tend to let the application describe how chares will behave in the future, which prevents a whole range of optimizations that task graph allow.

XKAAPI [GLMR13a] (previously KAAPI [HRF+10b]) is based on previous work on Athapascan-1 [GCRD98]. It provides a pragma-based interface (as well as OpenMP compatibility and extensions) to submit task graphs. It does catch the data passed to tasks, and can thus support for instance GPUs with discrete memory. Its current scheduling principles, inherited from the previous work, is mostly focused on efficient work-stealing that can work with very small tasks. It does not currently target supporting execution over the network.

The SuperGlue [Til15] runtime focuses on achieving best performance with very small tasks while supporting a fair set of features for task-based programming. By essence, the feature set will thus remain limited, but it allows to efficiently execute applications expressing very small tasks.

The Legion [BTSA12a] runtime is used to support the Regent [SLT+15] language. The main goal is to let the application precisely describe how tasks access data split into regions and subregions. This allows to automatically infer data dependencies and access conflicts, and generate the required communications. Legion then provides tools to easily map tasks to computation resources. This shows that it does not have made the step to integrate a notion of generic scheduler, and remains with only locality-oriented mapping.

StarSs designates a family of programming environments based on a common principle of pragmas for submitting tasks: COMPSs [BMC+15], OmpSs [BDM+11], GPUSs [ABI+09], CellSs [BPC+09], SMPSs [PBL08], ClusterSs [TFG+12], GridSs [BLS+03]. They are different runtimes which addressed different sets of hardware. Among the latest generations, OmpSs is based on the OpenMP standard, to which it adds extensions to improve the expression of parallelism and to open the way for dynamic optimizations. Such extensions are progressively included in the OpenMP standard, thus helping it to escape from static control flow. StarSs has really fully embraced making the application completely leave the control flow to the runtime, both in terms of computation and data. It however currently does not target implementing advanced dynamic scheduling like what will be discussed in Sections 3.1 and 3.2

Various PGAS programming environments have also been proposed and embed a runtime, notably UPC [EGC02, ZKD+14], XcalableMP [18], GASPI [SRG15], ParalleX [KBS09] with the HPX [KHAL+14] runtime, Chapel [CCZ07], X10 [CGS+05], ... The PGAS programming paradigm however typically does not allow applications to express much of the future behavior of the computation, the main control flow remains in the hand of the application. This prevents a variety of optimizations such as automatically deep pipelining and data prefetching of different iterations of a loop nest.

Lastly, the Big data trend has recently produced some dynamic runtimes such as Dask [Roc15] and Tensorflow [ABC+16a]. While initial versions such as DistBelief [DCM+12] were using rather

[7] `https://www.threadingbuildingblocks.org/`
[8] `https://apple.github.io/swift-corelibs-libdispatch/`

static approaches, the very irregular nature of the targeted applications quickly imposed using complete dynamic scheduling. The currently used scheduling heuristics are for now very simple, notably because in those fields very large performance speedup are obtained from application optimization and the community thus does not consider it worth spending time on optimizing the lower layers. That being said, such applications manipulate very large amounts of data which can not fit in the main memory, and a lot of care was taken to optimize data transfers.

2.3.4 StarPU, a general-purpose task-based runtime system

The runtime system discussed in this document, called StarPU[9] [12, 4, Aug09, AN08], also completely embraced making the application leave the whole control flow to the runtime. It was founded throughout the PhD thesis of Cédric Augonnet [Aug08, 43] (which I co-advised), and was the ground for various research projects I participated to and PhD theses I co-advised. It currently consists of almost 100,000 lines of C code under the LGPL free software license. It has become one of the references in the runtime domain, the founding paper [12, 4] has gathered more than 1000 citations.

StarPU focuses on the runtime part of Figure 2.6, and aims at providing a very flexible general-purpose interface that upper layers can interact with, and implementing a diversity of features and run-time optimizations based on information from the application. StarPU is not programmed through a specific language, it provides a library interface, on top of which computation libraries or task-based programming languages can be implemented. For instance the Chameleon[10] dense linear algebra library from the MORSE project[11] is based on it. StarPU is then both a host for state-of-the-art runtime heuristics and techniques for upper layers to benefit from, and an experimentation platform for research on runtime. It targets optimizing for general purpose, without specificities to a given end-user programming language or application pattern. That being said, customizability is also one of the goals here, notably the task scheduler can be provided by the application, to better fit the application's particular needs. To be able to combine the various optimization layers while keeping maintainability, genericity is privileged over extreme micro-optimizations (contrary to e.g. SuperGlue [Til15]). For instance, to efficiently exploit 32 cores, tasks must typically not be shorter than 100 μs.

To summarize, the overall goal is to provide a runtime layer ready for use by upper language or library layers, providing genericity, flexibility and state-of-the-art optimized performance, possibly at the expense of efficiency for small tasks.

2.4 Reaching real performance portability?

A lot of the runtimes described in the previous Section have adopted more or less a software stack such as typically shown on Figure 2.6. Task-based programming indeed allows to completely decouple the submission of the set of tasks (i.e. the application algorithm) from the actual implementations of the tasks.

On the task submission side, software layers or even domain-specific languages can be used to help programmers with expressing their high-level algorithms, without having to care about the technical details for the actual execution or even simply about the actual machine which will execute the application. They can concentrate on making their algorithms more parallel and let the runtime execute them as efficiently as possible.

On the task implementation side, various approaches are also possible. A first approach is to

[9] because it can address various architectures including CPUs, SPUs, GPUs, ... in brief *PU
[10] https://project.inria.fr/chameleon
[11] https://icl.cs.utk.edu/morse/

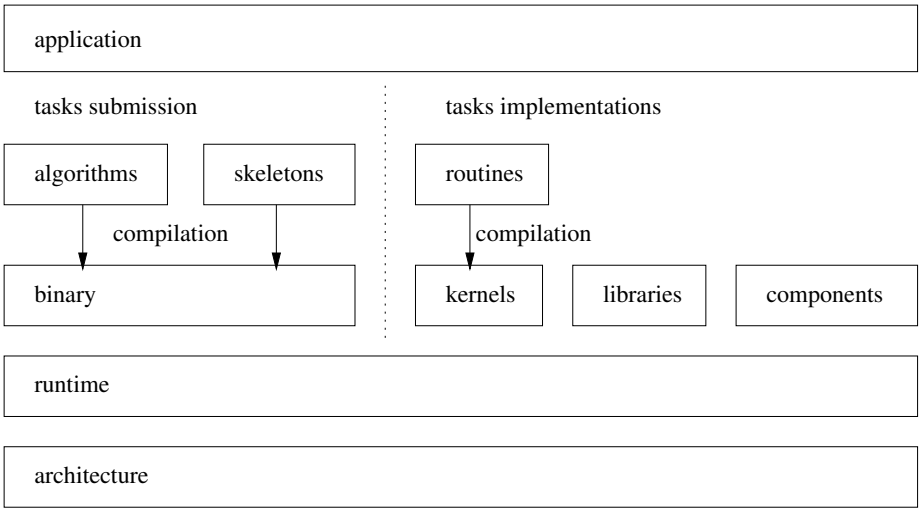


Figure 2.6: Complete software stack with separation between tasks submission and tasks implementations. Submission can be explicit, or possibly managed by skeletons, implementations can be provided multiple ways.

just pick up existing task implementations. For linear algebra for instance, the BLAS interface [LHKK79b] was defined to contain most operations that an application may need. Vendors can thus provide optimized implementations (e.g. MKL for Intel processors and cuBLAS for NVIDIA GPUs). Porting an application made of BLAS tasks to a newer architecture then boils down to plugging in the manufacturer-provided BLAS implementation.

Another interesting solution, for non-standard tasks, is to write only one implementation in some language, and let compilers generate binaries for the various targeted architectures. Typically, for most tasks a vendor-provided optimized implementation is available, but some are missing it. Even a not-so-optimized implementation compiled this way can be very beneficial, to e.g. be able to execute these tasks on the accelerator where data was produced and will be reused by the optimized tasks, thus saving costly transfers. If such implementation happens to still show up as a bottleneck, one can consider spending the time (or delegating the work) to write an optimized version, or to change the application algorithm. But at least there are ways to relatively easily get an initial version, and execution analysis as will be discussed in Section 3.7 can be used to determine where performance can be improved. At any rate, the management of the availability of more or less optimized implementations can remain independent from the actual application algorithm.

2.5 Task-based programming, a new horizon for programmers

While discussing with users of StarPU, we have noticed how much programming with tasks is really a change of programming paradigm for them and requires code restructuring, even if the STF interface (described previously in Section 2.1.3) allows to keep sequential-looking source code.

The first encountered constraint is having to split computation into tasks. Ideally these tasks would have rather stable performance so that performance models (as will be discussed in Section 3.1.2) can be used for advanced task scheduling (as will be discussed in Section 3.1). When the existing source code is a mere succession of BLAS calls, replacing them with task submission calls thanks to the Chameleon library is very straightforward. When the existing source code

is made of a series of functions which are called from a main loop, introducing a task layer in between is mostly syntactic sugar (which can even be supported through pragmas, as discussed in next Section). When the existing source code is mostly composed of lengthy functions without real structure, deep revamping is often required.

Along such restructuring, another issue which arises is global variables. In principle tasks should be *pure*, in the sense that they should access only the data given to it by the runtime, and never access global variables. In practice, users tend to be lax about this when they target only shared-memory systems. Of course, this will pose problem when moving to using GPUs with discrete memory, which can not access these global variables. This kind of cheating also lowers the correctness guarantees that the runtime can provide (as will be explained in Section 3.6) since the application does not expose all the semantic.

In the end, respecting these is basically *functional programming*, which is considered to be good programming style anyway, so doing this effort of *taskification* often leads to better source code overall, in addition to being able to leverage a task-based runtime.

The extra difficulty for users is to accept revamping their existing well-tested code, only to lose the handling of the control flow, and to have to trust dynamic runtimes to do a good job at scheduling instead. A parallel can be drawn with the historical migration from assembly to higher-level languages, which introduced constraints and made programmers have to trust compilers.

2.6 Can we have a common task-based programming interface?

Middleware layers shown on Figure 2.6 can be used to make application programming simpler. For instance, skeletons as provided by SkePU [EK10, DEK11, 15], which we integrated with StarPU during the PEPPHER european project[12] [BPT+11, 16, 13], aim at providing very high-level programming interfaces. We also integrated a PGAS-oriented language, XcalableMP, with StarPU [18].

Using such interfaces may be questioned, however, since modifying an application to use them means depending on the sustainability of the implementation, which may not be guaranteed unless the interface gets standardized and well-supported. Similarly, directly using the programming interface of a given runtime system means depending on it. Various people have expressed that they would like to carry out the move to task-based programming only once, and then be able to easily choose between runtime implementations.

In practice, task-based programming interfaces are quite similar, and sometimes the differences lie only in syntactic sugar around the STF programming style and some details on passing parameters to tasks. StarSs, OpenMP, as well as the GCC plugin on top of StarPU by Ludovic Courtès [Cou13] indeed use very similar sets of pragmas. The Chameleon dense linear algebra library is not using pragmas, but its principle remains based on STF: what looks like a function call is actually an asynchronous task submission. But while being fairly straightforward, having to switch between syntaxes in order to switch between runtimes would remain tedious.

It is thus worth wondering whether a common-denominator interface could be standardized, and all runtimes would implement directly it or provide a compatibility layer. Middleware layers and applications would then be *taskified* once, using that interface. Switching between runtimes would then be a matter of switching between implementations, just like the introduction of the standard MPI interface allowed to seamlessly switch between its implementations.

Settling down a common programming interface would also open the way to interesting source-to-source optimizing compilers to be applied independently from applications and runtime.

[12] `http://www.peppher.eu/`

```
def mysubmit(f):
  def submit(*args):
    return runtime_submit(f, args)
  return submit
```

(a) Python annotation implementation.

```
@mysubmit
def add(x,y):
  return x+y
```

(b) Python function annotation.

Figure 2.7: Example of Python function annotation.

Given a loop nest, polyhedral analysis can be used to catch the structure of the nest, and perform optimizations similar to what PaRSEC can currently achieve given the PTG form (see Section 2.1.3 above), and emit an optimized version with reorganized loops and optimization hints.

OpenCL[13] was supposed to be a standard parallel programming interface with great support for asynchonicity and heterogeneity for various types of architectures. StarPU provides an OpenCL-compatible interface, SOCL [Hen11, HDB12, Hen13a, HDB+14], written by Sylvain Henry (advised by Denis Barthou and Alexandre Denis), which allows OpenCL applications to benefit from the StarPU runtime optimization. Really achieving these optimizations however required adding a few extensions, notably to be able to express memory sharing and let the runtime get control over data transfers. Actually, OpenCL is both too low-level and too high-level. On the one hand, the SOCL work showed that the interface does not leave quite enough optimization control to the OpenCL implementation, and on the other hand when StarPU itself uses the OpenCL interface to drive accelerators, it is lacking some guaranteed control details over data transfers, notably. OpenCL also provides a very hairy programming interface which is not convenient for widespread use.

OpenACC [Ope13a] was supposed to become a standard for easily exploiting accelerators through a pragma-based programming interface, which could have become a task-based pragma programming interface. Efforts have however nowadays rather concentrated on OpenMP, which can be surprising since OpenMP was initially only for parallel loops with very directive control flow, tasks were only introduced in version 3.0 [ACD+09], without dependencies, and real task graphs are only possible with dependencies in version 4.0. The introduction of dynamism there is however making slow but good progress, with e.g. task priorities introduced in version 4.5, and data reduction and commutative access are being introduced for version 5.0, so the OpenMP standard looks more and more promising.

It thus seems that a convergence to OpenMP could be considered. The StarSs community is clearly heading in that direction with OmpSs, which acts as one of the experimental testbeds for extensions to be included in OpenMP. On the StarPU and XKAAPI side, a compatibility layer was introduced, KSTAR, lead by Olivier Aumage and Thierry Gautier, to provide an OpenMP interface on top of them, as well as a benchmarks suite KASTORS [31]. Some additional runtime code had to be introduced in StarPU and XKAAPI to support e.g. tasks starting and waiting for other tasks, but that was reasonable enough. The resulting performance is conclusive: for an FMM application, the OpenMP version was performing as well as the native StarPU version [AAB+16, AAB+17], provided the availability of the commutative data access extension.

Big data communities are demanding standard interfaces in higher-level languages than C, and typically lean towards Python, but there currently is no Python version of OpenMP. The Global Interpreter Lock (GIL) also prevents from any actual Python parallelism, but calling external routines such as BLAS in parallel is possible. It happens that Python includes a standard interface for asynchronous execution: *futures*, and function annotation is integrated in the language, called *decorators*. For instance, adding a `@mysubmit` decoration to a function definition as shown on Figure 2.7(b) does not immediately define `add` to $x, y \to x + y$, but gives that latter anonymous function to the `mysubmit` function, which returns the function that `add` eventually gets defined

[13] https://www.khronos.org/opencl/

to. Typically, `mysubmit` can be defined as shown on Figure 2.7(a), and thus calling `add` actually calls the `submit` function which submits to the runtime a call to the underlying function (i.e. `f(*args)`), and returns a Python *future*. The main program can then exactly look like sequential source code, only the function decoration makes it asynchronous. It will probably be interesting to establish standard decorations to open the way for replaceable task-based Python runtime implementations.

One of the remaining general issues, which arose already when we tried to introduce a pragma-based language on top of StarPU [Cou13], is expressing data access, for instance designating subparts of matrices. In the StarPU interface, to avoid the issue, data has to be registered and subdivided explicitly, which is actually useful to support arbitrary data structures (as will be discussed in Section 2.7.1). OpenMP annotations are however much less expressive, and would typically give the runtime only a pointer to the start of the data and its size. That is however not specific enough for accessing a subpart of a matrix, which involves the notion of *leading dimension*. Specifying it thus seems a necessity, but the precise notation is still an open question, and data structures beyond mere matrices will remain difficult to describe. The experience of the Regent [SLT+15] language on data description will probably be useful here.

More generally, it is questionable whether a standard language like OpenMP will provide enough flexibility for most use, or remain limited to the simple cases. The next Section describes the kind of interface flexibility which we have noticed to be useful and demanded. MPI is an example of standard which has grown a lot to provide such kind of flexibility, but that growth is questioned, the question will rise in the OpenMP case too. StarPU provides OpenCL and OpenMP interfaces only as options through SOCL and KSTAR, to keep the underlying C library interface available for the sophisticated features which may not be reasonable to express in OpenMP.

2.7 How to make a task-based interface generic, flexible, and featureful?

While designing the StarPU interface, we have usually privileged genericity and flexibility. This notably translated into providing several interfaces to express tasks, dependencies, and data, as explained in the next sub-Section. Providing a lot of features is more questionable: supporting features "behind the scene", i.e. without programming interface changes, poses only implementation questions, as will notably be discussed in Section 5.1, but when features involve the semantic of the programming interface itself, getting the interface right is tricky. Some examples are discussed here in Sections 2.7.2 and 2.7.3.

2.7.1 Genericity and flexibility come with diversified interfaces

Discussions with StarPU users have shown that for different applications (such as will be listed in Section 2.8), different programming interfaces are preferable.

For task submission, the STF programming paradigm, implemented by the `starpu_task_insert` function, has been widely adopted by users, for instance within the Chameleon[14] dense linear algebra Library. As was explained in Section 2.1.3, the resulting sequential-looking source code is clear and maintenance is productive. Users however often ask for more sophisticated interfaces for convenience. As a simple example, they would like to pass a variable-length array of variables instead of passing them as separate parameters. They also requested for `starpu_task_build`, which only builds the task conveniently like `starpu_task_insert`, but does not submit it, so the application can tinker with the task before submitting it. In other cases, it

[14] `https://project.inria.fr/chameleon`

is preferred to construct tasks by hand, by manually filling the structure returned by `starpu_task_create` (which was actually the only way originally provided by the first versions of the StarPU library). Control of the allocation of the task is sometimes also desired, e.g. to embed it into another structure, and then only `starpu_task_init` is used to set the default values. Such flexibility is not usually provided by a language-based interface.

The management of dependencies is also questionable. The STF paradigm makes them completely implicit, which helps a lot for productivity and was strongly requested before StarPU supported it. Some use cases however still prefer to express dependencies explicitly, or to put some additional dependencies to e.g. mitigate memory use. StarPU provides several ways to express these. They can be set between explicit tasks with `starpu_task_declare_deps_array`. They can be set a bit more implicitly through the use of *void* data, which do not contain any data, but that various tasks can be declared to write to and thus be serialized. Synchronization-only empty tasks can also be used to simplify synchronization schemes. Another interesting way to express dependencies is StarPU *tags*: they are *rendez-vous* points which are only identified by their number. This allows e.g. to make a task depend on some task which has not even been created yet, by just agreeing on the number (the tag) which identifies that task. Some use cases have showed them to be extremely convenient.

More widely, we have taken care of making the programming interfaces very generic, and as less dependent on hardware-specific features or interfaces as possible. This allows for instance to start with implementing a CPU-only version of the application and check correctness of the parallelization with it, before supporting GPUs by additionally using the few bits of GPU-specific programming interface to provide the GPU implementations of tasks. While at it, we added support for providing multiple implementations of tasks for a given architecture, possibly conditioned by a `can_execute` callback which can check at run-time whether hardware support is available (e.g. SSE extensions, CUDA capabilities, etc.).

Describing data has also required a lot of care. StarPU natively supports basic data structures (vectors, dense matrices, CSR/BCSR/COO sparse matrices), but support for arbitrary data structures was also made possible by making the application describe its own *data interface*. Instead of a language-based description such as used in the Regent [SLT+15] language, which would have inherent limitations, we let the application define its own structure which holds the description of the data (e.g. pointer(s) to start of data, elements and array(s) size(s), pointer to the root of a tree, etc.), and provide the method for transferring data. For instance, the transfer method for CSR sparse matrices boils down to calling `starpu_interface_copy` three times, for the value array, the column index array, and the row pointer array. This is enough for StarPU to then be able to transfer such data between CPU and GPU, NUMA nodes, etc. Two additional marshalling methods can be provided by the application, `pack_data` and `unpack_data`, to let StarPU then be able to store data on the disk (Out-of-Core) and exchange data over the network. In case optimized transfer implementations are possible (e.g. using low-level features such as `cudaMemcpy3DPeerAsync`), they can be additionally provided, but this is not required for getting an initial non-optimized version that already works. The implementation details will be described in Section 5.3. This possibility of generic data interface was notably used in the HI-BOX project, which will be described in Section 2.8, for supporting compressed h-matrices without having to actually describe to StarPU what an h-matrix is.

The support for partitioning data follows the same principle: StarPU provides *partitioning filters* for the basic cases (e.g. vector and matrix subdivision), but the application can ship its own arbitrary filters. Such filter simply computes, starting from a data interface, the sub-data interfaces, i.e. their respective data pointers, sizes, stride, etc. This can thus be used on arbitrary data structures.

More generally, the overall approach is that the StarPU API can be used in a simple way as a first approach, the only notions which have to be understood from the start is *tasks*, *codelets* (which gather the implementations for tasks) and *registered data*. This is enough to get a rough implemen-

tation of the application working, even if slowly. To get performance, the refined features (performance models for the scheduler, commutative access, flush hints for better Out-of-Core behavior, etc.), can be learnt progressively, and except in a few cases they *do not change the semantic of the program*, which means that they can not break the behavior of the initial rough version. The programmer can thus play with them at will to see how performance can be improved by providing the runtime with more information. In comparison, PaRSEC [BBD+13a] or HPX [KHAL+14] introduce various complex concepts [CL95b, DBB+14] which have to be understood before writing even simple programs.

2.7.2 Can we afford managing different versions of data?

The classical Write-After-Read (WAR) dependency means that tasks which write to a piece of data have to wait for tasks which need to read the previous content of the data. Such dependency may lead to less parallelism, and it may be desirable to rather make a copy of the previous content, so that all these tasks can run concurrently. Some runtimes like OmpSs [BDM+11] support this.

Supporting this automatically means not only more complex data management, but also poses questions of memory usage: we are here trading memory for parallelism availability. Whether this is a good thing is not guaranteed. As will be discussed in Section 3.2.4, optimizing for memory usage already involves a lot of different subsystems and heuristics. If making a copy really is decisive for performance, it could as well be implemented by hand, it is not clear whether some cases require an automatic decision.

We have thus for now not considered adding automatic support to StarPU: if the application programmer notices that copying can help with performance, it can be implemented by hand by using `starpu_data_register_same` to easily create a new data with the same structure, and `starpu_data_cpy` to copy over the content of the data.

2.7.3 Which semantic should we give to cancelling a task?

Being able to *cancel* a task has often been requested. Just like for the case of cancelling a POSIX thread which raised a lot of ground issues, the serious question is the precise *semantic* of canceling the task, and here more precisely what should be done with the descendants of the task. Some users have expressed that they would like descendants to be cancelled too, others would like to see descendants somehow able to survive, but a question arises for the validity of the data that was supposed to be written to by the cancelled task.

It could be argued, similarly to the case of the previous Section, and the case that will be described in Section 5.2, that users could implement the feature themselves, by introducing `if`s in their task implementation, to effectively make cancelled tasks not use any computation time. This will however disturb the calibration of performance models, and in case such tasks are scheduled to a GPU, data may be uselessly transferred to the GPU by the runtime. It thus seems that we will really have to add an interface for it. The precise semantic still needs to be refined, it is probable that several semantics will actually be proposed for the programmer to choose from.

2.8 Task-based success stories in HPC

In the past decade, we have worked with various research teams to use StarPU task-based programming in a wide range of HPC applications.

The dense linear algebra case was naturally the first testcase for StarPU. We actively worked with the University of Tennessee, Knoxville (UTK), notably through the MORSE associate team

project[15]. We got very conclusive result on the typical Cholesky [21], QR [8] and LU [AAD+11] cases, and generalized [7] the approach into the Chameleon[16] dense linear algebra library.

In the sparse linear algebra case, the PaStiX and qr_mumps solvers were successfully ported to task-based programming ([40, 27] and [ABGL13, ABGL14, 37], resp.) through the ANR SOLHAR project[17]. In the qr_mumps case, this even made 2D decomposition very simple to support, while implementing it by hand without task-based programming was considered unreasonable. The result was extremely wide parallelism which amply improved performance.

We have also worked on the *compressed* dense matrix case, within the context of the PhD thesis of Marc Sergent [46] (which I co-advised) and the HI-BOX DGA RAPID project[18]. These posed questions of memory management, which will be discussed in Section 3.2: throttling execution to avoid overflowing the memory, and supporting Out-of-Core execution to store unused data on the disk. In the case of hierarchically-compressed dense matrices (h-matrices), Out-of-Core support has always been considered unreasonable to implement without a runtime, while enabling Out-of-Core support of the StarPU port only required adding marshalling functions for h-matrix blocks, as was mentioned in Section 2.7.1. With the help of StarPU for managing task scheduling and data transfers, the industrial partners of the HI-BOX project could factorize matrices several orders bigger than other solvers which do not use a runtime. For instance, factorizing with double precision a 720 GB matrix on a system with 256 GB of memory could be achieved, leading to a 1600 GB result matrix. This is now used by AirBus, ArianeGroup, MBDA, and Thalès.

In order to get good performance for Fast Multiple Method (FMM) over StarPU [Bor13, ABC+16b, ABC+16], we had to improve the locking support for commutative access, as will be described in Section 3.3. Béranger Bramas additionally introduced a new scheduler, HeteroPrio, to better handle priorities in FMM: notably some tasks are completely independent, and can fill the scheduling gaps by being scheduled at last resort. Further development on this scheduler went on, as will be explained in Section 3.1.4.

For Conjugate Gradient applications [AGG+16], we enhanced StarPU's data partitioning to support asynchronicity, so the application can better express data coherency between domain pieces.

Some Finite Difference or Volume-based applications have been ported, notably seismic wave modeling [17], Computational Fluid Dynamics (CFD) for Ariane 5 booster take-off blast wave with FLUSEPA [CCRB17], and Galerkin methods (Raphaël Blanchard's PhD thesis). Image analysis applications have also been ported with the University of Mons [34, 32]

To summarize, task-based programming brought interesting performance improvement in various classes of applications, not only dense linear algebra, but also more irregular applications, for which dynamism of a runtime brings very useful load balancing. We will however see in the next Section that for other classes of application, using task-based programming raises issues.

2.9 A one-size-fits-all approach?

For some application cases, using task-based programming can be questionable.

A rather trivial case is when the application structure is very simple, such as a very regular stencil. Having to express computation as independent tasks and value propagation as data dependencies looks quite tedious, when an OpenMP version can quite easily be implemented and already provide very good performance. Even when using accelerators, calibration can be used to compute a static partition of the data, so the advantage of using a runtime system is not clear. It can be argued that in some cases the application may evolve and introduce for

[15] https://icl.cs.utk.edu/morse/
[16] https://project.inria.fr/chameleon
[17] http://solhar.gforge.inria.fr/
[18] https://imacs.polytechnique.fr/HIBOX.htm

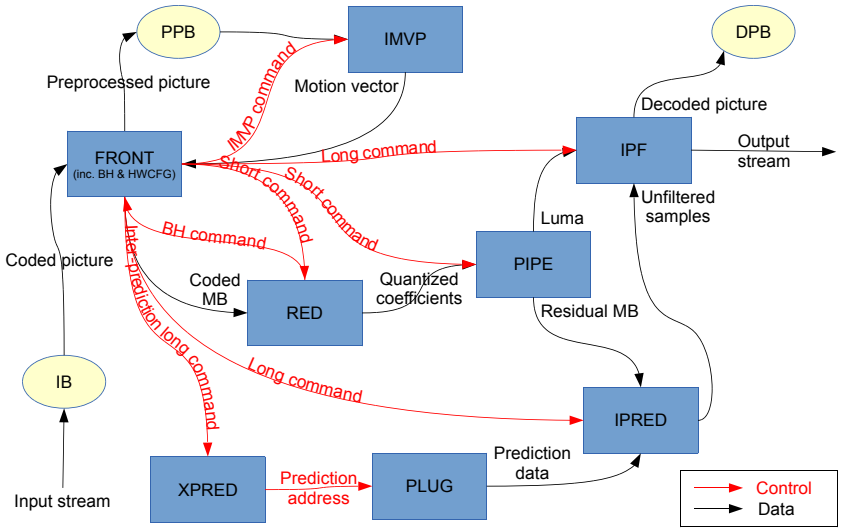


Figure 2.8: Actor graph for H.264 video decoding

instance Adaptive Mesh Refinement (AMR) which can introduce unpredictable irregularity and thus require dynamic load balancing, but a fair share of applications keep having too simple parallelism for a runtime to really bring enough benefit.

Apart from such cases, the issue which usually arises is how much overhead a runtime brings. We have seen in Section 2.3 that this may rule out quite a few runtimes, including StarPU. The underlying question is how long application tasks take. It happens that very often users just have no idea... In cases like Finite Element Methods (FEM), the basic computation elements are very small, and tasks can thus typically only take microseconds, which is too small when typical runtime overhead is also typically a few microseconds per task. This is a question of granularity of tasks, as will be discussed further in Section 3.4. It sometimes makes sense to group several elements so as to generate bigger tasks, like was done in FLUSEPA [CCRB17]. Some other applications however have an extremely fine-grain behavior which is difficult to aggregate, for instance graph algorithms would require an initial aggregation step whose cost may be prohibitive.

To efficiently cope with very fine-grain tasks, it was often suggested that a persistent kernel could be running on GPUs, picking up work from task queues [TPO10]. Fine-grain parallelism can also be quite naturally cast into an FPGA array. This actually leads to earlier work on *actor graphs* and *data flow* [Den74] models, which are very common for instance on embedded platforms. An example of actor graph is shown on Figure 2.8: data is flowing between actors which keep consuming and producing data at various pace at each independent actor *activation*. Scheduling actor activations has been extensively studied, and typically leads to static steady-state scheduling. During the PhD thesis of Paul-Antoine Arras [42] (which I co-advised) and in collaboration with ST Microelectronics, we tried to introduce dynamic runtime optimizations in an embedded platform. The idea was to keep the actor graph programming model (which can for instance provide a lot of correctness guarantees), and schedule tasks which are the instances of the actor activations. The main runtime issues we encountered were that tasks are extremely small (we can typically afford for the scheduler only a thousand CPU cycles per task), and having to adapt classical lightweight list scheduling to memory constraints [3]. We also worked on the model itself, to combine Dataflow Process Networks (DPN) and Kahn Process Networks, and add hierarchy possibilities, to get HDKPN [10], to benefit from the properties of both previous models. Experiments have shown that interesting results are obtained only when the application has a very irregular behavior. The case at stake was H-264 video decoding, which indeed exhibits ir-

regularities depending on the content of the video itself. For applications with static behavior, actor graphs are usually small enough to perform even expensive offline static mapping and scheduling.

2.10 Discussion

In this chapter, we have seen that while some applications do not need a task-based runtime system to easily get performance, and some classes of applications can not afford its overhead, a fair share of applications can greatly benefit from it. This has led to the birth of a very large variety of runtime proposals, with differing goals and thus compromises. The proposed programming interfaces are somehow diverse, but often seem to revolve around what we called STF, and the OpenMP standard includes more and more a common denominator of these interfaces.

To fully benefit from a runtime system, programmers however typically have to use its extensive API, and it is not clear whether OpenMP will be able to become as expressive as this. For instance, will it ever be possible to express h-matrices which were discussed in Section 2.8 with the OpenMP pragmas, to get Out-of-core support, if that feature even makes it to the OpenMP standard? A runtime API is however typically dense, and thus not adapted for real end-users; the StarPU API is not really meant to be, for instance. OpenMP can thus play a role of "good enough" simple task-based programming interface, like MPI has been compared to the other communication libraries. I however do not think OpenMP will likely be enough "for most uses", and a fair share of users, advanced programmers, will keep demanding for more advanced interfaces such as the StarPU API. OpenMP can then focus on the simpler general-purpose cases.

In the case where end-users do not program runtime systems directly, they can use domain-specific languages or middleware on top of runtime systems, as shown on Figure 2.6 page 15; these latter become what we call "application" in this document. I believe it is preferrable to separate these from the runtime, so that the respective communities can focus on improving their respective layer, and the API between them can be extended at will. This is what has been successfully going on between the HiePACS research team which designs Chameleon, and the STORM research team which designs StarPU.

The goal of a dynamic runtime system is to automatize as many aspects as possible: task scheduling, data placement, data transfers, etc. It should however not exclude letting the programmer control some of these. Actually it would be pretentious to claim that automatic heuristics always perform at least as well as what the programmer knows. It also allows her or him to leave control to the runtime progressively, making the transition to dynamic execution smoother, and allowing to analyze piece by piece why performance evolves (one way or the other...). Letting the programmer specify e.g. data placement or some task scheduling even allows her or him to perform offline analysis, that the runtime can then just refine during execution.

More generally, combining different tools for executing an application is very challenging. Projects typically want to provide as much support as possible, but this means that when using two projects for their respective features, some feature overlapping shows up. For instance, combining a Big data storage management layer with a task-based runtime with Out-of-Core support means finding the precise bits of API which can match. If e.g. the task-based runtime insists on using the POSIX file interface, and the storage management layer insists on providing only a key-value library-based interface, the combination can not happen, just because both tools implement the data storage format. It is thus essential for the two projects to provide several layers of API, to be able to find a pair which can indeed match. We will discuss another example, concerning CPU reservation, in Section 3.4.3.

Last but not least, for now most task submission paradigms express only *flat* task graphs. Some projects started experimenting with hierarchical task graphs, notably to let users express divid-

able tasks so the runtime can dynamically adapt the task granularity as will be discussed in Section 3.4.2. Such hierarchy seems to be useful more generally: similarly to how the notion of bubbles was generally useful for thread scheduling in our previous work [1, TNW07], it seems it would be useful to a larger extent. For instance, it can carry summaries of memory usage information, which could be used for expensive memory usage control heuristics, as will be discussed in Section 3.2.3, or make master-slave task distribution scale better, as will be described in Section 4.1. How to express such task graph hierarchy most easily however remains a challenge.

Chapter 3

Runtime-level opportunities for optimizations and features

Contents

This chapter describes the runtime side of task-based execution, and notably its main goal: the optimizations which can be achieved, notably thanks to the expressiveness of task submission. The information given by the task graph is indeed what allows a runtime to achieve better optimization than an Operating System would be able to do. The range of optimization is very wide, and the potential refinements are deep, for instance even scheduling the classical Cholesky factorization on heterogeneous platforms is tricky: a whole PhD thesis (which I co-advised) was spent on it [44]! This chapter is dedicated to shared-memory execution, distributed-memory execution will be discussed in chapter 4.

We first discuss the two main aspects of runtime execution: task scheduling and managing memory use and data transfers. We then discuss optimizing commutative data access, for which initially only a very simple solution was used, but the FMM application test case required a refined solution. The question of task granularity is then raised, and different solutions (task division, aggregation, leveraging parallel tasks) are discussed. The simulation of task-based application will then be introduced, which entails potential for execution correctness guarantees. The potential for feedback to the user will then be discussed, both for performance and for debugging.

3.1 Scheduling task graphs

Task graph scheduling has been studied for half a century [Cod60, Gra66] and extensively, even leading to meta-studies [KMJ94]. The problem at stake is typically to minimize overall execution time, given a set of tasks with precedences, and differing execution times among computation resources, which can be noted $R|prec, c_{ij}|C_{max}$. This problem, like scheduling in general,

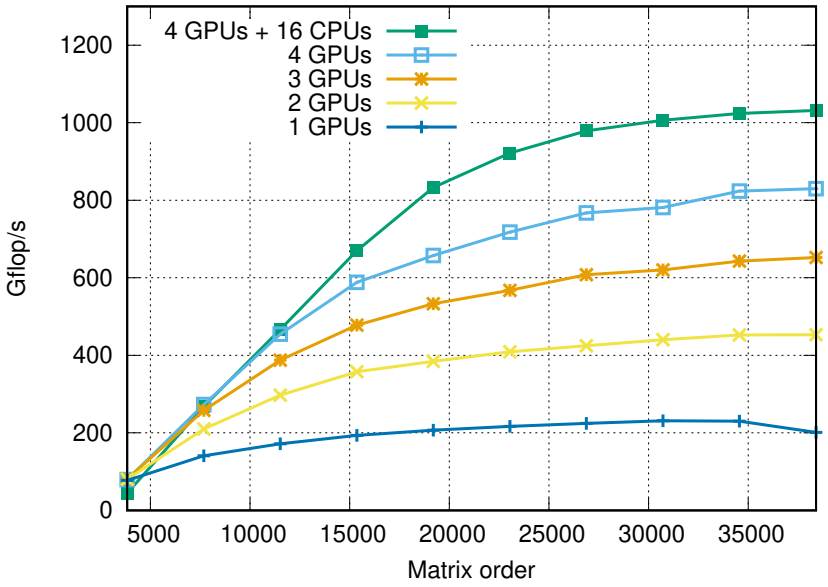


Figure 3.1: Performance of the QR factorization with StarPU

is NP-hard [GJ79, Ull75, BK09b], but in practice a lot of very effective greedy heuristics have been proposed. They generally consider only the *ready* tasks, i.e. tasks whose dependencies have completed. This allows to reduce the problem to independent tasks and make the problem simpler [LST90, BKSM+15]

The various schedulers available in StarPU are mostly based on two well-known generic heuristics, HEFT [THW99] and work stealing [BL99a]. They automatically achieve both task scheduling and trigger data prefetching, which entails load balancing and properly pipelining task execution and data transfers. That allows to get excellent performance, notably with dense linear algebra applications. For instance, Figure 3.1 shows the scalability of the QR matrix factorization over multiple GPUs. It also shows that using CPUs in addition to the GPUs brings an additional 200 GFlop/s performance boost, which happens to be greater than the performance of QR factorization on these CPUs alone, which is about 150 GFlop/s only. This super-linear effect happens because both the application and the architecture are heterogeneous, and scheduling heuristics can benefit from this, by scheduling on CPUs the tasks which GPUs are not *so* efficient for. The GPUs then process the remaining tasks more efficiently overall, leading to the super-linear effect.

In this Section, we will discuss various aspects of task scheduling within a runtime system. First the overall shape of a scheduler will be considered, and to what extent it can rely on performance models for tasks. We will then consider the algorithmic complexity of advanced heuristics, and how we could cope with it. We will examine taking energy consumption into account, and to what extent performance bounds can be automatically computed. The history of schedulers implemented in StarPU will then be explained, before discussing in more details the question of data transfers. We will eventually consider how we could effectively drag theoreticians into the picture.

3.1.1 How does a task scheduler fit in the runtime picture?

Ideally, the runtime scheduler would consider the whole task graph with all its dependencies to compute the whole application schedule. This is however far too expensive when typical task graphs contain thousands or even millions of tasks. Some approaches can be considered to

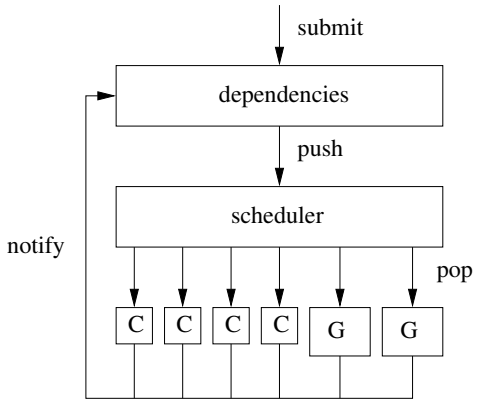


Figure 3.2: Runtime architecture around the scheduler.

mitigate the issue, as will be discussed in Section 3.1.3, but in general it is preferrable to make scheduling heuristics only consider the *ready* tasks, which can be scheduled immediately, and not investigate deeper in the task graph. This also allows to benefit from the abundant literature on scheduling independent tasks. Ignoring hints on the future ready tasks completely would be unwise since this is precisely one of the benefits of submitting tasks in advance, but we will discuss this in Section 3.1.3.2. For now we only consider ready tasks, which makes the whole picture simpler.

Figure 3.2 shows the typical software architecture around the core of the scheduler, which decides which computation unit will execute each task. Above the scheduler, layers of dependencies prevent tasks from entering the scheduler before they get ready. Below the scheduler, computation units request for tasks to execute when they become idle.

This is why in StarPU a task scheduler is basically composed of a *push* method, called when a task becomes ready, and a *pop* method, called when a computation unit becomes idle. The most basic task scheduler can then be a mere centralized task queue, as seen on Figure 3.3(a). This provides load-balancing, but does not permit data prefetching, since the scheduling decision is performed only just before execution. To be able to issue data prefetches, the scheduling decision has to be taken in advance, as shown on Figure 3.3(b). Scheduling all ready tasks in advance is however detrimental to performance when high-priority tasks become ready, and the established load-balancing actually becomes unbalanced. Typically, very low-priority tasks should be scheduled only at the latest. Figure 3.3(c) shows how a *scheduling window* can be used to only schedule a few tasks in advance. Eventually, it is not actually useful to decide in advance on which precise CPU core a task should be executed: to perform data prefetching into the main memory the runtime only needs to know that the task will execute on a CPU. Figure 3.3(d) shows how to actually combine the initial strategy with the refined strategies.

3.1.1.1 Building schedulers out of components

Implementing such scheduler structure in a monolithic way is laborious, and does not allow code reuse between the different strategies (each of which still makes sense on the various target systems). We have thus designed *modular schedulers* [SA14], which allow to easily implement and combine classical heuristic pieces stuffed in components [Szy03b] to build schedulers. For instance, the complex scheduler depicted in Figure 3.3(d) can be implemented with a combination of simple components (priority queues, decision functions, prefetch stages), as shown on Figure 3.4. More generally, this allowed to implement more complex schedulers than what we dared to implement with a monolithic approach. The construction of the graph of components simply follows the architecture of the system. Typically, a heuristic can be used at the memory

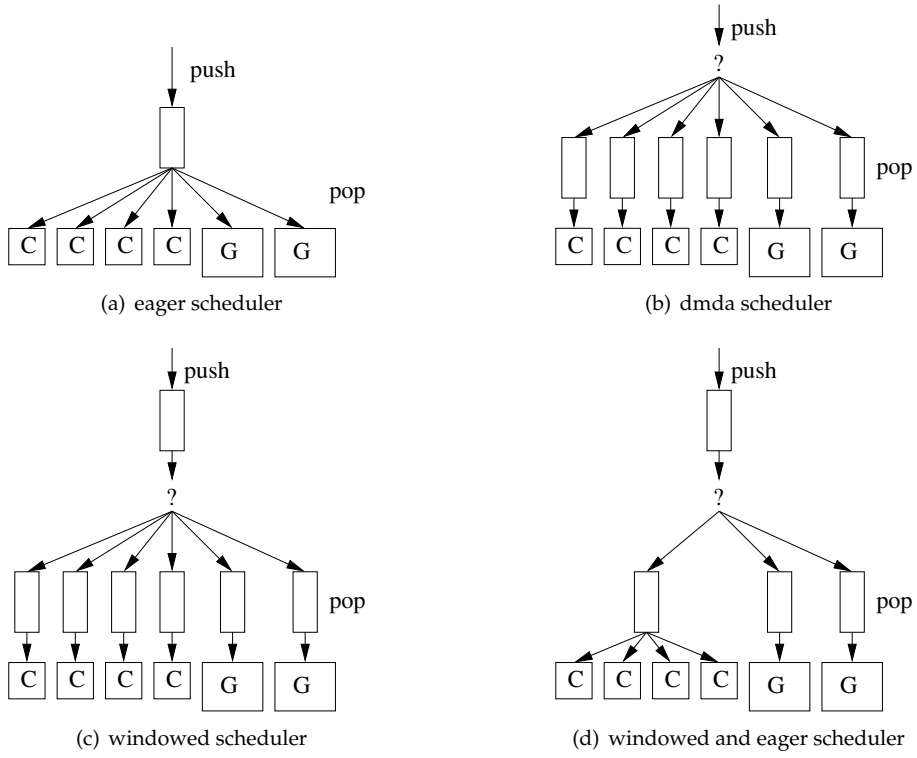


Figure 3.3: Scheme of schedulers with increasing complexity.

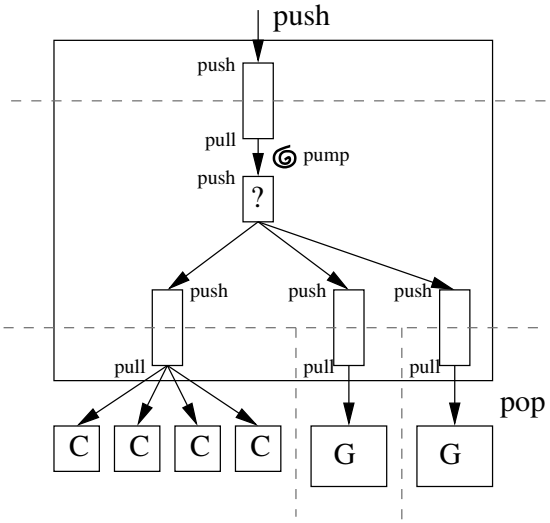


Figure 3.4: Modular implementation for the windowed and eager scheduler.

nodes level to determine a compromise between load balancing and data transfers, and locality-aware work stealing can be used between processing units within a given memory node.

Using components also allowed to completely move, out from scheduling heuristics, the implementation of the calibration phase used by StarPU to produce the performance models for tasks, which will be described in Section 3.1.2. A `perfmodel_select` component at the very top of the scheduler, not shown on the picture, filters out tasks which have not been calibrated yet, to schedule them with an eager heuristic which distributes them evenly to make calibration faster.

The scheduling components exchange tasks through *push* and *pull* operations. The graph of components can notably be decomposed in scheduling areas comprising the components between task queues, as shown with dashed lines on Figure 3.4. The top area gets tasks from pushes. The bottom areas see tasks pulled from computation unit idleness. The area in the center needs at least one scheduling *pump*, which is a mere `while` loop which pulls tasks from above and pushes tasks below, to make tasks progress between queues. Such pump has to keep making progress until task queues above are empty, i.e. the pull operation *failed*, or task queues below are full, i.e. *refused* the push operation. The crucial detail is that if some new task is pushed from above concurrently, the pump has to perform another iteration, in case the new task happens to be accepted by queues below, otherwise starvation can occur. Similarly, if a task is pulled from below, the pump has to perform another iteration, in case a task can fit in the resulting room.

This model actually highlights several ground runtime scheduling questions which arose while working on component-based schedulers in Bordeaux with Lionel Eyraud, and in Lyon with Louis-Claude Canon, Loris Marchal, and Adrien Remy. The first question is which runtime thread(s) should operate the pump.

- Making several threads operate the same pump means parallelizing scheduling, which can be beneficial for efficiency, but detrimental for scheduling decisions, since they may for instance decide at the same time to place a task on the same idle resource, which then becomes overloaded.
- When the thread which drives a GPU picks a task from a queue, thus leaving room, this thread should actually perhaps *not* operate the pump itself, and rather leave that work to another idle thread, so as to submit the obtained task to the GPU as quickly as possible.
- Similarly, when a task terminates on a GPU, the thread driving it has to release the corresponding descendent tasks, and thus push them to the scheduler. It should actually perhaps *not* operate the pump in that case either, to rather take another task from a queue to be executed on the GPU as quickly as possible, and leave the scheduling work to other idle threads.

So it seems that we need to separate the notion of notifying of a new situation (new tasks or available room) from performing the actual entailed scheduling work, to make better use of resources.

Such question is emphasized when scheduling heuristics have a non-constant cost per task push, because it for instance goes through the list of ready tasks it has not scheduled yet (the scheduling window). A simple enough but useful extension of this model would be to introduce *batched pushes*, for instance in the case of a task releasing several descendent tasks, which should rather be scheduled altogether instead of being scheduled separately at each respective push. It could seem that pushing lists of tasks would be enough to get this, but in practice, with e.g. synchronization tasks which can hide a whole tree of tasks to be released, the implementation would probably be much simpler by notifying the scheduler when a thread starts pushing tasks, and when it is finished with pushing tasks. This would be a sort of scheduling *cork* similar to the `TCP_CORK` socket option.

This entails another technical extension which would improve performance: while one thread is running such algorithm for optimizing the schedule, double-buffering should be used so other

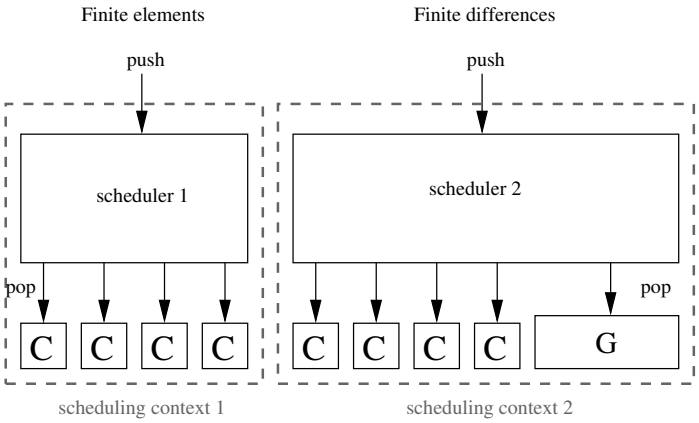


Figure 3.5: Two scheduling contexts share available resources. Two separate parts of the application submit tasks to the two separate schedulers.

threads can push yet more tasks without interfering.

Overall, using components to build schedulers is really appealing. It first allows to isolate the scheduling decision in a push/pull black box, all technical details being handled by other components. This typically means that scheduling theoreticians can work only on the scheduling decision function in Figure 3.4 and not have to care about any other runtime-related questions, and notably not the questions mentioned above.

Building schedulers out of components also lays the base for potentially *proven* schedulers. This component model seems similar to e.g. Kahn Process Networks, it would be interesting as future work to use the corresponding literature to rationalize the push/pull/pump mechanism up to be able to prove that the assembly of component will behave correctly, notably for liveliness. We could for instance check that there are pumps at appropriate places. It could even be interesting to design a domain-specific language to describe the assembly, likewise to the Bossa language [LMB02], which makes it more convenient to write, and let the compiler statically check the correctness of the assembly, rather than only checking it at runtime.

3.1.1.2 Introducing scheduling contexts

Yet another step in abstracting task scheduling was made through the notion of scheduling contexts introduced by the Master and PhD theses of Andra Hugo [Hug11, Hug14] (advised by Raymond Namyst, Pierre-André Wacrenier, and Abdou Guermouche). Applications are indeed often composed of several relatively independent computation parts, which may have differing scheduling requirements, for instance when coupling finite-element code with finite-difference code. The idea is then to partition the system, to dedicate resources to each scheduling context used to execute each part of the application with its own scheduler, as shown on Figure 3.5.

The question becomes how many resources should be given to each context. Andra proposed [Hug13, HGNW13] to use an approach where a supervisor observes the progression of each scheduling context. This can be measured accurately by making the application provide the number of flops computed by each task. A linear program can then be used over the whole application, taking into account the execution times of the different types of tasks of the different contexts, to determine an optimized amount of GPUs and CPUs to be attributed to each context so they progress at matching paces. This can also be performed periodically to compensate a behavior evolution of the contexts.

Andra even proposed [JBSH16] to control sharing a given GPU itself between contexts, by parti-

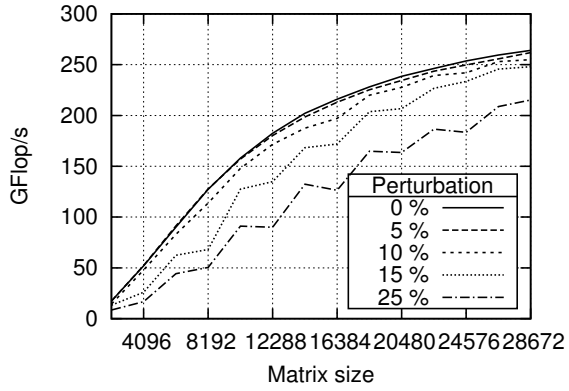


Figure 3.6: Performance variability depending on performance model perturbation.

tioning the set of Streaming Multiprocessors (SM) for task execution.

3.1.2 How accurate performance models can we have for scheduling tasks?

Most scheduling heuristics mentioned in the previous sub-Section assume the availability of estimations of tasks duration, to carefully pre-determine the execution ordering.

In StarPU this is done through automatic calibration [24]. The simpler version is to just compute the average completion time for various executions with the same data input size. More advanced performance models use linear or non-linear regression to be able to predict performance for varying sizes. The set of parameters to be taken into account in the regression is however not obvious. We have notably worked on modeling the duration time of sparse linear algebra computation kernels used by QR-mumps [SAB+15] and the kernels used by a Fast Multipole Method (FMM) [36].

This led us to a statistics-based methodology to establish multi-linear regression models (MLR) [36, SAB+15]. The principle is to have the application express all kinds of parameters which can potentially contribute to the algorithmic complexity: matrix dimensions, rank, tree level, number of interactions, etc., we will here take the instance from the application paper [SAB+15] which exhibits three parameters MB (height of the block), NB (width of the block), and BK (number of rows that should be skipped). Application execution then has to be performed over a wide range of input data, to capture a diversity of parameter values and the corresponding execution times. These timings can then be analyzed with R statistics to determine which parameters combinations are most significant. The list of these combinations (in the [SAB+15] instance, NB^3, $NB^2 * MB$, and $NB^3 * BK$) can then be included as such in the application source code. During subsequent executions of the application, the runtime will be able to automatically tune, from measurements on the target architecture (possibly different from the architecture where the initial statistics were performed), the scalar values to be used in front of each combination, thus getting for instance as a performance model $8.33 \times 10^{-1} + 1.59 \times 10^{-9}.NB^3 + 4.37 \times 10^{-7}.NB^2 * MB - 4.37 \times 10^{-7}.NB^3 * BK$.

Such performance models allow to provide fairly good duration predictions. Actually, it was shown [CJ09, BEDG18] that estimations do not need to be very accurate. As shown on Figure 3.6, when we tried to slightly disturb the models, performance remained stable, because dynamic heuristics can cope with such variation; only signification perturbations entailed real performance degradation.

3.1.3 Coping with algorithmic complexity of scheduling heuristics

The scheduling heuristics implemented in StarPU, which were mentioned in Section 3.1.1, use eager strategies and have constant scheduling cost per task. Such approaches are however short-sighted and can get really poor performance when e.g. locality between current tasks and future tasks has to be carefully taken into account to avoid data transfers.

A lot of theoretical work exists on scheduling dependent tasks, but their algorithmic complexity is usually $O(n^2)$, $O(n^3)$, or even $O(n^4)$, which are completely impractical for task graphs comprising thousands or millions of tasks. Some reasonable assumptions can be made to reduce the complexity, such as proposed in the PhD thesis of Florence Monna [Mon14] (advised by Safia Kedad-Sidhoum and Denis Trystram). For instance we can assume that GPUs always compute tasks faster than CPUs. We can also assume that a given machine only comprises a bounded number of different architectures (CPUs and GPUs, typically; at worse 2 different kinds of GPUs). We can also assume that there is only a bounded number of different types of tasks. Even the refined complexity can still be seen as too expensive to be applied as such, and we here discuss a few ways to get insight into the graph without introducing too much scheduling cost.

3.1.3.1 Application hints

The simplest way to provide the scheduler with insight into the future is to make the application provide scheduling hints. It can for instance set priorities on tasks to save a HEFT-based strategy from having to compute the upward ranks (which can not be computed in linear time in general), since the application often has a very good idea of what priorities should be. The application can also prefetch data into GPUs, to e.g. provide an initial partitioning of the data, which the runtime will be able to tune dynamically, instead of letting the runtime have to partition data blindly from scratch.

It would be interesting as future work to introduce an intermediate compile-time layer to compute such hints off-line and inject them automatically. For not too complex loop nests, a polyhedral analysis of the nest would indeed allow to compute critical paths algebraically, and inject the resulting task priorities directly into the generated code. Taking into account the data accesses could even allow to partition the task graph accordingly and emit initial prefetches.

It is not a problem for such hints to be wrong, at worse the obtained performance will be degraded, but the application execution will be correct. The application or an intermediate layer can thus experiment with them at will, the polyhedral analysis could for instance be very rough, even basic insights into the structure of the graph can bring very precious priority and partitioning information.

3.1.3.2 Incremental scheduling?

It was mentioned in Section 3.1.1.1 that it would be useful to make schedulers progress in batches. More generally, the question at stake is when scheduling should be done, and notably for schedulers which also consider the non-ready tasks. If the whole task graph has only one entry task, that task can be scheduled as soon as it is submitted, so that it can be processed while the rest of the graph is getting submitted and scheduled. If the task graph has several entry tasks, which one is best to execute first depends on the rest of the graph, so it would be best to let the application submit at least a part of the graph to get some insight before performing scheduling. Conversely, letting the application submit the whole graph may not be really useful. In practice, computing a precise schedule of the whole application in advance is not effective, since actual execution times vary from estimations, only dynamic strategies can have the performance robustness which was shown in Section 3.1.2. In the case of applications with a dynamic task graph, or dividable tasks as will be described in Section 3.4.2, this will not be possible anyway. A

partial view of the graph may however already be sufficient to compute a good-enough schedule. We have introduced a parameter which makes StarPU automatically wait for the application to have submitted a given number of tasks before calling the scheduler, thus getting incremental scheduling. This number can then be set to a reasonable enough value for the scheduling heuristic complexity. The application can also help by explicitly calling the `starpu_do_schedule()` function to mark the submission points where it is interesting to perform scheduling, because it has finished submitting a breadth-first layer for instance. Such approach will however not work if the task submission order is actually depth-first rather than breadth-first. Again, it would be interesting as future work to introduce a polyhedral analysis layer to automatically reorder the loop nest so as to submit the tasks in breadth-first order instead.

To even better benefit from such scheduling batches, the scheduling heuristic should be able to reuse the computations it performed in the previous batches. Theoretical work on such incremental scheduling is however lacking for now.

In cases where it is considered too costly to consider much more than the ready tasks (or the scheduling algorithm just does not support task dependencies), it can still be useful to take a small insight into the future by considering the tasks which will be ready *soon*. Indeed, for a given task B, when its last dependency A starts executing, the performance model for task A can be used to determine when A will finish, and thus when B will become ready. The scheduler can then pre-compute a schedule, and for instance trigger required data prefetches to the corresponding memory node. When A finishes and B becomes ready, hopefully the transfer will be finished already, and the scheduler will happily confirm the scheduling decision.

It is probably not useful for prefetching to take more insight than this. If it was, it would mean prefetching data for more than one breadth layer in advance, which probably means that the application has a poor computation/communication ratio, which will impact performance anyway, at least on systems with a little less bus bandwidth, and the application algorithm should be rethought to e.g. use Communication Avoidance strategies.

3.1.3.3 Hierarchical scheduling?

A way to efficiently get a glimpse into the task graph structure is to make the task graph hierarchical. Dividable tasks which will be described in Section 3.4.2 typically entail such hierarchy. This means that even if a scheduling heuristic has an $O(N^3)$ algorithmic complexity, with a two-level hierarchical graph which is for instance composed of $\sqrt{n}$ meta-tasks to be divided into $\sqrt{n}$ tasks each, applying the scheduling heuristic on the meta-tasks would have an $O(\sqrt{n}^3) = O(n.\sqrt{n})$ complexity, which would become reasonable. More levels of hierarchy could make even more costly heuristics reasonable.

The remaining question is what kind of decisions a scheduler would make for such a meta-task. They would be very probably very coarse since the performance models are probably not very accurate. But this could still be used as a visionary rough schedule which provides a good-enough visionary direction, and a simpler heuristic could then schedule the tasks to refine it.

3.1.4 A story of a few task schedulers in StarPU, from practice to theory

During the PhD of Cédric Augonnet [43], we implemented proof-of-concept versions of various well-known scheduling policies: list scheduling, random scheduling, work stealing [BL99a], ...

We notably implemented *dmda* [39, 11] which is based on the HEFT [THW99] heuristic. A first difference from HEFT is that it assumes that task priorities are provided by the application, instead of computing the upward rank, as was explained in Section 3.1.3.1. The fitness formula which is used for deciding which processing unit the task should be scheduled on is

$$T = \alpha.T_{compute} + \beta.T_{transfer}$$

to make a compromise between acceleration on GPUs and the cost of transferring data. The α and β parameters were introduced to be able to play with the compromise.

The other, main difference with HEFT is that since we implemented dmda as an online list-scheduling strategy, it only schedules ready tasks, and in particular does not consider the high-priority tasks which are not ready yet. The *dmdas* variant allows high-priority tasks to overtake other tasks, but they are still scheduled late if they are released late, and thus the scheduling decision may turn out to be inappropriate. The modular version, which was described in Section 3.1.1.1, gets a bit closer to HEFT by avoiding to schedule all ready tasks, but instead schedule just enough tasks to keep computation resources reasonably busy. High-priority tasks thus get scheduled after the few tasks which were scheduled in advance, and overtake them. They however can not overtake tasks which happen to have been started already.

The dmdas scheduler however proved to be quite effective and robust, even if its cost tends to become a problem with several dozens of CPU cores.

Our simple *ws* work-stealing policy was improved by Alfredo Buttari into a locality-aware work-stealing policy *lws*, which takes into account the hardware hierarchy to make processing units steal tasks from their neighbouring units before stealing from farther units. This allowed to provide scalability over several dozens of CPU cores, notably for the qr_mumps applications.

The *heteroprio* scheduling policy was initially implemented by Béranger Bramas to improve taking into account task priorities for a Fast Multipole Method application [ABC+14b], and then extended to support architecture heterogeneity [ABC+14a, ABC+16b]. The principle is to use one task queue per type of task, to sort the queues by acceleration factor between CPU and GPU execution (which is supposed to be the same for a given type of task), and to sort tasks on the queues by their priorities. GPUs can thus easily pick up the most accelerated and prioritized tasks while the CPUs take the least accelerated but prioritized tasks. Theoreticians have then studied it [ABEDK16] and improved it [BCED+16] and provided an approximation proof [BEDK17]. What was initially a very simple but quite effective heuristic for an application which did not perform so well with dmdas, actually ended up getting generalized and beating the well-known HEFT heuristic.

These stories show that people not directly involved in the StarPU project were indeed able to implement and improve scheduling heuristics, i.e. customizability of the scheduler did work, and actually even provided new ideas to theoreticians.

We have also started achieving the converse in collaboration with Louis-Claude Canon, Loris Marchal, and Adrien Remy: starting from a theoretical heuristic for independent tasks, and implementing it as a scheduler in StarPU. We also provide the heuristic with the application-provided priorities like dmdas. We used the modular way which was described in Section 3.1.1.1 to get the heuristic nicely isolated in a simple component. The preliminary results are promising, beyond the performance of dmdas. We will further discuss bringing theoreticians in in Section 3.1.8.

3.1.5 The compromise of optimizing for energy

As a proof of concept, we have tried to introduce energy awareness into the dmda scheduling strategy, by adding to the tasks completion times an energy penalty. A γ ratio has to be introduced to arbitrarily convert Joules into time, this is actually the necessary compromise choice between optimizing for time and optimizing for energy.

$$T = \alpha.T_{compute} + \beta.T_{transfer} + \gamma.Energy$$

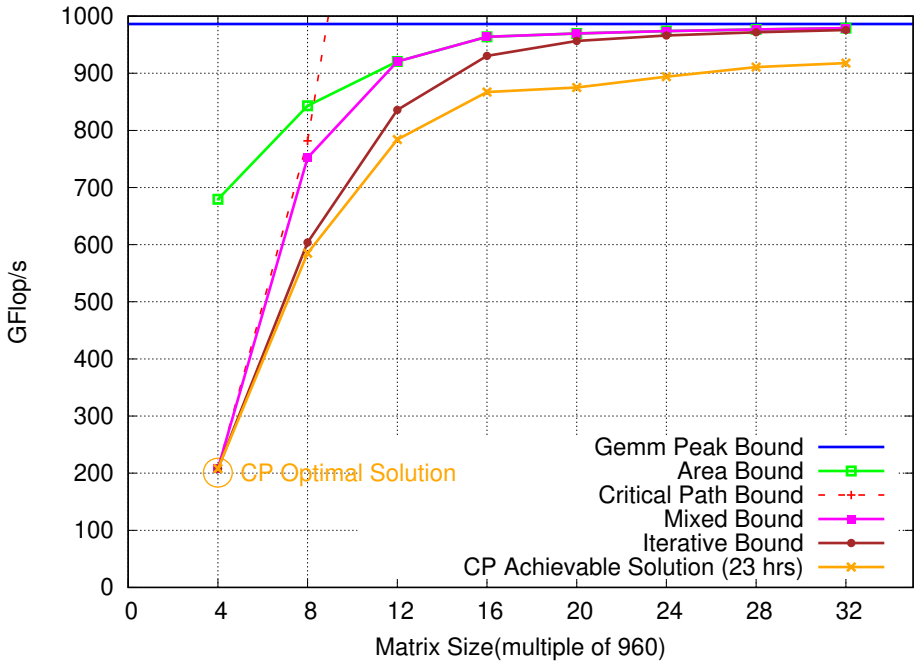


Figure 3.7: Theoretical performance bounds for the Cholesky factorization.

To be effective, this needs energy consumption models for the tasks, which is usually difficult to measure. The latest generations of NVIDIA GPUs, V100, provide a fine-grain consumption counter, but it is still updated only every few milliseconds, which is not so precise compared to typical kernel duration. It thus usually needs an offline calibration.

It is also necessary to measure the idle consumption of the system, otherwise the scheduler would only use the most energy-efficient processing units without caring about letting the rest of the system idle. Again, this can only be really measured offline.

3.1.6 How far are results from the optimum performance?

The various schedulers which were mentioned in Section 3.1.4 have kept improving performance. It is only natural to wonder how close they have gotten to the optimal performance. Since the problem is NP-complete, we do not easily have exact figures. A comparison commonly found in linear algebra literature is the GEMM performance, since this is the BLAS operation which usually has the best efficiency. This however ignores that linear algebra applications are not only composed of GEMM operations. Fairly good performance bounds can however be computed, as shown on Figure 3.7.

We first added to StarPU support for automatically computing the *area* bound [8], which ignores task dependencies and benefits from the fact that there are usually not so many different types of tasks and only a few types of processing units. We can indeed then efficiently optimize a linear program which only records how many tasks of each type are to be scheduled on each type of processing unit, and compute the resulting completion time. This works very well for dense linear algebra on large matrices since the parallelism is so wide that dependencies do not really matter any more.

The area bound is however too optimistic on small matrices since the critical path can not be ignored in that case. Using the critical path only is also too optimistic when the matrix is not

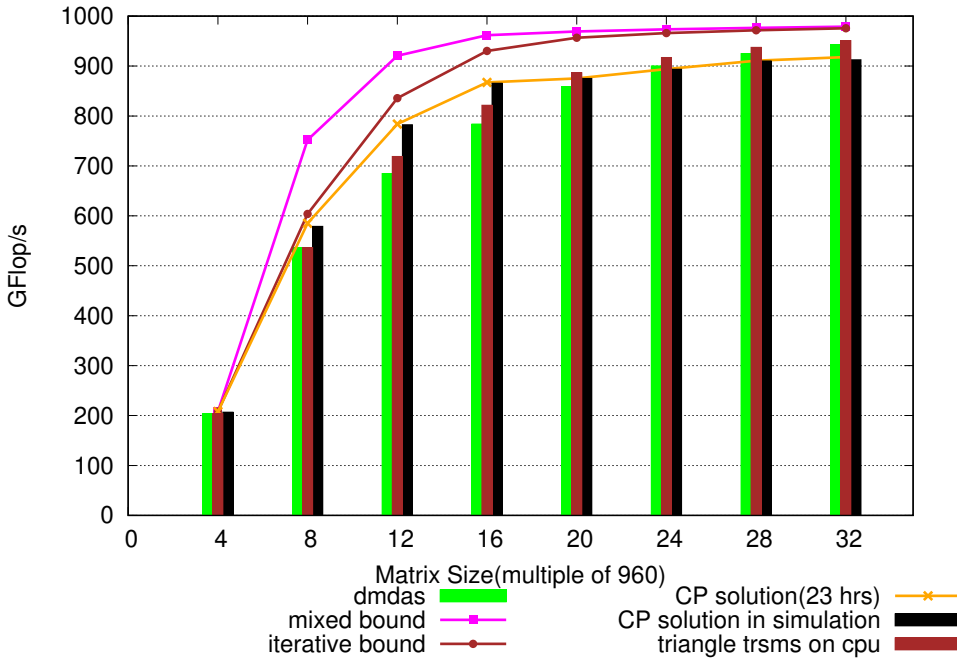


Figure 3.8: Comparison of measured performance against theoretical performance bounds.

so small. The *mixed* bound [23] combines the two bounds by adding the critical path to the area bound linear program. This was further improved into an *iterative* bound by iteratively adding critical paths as long as they are longer than the makespan computed by the linear program.

We also used a Constraint Programming formulation and ran CP Optimizer v12.4 for 23 hours. It was however only able to prove optimality of its result for the smallest matrix size, so in general it does not provide an upper bound, but an achievable performance.

Figure 3.8 compares some bounds with the performance obtained by some runtime schedulers. We can notice that the dmdas scheduling heuristic is not so far from the iterative bound, and a performance gap shows up mostly for medium-size matrices, which the PhD thesis of Suraj Kumar [44] (which I co-advised) tried to address. Among the attempts to close the gap, forcing the TRSM kernels which are far from the critical path, to be computed by the CPUs (as advised by the solution computed by the iterative and the CP bounds) does indeed improve performance. We also notice that with increasing matrix size, the CP solution indeed gets beaten by scheduling heuristics.

3.1.7 When should data transfers be scheduled?

To avoid seeing computation units getting stalled because they are waiting for the completion of data transfers, scheduling heuristics should take data locality into account. In the literature this is quite often done by minimizing the amount of data transfers, but this is actually not the best target, since bus bandwidth is available anyway, so it is best to just make use of it if that can make let some tasks execute earlier. This is actually why the approach used by dmdas, which was described in Section 3.1.4, works quite well in practice: it uses only a local view which balances the immediate computation with the time required to transfer data.

A question remains in both cases: when to actually start data transfers? In practice, starting them as soon as possible works quite well, provided that a high-priority transfer can somehow take

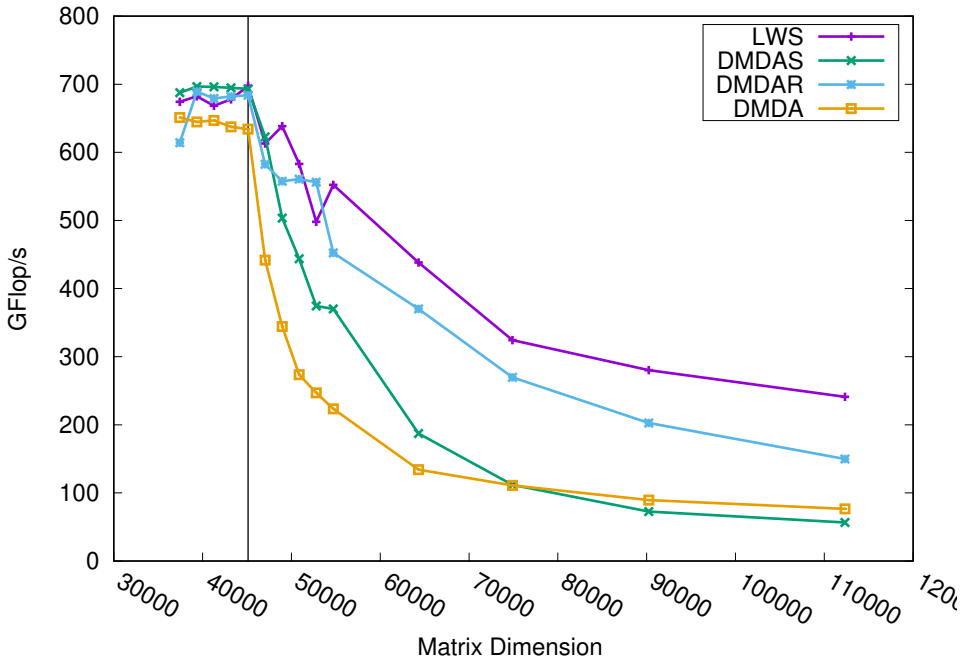


Figure 3.9: Performance of the Cholesky factorization with various schedulers, in Out-of-Core condition when matrix dimension is beyond the line at 45000.

over other transfers (i.e. at worse it has to wait for the completion of the currently ongoing transfer). This means that schedulers just need to avoid requiring too many transfers, and emit data prefetches as soon as they have determined them, and these will usually get nicely overlapped with computation without having to really determine data transfer ordering beyond just letting them get sorted by priority.

In theory there are a lot of situations where having to wait for an ongoing transfer can pose performance problem. In practice, if that is the case it probably means that the application is on the verge of having a poor computation/communication ratio, and the application algorithm should probably be rethought to require less communications. It would be interesting to make the runtime give as feedback how well overlapping happened, so the application programmer can know how close she or he is to the issue.

It is worth noting how much optimizing for locality conflicts with optimizing for critical path completion. Figure 3.9 shows the results obtained by different schedulers on an Out-of-Core version of the Cholesky factorization. It shows that as long as the matrix fits in the main memory, the dmdas scheduler performs best. When the matrix does not fit any more (beyond 45 000), the lws and dmdar heuristics are performing much better than dmda and dmdas. The gap between dmda and dmdar is not surprising: the difference is that dmdar ('r' as in *ready*) will first execute tasks whose data is already available on the target computation unit. The only difference between dmdas and dmdar is that dmdas takes task priorities into account, while dmdar just schedules and executes them in release order (and considers data availability as described above). The performance gap between them shows how detrimental it can be to insist on taking priorities into account while locality should here be privileged. This is confirmed by lws, which does not actually take data transfer time into account, but gets good performance just because it privileges locality.

That being said, other test cases have shown that the critical path has to be privileged for getting the expected performance. More precisely, for the tasks which are very close to the critical path,

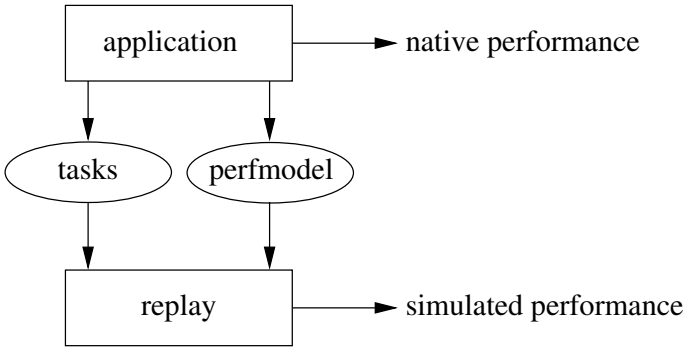


Figure 3.10: Making application test cases easy to run through simulation

priorities should be taken into account. For other tasks, it is best to use only rough priorities, to let the scheduler rather take locality into account and avoid data transfers. This tuning, between optimizing for the critical path and optimizing for locality, might also have to vary while execution progresses through the task graph. Such a balance between privileging locality and privileging prioritizing the critical path remains to be explored.

More generally, optimizing for memory use is highly challenging, as will be discussed in Section 3.2.4.

3.1.8 Bringing theoreticians in?

A lot of the work discussed above in this chapter has not actually been really confronted to the corresponding theoretical work on scheduling. I can see theoretical and practical reasons for this.

First, the optimization goals considered by theoreticians may not be what application programmers actually need. Notably, proven approximation ratios are not really useful in practice. Programmers indeed care much more about the performance in the common cases than in the worst case. The worst case itself is actually even not supposed to happen: the application programmer is not an opponent that a scheduler has to fight against, on the contrary, her or his interest is to get the scheduler to do a good job.

That being said, the worst cases, considered malicious in game theory, could be here rethought as *mistakes* from the programmer, which would actually be useful to detect and raise to the user, thus notifying where and why parallelism is lost. For simple applications, this is simple to spot with offline performance analysis as will be described in Section 3.7.1; for complex applications this is not enough, and average users do not want to have to understand such analysis anyway, so properly detecting these worst case would be useful.

As was discussed in Section 3.1.3, advanced algorithms from the literature also have often a high algorithmic complexity, and thus runtime programmers will tend to resort to much simpler heuristics and just adapt them to the particular class of application targeted by the runtime.

Ideally, theoreticians should be confronted with these use cases and provide appropriate heuristics. This is however not happening notably for an incidental practical reason: these applications are typically difficult to set up. They indeed often require various library dependencies and their configuration is often far from trivial. A symptom of this is the variety of projects meant to make installing HPC software simpler [1]. When even an HPC scientist has a hard time installing such software, a theoretician will just not even consider trying.

[1] conda, EasyBuild, Guix, Nix, Spack are compared in `https://archive.fosdem.org/2018/schedule/event/installing_software_for_scientists/`

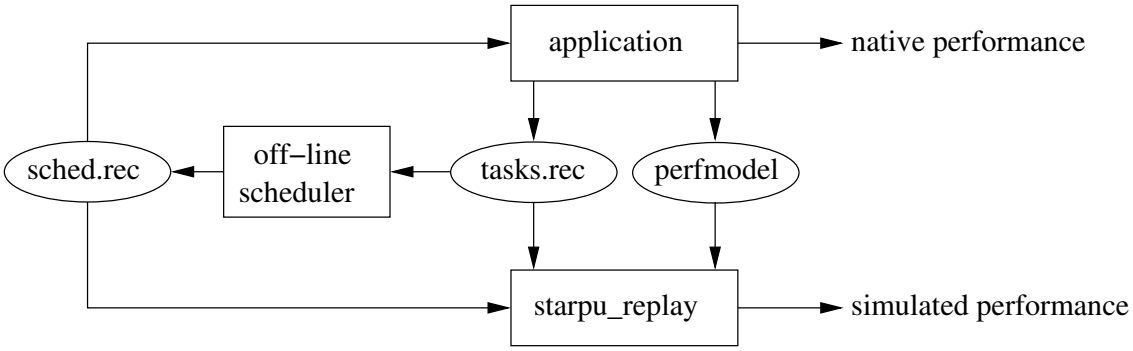


Figure 3.11: Injecting static off-line scheduling into execution.

We have started settling a solution thanks to execution tracing and simulation whose respective details will be discussed in Sections 3.7.1 and 3.5. The idea, as shown on Figure 3.10, is that authors of HPC software let the runtime running their application dump a task graph out of execution tracing, as well as the performance models for the tasks. A *replay* program can then resubmit the same task graph to the runtime, feeding a simulation layer with the performance models, and get performance result very similar to what can be measured with real execution. This replay program does not need the original application, or even to be run on the target architecture, it only needs the task graph and performance models. This means that theoreticians can just run it on their commodity laptop without installing anything else beyond the runtime itself. We have implemented a `starpu_replay` program doing this, and we started collecting task graphs in a task graph *market*[2], similar to the matrix market, for theoreticians to pick up various test cases for their algorithms.

The theoreticians' contribution can actually come two ways which can even be complementary. They can implement dynamic scheduling algorithms within the runtime and experiment with them with simulated executions (they can even ignore the algorithmic costs as a first approach), and then let HPC programmers use the algorithms implementations with reals application on real platforms. Theoreticians can also, as shown on Figure 3.11, implement offline static scheduling algorithms whose resulting schedule can be fed into the runtime in simulated executions. HPC programmers can then either use the static schedule as such in real executions, or run the offline static scheduler themselves.

Hopefully, these channels may help filling the gap between scheduling theory and HPC software stacks.

3.1.9 Discussion

In this Section 3.1, we have discussed how dynamic task scheduling can be achieved within a runtime system, and the relations with the corresponding scheduling theory literature.

It is worth noting that it is really not common for a runtime system to use advanced scheduling heuristics. StarPU is for instance one of the few generic-purpose runtime systems which really attempt to achieve HEFT-based heuristics (GAMA [Bar12] is another of the few examples); most systems use work-stealing and locality-aware eager schedulers. It is thus questionable why this is not more widespread.

One of the main reasons may be that runtime designers do to see it worth spending effort on advanced schedulers when more basic heuristics, once tuned, can already provide good-enough performance, and are easier to make robust. For instance, having to establish tasks performance

[2] `http://starpu.gforge.inria.fr/market/`

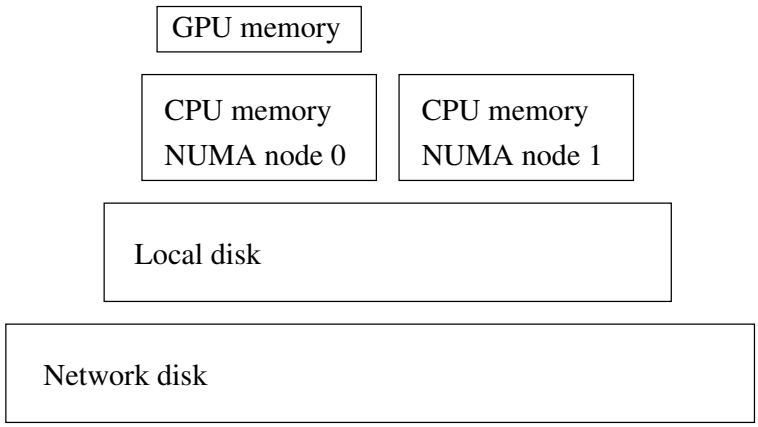


Figure 3.12: Hierarchy of memory levels

models is delicate. To make it easy to use, the calibration phrase needs to be automatically done behind the hood for instance. The StarPU scheduler modularity which was explained in Section 3.1.1.1 helps with this by moving the corresponding complexity out of schedulers, but that is complex to implement.

A more general reason may be that a lot of technical details arise when confronting theoretical heuristics with reality, not only the question of working out tasks performance models. For instance, in order to feed a GPU with tasks as efficiently as possible, one should always queue an additional task on the GPU while the previous is running, so that the GPU can start it as soon as the previous task is over. This means that schedulers have to provide several tasks in advance for GPUs, and the time of availability is less obvious. The question of which CPU should perform scheduling computation and when, as was discussed in Section 3.1.1.1, is also uncertain.

Theoretical scheduling heuristics, devised in a perfect world, thus have to be adjusted significantly when confronted with reality, to be able to give the expected performance improvements over simpler approaches. This makes it yet more difficult for runtime designers to pick them up from books. Hopefully, making theoreticians run a real runtime system themselves (as was explained in Section 3.1.8) will let them get a touch of these real-world issues and work on coping with these upstream.

3.2 Managing memory

The previous Section 3.1 was concerned mostly with task scheduling, and did not discuss memory management so much. That is however a growing concern: as shown on Figure 3.12, GPUs usually have discrete memory, whose content has to be managed; systems may also have different NUMA nodes, and the application data may not even fit in CPU memory, and disk space would thus have to be used to store data.

In this Section, we discuss the memory management aspects of a runtime. We first explain how we deal with data coherency between these different memory nodes. We then discuss dealing with the size constraint of memory nodes, by curing issues or by preventing them.

3.2.1 Managing data coherency between different memory nodes

Having to manage the coherency of different memory nodes possibly containing copies of the same data is a well-known problem, which is classically solved by using an MSI protocol. The

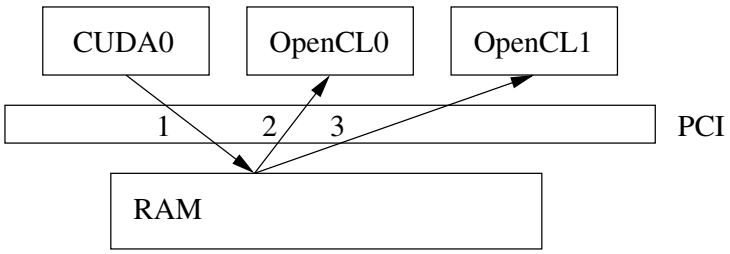


Figure 3.13: Using chained requests to set a dependency between request 1 and 2, and later reusing request 1 as a dependency for request 3.

interesting question is how to manage transfers between memory nodes, and avoid duplicate transfers.

StarPU uses a notion of chain of data requests [11], which is required when e.g. transferring data from a CUDA GPU to an OpenCL GPU, as shown on Figure 3.13: it has to be first transferred to the main RAM, and only then to the OpenCL GPU. If while the first transfer is going on it is determined that the data should be also transferred to another OpenCL GPU, only the data transfer from the main RAM to that GPU should be generated, and queued after the existing transfer from the CUDA GPU to the main RAM. In other words, the first transfer should be "reused" even if it is not finished. With more memory levels, some transfers may have to be reused even before they are started.

The generality of the notion of data request allowed to easily introduce Out-of-Core support into StarPU: the disk is simply a very large memory node which can be used as last resort when data does not fit in the main memory. It can be used like Operating Systems' *swap* space, in cases where the application's temporary data does not fit. It can also be used to store huge input (resp. output) matrices, from (resp. to) which StarPU will automatically load (resp. store) tiles as required by the task graph execution.

StarPU also uses several levels of data requests: fetch, prefetch, and idle fetch. Prefetch requests can be issued by schedulers as soon as they have decided where a task will be scheduled, and thus where data should be available when it starts. Fetch requests are used if the data is not available there when the task should start. Usually a prefetch request already exists and has just not been processed yet. It is then upgraded to a fetch request, which takes over all other prefetch requests. Idle fetches can be submitted by the application, to help the runtime with the distribution of data. They are processed only after prefetch requests, i.e. when the system bus would otherwise be idle. Idle fetches are also emitted by the runtime to write data back to the main memory, as will be described in the next Section. This allows to anticipate both the application's desire to eventually read the data from the main memory, and to make it trivial to release some memory from GPUs since the value is already saved in the main memory. Data requests are also sorted by the priorities of the tasks which triggered them, thus allowing to privilege transfers of data used on the critical path.

Such leveling and prioritization of data requests is really effective to optimize data transfer ordering, compared to what an Operating System can achieve without insight into the future. For now, the StarPU schedulers only emit prefetch requests which inherit the task priority. It would be interesting to try to refine them into prefetches vs idle fetches, and to possibly change the priorities, to decide roughly when data should be transferred, while letting the runtime core decide the details.

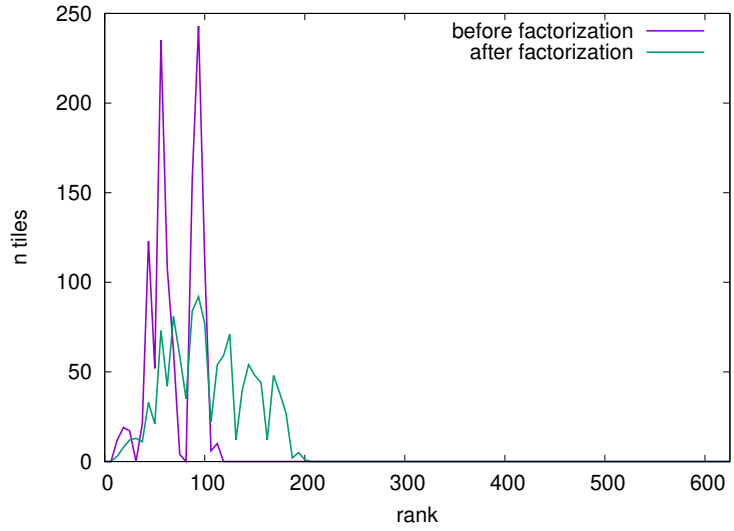


Figure 3.14: Distribution of tiles ranks before and after a Cholesky factorization.

3.2.2 How to deal with limited memory size?

The amount of memory available on a GPU accelerator is generally not so big, a dozen Gigabytes for instance. The main memory itself may also not even be large enough to contain all the application data. The runtime thus has to manage availability of free memory, to avoid overflows.

While the initial approach in StarPU, when free memory is not available any more, was to just evict half of the data to make room for new data, a more refined strategy is currently used, inspired by Operating Systems approaches. For each memory node, a list of allocated data is kept in Least-Recently-Used (LRU) order. Data transfers are periodically issued (*tidy* phases) to keep the head part of the list *clean*, i.e. a copy of the data is available in the lower level of the memory hierarchy. This allows to keep a fair part of the memory node trivial to release, thus making allocations easy even if the memory node is full of data.

The ordering of these lists of allocated data can be tuned by the application thanks to *wontuse* hints, which is similar to the POSIX' `madvise(MADV_DONTNEED)`: *wontuse* can be submitted by the application along the task graph to express which data will not be used by the task graph in the close future, and are thus good candidates for being written back and possibly even evicted. Conversely, data prefetches triggered by the scheduler act as *willmodify* hints, which allow to avoid spending bus bandwidth to write back data which will actually be modified again.

The combination of data writeback+eviction and data prefetch is however questioning: to what extent should data prefetches evict existing data from a memory node? Prefetching data means replacing old data with data which is known to be used in the future, but it could happen that a high-priority task gets released, which does need the data which has just been evicted. Prefetches should thus be issued only for data to be used in the *close* future, to avoid evicting data too aggressively.

Some applications even see their data *growing*. For instance, when using compressed dense matrices, the matrix tiles sizes tend to grow, because their ranks grow with the computation. Figure 3.14 shows for instance the distribution of tiles ranks before and after a Cholesky factorization of an h-matrix of the HI-BOX project[3]. While the tiles ranks of the input matrix are typically between 0 and 100 out of the 625 tile size, those of the output matrix are rather between 0 and 200.

[3] `https://imacs.polytechnique.fr/HIBOX.htm`

```
forall front f from fronts in topological order from leaves to root
  ! Wait for available memory
  do while (size(f) > avail_mem)
    wait_memory_release()
  end do

  ! Allocate front, avail_mem -= size(f)
  call activate(f)

  forall children c of f
    ! Assemble previously-activated child front into this front
    call submit(...)
    ...

    ! Submit release of child memory, avail_mem += size(c),
    ! to be achieved after executing assembly submitted above.
    call submit(deactivate, c)
  end do

  ! Factorize front
  call submit(...)
  ...
end do
call wait_tasks_completion()
```

Figure 3.15: Example of submission loop explicitly waiting for available memory.

Section 4.3 will additionally introduce the question of using some memory to cache data produced by other machines of the network, and Section 2.7.2 had additionally raised the question of copying data in case of WAR dependencies.

As a last resort to cope with such unexpected increase of memory use, a solution is to just stall the execution of the application, until some data gets evicted. We successfully used it for the HI-BOX project, and it allowed to factorize for instance 1600 GB matrices. In the context of Marc Sergent's PhD thesis [46] however, no disk is available, so we had to be particularly careful [29], by rather using over-estimations of memory usage, and stalling task submission itself before the overestimation overflows the memory available on the system. Once over-estimations are fixed and some cached data is flushed, submission can resume until the next stall. The first solution is not really satisfying, even if it can be kept as a last resort. It is better to use the second solution to avoid issues instead of fixing them.

3.2.3 Preventing memory challenges ahead of time?

More generally, we should rather prevent memory overflow concerns ahead of time instead of trying to cure them.

Decades ago, Dijkstra proposed a Banker's Algorithm [Dij65, Dij82]. The port of the sparse matrix solver qr_mumps to StarPU was made to use a similar strategy [ABGL14] to avoid overflows, in the context of the ANR SOLAR project[4]. The multifrontal method used there indeed involves variations of memory usage which can be overestimated, and can be naturally grouped in the tree activation step of the algorithm. As shown on Figure 3.15, this means that the task submission

[4] http://solhar.gforge.inria.fr/

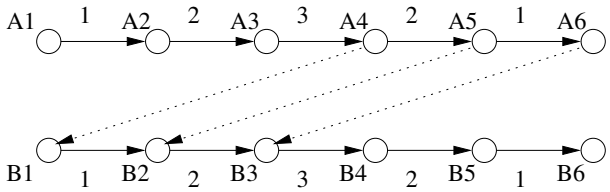


Figure 3.16: Task graph with memory use annotation and additional dependencies for respecting memory constraint 3.

loop of qr_mumps was easily split into phases, one per *front*, and each phase is not submitted until an overestimated amount of memory can be reserved for the memory requirements of the set of tasks for the phase. Memory is released during execution when tasks needing it are completed, thus letting the submission loop resume along.

Interestingly, this approach allows to dynamically adapt parallelism to the available amount of memory. If memory is very large, all tasks can be submitted without having to wait for any memory release. If memory is not so large, task submission will have to stall at some point, and wait for the release of the memory used by the first tasks, etc. If memory is really scarce, there is no other solution than completely serializing computation anyway. The only requirement is that allocations need to be performed in an order which can meet the memory constraint, i.e. the corresponding sequential execution needs to be safe. Otherwise, the submission loop would deadlock, waiting for the release of memory which will only happen after execution of tasks which have not been submitted yet. This guarantee can usually be provided at the application algorithm level.

The question at stake is how to generalize such approach, and to integrate it into the runtime system picture. In the qr_mumps case, integration can be achieved by just making it use a *blocking* memory reservation function instead of an explicit wait. The application algorithm is thus only composed of a loop using memory allocations and task submissions, that will occasionally stall on memory allocations.

Stalling the submission loop however prevents the runtime from getting insight into tasks which will be submitted after the stall. It would be preferable to let the application submission loop continue, and let whole sets of tasks stall *within the runtime*, just like for task dependency management. It would thus be useful as future work to introduce pseudo-tasks which only reserve memory. The application would then submit them within the task graph, and make the set of tasks using that reservation depend on such pseudo-tasks. The release of these pseudo-tasks would then be managed by a runtime memory scheduler. A simple one could be just following the sequential submission order, like happened for qr_mumps, but real dynamic Banker's algorithms could also be used instead, a proof of concept was implemented in StarPU by Arthur Chevalier [Che17] (advised by Pierre-André Wacrenier and Abdou Guermouche).

Making memory reservation a separate stage from the dependency stage may however prevent the task scheduler from performing some optimizations based on tasks dependencies, it would not be able to prefetch data for tasks which are about to become ready, for instance. During the PhD of Bertrand Simon [Sim18] (advised by Loris Marchal and Frédéric Vivien) it was proposed [MNSV18] to rather turn memory constraints into task graph dependencies: they proposed algorithms which add enough dependencies to the task graph to constraint it so that *any* task schedule will meet the memory constraint, while trying to keep the critical path minimized. Existing task schedulers can thus simply be kept unmodified, they will not be able to overflow memory, and will have a full view over the task graph. For instance, Figure 3.16 shows how adding dependencies (shown with dotted lines) can make sure that any schedule will require at most 3 units of memory, by preventing B1 from starting and allocating 1 unit before A4 has finished and released 1 unit, and so on with B2 and A5, and B3 and A6. This early work however

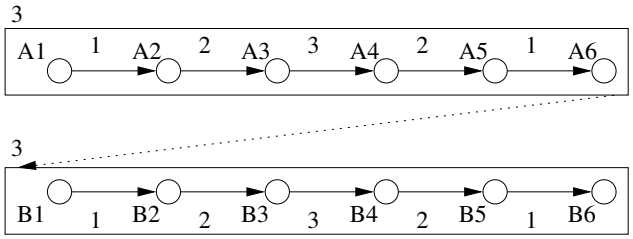


Figure 3.17: Task graph with memory groups and additional dependency for respecting memory constraint 3.

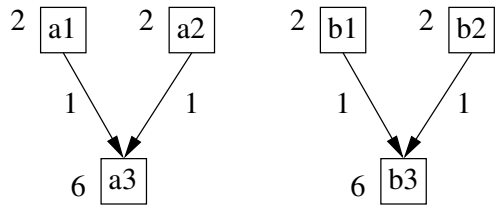
has a very high algorithmic complexity and produces many dependencies, while only the one between A4 and B1 is really essential, to prevent from the situation where A1 and B1 would start concurrently, and then it would not be possible to start A3 or B3 which require all 3 units. The other dependencies could be avoided by making B2 and B3 block when trying to allocate data while no unit of memory is available any more, and get unblocked when A5 and A6 release data units. In other words, it is the peak allocation of the group of task which is critical here, the rest can be managed opportunistically.

It would thus make sense as future work to create groups of tasks, and only take into account their allocation peaks. Dependencies could then be added between groups instead of precise tasks, as shown on Figure 3.17. This would reduce the cost of the memory scheduling heuristic just like was discussed in Section 3.1.3, the task groups themselves will be described in more details in Section 3.4.2. We would actually end up with a situation very similar to the qr_mumps example mentioned above; the difference is that if memory allows, groups could be unblocked in a different order than the submission order. For instance on Figure 3.18(a), a depth-first submission order (which is the safest order to use when the actual constraint is not known yet) would e.g. reserve memory for a3 before b1 and b2. This means that under memory constraint 8, a1, a2, b1, and b2 can not be processed in parallel since either b1 or b2 will have to wait for a3 to release memory, and thus wait for a1 and a2. Algorithms such as mentioned above could however determine that under memory constraint 8, the additional dependencies shown in dotted lines in Figure 3.18(b) allow any task scheduler to process a1, a2, b1, and b2 in parallel, then a3 and b3 sequentially, while being sure to respect memory constraint 8.

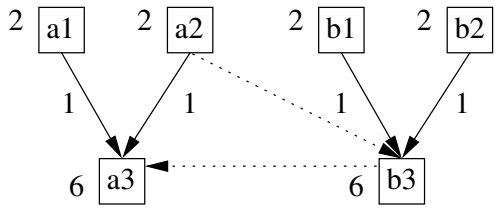
It is interesting to note that to reduce their complexity, such algorithms usually precisely benefit from the knowledge of a sequential order which respects the memory constraint, which an application can often just provide.

Figure 3.18(b) however also shows that the algorithm would have to arbitrarily decide whether to make a3 depend on b3, or the converse. It could be useful to devise less restrictive types of dependencies that task schedulers could take into account appropriately in such cases. Here we really only need to wait for three groups out of a1, a2, b1, or b2 before releasing either a3 or b3, whichever is ready for execution. This could be expressed by introducing notions of dependencies beyond one-to-one. On the incoming side, we could introduce *contribution* dependencies: several groups would explicitly contribute a given amount to the dependency (here 2 per group), and the dependency would be fulfilled when the total of contributions reaches a given threshold (here 6). On the outgoing side, we could introduce *exclusive* dependencies: several groups would subscribe to the dependency, and whichever group can be made ready first would "take" the dependency, and other groups would have to wait for its completion, and be made ready one after the other in the same way. Such extended dependencies could then be leveraged by memory constraining algorithms to reduce their overall complexity and number of emitted dependencies. For instance, Figure 3.18(c) shows how such dependency could be used in our example to get maximum efficiency.

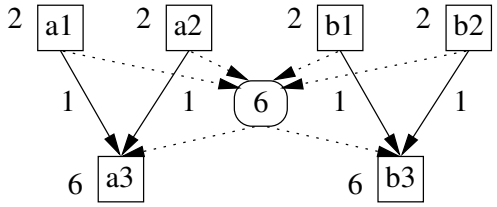
Another issue with such memory requirement model is with expressing data shared between



(a) Task graph with high parallelism potential, not reachable under memory constraint 8 by depth-first submission.



(b) Additional dependency to respect memory constraint 8 while reaching high parallelism.



(c) Advanced dependency to allow runtime flexibility.

Figure 3.18: Examples of introducing dependencies to respect memory constraint 8.

several tasks. Perhaps the model should actually be completely revamped, to only express data allocation and deallocation tasks, and to summarize computation tasks in between to reduce the memory algorithm complexity while keeping the knowledge of the parallelism structure for critical path minimization. This would bring it closer to the model we proposed during Paul-Antoine Arras' PhD thesis [42] (which I co-advised) for list scheduling under memory constraints [3, 9, 33]. Starting from the explicit allocation and deallocation tasks, we defined *memory sets* to include tasks between allocation and deallocation for a given data, and *memory clusters* which gather memory sets that intersect. We then ordered allocation tasks between memory clusters to meet memory constraints, and even make compromises between performance and guarantee against overflows.

All this being said, the over-estimations mentioned all along this Section may be questionable: if estimations are too large, fewer tasks will be submitted at the same time, reducing parallelism. The application can allow for more parallelism if it is able to refine the over-estimation of memory use during the execution of a set of tasks, by splitting the set into several parts with decreasing memory overestimation. In the case of qr_mumps, the 2D version (made possible by the use of the STF programming paradigm) generates so much parallelism that a simple overestimation was actually far enough. In the case of dense compressed matrices such as ACA [Beb00b], however, the compression ratio of matrix tiles can be typically as strong as 97% to even 99%. For a task whose input is such compressed tile and a dense tile, the compression ratio for the output can not be really bounded better than assuming an uncompressed result, i.e. typically $33\times$ to $100\times$ overestimation! In such case, requiring absolutely safe execution leads to very pessimistic estimations and thus poor parallelism, while data expansion is usually quite reasonable in practice. Figure 3.14 page 42 showed that it can indeed be $2\times$ only overall. A common way to avoid such huge overestimation is to just ask the user for what she or he thinks is a good estimation of data expansion. If execution overflows memory, it can be run again with a larger estimation. This is however not a particularly easy question to answer. As future work, we instead plan to try automatically using statistics to observe data expansion per task, cut the tail of the observation (which does not happen often anyway), and use that as an overestimation to be given to the algorithms mentioned above. The runtime system would then, as a last resort, use swapping in case data does not actually fit. Stochastic analysis on the whole execution behavior could even be used to be able to tell the user overall estimations such as "there are 99% chances that execution

will not overflow".

3.2.4 Discussion

To summarize, we are here faced with either making the computation fit in memory through careful scheduling constraining, or having to deal with several questions at the same time:

- optimizing task ordering for data locality
- writing back and evicting old data from memory nodes
- prefetching data which will be used

All these questions have unfortunately usually been addressed only separately. Task schedulers often optimize for data locality by caring about optimizing completion time, but they usually do not explicitly care about memory node overflow, and even data locality is often only an afterthought. Data prefetch is commonly employed by Operating Systems, but their short-sighted view prevents them from being aggressive enough to end up conflicting with data eviction (called swapping), so the two issues are usually managed separately. With the task-based programming paradigm, we have the unique opportunity to try to solve them altogether hand in hand, with insight into the future thanks to the task graph. It is generally agreed that data locality will be more and more critical for performance with the ever larger architectures to appear. As the previous Sections have shown, it is however far from obvious which approach should be considered, investigation should thus be pursued comprehensively.

We can however conclude from the existing work mentioned above that collaboration between scheduling heuristics, runtime system mechanisms, and application knowledge, might be key to success here: for instance applications can provide safe sequential order and runtime systems can help with memory accounting during execution. Conversely, even rough static task graph analysis can provide a lot of guidance for a runtime system to just refine it during the execution.

3.3 Taking advantage of commutative data accesses

Commutative data access is a case where it is interesting to make task-based programming deviate a bit from the pure sequential semantic of STF. This data access mode means exclusive access to a piece of data, but without a specified ordering. Put another way there is no inferred task dependency, the tasks are only prevented from running concurrently and can commute. This can typically be used when assembling contributions to a common buffer. For instance, in matrix inversion algorithms GEMM tasks can commute; if the application does not express commutativity the task execution ordering will be imposed by the task submission ordering, and it was shown [ABD$^+$11] that an unwise ordering can severely increase the critical path, and the application gets responsible for this. Expressing commutativity instead allows to just let the task order follow data availability time as it happens during execution, thus automatically optimizing the critical path. Commutative data access is currently being introduced into the OpenMP 5.0 standard.

The use case which raised the question of properly optimizing commutative data access in StarPU was a task-based Fast Multiple Method (FMM) implementation [ABC$^+$14] which required to perform a given computation on all pairs of a given set of data. A simplified version if shown on Figure 3.19. The resulting set of tasks has a very large potential for parallelism, but ordering it efficiently is not actually trivial, this is actually an instance of the classical dining philosophers problem [Hoa78].

```
for (i = 0; i < N; i++) do
  for (j = 0; j < N; j++) do
    if (i != j) then
      F(A[i]:RW | COMMUTE, A[j]:RW | COMMUTE)
    end if
  end for
end for
```

Figure 3.19: Use case for "commute" access which requires optimization.

The initial StarPU implementation, made for efficiency and scalability of the implementation itself, was using one ticket lock per data, taken by data address order to avoid deadlocks. In the example above it basically results in severe serialization. Conversely, using a centralized implementation (which can thus at will compute an optimized order) would not be scalable in general. Using a static schedule is not recommended either, since time of availability of data may diverge, and should rather be compensated by a run-time optimization.

We thus introduced the notion of *arbiter*, which is a centralized implementation, but can have several instances defined by the application. The idea is to let the application confine centralization to reasonable scopes. In our example, one arbiter would be defined for the set of data A, and other arbiters could be defined for other unrelated sets of data. To cope with the case where a task would access several pieces of data belonging to different arbiters, ticket locks are used on top of the arbiters.

This approach indeed allowed to get almost all parallelism out of the FMM application, and will probably be useful for other use cases. For now, defining arbiters is done manually. It should however be possible to perform a static analysis of the source code at compilation time, to detect the sets of data which are accessed in commutative mode by the same tasks, and thus automatically create arbiters for each of these sets. Once more, this does not have to be exact, at worse performance will be poor because only one central arbiter gets defined, or because as many arbiters are defined as there are data, thus getting serialization through the ticket locks.

3.4 How big should a task be?

Ideally, the size of tasks should be as small as possible, so as to capture as much of the application parallelism as possible. As was mentioned in Section 2.9, this however quickly meets the question of overhead caused by the runtime.

Measuring the actual runtime overhead is not so simple. Martin Tillenius proposed [Til15] to measure the scalability of running many independent tasks of a given duration. Figure 3.20 shows for instance how well StarPU performs with its lws scheduler: running tasks taking 256 μs does scale up to 45 cores, but beyond this, scalability gets limited by the runtime overhead. This actually gives a fair answer: to be able to scale over the whole 64-core system here, tasks should take at least a few hundreds of microseconds. This is in line with other results [AKG+15]: "1 ms is usually fine, 100 μs start getting real troubles". Some runtime systems such as Cilk [BJK+95], KAAPI [HRF+10b], or SuperGlue [Til15] however scale much better and can thus accommodate much smaller tasks, at the expense of available features. Martin Tillenius' PhD thesis [Til14], for which I was opponent at the mid-PhD defense, includes various studies on how implementing features, notably implementing data versioning, impacts the overhead.

Task size also has an impact over performance of the task implementation itself. GPUs typically need very large tasks to be able to utilize all of it computational power. Conversely, CPU cores do not need so large tasks, and since several dozens of them are available, small tasks need to be used to obtain enough parallelism for them all. When both CPUs and GPUs are available in a

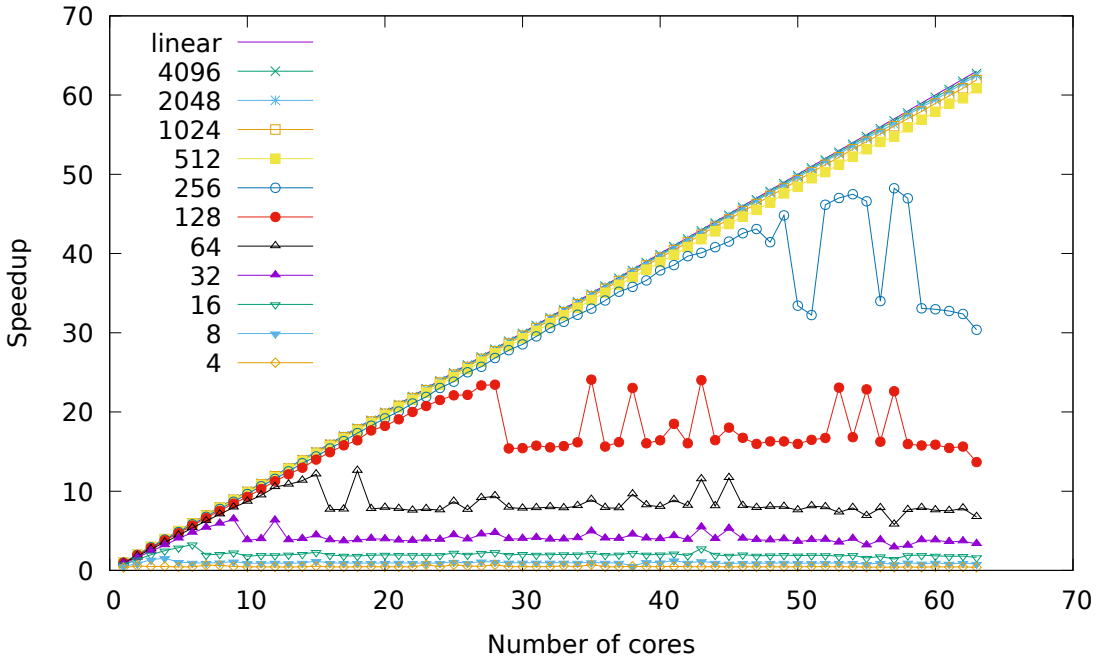


Figure 3.20: Scalability of StarPU with the lws scheduler, depending on task size in μs

given system, there is no good compromise, and it makes sense to even create tasks with various sizes, i.e. *introduce* heterogeneity of tasks to better exploit the heterogeneity of the architecture.

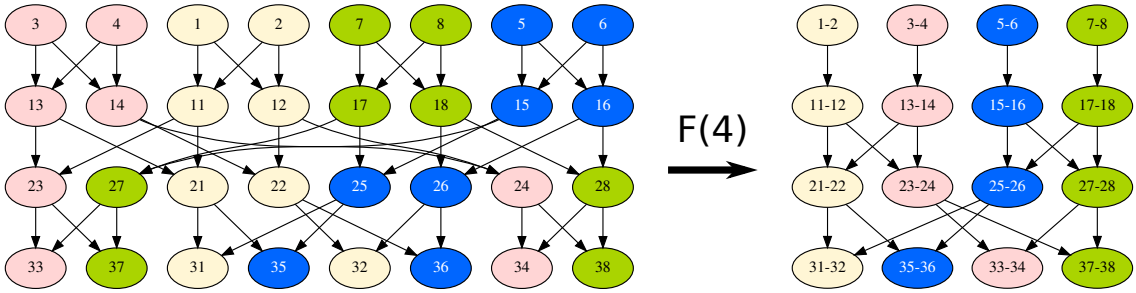
The compromise to be found is thus between questions of runtime overhead, task implementation efficiency, and load balancing. Various work such as the SCOOPP framework [SP99], or Capsules [MRK08] propose to automatically adapt task granularity, but their programming model departs significantly from task-based programming. In this Section, we will discuss some approaches which try to remain close to classical task-based programming. We first consider automatically assembling tasks, then automatically dividing them, and eventually examine how parallel task implementations can also be an answer.

3.4.1 Mangling the task graph

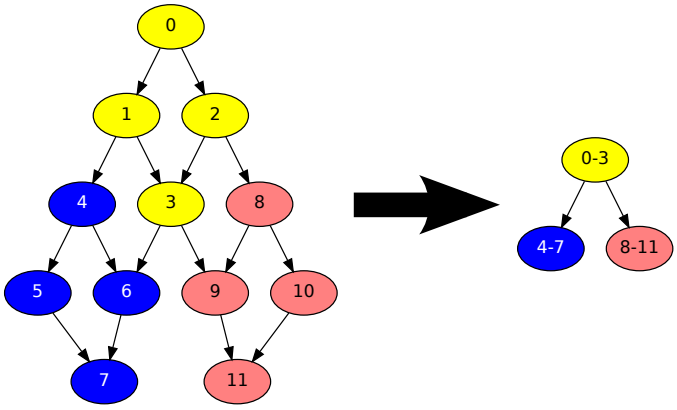
In Section 2.9, we had mentioned that for instance for Finite Element Methods (FEM), the smallest element of computation is extremely small, way below the reasonable size for a task, and that grouping computation could be performed at the application level to construct bigger tasks. Depending on the software architecture it may be relatively natural, and similar to blocked versions of loop nest for cache efficiency; it may on the contrary require unreasonable redesign.

The PhD thesis of Corentin Rossignon [45] (which I co-advised) proposed Taggre, which automatically mangles the task graph in its simplest form, before handing it to the runtime [28]. The principle is that the application can afford generating very fine-grain tasks, and operators are applied to aggregate these tasks efficiently until the resulting tasks get large enough to get efficiently processed by the runtime. Filters include simple cases such as aggregating consecutive tasks (Sequential operator) and aggregating tasks at the same graph depth (Front operator, whose effect is shown on Figure 3.21(a)), but also more involved operators such as the De-zooming operator, whose effect is shown on Figure 3.21(b).

An interesting aspect of this approach is that it is safe for the application semantic: the source



(a) Effect of the Front operator.



(b) Effect of the De-zooming operator.

Figure 3.21: Examples of task graph mangling.

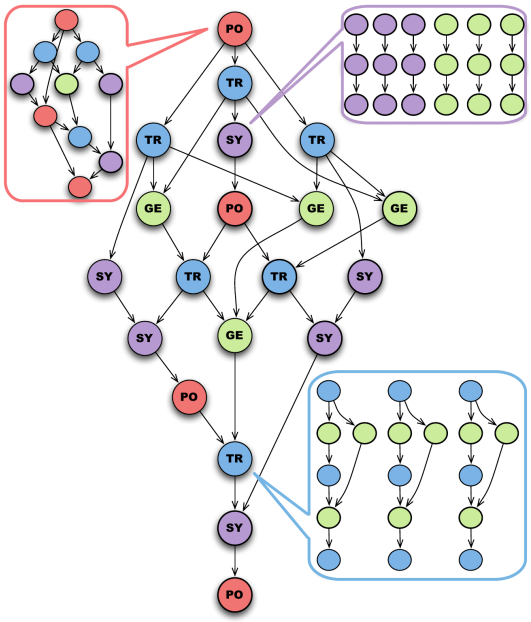


Figure 3.22: Recursive version of the Cholesky factorization, from [WBB+15]. A matrix tile would here be divided in 3×3 subtiles.

code does not have to be revamped to produce bigger tasks, the question has moved to only determining which sequence of operators should be applied on top of the fine-grain task submission, and at worse performance will be poor because tasks could not be aggregated, or parallelism was lost. In case the operators provided by Taggre do not produce interesting results, other operators can be invented, as long as they take care of not introducing dependencies loops.

Once more, expressing the task graph as a Parameterized Task Graph (PTG), as was explained in Section 2.1.3, would allow to perform such graph transformation algebraically, even before the instantiation of tasks. Allowing end users to just create tasks in a way that matches their algorithms is however much simpler than requiring them to write a parameterized version. Polyhedral analysis could also potentially be applied on the loop nest to extract the structure and aggregate iterations similarly.

Depending on applications, finding appropriate sets of tasks to be aggregated may however be very complex, in particular for very irregular application behavior such as graph algorithms. As was explained in Section 2.9, the task graph paradigm here shows its limits for very fine-grain parallelism.

3.4.2 Dividing tasks

Task aggregation as described in the previous Section somehow means rediscovering the application structure, while the application could just provide it. It is thus only natural to consider *dividing* tasks instead of aggregating them, to obtain a structured task graph. Task division could be done statically before execution, but such approach would not be able to adapt to run-time imbalances, we here discuss run-time task division.

Dividing work is the essence of the Cilk [BJK+95] language, the potential division is made opportunistically at function calls, as was detailed in Section 2.3.3. The functional programming interface proposed by Sylvain Henri [Hen13b, Hen13a] on top of StarPU follows a similar idea: at function call, the functional layer decides to submit one task or a task subgraph.

More generally, *recursive* task graphs were introduced in various runtimes. XKAAPI [GLMR13a] inherited it from its predecessor KAAPI [HRF+10b] and Athapascan-1 [GCRD98], OmpSs proposes it, as well as PaRSEC [WBB+15]. DuctTeip [ZLT16] also uses it to make distributed execution scale on large platforms, as will be discussed in Section 4.5. For instance, Figure 3.22 shows how tasks of the Cholesky factorization can be decomposed into task subgraphs. Here matrix tiles are divided in 3×3 subtiles: for instance at the bottom of the Figure a TRSM task (here denoted TR) gets subdivided into a subgraph of 9 TRSM subtasks, and 9 GEMM subtasks.

How subtasks should be expressed by the application and modeled within the runtime is a delicate question. A few runtime systems implement it by making tasks just submit the subtasks and wait for their completion. Making a task wait for another task is however technically problematic: making the thread that executes the task just wait would mean wasting the computation power of the CPU running it. Starting another thread on the same CPU makes thread and stack management much more complex. Instead, the wait function could actually run other tasks, but if one of such other tasks also requests to wait for yet another task, we can not resume the very first task before these, unless using a separate stack. Taking a step back, the main task does not really need to explicitly wait for its subtasks, we only need to declare it completed once its subtasks are completed. A preferable approach is thus to introduce into the runtime a notion of *dependency for the end of the task*: instead of delaying the start of the task, such dependency delays the *completion* of the task. The task implementation can thus just submit subtasks, add that kind of dependencies from the subtasks to itself, and return, and the runtime will handle the dependencies. Such implementation even supports nested subtasks.

An important concern with dividable tasks is the management of the coherency between data and subdata. For instance with dense linear algebra, tasks will take whole tiles as parameters, while subtasks will take subtiles, which may be parts of the former tiles, and thus require appropriate dependencies and proper pointer computation or even subtile transfers. In the StarPU implementation for dividable tasks, data is registered several times, for each granularity, and coherency is guaranteed as discussed below, similarly to asynchronous partitioning.

In StarPU, we have implemented dividable tasks by introducing the notion of *bubble of tasks* (inspired by the bubbles of threads introduced by my PhD [1, TNW07]). The principle is that at runtime, a task may be turned into a bubble before execution, the bubble generates a subgraph of tasks and waits for its termination. Bubbles behave very much like tasks, which allows to keep the same implementation path, except in a few places, which provide dependency and coherency optimizations described below.

Dividable tasks are most often implemented with a fork-join scheme. This is shown on Figure 3.23 which represents the bottom of the Cholesky task graph with SYRK and TRSM tasks subdivided to process 2×2 subtiles. Here, the whole task subgraph for the SYRK task has to wait for the completion of the whole task subgraph for the TRSM task. In StarPU, the bubble implementation does not introduce dependencies between tasks and subgraphs, and instead automatically introduces (P)artitioning and (U)npartitioning coherency pseudo-tasks, as shown on Figure 3.24. (P)artitioning pseudo-tasks subdivide tiles into subtiles, which subtasks from the subgraph for TRSM can naturally depend on, instead of depending on the POTRF and GEMM tasks. Similarly, subtasks from the subgraph for SYRK naturally depend on the subtiles produced by subtasks from the subgraph for TRSM. Eventually, the (U)npartitioning pseudo-task collects the subtiles into the tile for the POTRF task, and thus naturally depends on the subtasks from the subgraph for SYRK. This way, subtasks from both subgraphs can run concurrently. At the extreme, if the whole task graph is subdivided into subgraphs, the union of the subgraphs is exactly the Cholesky task graph for the smaller tile size, i.e. we *do not miss* any parallelism, and that even works recursively.

Introducing bubbles also allows to raise the commutative data access mode (which was detailed in Section 3.3) to a higher level: by specifying a commutative data access on bubbles, whole bubbles themselves can commute, thus allowing subgraphs to be reordered, which would be

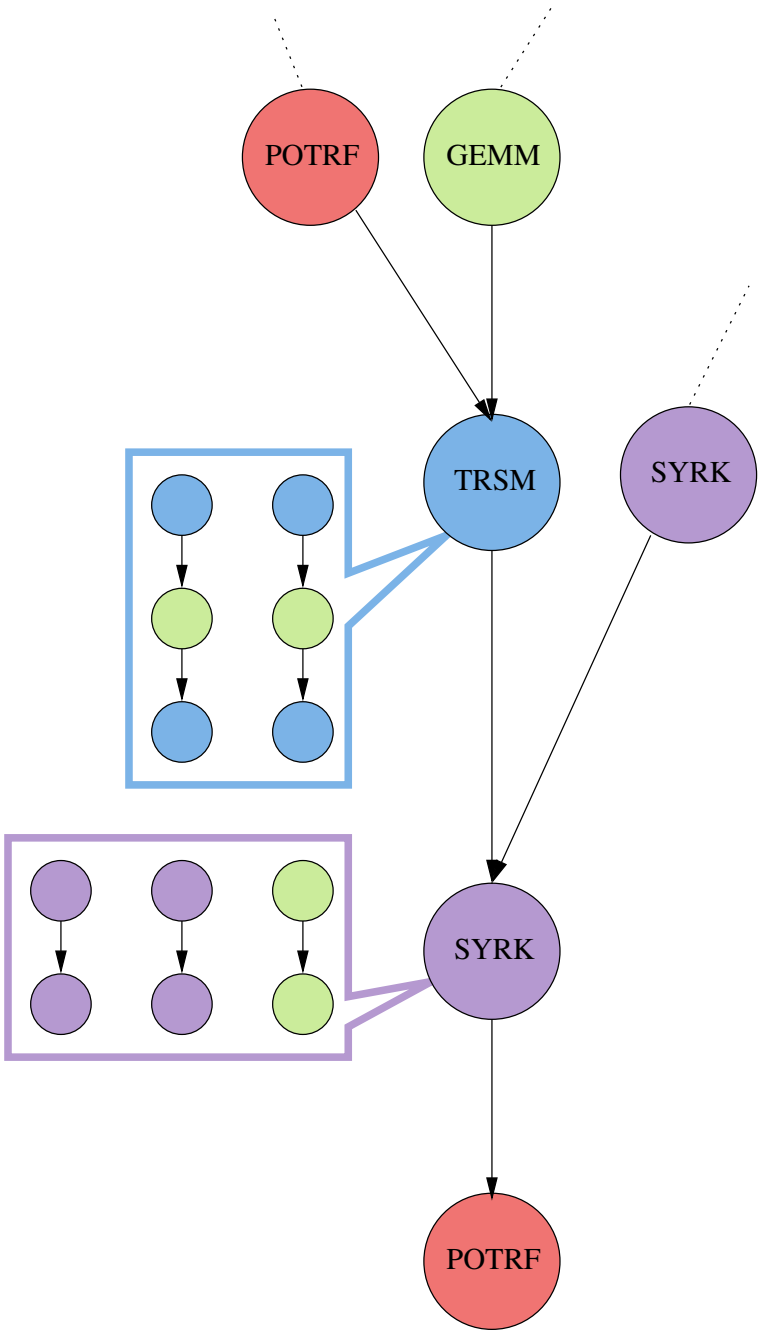


Figure 3.23: Subdivision of TRSM and SYRK, using subgraphs implemented through fork-join, thus strictly separating execution of subgraphs of tasks.

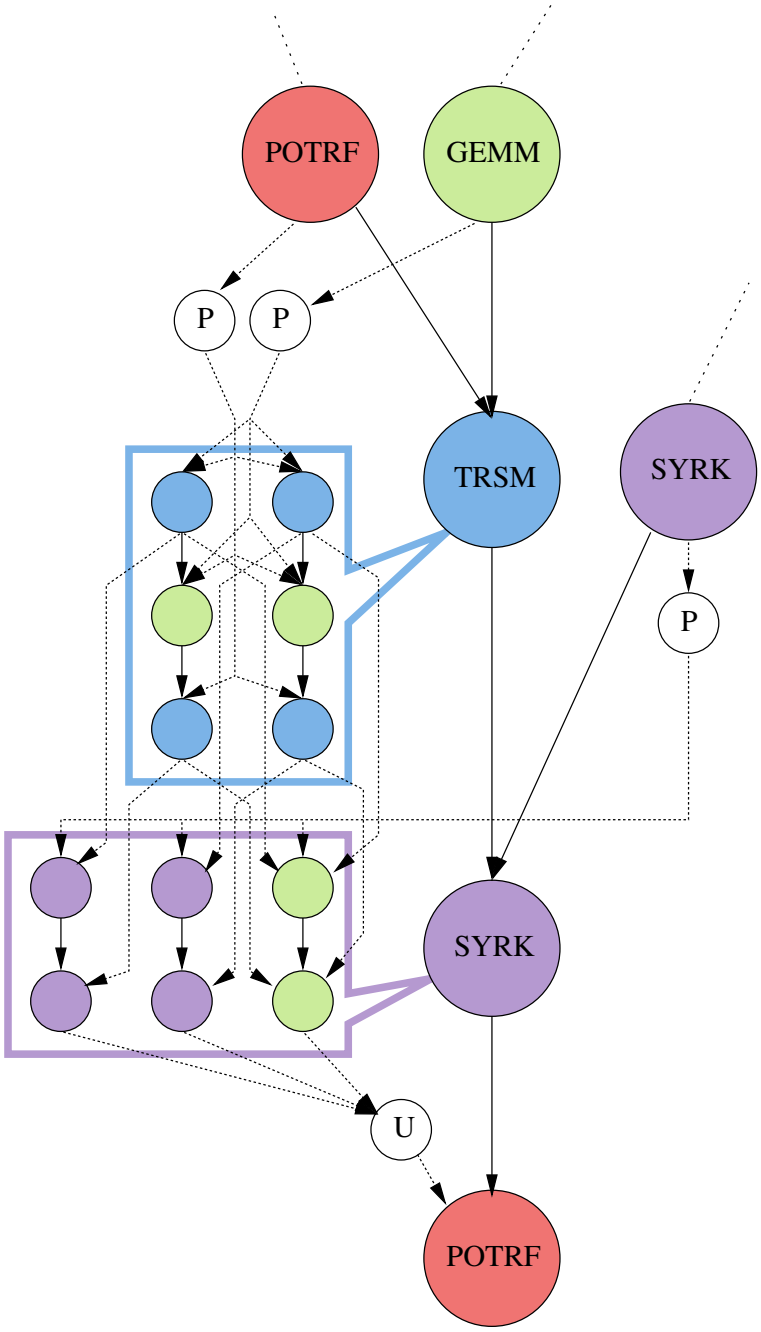


Figure 3.24: Subdivision of TRSM and SYRK, using subgraphs implemented through bubbles, allowing for fine-grain inter-subgraph dependencies, shown in dotted lines, which are automatically generated.

way more complex to express with only the complete task graph for the smaller tile size.

An interesting consequence of using dividable tasks is that submission of subgraphs does not have to be done in the application's main thread, it can be deferred, and thus achieved in parallel by various threads. The task graph submission can indeed become very costly, as will be discussed in Section 4.5 in the distributed case where this is exacerbated, and thus deserves parallelizing.

Version 10 of the CUDA runtime API provides support for submitting task graphs to GPUs, letting them handle dependencies etc. efficiently. The recursive task subdivision could thus proceed even further within the GPU itself. The API however provides limited flexibility, missing features such as the commutative data access which was described in Section 3.3.

A remaining question is how to decide whether to subdivide tasks, and how much, which is the focus of Léo Villeveygoux's ongoing PhD thesis (advised by Abdou Guermouche, Raymond Namyst, and Pierre-André Wacrenier). For dense linear algebra, the rule of thumb is typically [WBB+15] to use 1152×1152 tiles for proper efficiency on the GPUs, and 192×192 subtiles (i.e. 6×6 smaller) for CPUs. A strategy could be that CPUs privilege executing subtasks, and if none of them exists, just subdivide a task. The runtime could also systematically subdivide bubbles into a proportion of small and big tiles suited to the execution platform. It is worth wondering whether this could be generalized into just dividing when not enough parallelism is available. Ideally enough, the application would just submit one task for the whole execution, and let it get divided several times to exactly fit the required amount of parallelism. The runtime would then be able to completely control the flow of bubble refinement: it would be able to divide some bubbles, observe the entailed execution behavior, and infer from this the strategy to be used for dividing more bubbles, etc. This is also interesting for making the application quite naturally express the overall *structure* of the task graph, which would be widely useful to get an overview instead of having to rediscover it, for hierarchical scheduling (as was discussed in Section 3.1.3.3), or memory usage scheduling (as was discussed in Section 3.2.3), or making master-slave task distribution scale better (as will be discussed in Section 4.1).

Eventually, how to express such recursive task graph is yet an open area. The current state of bubbles in StarPU is such that they have to be expressed by hand with the StarPU API. We will need to get inspired from the DuctTeip, OmpSs, and XKAAPI work to figure out a convenient API to express them. Hopefully, the OpenMP standard will some day define a language-based way to express them with the fine-grain inter-subgraph dependencies, but this is for now very uncertain.

3.4.3 Leveraging parallel tasks, i.e. other runtime systems

In cases such as dense linear algebra, using dividable tasks may be overkill: for a given task, instead of dividing it and having to schedule the subtasks over CPUs, it may be beneficial to rather use a *parallel* implementation for the task, which will indeed divide the work, but may achieve a better job at scheduling it. Typically, it is very hard to beat the parallel MKL implementations of BLAS operations.

The overall picture can be seen on Figure 3.25: for instance, the runtime system schedules tasks either on GPUs or whole sockets. The actual execution in parallel on sockets is left to the task implementation. The runtime can either provide existing threads for this (so-called SPMD mode), or let the implementation manage its own threads (so-called fork-join mode). The latter can typically be used to just delegate the implementation to a library or even another runtime. Libraries can be used for well-known operations: for dense linear algebra for instance, a parallel implementation of BLAS operations such as MKL [int09a] can be just called. In other cases, the user may want to use a parallel programming environment such as OpenMP or TBB to implement the parallel version of the task. In both cases, the question of encapsulating the underlying

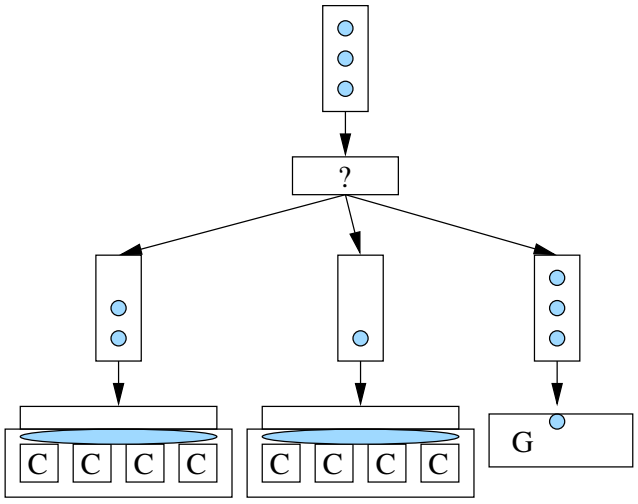


Figure 3.25: Scheduling and running parallel tasks with an existing modularized scheduler, by just introducing a component which runs a task in parallel.

runtime arises. We indeed end up with a main runtime which is supposed to tell sub-runtimes which resources they should use for executing the parallel tasks, i.e. runtime *composition*. The lithe [PHA10] framework was proposed to achieve a similar goal, the required adaptations were however really cumbersome and that did not get adopted by the community. A simpler way is to just *bind* the thread calling the parallel implementation to the CPUs it is supposed to use, and let sub-runtimes read that binding and use it. Runtimes however very often rather discover the available resources on the system, notably through our HwLoc [14] library, and ignore existing binding. Another way could then be introducing an HwLoc plugin that restricts the apparently available resources. A last issue but not the least, in most cases it is beneficial to run several parallel tasks concurrently, e.g. on different CPU sockets. The sub-runtime systems used for parallel implementations thus have to support getting invoked concurrently, which is unfortunately typically not the case of OpenMP implementations. At least BLAS implementations usually behave correctly.

Eventually, these parallel tasks need to be scheduled. In simple cases such as dense linear algebra, tasks can be just always run in parallel over whole CPU sockets. This indeed allows to just use a large tile size, which will be efficient both on GPUs and on whole CPU sockets. This makes the scheduling relatively simple: only few execution choices are available (GPUs and CPU sockets), and the computation power imbalance between a GPU and a whole CPU socket is not so big. With the modular approach to build schedulers, which was described in Section 3.1.1.1, existing schedulers can simply be leveraged as such as shown on Figure 3.25, by letting them only see the few available choices. Quite often however, it is preferable to decide whether to run a task in parallel or not, or even how many CPU cores should be used for the parallel task, i.e. *moldable* tasks. For instance, parallel implementations of the POTRF task of the Cholesky factorization usually do not scale so well, so it would make sense to rather run POTRF tasks sequentially. But POTRF tasks are on the critical path of the task graph, and the very first and the very last POTRF tasks of the task graph should rather be executed as parallel tasks, to make them terminate as quickly as possible, at the expense of efficient use of CPU time. During the PhD thesis of Cédric Augonnet [43], we implemented only proof-of-concept dumb strategies. The PhD thesis of Terry Cojean [Coj18, CGH+16] (advised by Abdou Guermouche, Raymond Namyst, and Pierre-André Wacrenier) proposed various parallel task modular schedulers for StarPU which indeed provided important performance improvements, even beating MKL in some cases. Similarly to the discussion of Section 3.1.8, it would here be useful to bring theoretical work on scheduling moldable tasks [DMT04, BHKS+16] in as well. Adapting the theoretical state of the art to the

reality of a runtime will however probably require substantial effort.

3.4.4 Discussion

We have discussed several approaches around task granularity. Task aggregation is mainly just contradictory with task division or parallel tasks, but whether to divide tasks or schedule them in parallel, or even both, is questionable. For instance, as mentioned above, the MKL mostly achieves a very good job at running BLAS operations in parallel, so dividing them in subgraphs will probably not bring performance improvement. Executing not-so-dense tasks such as POTRF in parallel however makes poor use of CPUs, and running them sequentially can pose problem on the critical path. In such a case it could make sense to divide POTRF tasks, and let the resulting sub-tasks get intermixed with parallel tasks. Some of these POTRF tasks are however so critical that it may be better to still let MKL process them in parallel. Deciding between the two cases is challenging: measuring the speedup of a parallel implementation is straightforward, but measuring the speedup of a subgraph implementation is questionable. We could use the total processing time of the set of subtasks of the subgraph, but this would not account for the availability of parallelism in the subgraph. How the subtasks get intermixed with other tasks by the scheduler can also largely change the overall efficiency of the parallel execution of the subgraph. Investigation to capture characteristics of subgraphs will probably be useful, but the scheduler behavior will be critical for such a choice.

3.5 Predicting application execution time

The previous Sections have discussed how to improve execution performance. In the context of the ANR SONGS project[5], a very different idea emerged: since StarPU has performance models for tasks as was described in Section 3.1.2, how about combining the SimGrid simulator [CLQ08] and StarPU to *simulate* the execution of a task-based runtime system, so as to be able to *predict* the performance?

3.5.1 Principle and implementation

The principle we used is shown on Figure 3.26. A StarPU application is first run on the target system. Performance models for the tasks are generated as was described in Section 3.1.2 from profiling this execution. The application is then run again, but with a version of StarPU compiled against the SimGrid simulator. In that case, instead of calling the application-provided implementations of the tasks, StarPU just tells SimGrid how much time the tasks take according to the performance model, and SimGrid accounts for it, considering all synchronizations involving the different tasks, data transfers, etc. which happen within StarPU, and maintaining a notion of *virtual time*. Eventually, SimGrid can tell the virtual completion time of the last task, thus a makespan of the simulated execution.

The implementation is actually relatively straightforward. As shown on Figure 3.27, the principle is essentially to replace POSIX threads calls with equivalent SimGrid calls. SimGrid can thus capture all synchronizations between the StarPU threads running the tasks, and thus the dependencies between executions of the different tasks. The virtual delays which replace the calls to the tasks implementations thus get correctly intermixed with the synchronizations.

It is worth noting that the Dongarra team at the University of Tennessee had attempted to achieve a similar simulation. They however did not use a complete simulation platform such as SimGrid,

[5] `http://infra-songs.gforge.inria.fr/`

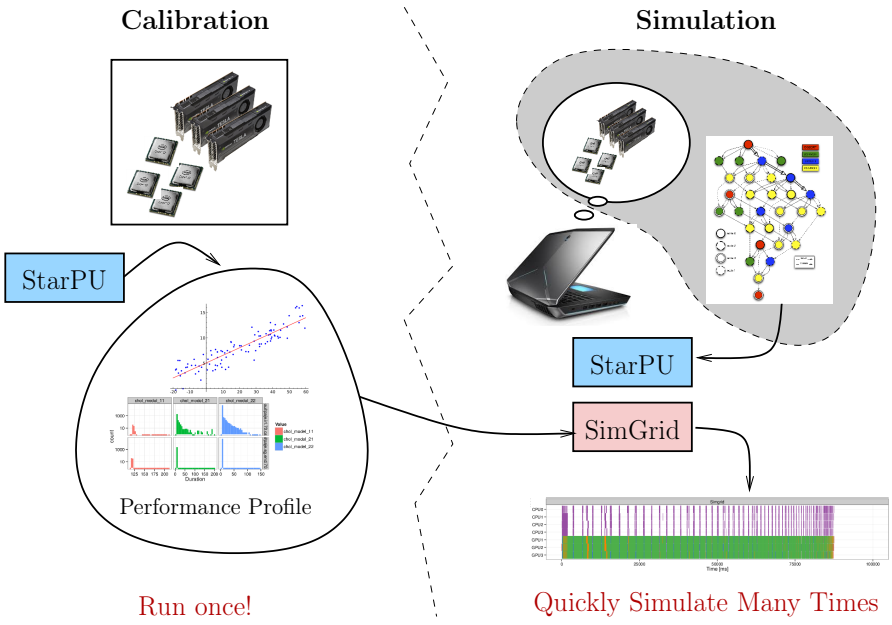


Figure 3.26: Simulation workflow using StarPU and SimGrid, from Arnaud Legrand and Luka Stanisic

```
starpu_driver() {
  while(running) {
    pthread_mutex_lock();
    /* fetch task to run */
    task = ...;
    pthread_mutex_unlock();

    /* call implementation */
    task->f(task->d);

    pthread_mutex_lock();
    starpu_finished(task);
    pthread_mutex_unlock();
  }
}

starpu_task_submit(task) {
  pthread_mutex_lock();
  ...  submit task ...
  pthread_mutex_unlock();
}

/* Application source code */
main() {
  for ()
    starpu_task_submit(t);
}
```

(a) Native version using POSIX threads and task implementation.

```
starpu_driver() {
  while(running) {
    xbt_lock();
    /* fetch task to run */
    task = ...;
    xbt_unlock();

    /* account task duration */
    msg_sleep(task->perfmodel);

    xbt_lock();
    starpu_finished(task);
    xbt_unlock();
  }
}

starpu_task_submit(task) {
  xbt_lock();
  ...  submit task ...
  xbt_unlock();
}

/* Application source code */
main() {
  for ()
    starpu_task_submit(t);
}
```

(b) SimGrid version using SimGrid threads and virtual delay.

Figure 3.27: Implementation principle for combining StarPU with SimGrid.

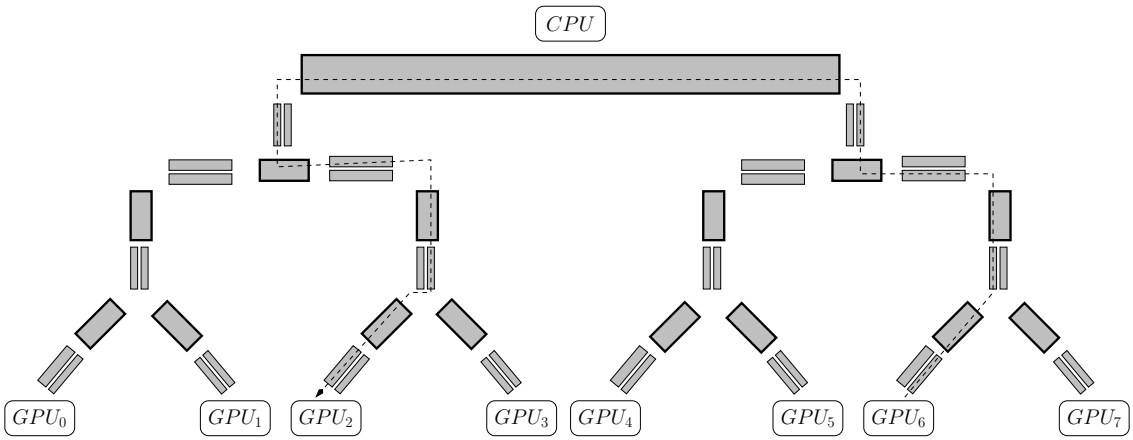


Figure 3.28: SimGrid model of the PCI tree

and thus precisely had issues with getting synchronizations between threads properly considered, and obtained inaccurate results.

Task execution on GPUs can be simulated the same way. Data transfers however need more modeling. Figure 3.28 shows the model used for the PCI tree of a system comprising 8 GPUs. The tree information is automatically retrieved on the target system from our HwLoc [14] library. The bandwidths of the different PCI links are inferred from offline benchmarking of data transfers between the main memory and GPUs, and between GPUs. StarPU can eventually describe this model as a SimGrid network, and tune SimGrid's modeling of network protocols to behave like the PCI bus. GPU transfers can then be replaced by SimGrid network transfers, and SimGrid will properly account for link bandwidth contention. We also had to introduce hardware specific particularities such as transfer serializations for the same pair of ends.

The principle of simulating the use of MPI is straightforward, since SimGrid provides an MPI implementation, SMPI. A lot of SMPI optimizations however had to be fixed to support being used concurrently with SimGrid's pthread-like interface.

Overall, the simulation principle proposed here is quite *coarse-grain*: we assume that the performance models for tasks and the PCI bandwidths properly capture the overall behavior of the microscopic activity. Other approaches usually use cycle-accurate simulators such as Barra [CDDP10] or Multi2Sim [UJM+12], but this is extremely costly and makes simulation much longer than real execution, while by simulating at the task granularity level, SimGrid simulation is much faster than real execution. Cycle-accurate simulators can however still be used to generate the performance models without having to actually execute the tasks implementations.

An example of the obtained performance is shown on Figure 3.29, more details are provided in the published papers [20, 6]. The simulated performance is extremely close to the native performance, and even when it diverges, the overall behavior and thresholds are correct. Quite often, discrepancy between simulation and native execution was actually a sign of issue in the native execution, such as a rogue daemon, a driver configuration issue, etc., or a hardware flaw. For instance, execution on the hannibal system shows a sudden drop of performance around matrix dimension 66 240 which is due to terribly bad performance of matrix subtile data transfers with the Quadro FX 5800 cards when the tile stride parameter gets that large. We thus had to introduce this flaw in the simulation to get accurate results. If the system has NUMA nodes, however, the accuracy is lower: the modeling proposed here does not take into account the position of tasks and data among NUMA nodes. The deviation remains small enough for a few NUMA nodes, but becomes unacceptable for very large NUMA systems, which fortunately are not very common. Properly modeling NUMA nodes remains for now a strong challenge.

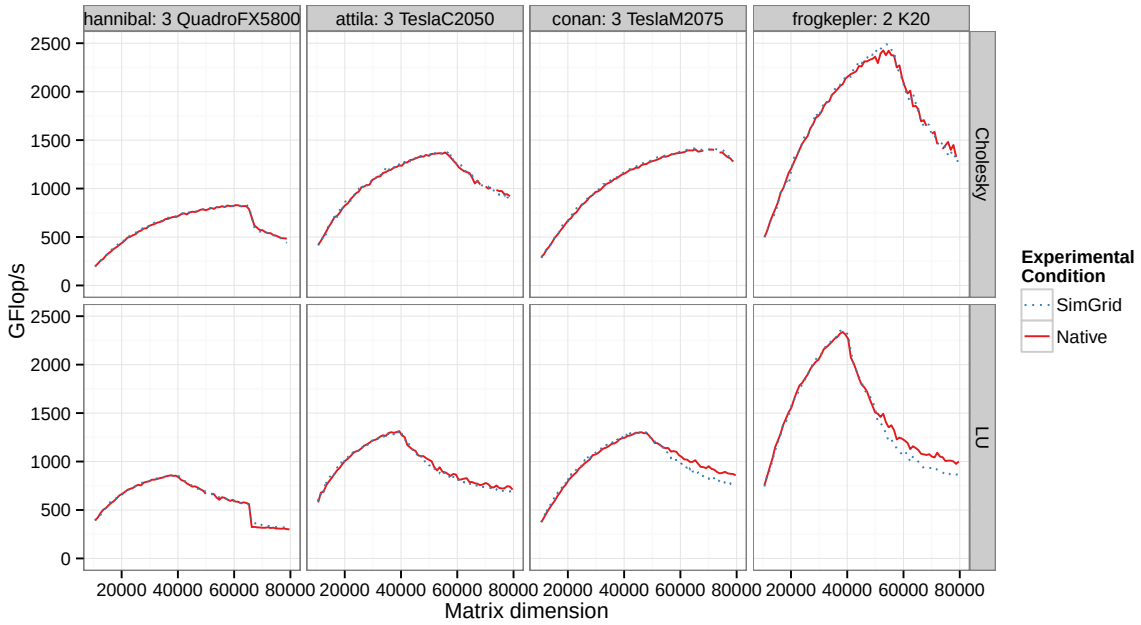


Figure 3.29: Performance comparison between native execution and simulated execution for various architectures.

As a result, provided that the measurements for the performance models can be trusted, and leaving the large NUMA case apart, we can be very confident with analyzing and optimizing simulated executions instead of running actual computations.

3.5.2 A wide new scope of possibilities

Such accurate simulations open up for a very large range of possibilities.

A first consequence is that simulated execution can easily be performed on a commodity laptop, instead of spending costly CPU.hours on the target platform, whose access may also not be convenient, or which may simply be offline for repair!

More generally, since SimGrid is deterministic, this opens up for real reproducibility: given the application source code, exact versions of StarPU and SimGrid, and the performance models for target platform, the simulated performance can be reproduced *exactly*. This gets away from all the usual issues with reproducing measurements: driver or firmware versions, BLAS implementation version, etc. or simply aging hardware.

This is extremely interesting for bringing theoreticians in, as was discussed in Section 3.1.8: they do not need to get an account on the target system, learn to build and run the whole software stack, fight with driver issues, etc. This also allows them to ignore some issues: for instance, setting the PCI bandwidth model to infinity allows to ignore data transfer issues while working on task scheduling. The cost of the task scheduler itself can also be ignored, to delay optimizing it until after getting interesting results. For instance, almost all of Suraj Kumar's PhD thesis [44] was achieved with simulations, and native execution was only used to confirm the obtained results. This allowed this PhD to have both a strong foot in numerical analysis and another in scheduling theory.

Beyond reproducing performance for a target system, it is also easy to modify the simulated platform. For instance, checking how performance evolve when modifying the PCI bus bandwidth model allows to easily determine whether some scheduling symptom is related to saturation of

the PCI bus, and similarly for the GPU memory size. Adding simulated GPUs also allows to check multi-GPU behavior. This can actually even be used as a tool for hardware provisioning, to check whether the PCI bus is fast enough for a given number of GPUs and a given application, or a faster bus is preferable.

Simulation can also be used for debugging: since the execution is deterministic, it is possible to set a breakpoint at an exact point in time where misbehavior was observed through offline analysis which will be described in Section 3.7.1. If a user has scheduling performance issues, she or he can provide the application source code and the performance models corresponding to her or his system, and the designer of the scheduling heuristic will be able to reproduce the issue.

To summarize, unsurprisingly enough, simulation provides the same kinds of benefits that it provides to other sciences[6].

3.6 Guaranteeing and verifying execution correctness

The previous Section has proposed an approach for execution reproducibility. An other appealing notion would be guaranteeing or at least verifying execution correctness, which is one of the focuses of the ongoing HAC-SPECIS project[7].

Guaranteeing or verifying the correct execution of a complete dynamic application would look daunting. The task-based programming paradigm however cleanly splits this in several parts.

- Guaranteeing that the task-based algorithm is correct. This part is up to the application. It is made simpler thanks to task-based programming because proving it or debugging it can be done assuming a sequential execution, since the semantic of the parallel execution is supposed to be the same.
- Guaranteeing that task implementations are correct. This part may be delegated to the implementors of the tasks, which can typically be the hardware manufacturer. Again, this part is made simpler because proving it or debugging it can be done with simple executions of the kernel: provided that the implementation does not use any global state, concurrent executions will behave identically. Compiler help could probably be used here to make sure that implementations do not access global state, or outside the given buffers.
- Guaranteeing that the runtime correctly respects the semantic of the task graph (such as task dependencies). This part is up to the runtime, but is then the guarantee is provided to all applications using this runtime. It is interesting to note that it is actually easier to treat a generic-purpose runtime, which just blindly processes tasks and avoids making specific cases, than treating an application-specific runtime, which has not been completely generalized yet and thus still contains remnants of specific cases. We here discuss this runtime part.

Treating the whole dynamic runtime at the same time seems overwhelming. The approach we are currently considering in the HAC-SPECIS project is to treat subparts separately: dependency tracking, data management, task scheduling, etc. Task scheduling itself can be treated in pieces: the modular approach (which was described in Section 3.1.1.1) to build task schedulers provides a way to decompose analyses, and the simplest scheduling components can probably be proven correct relatively easily. Similarly, we hope to be able to split the runtime into parts which are small enough to apply for instance model checking. Covering parts separately however does not provide guarantees on the overall behavior, which could e.g. risk *livelocks*. An overall model will

[6] A funny note is that most of the time here we simulate the execution of applications which themselves compute scientific simulations.

[7] `http://hacspecis.gforge.inria.fr/`

thus probably be required, similarly to a Petri network which has been proposed by Mironescu and Vinţan [I. 14] to model the StarPU runtime.

In some cases, stochastic analysis may be useful to provide some probabilistic level of guarantees. For instance, as was detailed in Section 3.2.3, for some applications whose data expands over time, and for which fully-guaranteed over-estimations can only provide completely unreasonable results, it would be interesting to still be able to e.g. claim that there are 99% chances that execution will not overflow the available memory.

3.7 Providing feedback to application programmers

Since task-based execution is entirely controlled by the dynamic runtime system, it is essential for a runtime to provide proper feedback on how execution proceeded, to be able to understand where performance gets lost for instance: are there scheduling issues, or is the task graph simply not expressing enough parallelism?

As was explained in Sections 2.2.1 and 2.5, it is delicate for application programmers to accept losing control over execution and having to trust a runtime system. This is also a reason why a runtime should provide feedback, so the programmer can still "feel" how computations are going on, and make runtime acceptance easier.

3.7.1 On-the-fly and post-mortem feedback

Providing feedback during the execution itself poses questions of efficiency of the feedback: if complex analyses are performed during execution, they will disturb it. That is why on-the-fly feedback is usually limited to simple statistics such as percentage of time spent running task, or spent scheduling, or wasted in idleness or waiting for data transfers.

Most analyses are thus rather performed post-mortem. For a start, statistics over the whole execution can already provide figures worth considering. For instance, an efficiency decomposition was proposed by Agullo *et al* [ABGL13]. Classically, one would consider the efficiency ratio between the time $t(1)$ used by a sequential implementation to process the computation and the aggregated CPU time $t(p)$ used by a task-based implementation to process the same computation on p CPUs:

$$e(p) = \frac{t(1)}{t(p)}$$

But with more precise feedback, $t(p)$ can be decomposed into the CPU time usefully spent computing tasks $t_t(p)$, the CPU time spent in runtime overhead $t_r(p)$, and the CPU time wasted in idleness $t_i(p)$, which results in

$$e(p) = \frac{t_t(1)}{t_t(p) + t_r(p) + t_i(p)}$$

which can be decomposed in

$$e(p) = \frac{t_t(1)}{t_t(p) + t_r(p) + t_i(p)} = \overbrace{\frac{t_t(1)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(tp)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(tp)}{t_t(p) + t_r(p) + t_i(p)}}^{e_s}$$

e_t measures the *task efficiency*, i.e. how decomposing the computation into separate tasks has made the overall purely computational time longer. e_r measures the *runtime efficiency*, i.e. how

Figure 3.30: Gantt chart visualization of an execution trace.

much the runtime overhead impacts execution time. e_s measures the *scheduling efficiency*, i.e. how well the task graph happened to be parallelized by the scheduler. This decomposition allows application programmers to determine how much performance is lost in their parallel implementation (e_t), in a loss of parallelism or misbehaving scheduler (e_s), or simply wasted in runtime overhead (e_r). The runtime efficiency can be further split, to e.g. separate out the time spent on scheduling tasks (which depends on the choice of scheduler) from the rest of the runtime overhead (which does not really change with the scheduler).

More advanced post-mortem performance analysis is usually provided through *execution traces*. To make tracing efficient during execution, dedicated trace formats can be used (StarPU uses FxT [DNW05]), but they are usually converted to more standard formats such as Paje [KSM00] or OTF [KBB+06] before visualization.

Tracing exactly the execution of each and every task can become prohibitive when running over long periods of time and at large scale. In such case, it would be preferable as future work to introduce *sampled* traces: for instance only periodic snapshots of the on-the-fly statistics would be recorded.

3.7.2 How to visualize the execution of a task-based application?

Execution traces provide very dense information on the behavior of the runtime. Depending on the focus of the user, different visualization strategies are needed.

3.7.2.1 Gantt chart visualization

A classical Gantt chart view of an execution trace is shown on Figure 3.30, as produced by the Vite [CFJ+] tool. The application case is a 10×10 tiles Cholesky factorization. We here have 9 CPU cores at the top and 3 GPUs at the bottom. The long red bars show idle time, and the different colors show the different types of tasks. At first sight we can notice that with such a small case, the scheduler used here (modular-heft, which was described in Section 3.1.1.1) scheduled almost

Figure 3.31: Trace visualization beyond only Gantt charts.

no task on CPUs, because GPUs are so much faster that CPUs would actually be lagging behind. The couple of exceptions are POTRF tasks, for which GPUs are not *so* much faster. We also notice that the end of the execution is lacking parallelism, leading to GPUs progressively getting idle. We can also notice that a lot of data transfers, as shown in black arrows, happen directly between GPUs instead of flowing through the main memory.

This simple kind of Gantt chart visualization is enough for small cases, or for taking a precise look at a known issue. For large cases, trace visualization is too dense to get a grasp of execution behavior, and idle time is typically showing up in invisible tiny pieces all along. To get a better overview, on Figure 3.30 we have added a few statistics along the Gantt chart, such as GFlop/s rate and data transfer bandwidth. In collaboration with the POLARIS team in Grenoble [26, 5, 35], we revamped the visualization into several panels, as shown on Figure 3.31, which can be easily tuned through R code. The application case is a 75×75 tiles Cholesky factorization over two MPI nodes, each node is comprised of 5 CPU cores and 2 GPUs (execution over MPI will be discussed in Section 4).

Panel 2 displays the classical Gantt chart, but idle periods have been separated out to Panel 3 to make them more visible. Panel 5 shows the number of submitted tasks over time, which clearly shows that the mere submission of the task graph takes about 1/6 of the overall execution (this will be discussed in Section 4.5). Panel 4 shows the number of ready tasks, and can be related to idle time shown on Panel 3: at the beginning of the execution, node 1 (in blue) has very few ready tasks, which is already surprising since the beginning of the Cholesky task graph exhibits a lot of parallelism. Panel 6 (resp. 7) shows the overall GFlop/s rate achieved by CPUs (resp GPUs) on each node, and indicates that except during idle times, it remain very high. Panel 8 shows the

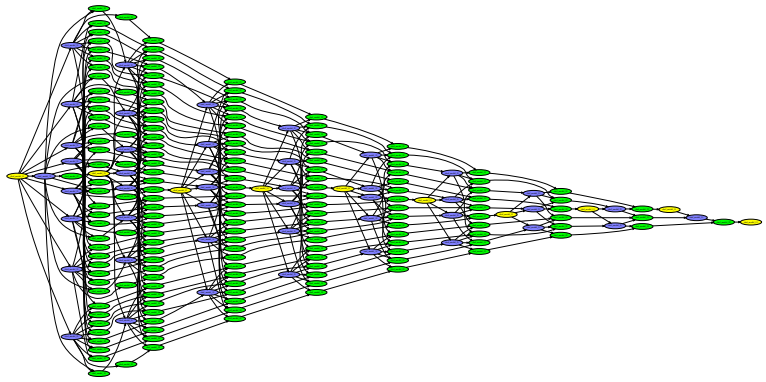


Figure 3.32: Graphviz [EGK+01] rendering of the task graph.

overall data transfers with GPUs and does not provide really conclusive information in this case. Panel 9 shows the overall data transfers between MPI nodes, which remains relatively low until the idle period around 2/3 of the overall execution. We had to additionally introduce Panel 10 to understand an erratic behavior here: it shows the number of on-going MPI transfers. It exhibits a ping-pong behavior, which is really not expected from an MPI implementation. Investigation has shown that the MPI implementation used here (OpenMPI) is aggregating messages too aggressively, thus the surprising sudden termination of a lot of transfers at the same time, and leading to almost no pipelining between computation and communication.

Panel 1 shows a much more advanced figure. Each horizontal line shows in green dots the execution times of the tasks of a given outer loop iteration of the Cholesky factorization. The first dot of a line thus represents the POTRF task, then further dots represent TRSM, SYRK, and GEMM tasks for the same outer iteration. Black lines are used to emphasize the execution time of the first and last tasks of the loops. The top line of the Figure thus shows that a lot of tasks from the very first loop iteration, even if submitted before the tasks from the other loops, iteration, are executed very late, long after execution of tasks from much later loop iterations. In other words, this Figure shows how much tasks from different loop iterations get *pipelined*, which is good for progressing on the critical path.

In the dense linear algebra case, it would also make sense to produce videos showing the evolution of tiles themselves (task execution but also data transfers). On the Cholesky algorithm for instance, it would show the tasks treating the top left corner of the matrix, then activity quickly expanding over the whole matrix, and finishing at the bottom right corner of the matrix.

The question is however how the two analyses above could be generalized. Loop pipelining can generally be useful as soon as the task graph is not regular, so we have added to StarPU functions that the application can use to explicitly tell the runtime the loop iterations and let visualization display them. Such addition to the application source code could easily be automatically done through a source-to-source compilation tool, making the Panel 1 analysis generic. The matrix video analysis is more questionable. It can however be argued that underlying computations usually have coordinates, if not exactly matrix indexes like in the linear algebra case. We have thus allowed to annotate data registered to StarPU with coordinates, which the application can set arbitrarily. More work is now needed to benefit from these pieces of information for more advanced visualization.

3.7.2.2 Visualization beyond Gantt charts

Many different visualizations are interesting to construct beyond mere Gantt charts.

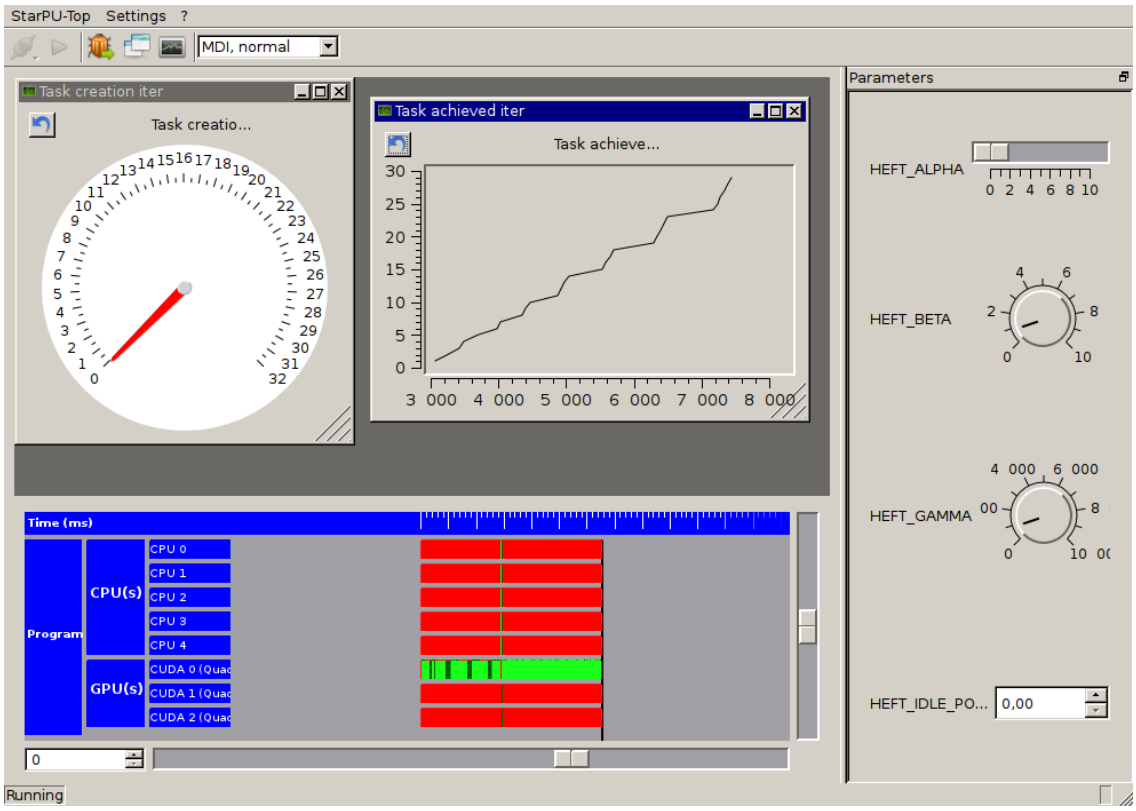


Figure 3.33: StarPU-top: proof of concept for an online task-based profiling tool.

Basically enough, the execution trace provides the task graph itself, which can thus be drawn automatically as shown on Figure 3.32. Providing such a feature is actually very useful for programmers to easily check that the generated task graph really is what they imagined. Depending on the graph rendering technique being used, it can also show the parallelization potential of the graph. Here we can easily see that the Cholesky task graph very quickly exhibits a lot of parallelism on the left, but the tail of the graph on the right remains largely sequential.

The Dask runtime [Roc15], geared toward more interactive use, proposes to dynamically show the Gantt chart as it gets executed: completed tasks disappear on the left, and scheduled tasks appear on the right. This allows to actually see the reasons for the scheduler decisions, according to the tasks already scheduled, which does not show up on complete Gantt charts where all tasks show up, not only the tasks scheduled at a given time. We have proposed a similar approach, called StarPU-Top, shown on Figure 3.33, which displays such a dynamic Gantt chart, but also various statistics, and tuning buttons which can be used to interactively experiment various values of the scheduling parameters (such as α, β, and γ of the dmda scheduling policy which were described in Section 3.1.4).

The modular approach to build schedulers which was described in Section 3.1.1.1 has also logically enough led to producing a visualization of the graph of scheduling components, and how tasks flow between components. Figure 3.34 shows the modular-heft scheduler in action on the 10×10 tiles Cholesky factorization execution, at one of the two moments where a task is eventually getting scheduled on a CPU. We can indeed notice that the priority queues dedicated for GPUs are already filled with many tasks, and thus the scheduler considers it worth executing a task on a CPU. Such a scheduling replay animation is extremely precious to check that a modularized scheduler actually behaves how it is supposed to.

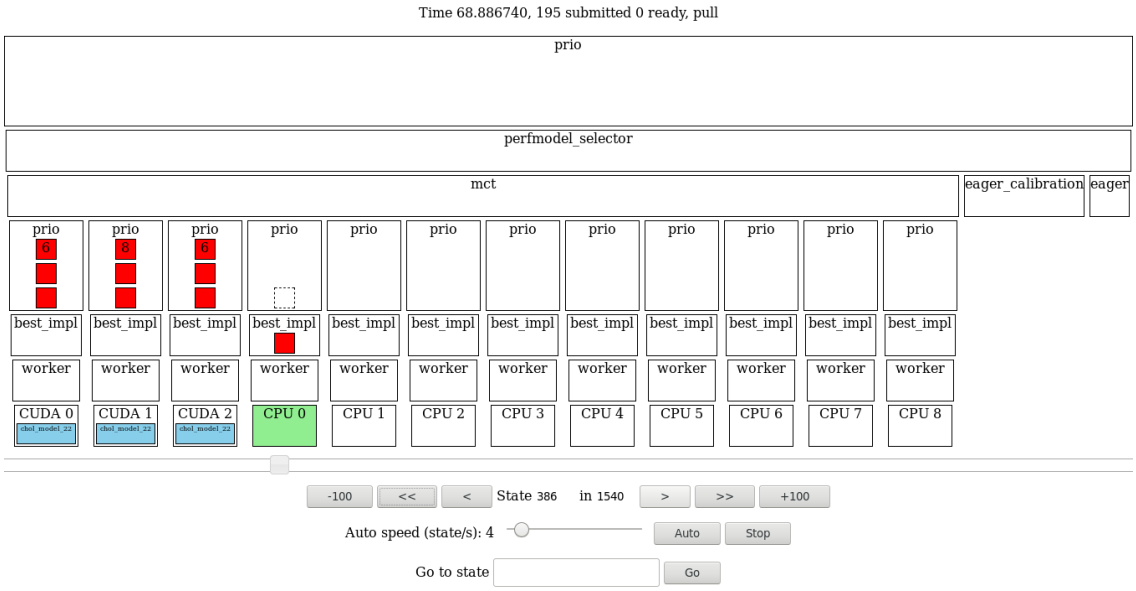


Figure 3.34: Visualization of the modular-heft scheduler, CPU 0 is shown pulling a task.

3.8 How to debug task-based application?

As was mentioned in Section 3.6, the separation between task graph submission and tasks implementations allows to debug them separately: tasks submission can be run in sequential order, and tasks implementations can be debugged in isolation. Overall behavior issues can however show up, and visualization tools described in the previous Section usually tell *when* issues happen, but often not *why*, and debugging tools specific to task-based programming need to be introduced.

Figure 3.35 shows the Temanejo [BGNK11] task-based debugging interface, written by S. Brinkmann *et al*. The principle is that some calls to its Ayudame library are added within the runtime at relevant places such as task submission, task scheduling decision, task start and completion, etc., which can be achieved in a couple of days. During execution of the application with the runtime, this library transfers information to Temanejo through a socket, allowing to dynamically display the task graph and execution progression. Conversely, Temanejo can tell the library to *block* under a given condition, this allows to set *breakpoints on tasks*. Once the breakpoint is reached, Temanejo can attach a gdb instance to the application, allowing to easily debug the task itself or the scheduler.

A similar approach using simulation was mentioned in Section 3.5.2. If profiling tools described in the previous Section provide a date when erratic behavior shows up, one can just set a breakpoint on that precise date and run the simulation again. Since simulation is deterministic, execution will stop exactly at the requested situation.

Livelocks are much more difficult to catch. They would usually not show up in simulations, but appear in native runtime executions. They are typically due to the use of *trylock* primitives in a situation where locks are rarely available. They would not show at reliable dates and are thus difficult to catch in a debugger. We have thus introduced a *watchdog* thread which can periodically check that the runtime has made progress since the last check. If no progress was achieved during a configurable amount of time, a signal is raised, to interrupt a debugger or generate a core file. It thus allows to inspect how parts of the runtime are interlocked within the period of inactivity.

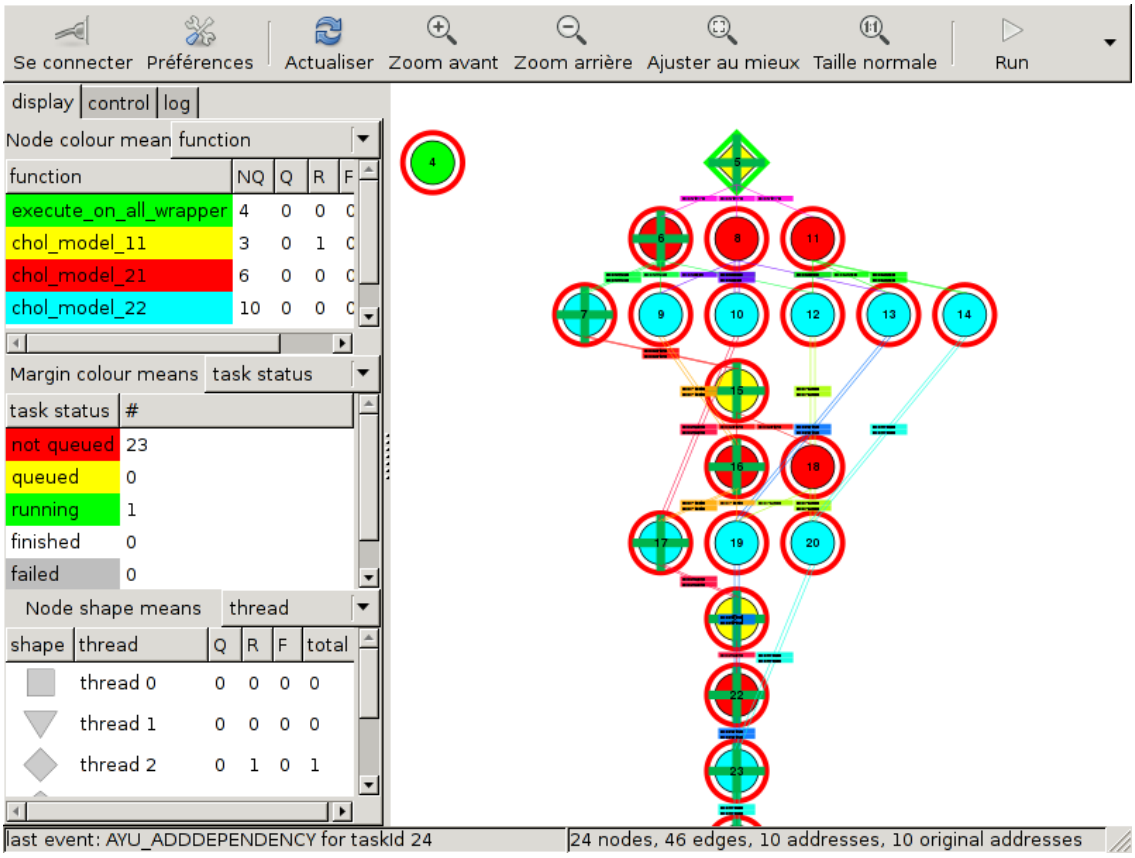


Figure 3.35: The Temanejo task-based debugger

3.9 Discussion

This chapter has shed some light on the hugely wide spectrum of support that a runtime system can provide thanks to task-based programming: not only task scheduling and memory management, but also execution simulation, guarantees, and feedback.

It is important to notice that this is really the task-based programming paradigm which made all of this possible. Thanks to the pure calling interface (tasks are not supposed to access data beyond their parameters), the runtime can conduct the task graph execution at will, benefiting from the knowledge of the task behavior and the graph shape. Thread programming interfaces, on the contrary, typically do not make appplication programmers express which thread accesses which data, and while automatic detection strategies can work, they can not be as effective as mere declaration of a task graph and task data access. One significant aspect however is the stability of performance for tasks. As was shown in Section 3.1.2, it does not have to be perfect, but for a runtime to be able to take smart decisions, at least rough stability is required.

We were able to integrate some well-known principles such as software components, Least-Recently-Used (LRU), and dining philosophers, to solve some of the issues that a task-based runtime meets. For a wide range of other encountered challenges, we however had to collaborate with other research teams in order to provide appropriate solutions. For instances, scheduling heuristics (Section 3.1) were obviously discussed with scheduling experts, multi-linear performance models (Section 3.1.2) were designed with statistics experts, simulated execution (Section 3.5) was designed with simulation experts, execution guarantees (Section 3.6) will benefit from collaboration with compilation experts and correctness experts, and trace visualization (Section 3.7.2) is an on-going work with visualization experts. In the simulation case, for instance, missing the simulation expertise would very probably have made us fail like the Dongarra team, as detailed in Section 3.5.1.

In each case, the corresponding experts have also been gladly interested in the use cases provided by the task-based runtime, which provides them with as many test cases as applications ported to the runtime. In many cases, future work will thus actually be pursued by the experts themselves, using the runtime as a testbed. Conversely, applications will directly benefit from the resulting improvements, and when programmers make their applications available for testing by all these experts (e.g. through platforms such as the task graph market which was explained in Section 3.1.8), they actually ensure that their particular application cases will be considered and optimized.

Integrating all these contributions altogether in the same runtime is however very challenging, we will discuss the implementation part in Chapter 5.

The concept of bubble, i.e. keeping track of the structure of task graph, will probably be fruitful in many aspects of a task-based runtime. A few examples have been discussed in Section 3.4.2, but a lot more uses are probably yet to devise, for instance in execution visualization.

This chapter was however only about execution on a shared-memory system, in the next Chapter we discuss extending task-based programming to executing over the network.

Chapter 4

Porting task graph execution to distributed execution

Contents

In the HPC field, the Message Passing Interface (MPI) is very widely used to address distributed systems, and allows to operate very large platforms. The adoption of multicore systems has additionally raised the question of intra-node parallelism. A simple approach is to run several MPI nodes on the different cores of the shared-memory machines, and MPI implementations indeed implement shared-memory message passing optimizations. To make better use of the availability of shared memory, it is also common to combine MPI with the use of threads or OpenMP, or even tasks. Combining MPI with MPI itself has even been proposed [HDB+13]. Leveraging GPUs can also be achieved by combining MPI with CUDA, or by using a CUDA version of MPI [Law09].

The purpose of this chapter is to discuss how distributed systems can be driven with only task-based programming, thus making task-based application seamlessly support distributed systems. We will use MPI as an example of underlying communication interface, but the principles exposed here would apply to other communication interfaces as well. We first discuss a master-slave design, then a fully distributed design. We then explain why caching data is needed and how to manage it. We will show the shortcomings of the classical MPI implementations for such workloads and how escaping the MPI interface allows to achieve a better interaction between the task runtime and the network communication runtime. We will then discuss problems which remain so as to scale the proposed model at large: task submission can become a bottleneck, load balancing should avoid global barriers, collective operations need to be optimized, and node failures have to be dealt with.

4.1 Can we make master/slave designs scale?

A simple approach for driving distributed systems is to use a master/slave paradigm. In essence, this is just an additional execution level, which can cope with task scheduling and data transfers just like when driving GPUs with discrete memory. This means that a task-based application does not need to be modified to be executed with such paradigm. The master/slave approach however poses scalability concerns: if the master has to decide all of the scheduling of each and every task on the platform, it will be quickly overwhelmed as the number of slaves increases.

OmpSs and ClusterSs use this paradigm (resp. [BPD+12] and [TFG+12]). To mitigate the scalability concerns, the task graph is expressed hierarchically by the application. The master thus only has to distribute the tasks and data at the highest levels of the hierarchy. Unrolling the corresponding subtasks can be done locally on slaves without interfering with other nodes. Results have for instance been shown [TFG+12] for 64 nodes of 4 cores each.

We implemented the StarPU support for the master/slave paradigm by just reusing the existing Xeon Phi accelerator and SCC support and adding the MPI set of communication methods. The MPI situation is indeed very similar to these architectures which can simply run the same program as on the master, and just need to exchange control messages to be told which tasks should be executed. This support however does not include hierarchical task graph unrolling yet, so only graphs composed of large tasks will scale. It will however probably be very natural to use the notion of bubble, which was described in Section 3.4.2, to express a hierarchy of tasks which can be unrolled locally.

A remaining issue with such hierarchical unrolling is that data exchanges are then typically achieved at the network levels of the hierarchy. It can be argued that this allows for bigger chunks of data to be transferred, but it will also reduce reactivity, since all related tasks will have to be finished before being able to send their contribution over the network, thus reducing pipelining potential. This is actually similar to the fork-join dependency issue which was explained in Section 3.4.2, which was solved within a shared-memory node, but is much less obvious to solve with a distributed system. A possible solution would be to differentiate the hierarchical level used for distributing tasks from the hierarchical level used for exchanging data, but this severely twists the execution model.

Using only one master node is also questionable at very large scale: even if it only distributes high-level tasks to be subdivided locally, doing so at large scales will probably be too demanding for a single master node. Continuing with the same hierarchical principle, the master should rather delegate some of the work distribution to sub-masters, by unrolling even fewer levels of the task graph hierarchy itself and letting sub-masters unroll more levels. Such an approach will however aggravate the data exchange concern mentioned in the previous paragraph.

4.2 Extending the STF principle to distributed execution for scalability

Instead of introducing a master/slave layer which requires a recursive application description in order to scale, we preferred to try keeping up as much as possible with the STF programming model which was described in Section 2.1.3. This resulted in two approaches for supporting network communications in a task-based programming model. The first approach, described in the next sub-Section, remains largely inspired from the MPI programming model, by making explicit the message sends and receives. The second approach, described in the subsequent sub-Section, makes them automatically generated by the runtime, thus sticking exactly to the STF programming model.

4.2.1 Task programming model with explicit communications

Figure 4.1(a) shows an MPI source code for a simple ring test case: a piece of data is propagated from node to node, starting from node 0, and incremented at each step. Each node thus receives the data from the previous node, increments it, and sends it to the next node. This is naturally synchronous, since we need to wait for the completion of MPI_Recv before running the increment, and wait for the increment completion before calling MPI_Send.

The principle of the MPI+tasks programming paradigm is to replace the computation part with the execution of a task. This however requires synchronization between the task programming model and the MPI programming model. Figure 4.1(b) shows that such synchronization (shown in blue) is very verbose, but required anyway. We have to tell the runtime that the data will be written to by MPI_Recv (so that e.g. a cached copy of the old value in the discrete memory of a GPU is discarded), and when the write is complete. We have to wait for the completion of the task before sending the result, but also to tell the runtime that MPI will read the data from the main memory (the computation might have happened on a GPU with discrete memory, in which case a transfer to the main memory is needed), and when it is finished with reading (so that e.g. other tasks can overwrite the data).

It is much more convenient to hide such synchronization in helpers provided by the task-based runtime. Figure 4.1(c) shows the resulting source code, actually very similar to the original pure MPI version of Figure 4.1(a): the computation has been replaced by a task submission, and the MPI operations have been replaced by communication submission. All synchronization questions of reception are hidden inside starpu_mpi_irecv_submit which calls starpu_data_acquire_cb, in order to call MPI_Irecv in a callback. This means that it is actually asynchronous, and starpu_task_insert can be called immediately after that, and proper dependencies will be automatically introduced between the completion of the MPI reception and the start of the task. Similarly, starpu_mpi_isend_submit is asynchronous and will be properly synchronized.

We end up with a completely asynchronous expression of the application [11]: the whole task graph is submitted to the runtime, including MPI receptions and emissions, and the runtime will handle all required synchronizations. The runtime can thus easily pipeline execution at will, performing MPI transfers as soon as possible when computations are completed, without being hindered by a sequential expression of the application. Turning the original MPI code from Figure 4.1(a) to Figure 4.1(c) also happens to be quite seamless by just requiring to replace MPI calls by their runtime-provided equivalents, it is mostly the logical extension of *taskifying* the application to make it completely asynchronous.

4.2.2 Task programming model with implicit communications

Using explicit MPI communications, as described in the previous sub-Section, makes a lot of sense when migrating an existing MPI application to task-based programming. When starting from a task-based application, having to determine the required MPI communications however departs from the STF programming model, it is more convenient to see them automatically inferred.

Figure 4.2 shows how this can be expressed by the application. starpu_mpi_data_set_rank needs to be called first, to tell the runtime which node the initial value for the data is on. starpu_mpi_task_insert can then be used instead of starpu_task_insert. The only difference is that the _mpi variant will automatically generate the required MPI transfers. It can do so since all nodes know exactly where the data is at each step of the submission loop. This however requires to make all nodes submit the whole task graph, thus the extension of the "for" loop, so that all nodes are made aware of all the progression of the data among nodes. Each node determines for each call whether it has to actually execute the task (indicated by STARPU_ON_NODE), or only generate the required data transfers if needed. Unrolling the whole task graph on all nodes can become

```
for (loop = 0; loop < NLOOPS; loop++) do
  if ( ! (loop == 0 && rank == 0)) then
    MPI_Recv (&data, prev_rank, ...);
  end if

  increment (&data);

  if ( ! (loop == NLOOPS-1 && rank == size-1)) then
    MPI_Send (&data, next_rank, ...);
  end if
end for
```

(a) Pure MPI programming model, synchronous.

```
for (loop = 0; loop < NLOOPS; loop++) do
  if ( ! (loop == 0 && rank == 0)) then
    starpu_data_acquire (data_handle, STARPU_W);
    MPI_Recv (&data, prev_rank, ...);
    starpu_data_release (data_handle);
  end if

  starpu_task_insert (&increment_cl, STARPU_RW, &data_handle, 0);
  starpu_task_wait_for_all ();

  if ( ! (loop == NLOOPS-1 && rank == size-1)) then
    starpu_data_acquire (data_handle, STARPU_R);
    MPI_Send (&data, next_rank, ...);
    starpu_data_release (data_handle);
  end if
end for
```

(b) MPI+tasks programming model, requires synchronization between MPI and tasks.

```
for (loop = 0; loop < NLOOPS; loop++) do
  if ( ! (loop == 0 && rank == 0)) then
    starpu_mpi_irecv_submit (data_handle, prev_rank, ...);
  end if

  starpu_task_insert (&increment_cl, STARPU_RW, &data_handle, 0);

  if ( ! (loop == NLOOPS-1 && rank == size-1)) then
    starpu_mpi_isend_submit (data_handle, next_rank, ...);
  end if
end for
starpu_task_wait_for_all ();
```

(c) Pure tasks programming model, completely asynchronous.

Figure 4.1: Various expressions of a ring test case. Each node receives a token from the previous node, increments it, and sends it to the next node.

```
starpu_mpi_data_set_rank (&data_handle, 0);
starpu_mpi_data_set_tag (&data_handle, 42);
for (loop = 0; loop < N * NLOOPS; loop++) do
  starpu_mpi_task_insert (&increment_cl, STARPU_RW, &data_handle,
                          STARPU_ON_NODE, loop % N, 0);
end for
starpu_task_wait_for_all ();
```

Figure 4.2: Ring test case with implicit MPI transfers.

```
ASSERT(P * Q == N);
for (m = 0; m < MT; m++) do
  for (n = 0; n < NT; n++) do
    starpu_data_set_coordinates (A[m][n], 2, m, n);
    starpu_mpi_data_set_rank (A[m][n], (m % P) * Q + (n % Q));
    starpu_mpi_data_set_tag (A[m][n], m * NT + n);
  end for
end for

for (k = 0; k < MT; k++) do
  starpu_mpi_task_insert (POTRF, STARPU_RW, A[k][k], 0);

  for (m = k+1; m < MT; m++) do
    starpu_mpi_task_insert (TRSM, STARPU_R, A[k][k], STARPU_RW, A[m][k], 0);
  end for

  for (n = k+1; n < NT; n++) do
    starpu_mpi_task_insert (SYRK, STARPU_R, A[n][k], STARPU_RW, A[n][n], 0);

    for (m = n+1; m < MT; m++) do
      starpu_mpi_task_insert (GEMM, STARPU_R, A[m][k], STARPU_R, A[n][k],
                                    STARPU_RW, A[m][n], 0);
    end for
  end for
end for
starpu_task_wait_for_all();
```

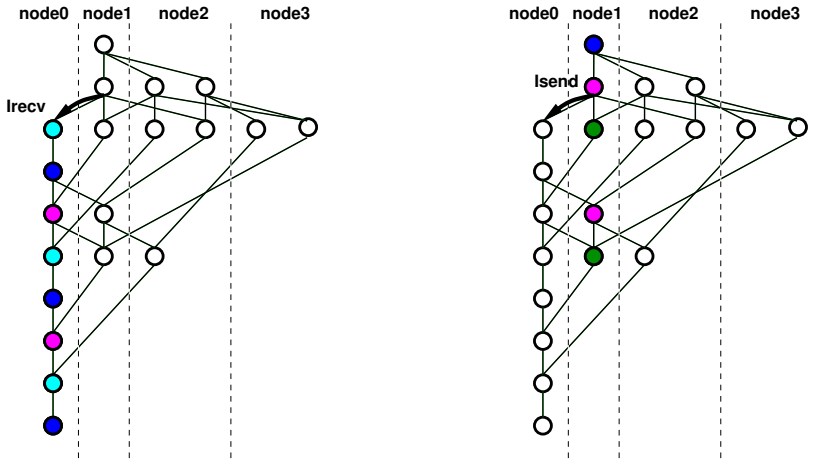
Figure 4.3: Distributed version of the task-based Cholesky factorization.

prohibitive, we will discuss this issue in Section 4.5.

An important detail is that a tag also has to be set to the same value on all nodes for the data. This is needed when several pieces of data are involved, so that the generated sends and receives get to match accordingly. Deciding a different but deterministic tag value for each piece of data is often relatively simple (by e.g. using the matrix tile coordinates), but that might become a concern. Adding a verification mechanism could be useful to make sure that tags are really set coherently among all nodes, possibly with the help of the compiler to identify the data.

In Figure 4.2, we have used STARPU_ON_NODE to make the computation performed on different nodes. This is generally not actually needed. For dense linear algebra for instance, the 2D block-cyclic tile distribution [Sus97] is well-known for providing optimized distributed execution, by limiting the amount of data transfers while keeping appropriate pipelining [DW95]. The distributed version of the Cholesky factorization can then be implemented as shown on Figure 4.3 [41, 38]. The preamble makes sure that the P and Q parameters of the 2D block-cyclic distribution correspond to the number of MPI nodes N. For each tile, it then sets the tile coordinates (useful for high-level feedback as was discussed in Section 3.7.2), the 2D block-cyclic MPI rank, and the tag to be used for data transfers. The loop nest can then remain exactly like the pure STF version shown on Figure 2.2 page 6. The runtime will determine coherently on each node where tasks should be executed thanks to a deterministic policy, such as "a task shall be executed on the node which contains the biggest data that the task writes to". The result of the policy is straightforward here, and exactly follows the task distribution used by the former state-of-the-art ScaLAPACK [CDD+96].

The important point here is that we have kept the task loop nest free of any real notion of distributed execution, to only concentrate on the task graph algorithm with a sequential-looking source code. The preamble provides the data mapping over the network, which automatically



(a) Task graph unrolling on node 0

(b) Task graph unrolling on node 1

Figure 4.4: Example of the Cholesky task graph distributed on four nodes, here nodes 0 and 1 determine the tasks they should execute, and the MPI requests they should generate.

entails the task distribution through an execution policy which can also be customized in the preamble. If the chosen data mapping brings poor performance, it can be easily tuned at will without any fear of breaking the semantic of the application. Dynamic data remapping may be needed during execution, this will be discussed in Section 4.6.

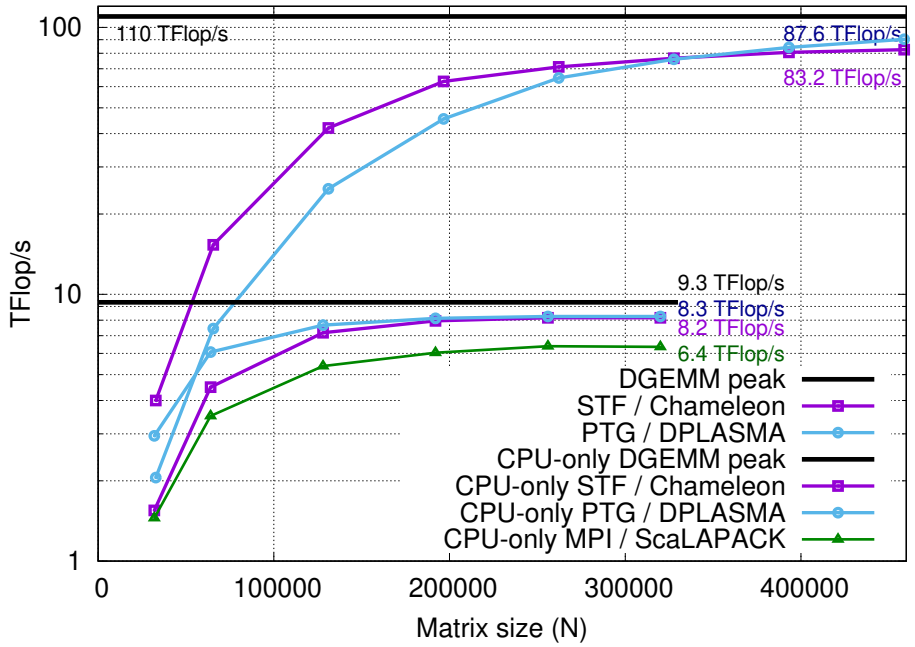
The end result is as shown on Figure 4.4: while each node unrolls the whole task graph, each node creates only the tasks it will have to execute, and submits the required MPI_Irecv and MPI_Isend requests, which by construction match with each other.

Both the Quark-D [Don13] and DuctTeip [ZLT16] projects have also followed this approach, driven by the same goal of a scalable solution. DPLASMA [BBD+11] benefits from the parameterized PTG expression of the application to achieve independent unrolling of the task graph, the difference is that it can easily make each node unroll only its required part of the task graph, this will be discussed in more details in Section 4.5.

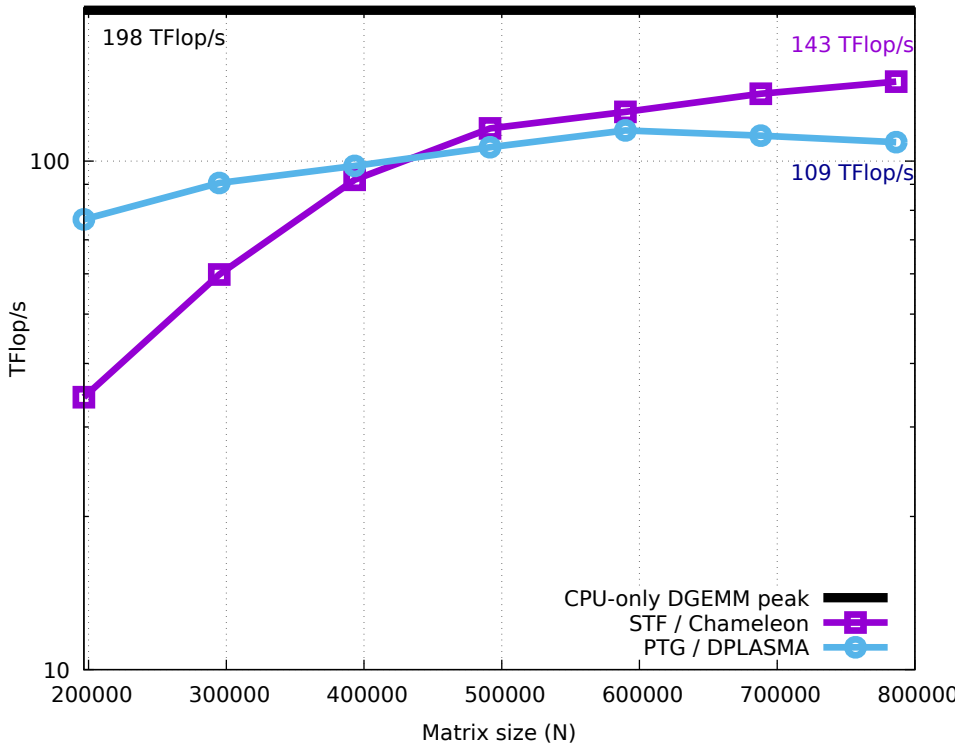
This approach does achieve interesting results on quite large clusters. In the context of Marc Sergent's PhD thesis [46] (which I co-advised) we had the opportunity to experiment over more than 6000 cores [2], and it showed performance comparable to the hand-tuned industrial code of CEA. In some cases performance was even greater, and we determined that this was because the hand-tuned pipeline of the CEA solver was not aggressive enough (2 layers of panels), whereas execution as described above does pipeline as much as theoretically possible. CEA spent some engineer time to improve their pipeline to 3 layers of panels, and catch up with the performance obtained by the StarPU runtime.

Figure 4.5 shows the performance of the Cholesky factorization of the Chameleon linear algebra library on top of StarPU on the CEA platforms, compared with the performance obtained with DPLASMA. The bottom of Figure 4.5(a) shows that on CPUs only, the approach performs much better than the former state-of-the-art ScaLAPACK. The top of the Figure and Figure 4.5(b) show that even with GPUs, the Chameleon and DPLASMA performance are comparable, it is not obvious to claim one is really performing better. At this scale of execution, we would need advanced performance analysis tools such as was described in Section 3.7.2.2 to determine why one is performing better than the other in the various cases shown here.

In the following Sections, we will discuss various important aspects of such a runtime-driven distributed execution.



(a) Results on 144 nodes of TERA-100 cluster (1152 CPU cores and 288 GPUs). The Y-axis uses a logarithmic scale.



(b) Results on 256 nodes of Occigen cluster (6144 CPUs). The Y-axis uses a logarithmic scale.

Figure 4.5: Performance of a distributed Cholesky factorization.

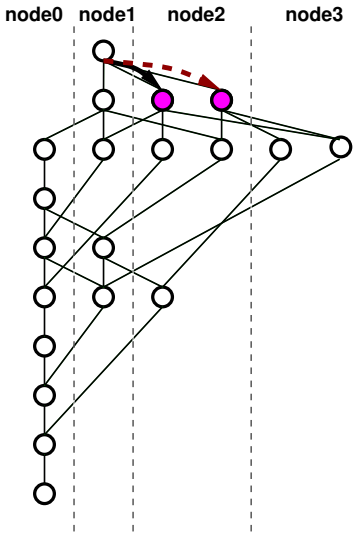


Figure 4.6: Example of duplicate communication.

4.3 Caching remote data at the expense of memory usage

A very common case, that all distributed runtimes have to optimize, is when a node needs the same data from a remote node for multiple tasks. Figure 4.6 shows an example, which in the Cholesky case takes place for almost all POTRF tasks, whose result is used by a series of TRSM tasks. In such a case the runtime should avoid transferring the data as many times.

This means *caching* the received data, and the immediate question is when to *flush* the cached data, since keeping it in memory exacerbates the memory use concerns which were detailed in Section 3.2. Once the whole task graph is submitted to the runtime, one can assume that after the task graph execution, cached data will not be used, and thus it can be flushed right after completing the last task of the graph using it.

As was discussed in Section 3.2.2, one of the ways to cope with the limited amount of available memory is however to throttle task submission, and thus the task graph is known only progressively during execution. The runtime hence can not know during execution whether some tasks using the cached data will be submitted later on. More generally, with dynamic task submission, the runtime can not know whether cached data will be re-used later on. In such a case, the application can specify by hand when a piece of data will not be re-used by the rest of the task graph, Figure 4.7 shows an example on the Cholesky factorization. This allows, even when throttling task submission, to release data received from other MPI nodes as early as possible. Such locations are actually also good candidates for *wontuse* hints to optimize data eviction from GPUs, as was described in Section 3.2.2. Such hints do not change the semantic of the computation, at worse they will entail spurious re-transmissions. They could probably be introduced automatically at compile-time thanks to a loop nest analysis, which can easily determine which tiles will never be accessed again in further interations of the loops.

4.4 Classical MPI implementations do not cope well with task-based workload

In this Section, we discuss the issues we have had in StarPU-MPI with various MPI implementations, which happen to have been met by other task-based runtime systems using MPI to dis-

```
for (k = 0; k < MT; k++) do
  starpu_mpi_task_insert (POTRF, STARPU_RW, A[k][k], 0);

  for (m = k+1; m < MT; m++) do
    starpu_mpi_task_insert (TRSM, STARPU_R, A[k][k], STARPU_RW, A[m][k], 0);
  end for

  starpu_mpi_cache_flush (A[k][k]);

  for (n = k+1; n < NT; n++) do
    starpu_mpi_task_insert (SYRK, STARPU_R, A[n][k], STARPU_RW, A[n][n], 0);

    for (m = n+1; m < MT; m++) do
      starpu_mpi_task_insert (GEMM, STARPU_R, A[m][k], STARPU_R, A[n][k],
                                    STARPU_RW, A[m][n], 0);
    end for

    starpu_mpi_cache_flush (A[n][k]);
  end for
end for
starpu_task_wait_for_all();
```

Figure 4.7: Adding explicit cache flushes to perform well with memory limitations.

tribute tasks.

A first very widely known issue is that most MPI implementations are not really thread-safe, and thus to avoid obscure bugs it remains a must to keep all MPI operations in a dedicated *communication thread*.

A second issue is the scalability of the receive queue. In the initial version of StarPU-MPI, shipped in StarPU 1.1, we were passing the data tags (which were described in Section 4.2.2) to the MPI implementation, to let it achieve the tag matching. This means that in the Cholesky case for instance, StarPU-MPI would, right from the start of the application, post all MPI receptions for all data contributions that will be received during the execution. This makes sense since the runtime does not necessarily know that the extreme bottom-right tiles will not be computed before long. This however also means that the MPI layer is overflown with e.g. thousands of MPI receptions on separate tags, which is often not well supported by MPI implementations. The issue is typically that MPI implementations delegate reception to the hardware, which have limited resources (e.g. around a hundred requests), and thus can not cope with too many requests at the same time. When the limitation is overcome, a fallback non-optimized implementation is used, and entails poor performance.

To avoid such shortcoming from the MPI implementation, in StarPU 1.2 we have implemented the tag matching in StarPU-MPI itself. It means that all communications are composed of an envelope followed by the corresponding data. The only pending MPI receptions are thus the reception of the next envelope, and the data receptions for which the envelope was already received (and thus for which data will probably land very soon). This approach allowed us to get the results which were shown in Section 4.2.2.

This can be seen as a sign that MPI implementations do not really properly handle task-based workloads, which use extremely aggressive task graph pipelining, and thus very irregular and long-pending data requests. It happens that Alexandre Denis' work on communication libraries [DT16], conducted in the NewMadeleine library, notably aims at optimizing irregular communication patterns. Such optimizations are not often seen in MPI implementations, because applications do not (yet) tend to use such patterns, but the latter may be precisely just because MPI implementations do not support them very well... In StarPU 1.3, we have respawned

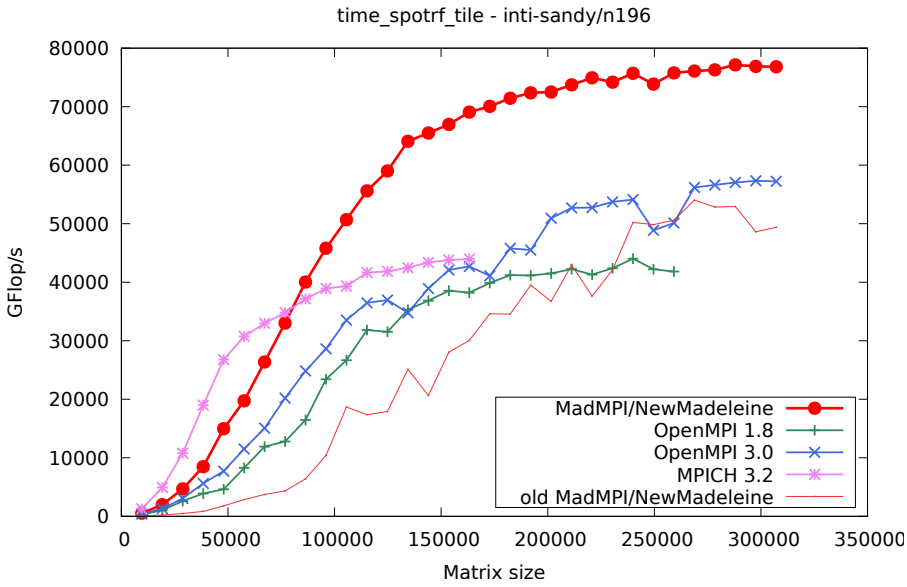


Figure 4.8: Performance of the StarPU-based Cholesky factorization with different MPI implementations.

the StarPU 1.1 implementation, used only when running with the madmpi MPI interface of the NewMadeleine library, to let it handle all the tag matching. We have also even disabled the communication thread in that case since NewMadeleine really is thread-safe. This means dropping the useless inter-thread synchronization when data has been computed and is thus available for sending over the network: the thread which terminated the computation submits the the transfer request itself immediately. We also replaced some of the MPI calls with native Madeleine calls (the nmad interface), in preparation of using extensions of Madeleine to bring interactions between the task runtime and the communication runtime beyond the MPI interface.

The results are compelling. Figure 4.8 shows the result of an execution of the Cholesky factorization on 196 nodes connected through InfiniBand, and using NewMadeleine indeed provides much better performance. The exact reasons are yet to be investigated, but it is probably overall due to the reactivity of NewMadeleine thanks to the runtime just telling it exactly which communications are to happen. This application case, and more generally task-based applications, are actually a very interesting test case for Alexandre Denis' researches on behaviors of MPI implementations with irregular workloads.

4.5 Making fully-distributed STF submission scale to exa?

As was detailed in Section 4.2, a pure STF programming model would normally mean that all nodes unroll the whole task graph, to determine coherently which node needs to execute which task and issue which transfers.

Unrolling the whole task graph can however become questionable as the number of nodes increases. Figure 4.9 shows in red (non-pruned) how the task submission time can get closer to the overall execution time with an increasing number of nodes, to the point that execution gets delayed by the mere lack of tasks being submitted.

The overhead at stake here is actually the mere cost of passing task parameters to starpu_mpi_task_insert and that function parsing them. In Chameleon, we have mitigated the issue by in-

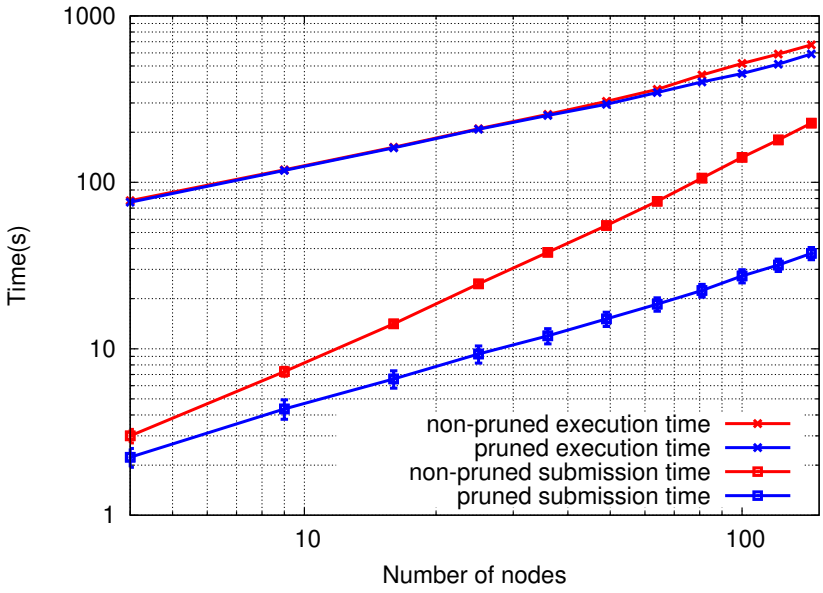


Figure 4.9: Submission time vs execution time in the Chameleon distributed Cholesky implementation, without (red) and with (blue) submission pruning.

troducing *task graph pruning*: if none of the data used by the task belongs to the current node, starpu_mpi_task_insert is not even called, thus saving the corresponding overhead. This is hidden inside helper functions so that the main application source code does not exhibit this optimization. The result is shown in blue (pruned) on Figure 4.9. The submission time progresses way more slowly, it still progresses faster than execution time, and extrapolating the curves roughly seems to indicate that overhead issues may show up again around the million-node order of magnitude.

At this point, the remaining overhead comes merely from the $O(n^3)$ loop nest itself. Making the helper functions *inlined* allows the test (for submitting the task or not) to be factorized by the compiler in the loop nest itself, which buys a few more orders of magnitude for scalability. Going further down, it makes sense to turn the P and Q parameters of the 2D block-cyclic distribution constant, i.e. to build an optimized binary dedicated to a given number of nodes. The compiler can then optimize this kind of loop resulting from the inlined test:

```
for (i = 0; i < ntiles; i++)
  if (i % Q == myrank) {
    task_submit(...);
  }
```

into

```
for (i = myrank; i < ntiles; i+= Q)
  task_submit(...);
```

which reduces the loop nest complexity from $O(n^3)$ to $O(n^3/Q)$, and thus buys a few other orders of magnitude for scalability. Unfortunately, usual compilers such as gcc or icc do not perform this kind of optimization yet, but a source-to-source compiler such as ppcg using libisl does make the optimization with polyhedral analysis. Figure 4.10 shows an projection of the submission cost. We here only ran the task submission loop, while assuming that each node provides

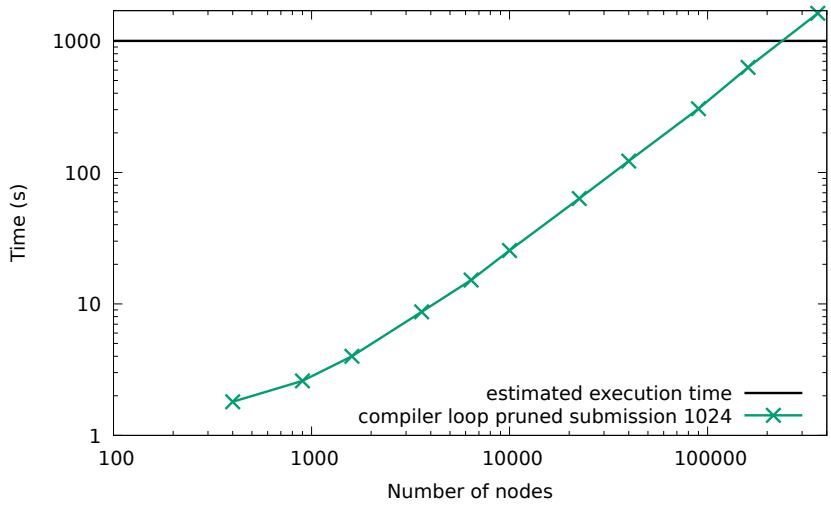


Figure 4.10: Large-scale projection of the task graph unrolling cost.

500 GFlops with a 1024 tile size, and have tuned execution parameters to weak scaling values giving 500,000 GFlop to compute to each node, thus an execution time fixed to 1000 s. We have measured the corresponding task submission time with all optimizations mentioned above enabled. The curves cross around 250,000 nodes, i.e. submission time reaches execution time at 0.125 EFlops. With faster nodes and bigger tile sizes, ExaFlops is not so far.

Of course, such projected figures are far-fetched, and keeping clenched on the STF programming model may not be reasonable any more at such a scale, when DPLASMA [BBD+11] can easily avoid such task submission cost issue altogether, thanks to the parameterized PTG representation of the task graph which allows to exactly unroll what is needed.

The DuctTeip [ZLT16] project took another approach to avoid the issue. As was described in Section 3.4.2, task graphs can be expressed in a recursive way. Application execution with DuctTeip unrolls the first levels of recursion for the whole task graph on all nodes, just like we described for STF in Section 4.2.2. But similarly to the master/slave approach detailed in Section 4.1, thanks to the recursive description of the task graph, further levels of recursion can be unrolled only on nodes which actually have to execute the corresponding tasks, and are handled locally only by the Superglue [Til15] shared-memory runtime. It is indeed useless for all nodes to unroll all the task details of all other nodes, they only need to fully know the highest levels of recursion to generate the required network transfers. This saves a lot of task submission overhead, and as was mentioned in Section 3.4.2, recursive submission can be achieved in parallel, thus even reducing the impact of the remaining submission overhead. A similar approach will probably be possible in StarPU itself thanks to the notion of bubbles. The TaskUniVerse [Zaf18] project proposes Unified Task-based Programming (UTP) which generalizes the DuctTeip recursive approach, and can drive several backends, including StarPU, and does improve scalability.

Such an approach however meets the same issue as was mentioned for the master/slave paradigm in Section 4.1 : communications between nodes will be expressed only at the network levels of recursion, introducing fork-join synchronization points which may hinder parallelism. Global unrolling thus has to be conducted deep enough to avoid such large synchronizations. Whether at very large scale we can find a balance between globally unrolling enough for parallelism, but not too much to avoid the overhead, remains to be investigated.

4.6 Redistributing data dynamically while keeping the STF principle?

In the previous Sections we have mostly assumed that the data mapping does not evolve over time, and thus task mapping is fixed. For dense linear algebra, the static 2D block-cyclic distribution [Sus97] is indeed a good rule of thumb [DW95]: contributions between tiles naturally flow along columns and rows, and this distribution tends to minimize the number of nodes involved in both cases.

For less regular cases, such as sparse or compressed linear algebra, or stencil applications using Adaptative Mesh Refinement (AMR [BC89]), the computation load can evolve during execution, and in an unpredictable way. Hardware faults can also entail the loss of a node, as will be detailed in Section 4.8, thus requiring a redistribution of the load.

In some cases, the application itself may determine how to rebalance the load. In the STF programming model, this can be achieved by just submitting to the runtime some *data migrations* within the submission loops. Provided that all nodes achieve the same data migration at the same point within the submission loop, the MPI transfers generated after the migration will be globally coherent too. The interesting point is that since this is triggered during task submission, it does not introduce any global synchronization in the execution itself. It will only generate a few additional data transfers (for the actual data migration), which can be overlapped with the execution of unrelated tasks which were submitted before or after the migration submission. We have made initial experimentations on synthetic stencil-based cases, and we plan to experiment this with AirBus on their FLUSEPA Navier-Stokes Solver [CCRB17]. This being said, it would however be interesting to relieve the application from having to determine which data migration should be achieved to balance the load.

The DuctTeip project uses a completely distributed approach to achieve dynamic load balancing [ZL18]. The principle is to avoid any kind of global exchange of information which could limit scalability. Pairs of busy and idle nodes are discovered opportunistically and exchange pieces of the task graph, to try to compensate for the idleness. Data itself is not really migrated, the tasks are just executed remotely and the result sent back to the node which was supposed to execute the tasks. This is beneficial if the computation is dense enough compared to the back and forth communication requirement. Such a completely local approach is seducing for scalability, we could think of really migrating the data, and thus all the following tasks, to more permanently compensate for the load imbalance. This would however ignore locality, and may entail more and more required MPI communications as pieces of the task graph get redistributed without a global vision.

It happens that taking a global view of the workload (possibly hierarchical to make it scale better) might actually not pose performance issues. Global synchronization could indeed be achieved at submission time, which can be far ahead of execution time. Provided that submission is not too costly (as discussed in the previous Section), there would be a large time interval between submission and execution. As shown on Figure 4.11, this means that we may afford stalling submission while a global data redistribution is computed according to the global load imbalance knowledge (e.g. with a graph repartitioner such as Scotch [PR96]). In the meanwhile, the execution of tasks submitted with the previous data distribution could continue. Submission can then resume with the new distribution, and the corresponding tasks can even intermix with tasks submitted with the previous distribution. This would allow to really take a global view into account when optimizing for locality. Experimenting with this approach will be part of the PhD thesis of Romain Lion (which I co-advise) which has just started in the context of the EXA2PRO H2020 FET-HPC European project.

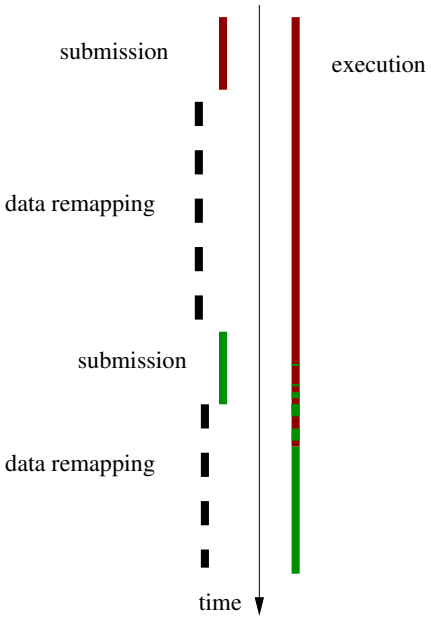
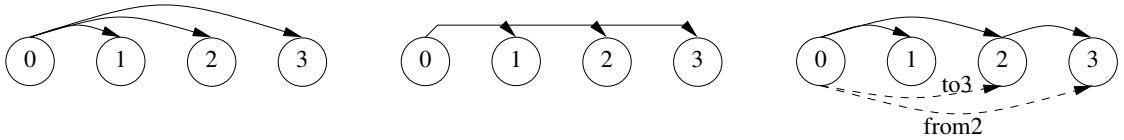


Figure 4.11: Dynamic data remapping thanks to submission/execution decoupling. Red tasks, submitted first, can execute while a data remapping is computed, and green tasks submitted with the new data mapping can be executed along red tasks, while yet another data remapping is computed.



(a) Without optimizations.

(b) MPI-provided optimization.

(c) Opportunistic optimization driven by control messages.

Figure 4.12: Different strategies to achieve collective operations.

4.7 Optimizing collective operations opportunistically: cooperative sends

A very common need is to broadcast the result of a computation to many different MPI nodes. In the Cholesky case for instance, the results of POTRF tasks have to be broadcast to all MPI nodes which will perform the corresponding TRSM tasks. Making the source of the data have to transmit it to all destinations one by one as shown on Figure 4.12(a) would make poor use of the network.

Communication interfaces like MPI provide optimized implementations of such collective operation, which can be represented as shown on Figure 4.12(b). One way to integrate this in the STF programming model would be to introduce explicit collective operations within the application algorithm. In the Cholesky case, it would mean submitting a broadcast request between submitting the POTRF task and submitting the TRSM tasks. Designating the exact set of nodes to be involved in the communication is however tricky. In the case of dense linear algebra, rows and columns of the 2D block-cyclic distribution easily come as candidates to be declared as MPI communicators which can then be used to broadcast data. This is however a very restrictive case, and having to explicit the collective operation is questionable. When dynamic load balancing

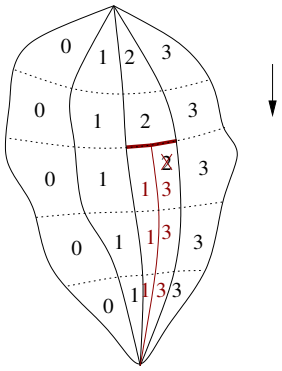


Figure 4.13: Asynchronous checkpoint and restart with the task-based programming model.

approaches described in the previous Section are used, this becomes even more tricky.

Another way would be to just automatically discover that a broadcast scheme is taking place. This means aggregating the separate communication requests automatically generated by starpu_mpi_insert_task. Once more, the runtime happens to have the time between submission and execution (more precisely between the generation of the communication request and the availability of the data to be transmitted) to collect these requests, i.e. to make a list of all the destinations for the same data. An optimized communication tree could then be established and used for broadcasting the data efficiently, i.e. an opportunistic *cooperative send*. To reduce the part of task graph that nodes will have to unroll, the communication tree would be only determined on the source node of the data. That node would then send control messages (shown in dashed lines on Figure 4.12(c)) to the nodes involved in the tree, to tell them their role in the tree. Such control messages can be sent very early, between submission time and execution time, so that all nodes are ready with the optimized communication tree even before the data to be broadcast is computed. This can for instance be triggered by the same *wontuse* hints as was described in Section 3.2.2 for memory management and was mentioned in Section 4.3 for cache flush, set by the application itself, or by a compiler with polyhedral analysis.

It would then become simple to tune the communication tree: binomial trees can be a good default, but in some cases it could be essential that the first nodes (for which the data was requested in task submission order) get the value earlier, to improve computation pipelining; this could be configured by selecting a different policy. We are working on this with Alexandre Denis as well, to possibly even delegate this to the communication library, which has much better knowledge on the network topology and network link utilization.

4.8 Leveraging task graphs for checkpointing and resiliency

As a last issue, but not the least, with scalability over large platforms come a high probability of hardware failures, which can entail the loss of whole computation nodes.

A common way to compensate such loss is to perform periodical checkpoints of the data of the application, to be able to restart it from the last complete checkpoint. This however usually means a global stop-and-restart of the whole application, which is more and more costly with larger platforms. Since with larger platforms the failures are also more and more frequent, this cost can not be ignored at some scales. Without using a programming layer such as task graphs, supporting partial restart requires deep application modifications.

The task-based programming model could provide distributed checkpoint and restart. The idea is sketched on Figure 4.13, where the execution of a task graph is distributed on 4 nodes. At

regular positions in the task submission loop, a checkpoint request would be made, shown in dotted lines. The precise positions could be set at appropriate places by the application or by a compiler (e.g. to minimize the amount of live data which has to be save), or it could just be set automatically by the runtime, the only condition is that the position needs to be coherent among all nodes, so as to get by construction a *consistent global state* [CL85]. When a node would fail, here node 2, neighbour nodes would restore the data from the checkpoint, and restart submitting the tasks that node 2 was supposed to execute starting from the checkpoint. In the meanwhile, node 0 would be completely unaffected. Generally speaking, with a lot more nodes, only nodes which exchange data with the failing node would need to be involved in the recovery. A dynamic load balancing strategy such as described in the previous Section would additionally help with redistributing the load of node 2 to node 0 in addition to nodes 1 and 3.

This sketch raises several questions. The pure STF programming model, where all nodes unroll the whole task graph, does mean that nodes 1 and 3 are theoretically able to unroll the part of the task graph that node 2 was supposed to achieve. "Restarting submitting tasks" would however mean branching in the middle of the application submission loop, at the position of the checkpoint. Help from the compiler could probably be used to achieve this while catching values of local variables etc. without revamping the application source code.

Node 2 may also have already sent some data to other nodes, the repetition of these emissions would have to be discarded. Conversely, data that node 2 would have already received would need to be re-emitted for the re-execution. Nodes 1 and 3 would also have already posted reception requests for further data coming from node 2, they would have to be redirected.

The management of all these issues would be quite costly. It would however take place independently from the rest of the execution, which can continue to the extent of data dependencies. We can thus hope that the only noticeable effect would be a sudden load imbalance, to be compensated dynamically. This whole recovery principle will also be part of the PhD thesis of Romain Lion.

Another way to cope with the loss of a computation node is to modify the application algorithm to replicate some computations so that several nodes end up containing a copy of the result. It would be useful to make the application explicitly tell the runtime which data, among several nodes, are to contain the same value thanks to this, so that the runtime would be able to automatically fetch the value from surviving nodes.

4.9 Discussion

Along this chapter, we have tried to keep the STF programming model as much as possible, while introducing strategies for caching data, redistributing the workload, and coping with hardware faults. The result is that for instance, the actual Cholesky implementation used by Chameleon on top of StarPU, shown in Figure A.1 of Appendix A, page 102, is very close to the implementation which was sketched in Section 2.1.3, page 6. The few additional lines only affect performance, not the computation semantic, so that programmers can feel safe with playing with them. This version does not support load balancing and checkpointing yet, but it did produce the competitive performance shown on Figure 4.5 page 77.

As was seen in Section 4.5, unrolling the whole task graph on all nodes becomes more and more questionable with an increasing number of nodes. Hierarchical approaches such as proposed by DuctTeip and bubbles will be required to possibly keep the optimizations described in this chapter, made possible by the STF programming model, while getting task submission to scale. Using a master/slave design as was described in Section 4.1 would probably also make supporting the optimizations described in this chapter quite simple. Whether that can scale by using a hierarchy of masters is yet to be investigated.

This chapter was discussing the use of MPI for network transfers. Beyond the mere issues encountered with MPI implementations which were described in Section 4.4, the choice of MPI may be questionable. As was mentioned in Section 4.7 in the case of collective operations, some optimizations may require the use of more advanced communication interfaces than just MPI, such as NewMadeleine. Using a PGAS communication library such as GASPI [SRG15] would also make a lot of sense: the runtime only wants to expose data from one node to another node. In practice, the programming interface provided by PGAS implementations is often not low-level enough, for instance they would lack fine-grain management of asynchronous coherency requests.

Caching data as was described in Section 4.3 is crucial for performance, but takes yet more room in the main memory. If available memory gets really tight, it would make sense to flush some of the cache unexpectedly. This would however require to notify each node which provided the data that they will have to send it again in the future, which twists the very deterministic scheme that the STF programming model settled. "In the future" is also a sign of another concern. Transmitting data between nodes as soon as possible allows to release tasks earlier, thus favoring parallelism. But it also means monopolizing the corresponding memory area until tasks actually get to run and complete. Using synchronous sends allows to make the sender wait for the receiver to consider that it can afford allocating the buffer for reception. The interaction with the data management that was detailed in Section 3.2.3 is however to be settled: how should a memory allocation manager decide whether to reserve memory for local tasks, or reserve memory for data coming from other nodes?

Chapter 5

Implementation notes for a task-based runtime system

Contents

Implementing a task-based runtime system is an arduous task, full of traps. In this chapter, we discuss a few noteworthy questions on the implementation itself.

We first explain how we tried to keep StarPU extensible, and how we interacted with users to achieve it. We then discuss how StarPU can manage different data layouts, and why it uses several layers of memory allocations. We will then describe how StarPU can more or less successfully leverage low-level software interfaces altogether. Eventually, we will discuss the interactions between conducting research on a runtime system, and working on its implementation.

5.1 Maintaining the potential for extensibility

While conducting our research on task-based runtime systems, we have tried to implement the research contributions in a common software implementation, StarPU. Managing to integrate the various extensions requires to carefully design the overall architecture, with appropriate modularity. This avoids integrating all features in a single ever-increasing core, even if it means some overhead, to improve maintainability.

For instance, as shown on Figure 5.1, the low-level module for managing dependencies supports explicit dependencies between tasks. Inferring the dependencies from the STF programming model is done in a separate "implicit dependencies" module which simply creates explicit dependencies between tasks, instead of introducing another type of dependencies. This actually made it easier to introduce commutative data access. Similarly, the bubbles module is mostly just an interface on top of the existing notions of dependencies and asynchronous data partitioning.

Data coherency operations, such as the synchronization needed for the application to be able to print output data from the main memory, is also implemented simply by submitting empty

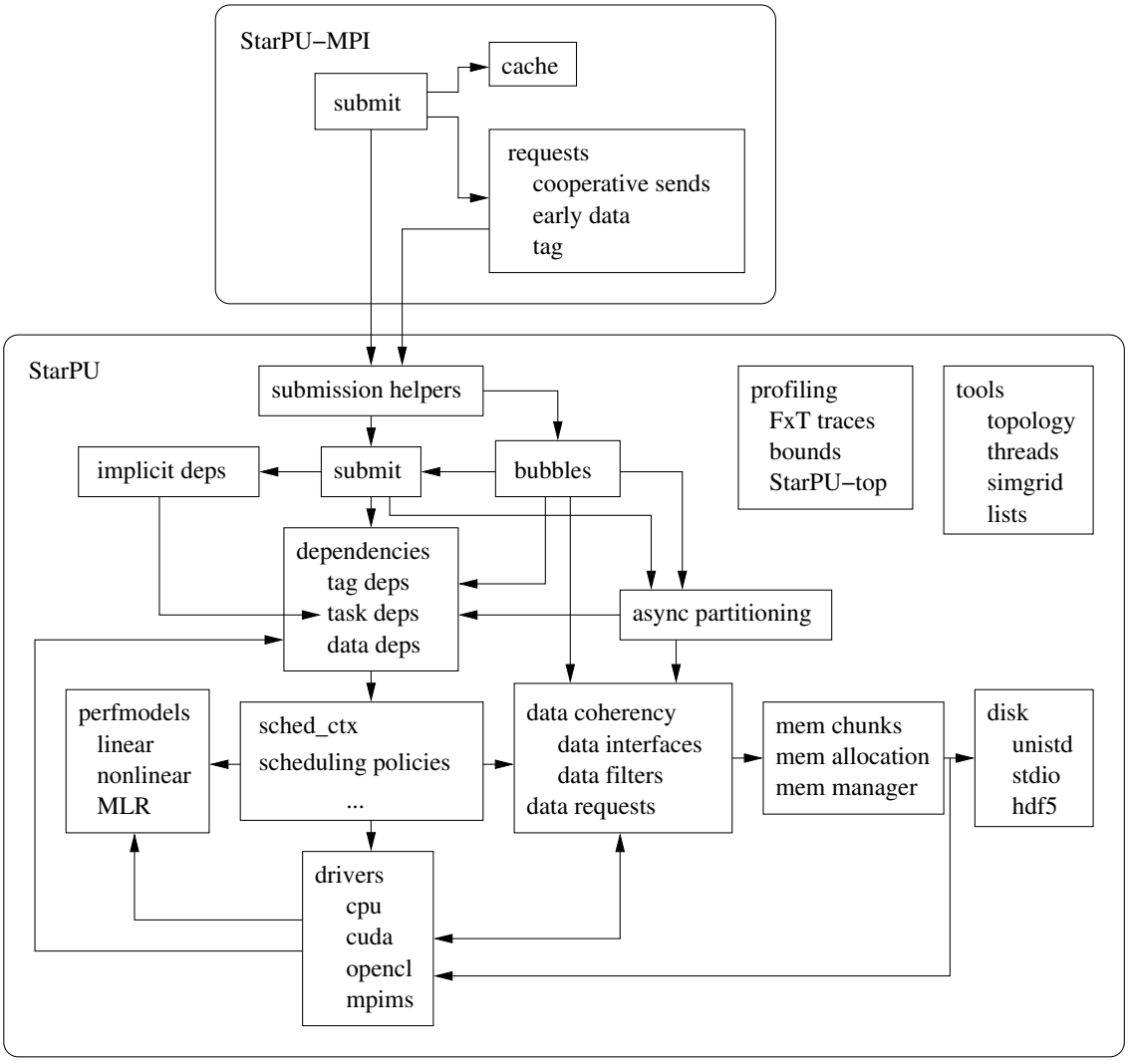


Figure 5.1: Overall structure of the StarPU runtime system.

tasks which access the data from the main memory, thus benefiting from all the common data dependency tracking and data transfer support, and showing up in the task graph.

Being able to access the same matrix with different tile sizes concurrently was often requested, but the initial implementation attempt, which was done within the data coherency module itself, was not reasonable. Later, we implemented it as a separate "asynchronous partitioning" module which just uses synchronization tasks and standard data coherency operations. This layering brings overhead, but makes it straightforward to maintain.

StarPU-MPI is a completely separate layer, which interacts with the StarPU core through standard interfaces. For instance, the interaction between MPI data requests and task execution in StarPU is achieved by using the standard data coherency requests provided by StarPU, and thus naturally show up in the task graph. This means that writing another distributed layer on top of another communication library would simply use the same principle, and not need any further development within StarPU itself.

Similarly, OpenMP and OpenCL interfaces have been implemented on top of StarPU as separate layers, KSTAR and SOCL, and only a few features have been added to StarPU to provide the required additional support, such as task continuation for OpenMP.

5.2 Interacting with users to invent the right features

What a task-based programming interface should look like is still debated. We should both try to anticipate the needs of application programmers, and try to generalize them. It thus means discussing with them to fully understand their application needs, but also *not* just implement the immediate solution to the needs. Ideally a programming interface would provide a wide range of features from a very small set of basic operations, and as explained in the previous Section, more advanced features be implemented as mere small layers on top of the basic operations.

This actually means perhaps refusing to write an unsound implementation, and wait for taking some step back from the immediate needs and let ideas settle down. This is for instance what happened for asynchronous partitioning mentioned in the previous Section, and eventually led to an elegant implementation. In the context of memory accounting that was mentioned in Section 3.2.3, we also actually let qr_mumps maintainers implement a proof of concept, only adding to StarPU the very basic tools to make it possible. We then considered how the concept could be generalized, and integrated it in StarPU as a separate module.

Getting to understand the programmer's needs is however delicate when the source code for the application is not available (e.g. for secrecy reasons), and only descriptions are available. In such a case, it is very useful for the programmer to provide at least a dumb equivalent of the application, to serve as a proxy application, which can even be integrated in the runtime testsuite, so that the programmer can be sure that her or his particular use case will remain supported.

5.3 Supporting different data layouts seamlessly

Section 2.7.1 explained that StarPU supports various data interfaces (vectors, dense matrices, sparse matrices, etc.) and data partitioning. Making this possible required a careful separation of implementation bits. Managing data indeed involves a series of loosely related notions:

- Describing the shape of the data itself, e.g. two buffers of contiguous data.
- Describing how data can be partitioned into subdata.
- Managing coherency between copies stored in different memory nodes.

- Managing data transfer requests.
- Accounting memory reservation.
- Actually allocating buffers and transferring data, as implemented e.g. by the CUDA driver.

When an application programmer uses her or his specially-crafted data structures, she or he only has to implement the first two points, which are respectively called *data interface* and *data filters* in StarPU. These are then called as needed by StarPU. For instance, when a task is to be run on a GPU, the data coherency manager first calls the application-provided `allocate` data interface method to allocate the target buffers for task parameters. This method merely calls starpu_malloc_on_node with an appropriate size, and that function actually allocates a buffer on the GPU and accounts for the memory reservation. The data coherency manager then creates requests to perform the data transfers. Handling such a request means calling the application-provided `any_to_any` data interface method to describe the data transfer. This method merely calls starpu_interface_copy with the appropriate pointers and sizes, and that function calls the actual GPU methods for starting the transfer. This notion of data interface thus decouples the layout of the data from all the rest of StarPU's data management, the data interface methods being just called at the exact time when customization is needed to describe the particular data structure of the application.

5.4 Why distinguishing several layers of memory allocation notions?

Figure 5.1 page 90 shows several layers of memory: chunks, allocation, and management. These indeed correspond to different notions of allocated data.

Memory management does not actually allocate data, but manages memory reservations, as used for preventing memory overflow, as was detailed in Section 3.2.3. It thus only accounts for memory that will potentially be really allocated, possibly using pessimistic approximations.

Memory allocation performs the real data allocation, using the proper underlying system-provided methods. For the main memory this could be a mere malloc, but when CUDA GPUs are available, this is replaced by cudaHostAlloc, which pins the allocated memory for CPU-GPU transfers to be asynchronous. For GPU memory, StarPU does not directly call cudaMalloc for small allocations, because that function is terribly costly. An intermediate suballocator is implemented, to reduce the number of times cudaMalloc is called.

Memory chunks manage data as seen by the application through the data interfaces described in the previous sub-Section. A memory chunk can contain several memory allocations if the application-defined data interface uses several buffers for a given piece of data (e.g. for sparse matrices). The chunks implementation also supports a cache of allocated chunks, to save calling the low-level interfaces when buffers of the same size can be reused, which is extremely common with dense linear algebra for instance.

We can notice that it is actually the notion of customizable data interface detailed in the previous sub-Section which makes the distinction between memory chunks and memory allocation necessary, since a chunk can be composed of several allocations. Memory management is on a completely different level, aiming for long-term reservations accounted at submission time, and providing an upper bound of the actual memory allocations which will happen at execution time.

5.5 Leveraging underlying low-level interfaces altogether, and influencing them

On heterogeneous platforms, a runtime system has to handle interactions between various system-provided layers that perform computations and data transfers. Testing efficiently for the termination of requests queued to different low-level layers is however a concern. Mainstream progression managers such as implemented in glib are often based on the availability of POSIX file descriptors which can be passed to a single `select` call. HPC layers however typically do not provide such a standard interface for synchronization, and rather provide interfaces whose flexibility vary a lot, some of them are actually even problematic to use in a runtime system.

The HDF5 library for instance, which StarPU can use to store data arbitrarily in a single file on the disk, does not provide an asynchronous interface yet. We thus had to use a separate thread which keeps getting blocked while waiting for the completion of requests. The standard AIO interface as provided by GNU/Linux is actually also implemented on top of the synchronous system calls by the GNU/Linux system itself by creating such a thread; only the Linux-specific non-standard AIO interface provides a really asynchronous interface to the disk.

The CUDA interface was initially rather high-level and its apparent asynchronous support was actually ridden with implicit synchronizations. This was fixed by the introduction of CUDA streams, which allow to monitor various computations and data transfers separately. The remaining issue is that we have to teach programmers to use them and avoid cudaMalloc/cudaFree when writing implementations of tasks, otherwise the implicit synchronizations from CUDA strongly hinder pipelining.

The OpenCL interface, on the contrary, was from the start meant to be asynchronous, and provides very fine-grain synchronization control possibilities.

The programming interface for Xeon Phi as an accelerator actually comes in two layers. COI is a high-level interface which is very convenient for end-user applications by hiding data transfers. SCIF is on the contrary a low-level interface which allows to get control on data transfers. Providing these two interfaces separately makes a lot of sense. OpenCL probably suffered from providing a quite low-level interface, while CUDA suffered from providing initially a too high-level interface, so separating both aspects into COI and SCIF was probably a good way to make both kinds of usage satisfied.

The MPI interface provides, for the most useful communication operations, asynchronous versions (called non-blocking) whose termination can then be tested in various ways.

A concern which is however common to these asynchronous interfaces, is how to efficiently check for termination of several requests. *Probing* functions are usually available, to be able to test for the termination of a specific request. MPI allows with MPI_Testsome to check several requests at the same time. When to actually call such probing is however questionable [Tra09, TD09b]: doing it only between tasks executions means introducing a reactivity latency which is as long as a typical task duration. Introducing another thread to probe more often might require a dedicated core to avoid disturbing computation. Another way is to make a thread *wait* for the completion of operations, thus not consuming CPU time. The interfaces provided to achieve this are however usually not suited to the case at stake here, because they make the caller pass the set of requests to be looked after. A runtime would rather typically make the thread just wait for *any* request termination, not only the requests which were queued when the thread started the wait, but also requests which get queued later on. The POSIX `select` interface can easily be interrupted by sending a signal or writing to a dedicated pipe, only to be restarted with addition file descriptors to be monitored. Blocking functions provided by HPC interfaces however usually do not provide such a feature, which thus hinder their usability for a runtime, since we then can not dynamically add requests to be checked for. We will work with Alexandre Denis on this aspect with the NewMadeleine communication library, to improve the relation between a runtime system and

the underlying library, beyond what the MPI interface can express.

The HiHAT project[1] aims at providing a software stack which runtime systems could build on at their preferred level. I have participated to discussions on the asynchronous operation interface to raise this "wait" issue; this turned into the requirement of being able to combine events to be waited for, and to provide user-defined asynchronous events, thus allowing to interrupt a wait operation similarly to the POSIX pipe way.

Another common way to notify the termination of requests is using *callbacks*. For instances CUDA supports inserting callbacks within the stream of kernels and transfers execution, and NewMadeleine supports setting a callback on communication termination. This approach provides a lot of flexibility since the runtime can then synchronize its preferred way. The question is which context the callback gets called in, and what it is allowed to achieve. Callbacks may get called from calls to the API, or even from a separate thread started by the underlying layer, whose status is often not really defined. Using more than synchronization release functions in the callback can even risk recursion: if the callback e.g. submits subsequent operations itself, they might succeed immediately, trigger other callbacks, etc. The CUDA API actually simply forbids making CUDA calls from callbacks. In the end, callbacks are typically used to just notify another thread, with the entailed overhead.

5.6 Developing and maintaining a runtime system for research

In Section 2.3 we had detailed the availability of a wide range of runtime systems. Quite a few of them are very similar, we have for instance often been asked for the differences between OmpSs, KAAPI, and StarPU, since their programming models are very similar, and they provide similar sets of features. The details vary of course, but over time differences appear and disappear as each project spends various implementation efforts. The real differences appear in the ground research topics that the respective teams are conducting. Actually, the fact that these runtime systems seem to have converged to similar models and features may be a sign that these are the proper models and features to be implemented.

It would make sense to agree on a common runtime implementation which various research teams would contribute to. It is however much easier for research teams to work on their own runtime system, whose implementation is well understood within the team, and is suited to its research goals. For a newly-created research team, it is also much easier to start working with a new simple runtime implementation in which it is convenient to conduct the targeted research, rather than having to dive into the complex implementation of an existing complete runtime system. Similarly, it may look simpler to users to rewrite a runtime suited exactly to their application needs than having to either wait for maintainers of an existing runtime to implement the missing features, or to contribute them to the runtime. Sometimes the runtime actually already provides the desired features, but understanding how to achieve it from the documentation sometimes look more complex than just reimplementing it. Most often, such reimplementation progressively gets more and more features, up to actually becoming as complex as the other runtime systems, and thus similarly hard to dive into.

A first concern with this variety of implementations is the corresponding variety of programming interfaces. Back in the days of research on threading libraries, a wild variety of threading interfaces were proposed, but the introduction of the POSIX threads interface eventually got consensus. Similarly, MPI provided a common interface at a time when various communication libraries were competing. As was detailed in Section 2.6, OpenMP could be a promising candidate for a common programming interface, I do not think we are exactly there yet, though. One of the issues may be that it is a language extension, which is more difficult to implement than the POSIX threads and MPI interfaces which expose a library interface. Perhaps a library interface

[1] `https://hihat-wiki.modelado.org/`

will have to be additionally introduced to really get a consensus. A common standard would also need to be able to evolve fast enough to be adopted both by research projects and users.

A second concern with the variety of implementations is that maintenance effort is scattered. Some research-oriented runtime systems are only experimentation boxes for getting results and do not aim at being actually adopted by end-users. Some other systems aim at wide adoption, but this means that they have to perform *well*, and notably to make end-users accept using a runtime at all, as was discussed in Section 2.2.1. This means that for a start, research contributions need to be assembled in the same implementation, so that users can benefit from them altogether. A lot of implementation details also have to be fixed to get proper performance, we for instance had to integrate an efficient implementation of prioritized lists. Less trivial details also need to be supported, even if they are not at the core of the research conducted along the runtime: for instance, the optimized support for commutative data access which was described in Section 3.3 was really important for getting the FMM application to scale, was tricky to implement, but did not involve really innovative solutions. And all of this has to actually work, even in corner cases, which means tedious bug tracking, Continuous Integration, etc. to get a really strong implementation. The whole support also needs to be properly documented. Getting an implementation to work well enough to be adopted is thus a lot of hard engineering work, which laboratories do not necessarily have budget for, and are thus rather funded on the side of research projects. Some projects such as OCR and HiHAT are attempting to provide a common framework where research could be conducted, similarly to one of the goals of OpenMPI, they are however not there yet.

It is thus questionable whether it is worth spending the time and energy to make a research runtime system actually able to get adopted by end-users, or rather just focus on experimentation boxes. For the StarPU project we did choose to try the former, to bring as much of the state-of-the-art to applications, possibly at the expense of some missing optimizations, and we did not spend too much effort on providing high-level languages, but just provide compatibility layers with standard interfaces such as OpenCL and OpenMP. The result seems worth it. As was mentioned in Section 2.8, we have gathered various application test-cases, and StarPU is actually used for instance by AirBus customers for its Out-of-Core support. But as mentioned in the previous chapters, the strong position of StarPU also made it an interesting testbed for various research not directly related to task-based programming, such as communication optimization, performance visualization, correctness, etc. which benefit from the gathered application test-cases.

Chapter 6

Conclusion and perspectives

This document has shown that even when concentrating on the runtime part of task-based programming, a wide range of interesting challenges arise. The keys are that task graphs provide rich information that various parts of a runtime can benefit from, and that the time interval between task submission and task execution can be used to perform optimizations without delaying execution. In the context of the StarPU project, we have explored many aspects of such a runtime, from task scheduling to execution simulation, using various approaches in collaboration with diverse research teams, ranging from theoretical background to visualization expertise.

More precisely, we have explored how to build schedulers in a flexible way, allowing to leverage advanced task scheduling heuristics for real applications. We have experimented with respecting memory size constraints thanks to memory overflow recovery and avoidance approaches, using task submission throttling or rather introducing additional task dependencies. We have discussed aggregating tasks, or conversely dividing tasks to automatically adapt their granularity, but also more generally to convey a hierarchical structure of the graph, which may then be used by various heuristics. We have proposed to simulate the execution of the runtime, so as to be able to predict the execution time of applications, which entails a whole class of new possibilities, both for users and researchers. We provided with ways to verify and debug the application, and visualize performance results. We introduced a completely distributed execution model with competing results. All of this was integrated into a single piece of software, StarPU, and was achieved with minimal modifications to the main application source code, provided that it is already expressed as tasks, notably with the commonly-used Sequential Task Flow (STF) paradigm.

For some of these contributions, we were inspired by the state of the art in Operating-Systems, but extending it thanks to the task graph information. Conversely for other contributions, we used theoretical heuristics, by adapting them to the realities of a runtime system. For most of these contributions, collaboration with research teams comprising diverse expertise was key to success. The PhD thesis of Suraj Kumar for instance, was co-advised by Olivier Beaumont and Lionel Eyraud for the scheduling theory side, by Emmanuel Agullo for the numerical algorithms side, and by myself for the runtime side. That thesis acted as a bridge between the scheduling theory side and the numerical algorithms side through the runtime side. More generally, the availability of real-world applications on top of StarPU makes it an interesting experimentation platform, and it is now used as such for instance for testing task scheduling heuristics in the wild, for improving the NewMadeleine network communication library, or the SimGrid simulator. It also provides interesting test cases for application trace visualization and analysis. Conversely, application programmers can use StarPU to easily try new numerical algorithms, all runtime concerns being handled automatically.

Perspectives

Naturally enough, task scheduling remains a challenge in general. We have observed that improvement needs to be achieved in the balance between optimizing for the critical path and optimizing for locality to avoid data transfers, and in simply really scheduling the data transfers. On the parallel tasks side, a lot of theoretical work on scheduling moldable tasks remains to be exploited in real conditions in a runtime system. Generally enough, scheduling heuristics are quite expensive, conceiving incremental scheduling strategies would thus be a decisive step to make them practical. Hierarchical scheduling through the notion of bubbles could also be an option; what this could really look like however remains to be devised. All of this should be conducted in close collaboration with theoreticians, and hence it is essential to make it very simple for them to experiment with scheduling within a runtime system, for instance with simulated execution and the availability of various real-world test-cases on a task graph market. The modular way of writing a scheduler which was presented also helps a lot to eliminate most concerns from the sight of the scheduling core that theoreticians are concerned with. The modular API would however deserve a formal model to be able to prove its correctness.

Concerning the management of memory use, it is unclear how it should be addressed. Several concerns arise at the same time: either managing to fit in the available memory, or having to handle at the same time data prefetching, data eviction, and in the case of distributed execution, cached data. Which model should be used is uncertain: considering memory usage at the task granularity is probably way too expensive, but should we then create groups of tasks, or rather introduce pseudo-tasks which represent allocation and release of memory, or groups of allocations and releases? The exact interaction between memory control strategies and the rest of the runtime system is also unsettled, we have for instance in this document sketched some extended dependency notion which could help to both make strategies less costly, and leave more control to the runtime system for dynamic refinements. In the cases where no memory consumption guarantees can be asserted on tasks themselves, stochastic analysis would be needed to be able to at least provide probabilistic estimations of overflow avoidance.

Elaborating on the notion of hierarchical task graphs will probably be key to unlock various challenges, not only to get dividable tasks which can thus accommodate the heterogeneity of available computation resources, but also to provide a hierarchical structure of the application. It allows to mitigate the cost of the task graph submission, but can also convey precious overall information on the task graph, that can be used for overall task scheduling, memory management, distributed execution, etc. At the lowest hierarchical levels, it will be challenging to automatically determine whether an optimized parallel implementation should be used for moderately big tasks, or whether recursion should be pursued down to small tasks to be run on single cores. How hierarchical task graphs should ideally be expressed at the language level is also yet unsettled and will be questioned before a potential inclusion in the OpenMP standard.

On the OpenMP standard side, a lot remains to be integrated to fully cover the features provided by an advanced runtime such as StarPU. OpenMP notably supports offloading only array sections for now, not even sub-matrices. To fully benefit from advanced scheduling which requires task performance models, the standard will also need to define a way to specify on tasks which kind of performance model should be used, and notably the parameters to be used for regression-based models.

A lot of information could be obtained from a polyhedral analysis of the task submission loop. Figure A.1 of Appendix A, page 102, shows the hints that we introduced to enable advanced optimizations of the Cholesky factorization. These could actually be automatically injected by a source-to-source compiler, thus keeping the application algorithm intact. Such a compiler could even reorder loops to submit tasks in a more breadth-first order that would help for e.g. incremental scheduling. A compiler pass could also be needed for seamless task graph checkpoint/restart support, to automatically generate a task submission loop which can be properly restarted at

the checkpoint location (which could even be determined automatically from the amount of data liveness). Compilers could also help with checking the behavior of the application program, to e.g. track improper data accesses from tasks.

We see more and more communities adopting programming environments at a much higher level than C and Fortran, such as Python with numpy, Julia, or notebooks such as Jupyter. For now, task-based programming is typically hidden behind the numerical libraries called from such environments, but it would make sense to bring task-based programming at the front of these high-level layers, in relation with the underlying task-based execution. High-level languages could be used to submit the highest levels of a hierarchical task graph, lower levels being submitted by the numerical libraries. Job submission on computation clusters could even be managed automatically to execute whole subgraphs. To put it simply, it would mean taking the BigData approach for cloud computing, thus helping with the convergence of HPC and BigData. The Out-of-Core support explained in Section 3.2.1 could also benefit from the BigData experience on disk storage, HDFS could for instance be leveraged to store data which does not fit in the main memory.

Understanding the behavior of task graph execution is necessary to be able to improve performance. We will thus need ways to represent the quintessence of this behavior, filtering out useless information, and pinpointing erratic behavior. On very large platforms, we can not just take a thorough trace of events, sampling and aggregation approaches will be needed. Statistical analysis could then be used to spot noteworthy behavior. The important point is then to relate such analysis to the application-level notions, such as which part of the matrix is concerned by the erratic behavior. Details such as matrix tile coordinates should thus be propagated to the visual rendering, which would then be able to represent execution in a way most suited to end-users.

Applying model-checking on an application can be made simpler by the task-based paradigm, since that separates the task submission source code from the actual implementations. Checking the runtime system itself will however remain a challenge, we hope that its modularity can permit checking modules separately, and then checking the overall behavior.

Proper distributed execution will be key to scalability over very large platforms. Using a hierarchy of masters, sub-masters, etc. is probably worth trying; but the distributed approach exposed in this document, consisting in unrolling the task graph independently on each node without synchronization, is really appealing. Task graph pruning as explained in Section 4.5 will probably quickly show its scaling limits, we will need to generalize a refined use of hierarchical graphs to better control the graph unrolling, while avoiding too coarse dependency tracking. Advanced interaction with the communication library will probably also be decisive to optimize the irregular flux of data transfers entailed by task graphs. In this document we sketched the potential for opportunistic collective sends; the runtime could more generally tell the communication library ahead of time which *future* transfers will be submitted, to help it with aggregation strategies for instance. Dynamic workload redistribution driven by the application is already possible with StarPU without global synchronization, we plan to experiment with it on a Navier-Stokes Solver. Completely automatic redistribution support would be compelling but it will be a challenge to make it scalable. Hierarchical task graphs would probably be very useful to get a global vision to achieve appropriate redistribution at a reasonable cost. Last but far from the least, the task graph paradigm can probably be leveraged for fault resiliency by automatically restarting parts of the task graph. This will pose a lot of challenges, but at least a proof of concept can probably be achieved for a start.

Appendix A

```
for (k = 0; k < A->mt; k++) {
    RUNTIME_iteration_push(chamctxt, k);

    tempkm = k == A->mt-1 ? A->m-k*A->mb : A->mb;
    ldak = BLKLDD(A, k);

    options.priority = 2*A->mt - 2*k;
    INSERT_TASK_zpotrf(
        &options,
        ChamLower, tempkm, A->mb,
        A(k, k), ldak, A->nb*k);

    for (m = k+1; m < A->mt; m++) {
        tempmm = m == A->mt-1 ? A->m-m*A->mb : A->mb;
        ldam = BLKLDD(A, m);

        options.priority = 2*A->mt - 2*k - m;
        INSERT_TASK_ztrsm(
            &options,
            ChamRight, ChamLower, ChamConjTrans, ChamNonUnit,
            tempmm, A->mb, A->mb,
            zone, A(k, k), ldak,
                  A(m, k), ldam);
    }
    RUNTIME_data_flush( sequence, A(k, k) );

    for (n = k+1; n < A->nt; n++) {
        tempnn = n == A->nt-1 ? A->n-n*A->nb : A->nb;
        ldan = BLKLDD(A, n);

        options.priority = 2*A->mt - 2*k - n;
        INSERT_TASK_zherk(
            &options,
            ChamLower, ChamNoTrans,
            tempnn, A->nb, A->mb,
            -1.0, A(n, k), ldan,
             1.0, A(n, n), ldan);

        for (m = n+1; m < A->mt; m++) {
            tempmm = m == A->mt-1 ? A->m - m*A->mb : A->mb;
            ldam = BLKLDD(A, m);

            options.priority = 2*A->mt - 2*k - n - m;
            INSERT_TASK_zgemm(
                &options,
                ChamNoTrans, ChamConjTrans,
                tempmm, tempnn, A->mb, A->mb,
                mzone, A(m, k), ldam,
                       A(n, k), ldan,
                zone,  A(m, n), ldam);
        }
        RUNTIME_data_flush( sequence, A(n, k) );
    }
    RUNTIME_iteration_pop(chamctxt);
}
```

Figure A.1: Verbatim Cholesky implementation from Chameleon's compute/pzpotrf.c. Purple lines provide the runtime with priorities for the dmdas scheduler. Blue lines provide hints for cache flushes and LRU. Green lines provide iteration numbers to the runtime for advanced performance feedback.

Bibliography

[ABC+14] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based FMM for multicore architectures. *SIAM Journal on Scientific Computing*, 36(1):C66–C93, 2014.

[ABC+16a] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, , Xiaoqiang Zheng, and Google Brain. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[ABC+16b] Emmanuel Agullo, Berenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based FMM for heterogeneous architectures. *Concurrency and Computation: Practice and Experience*, 28(9), June 2016.

[ABD+90a] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A Portable Linear Algebra Library for High-performance Computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[ABD+11] Emmanuel Agullo, Henricus Bouwmeester, Jack Dongarra, Jakub Kurzak, Julien Langou, and Lee Rosenberg. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In José M. Laginha M. Palma, Michel Daydé, Osni Marques, and João Correia Lopes, editors, *High Performance Computing for Computational Science – VECPAR 2010*, pages 129–138, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[ABGL13] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Multifrontal QR Factorization for Multicore Architectures over Runtime Systems. In *19th International Conference Euro-Par (EuroPar 2013)*, volume 8097, pages pp. 521–532, Aachen, Germany, August 2013.

[ABGL14] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems. Rapport de recherche IRI/RT–2014-03–FR, IRIT, Université Paul Sabatier, Toulouse, novembre 2014.

[ABI+09] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th Euro-Par Conference*, Delft, The Netherlands, August 2009.

[ACD+09] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[ADD+09a] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[AKG+15] D. Akhmetova, G. Kestor, R. Gioiosa, S. Markidis, and E. Laure. On the application task granularity and the interplay with the scheduling overhead in many-core shared memory systems. In *2015 IEEE International Conference on Cluster Computing (CLUSTER)*, volume 00, pages 428–437, Sept. 2015.

[ATNW11a] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[Bar12] João Barbosa. Gama framework: Hardware aware scheduling in heterogeneous environments. Technical report, University of Texas at Austin, 09 2012.

[BBD+10b] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for high performance computing, SEP 2010. UT-CS-10-659.

[BBD+11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. In *Proceedings of the 25th IEEE International Symposium on Parallel & Distributed Processing Workshops and Phd Forum (IPDPSW'11), PDSEC 2011*, pages 1432–1441, Anchorage, United States, mai 2011.

[BBD+12a] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, and Jack Dongarra. From serial loops to parallel execution on distributed systems. In *European Conference on Parallel Processing*, pages 246–257. Springer, 2012.

[BBD+13a] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack Dongarra. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering*, 15(6):36–45, November 2013.

[BC89] Marsha J Berger and Phillip Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of computational Physics*, 82(1):64–84, 1989.

[BCOM+10a] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, pages 180–186, Pisa, Italia, February 2010. IEEE Computer Society Press.

[BDM+11] Javier Bueno, Alejandro Duran, Xavier Martorell, Eduard Ayguadé, Rosa M. Badia, and Jesús Labarta. Poster: programming clusters of gpus with ompss. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 378–378, New York, NY, USA, 2011. ACM.

[Beb00b] Mario Bebendorf. Approximation of Boundary Element Matrices. *Numerische Mathematik*, 86:565–589, 2000.

[BEDG18] Olivier Beaumont, Lionel Eyraud-Dubois, and Yihong Gao. Influence of Tasks Duration Variability on Task-Based Runtime Schedulers. Research report, INRIA, February 2018.

[BGNK11] Steffen Brinkmann, José Gracia, Christoph Niethammer, and Rainer Keller. TEMANEJO - a debugger for task based parallel programming models. *CoRR*, abs/1112.4604, 2011.

[BHKS+16] Raphaël Bleuse, Sascha Hunold, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. Scheduling Independent Moldable Tasks on Multi-Cores with GPUs. Research Report RR-8850, Inria Grenoble Rhône-Alpes, Université de Grenoble, January 2016.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[BK09b] Peter Brucker and Sigrid Knust. Complexity results for scheduling problems. Web document, URL: `http://www2.informatik.uni-osnabrueck.de/knust/class/`, 2009.

[BKSM+15] Raphael Bleuse, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. Scheduling Independent Tasks on Multi-cores with GPU Accelerators. *Concurr. Comput. : Pract. Exper.*, 27(6):1625–1638, April 2015.

[BL99a] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[BLKD09a] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Computing*, 35(1):38–53, 2009.

[BLS+03] Rosa M. Badia, Jesús Labarta, Raül Sirvent, Josep M. Pérez, José M. Cela, and Rogeli Grima. Programming grid applications with grid superscalar. *Journal of Grid Computing*, 1:151–170, 2003.

[BMC+15] Badia, R. M., J. Conejero, C. Diaz, J. Ejarque, D. Lezzi, F. Lordan, C. Ramon-Cortes, and R. Sirvent. COMP Superscalar, an interoperable programming framework. In *SoftwareX*, volume 3-4, pages 32–36, December 2015.

[BPC+09] Pieter Bellens, Josep M. Pérez, Felipe Cabarcas, Alex Ramírez, Rosa M. Badia, and Jesús Labarta. Cellss: Scheduling techniques to better exploit memory hierarchy. *Scientific Programming*, 17(1-2):77–95, 2009.

[BPD+12] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. Productive programming of gpu clusters with ompss. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 557–568, May 2012.

[BTSA12a] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[CDD+96] Jaeyoung Choi, James Demmel, Inderjiit Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David Walker, and R Clinton Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers—Design issues and performance. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 95–106. Springer, 1996.

[CDDP10] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A parallel functional simulator for gpgpu. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 351–360, Aug 2010.

[CFJ+] Kevin Coulomb, Mathieu Faverge, Johnny Jazeix, Olivier Lagrasse, Jule Marcoueille, Pascal Noisette, Arthur Redondy, and Clément Vuchener. Vite's project page. `http://vite.gforge.inria.fr/`.

[CGH94] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[CJ09] Louis-Claude Canon and Emmanuel Jeannot. Evaluation and optimization of the robustness of dag schedules in heterogeneous environments. *IEEE Transactions on Parallel and Distributed Systems*, 99(RapidPosts):532–546, 2009.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[CL95b] Michel Cosnard and Michel Loi. Automatic task graph generation techniques. In *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 2, pages 113–122. IEEE, 1995.

[CLQ08] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation (UKSim)*, April 2008.

[Cod60] E. F. Codd. Multiprogram scheduling: Parts 1 and 2. introduction and theory. *Commun. ACM*, 3(6):347–350, June 1960.

[CVZB+08a] Ernie Chan, Field G Van Zee, Paolo Bientinesi, Enrique S Quintana-Orti, Gregorio Quintana-Orti, and Robert Van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 123–132. ACM, 2008.

[DBB07] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment, 2007.

[DBB+14] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. PTG: an abstraction for unhindered parallelism. In *Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on*, pages 21–30. IEEE, 2014.

[DBMS79] Jack J Dongarra, James R Bunch, Cleve B Moler, and Gilbert W Stewart. *LINPACK users' guide*. Siam, 1979.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large Scale Distributed Deep Networks. In f. pereira, c. j. c. burges, l. bottou, and k. q. weinberger, editors, *advances in neural information processing systems 25*, pages 1223–1231. curran associates, inc., 2012.

[DEK11] Usman Dastgeer, Johan Enmyren, and Christoph W. Kessler. Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems. In *Proceeding of the 4th international workshop on Multicore software engineering*, IWMSE '11, pages 25–32, New York, NY, USA, 2011. ACM.

[Den74] J. B. Dennis. First version of a data fow procedure language. In Springer Berlin Heidelberg, editor, *Programming Symposium*, pages 362–376, 1974.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[Dij65] Edsger W. Dijkstra. Een algorithme ter voorkoming van de dodelijke omarming. `http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF`, 1965. circulated privately.

[Dij82] Edsger W. Dijkstra. The mathematics behind the banker's algorithm. *Selected Writings on Computing: A personal Perspective*, 1982.

[DMT04] Pierre-Francois Dutot, Grégory Mounié, and Denis Trystram. Scheduling Parallel Tasks: Approximation Algorithms. In Joseph T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 26, pages 26–1 – 26–24. CRC Press, 2004.

[DNW05] Vincent Danjean, Raymond Namyst, and Pierre-André Wacrenier. An efficient multi-level trace toolkit for multi-threaded applications. In *EuroPar*, Lisbonne, Portugal, September 2005.

[Don13] Jack Dongarra. Architecture-aware algorithms for scalable performance and resilience on heterogeneous architectures. 3 2013.

[DT16] Alexandre Denis and François Trahay. MPI Overlap: Benchmark and Analysis. In *International Conference on Parallel Processing*, 45th International Conference on Parallel Processing, Philadelphia, United States, August 2016.

[DW95] J. Dongarra and D. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.

[EGC02] Tarek El-Ghazawi and Francois Cantonnet. UPC performance and potential: A NPB experimental study. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 17–17. IEEE, 2002.

[EGK+01] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. Graphviz — open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

[EK10] Johan Enmyren and Christoph W. Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[FR09] Mathieu Faverge and Pierre Ramet. A NUMA Aware Scheduler for a Parallel Sparse Direct Solver. In *Workshop on Massively Multiprocessor and Multicore Computers*, page 5p., Rocquencourt, France, February 2009. INRIA.

[GCRD98] F. Galilee, G.G.H. Cavalheiro, J.-L. Roch, and M. Doreille. Athapascan-1: Online building data flow graph in a parallel language. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 88 –95, oct 1998.

[GGHVDG01] John A Gunnels, Fred G Gustavson, Greg M Henry, and Robert A Van De Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)*, 27(4):422–455, 2001.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[GLMR13a] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1299–1308. IEEE, 2013.

[Gra66] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[HDB+13] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. Mpi+ mpi: a new hybrid approach to parallel programming with mpi plus shared memory. *Computing*, 95(12):1121–1136, 2013.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

[HRF+10b] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. Multi-gpu and multi-cpu parallelization for interactive physics simulations. In Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 235–246. Springer Berlin / Heidelberg, 2010.

[I. 14] I. D. Mironescu and L. Vinţan. Coloured petri net modelling of task scheduling on a heterogeneous computational node. In *2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 323–330, Sept 2014.

[int09a] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, Santa Clara, USA, 2009. ISBN 630813-054US.

[KBB+06] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the open trace format (otf). In Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, pages 526–533, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[KBS09] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *2009 International Conference on Parallel Processing Workshops*, pages 394–401. IEEE, 2009.

[KHAL+14] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[KK93a] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[KMJ94] A. A. Khan, C. L. McCreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. In *Proceedings of the 1994 International Conference on Parallel Processing, volume II*, pages 243–250, 1994.

[KRSS94] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *Parallel Programming Laboratory Technical Report #95-03*, 1994.

[KSM00] Jacques Chassin De Kergommeaux, Benhur De Oliveira Stein, and Montbonnot Saint Martin. Paje: An extensible environment for visualizing multithreaded program executions. In *Proc. Euro-Par 2000, Springer-Verlag, LNCS*, pages 133–144, 1900.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28:690–691, September 1979.

[Law09] O.S. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –8, 31 2009-sept. 4 2009.

[Lei09] Charles E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51:522–527, 2009.

[LFB+14] Xavier Lacoste, Mathieu Faverge, George Bosilca, Pierre Ramet, and Samuel Thibault. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 29–38. IEEE, 2014.

[LHKK79b] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[LMB02] Julia L Lawall, Gilles Muller, and Luciano Porto Barreto. Capturing OS expertise in an Event Type System: the Bossa experience. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 54–61. ACM, 2002.

[LST90] Jan Karel Lenstra, David B Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 1990.

[MNSV18] L. Marchal, H. Nagy, B. Simon, and F. Vivien. Parallel scheduling of dags under memory constraints. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 204–213, May 2018.

[Mon14] Florence Monna. *Scheduling for new computing platforms with GPUs*. PhD thesis, Université Pierre et Marie Curie - Paris VI, November 2014.

[MRK08] HasnainA. Mandviwala, Umakishore Ramachandran, and Kathleen Knobe. Capsules: Expressing composable computations in a parallel programming model. In Vikram Adve, MaríaJesús Garzarán, and Paul Petersen, editors, *Languages and Compilers for Parallel Computing*, volume 5234, chapter Lecture Notes in Computer Science, pages 276–291. Springer Berlin Heidelberg, 2008.

[Ope13a] OpenACC-Standard. The OpenACC application programming interface, May 2013.

[PBAL09a] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[PBF10] Artur Podobas, Mats Brorsson, and Karl-Filip Faxén. A comparison of some recent task-based parallel programming models. In *3rd Workshop on Programmability Issues for Multi-Core Computers*, Pisa, Italy, 2010.

[PBL08] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A dependency-aware task-based programming environment for multi-core architectures. Proceedings of the 2008 IEEE International Conference on Cluster Computing, Sept 2008. pp. 142-151.

[PHA10] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 376–387, 2010.

[POP16] A. Pereira, A. Onofre, and A. Proenca. Tuning pipelined scientific data analyses for efficient multicore execution. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 751–758, July 2016.

[PR96] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In Heather Liddell, Adrian Colbrook, Bob Hertzberger, and Peter Sloot, editors, *High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer Berlin / Heidelberg, 1996.

[Rei07b] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[Roc15] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference (SciPy 2015)*, pages 130–136, 2015.

[RSL93a] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26:28–38, June 1993.

[Sim18] Bertrand Simon. *Scheduling task graphs on modern computing platforms*. Theses, Université de Lyon, July 2018.

[SLT+15] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 81. ACM, 2015.

[Sod15] A. Sodani. Knights landing (knl): 2nd generation intel xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24, Aug 2015.

[SP99] João Luís Sobral and Alberto José Proença. Dynamic grain-size adaptation on object oriented parallel programming the SCOOPP approach. In *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, IPPS '99/SPDP '99, pages 728–732, Washington, DC, USA, 1999. IEEE Computer Society.

[SRG15] Christian Simmendinger, Mirko Rahn, and Daniel Gruenewald. The gaspi api: A failure tolerant pgas api for asynchronous dataflow on heterogeneous architectures. In Michael M. Resch, Wolfgang Bez, Erich Focht, Hiroaki Kobayashi, and Nisarg Patel, editors, *Sustained Simulation Performance 2014*, pages 17–32, Cham, 2015. Springer International Publishing.

[Sus97] Susan Blackford. The Two-dimensional Block-Cyclic Distribution, May 1997.

[SYD09b] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. IEEE, 2009.

[Szy03b] Clemens Szyperski. Component technology: what, where, and how? In *Proceedings of the 25th international conference on Software engineering*, page 684–693, May 2003.

[TD09b] François Trahay and Alexandre Denis. A scalable and generic task scheduling system for communication libraries. In *IEEE International Conference on Cluster Computing*, New Orleans, LA, United States, August 2009. IEEE Computer Society Press.

[TDB10] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

[TDH+18] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, Apr 2018.

[TFG+12] Enric Tejedor, Montse Farreras, David Grove, Rosa M Badia, Gheorghe Almasi, and Jesus Labarta. A high-productivity task-based programming model for clusters. *Concurrency and Computation: Practice and Experience*, 24(18):2421–2448, 2012.

[THW99] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, HCW '99, page 3, Washington, DC, USA, 1999. IEEE Computer Society.

[Til14] Martin Tillenius. *Scientific Computing on Multicore Architectures*. PhD thesis, Uppsala University, 2014.

[Til15] Martin Tillenius. SuperGlue: A shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM Journal on Scientific Computing*, 37(6):C617–C642, 2015.

[TNW07] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building portable thread schedulers for hierarchical multiprocessors: The bubblesched framework. In *Euro-Par 2007 Parallel Processing*, volume 4641, pages 42–51. Springer, 2007.

[TPO10] Stanly Tzeng, Anjul Patney, and John D. Owens. Poster: Task management for irregular workloads on the gpu. In *Proceeding of NVIDIA GPU Technology Conference*, 2010.

[Tra09] François Trahay. *De l'interaction des communications et de l'ordonnancement de threads au sein des grappes de machines multi-cœurs*. PhD thesis, 2009. Thèse de doctorat dirigée par Namyst, Raymond et Denis, Alexandre Informatique Bordeaux 1 2009.

[UJM+12] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 335–344, New York, NY, USA, 2012. ACM.

[Ull75] Jeffrey D. Ullman. NP-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[WBB+15] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. Hierarchical DAG scheduling for Hybrid Distributed Systems. In *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Hyderabad, India, May 2015.

[Yar12] Asim YarKhan. *Dynamic task execution on shared and distributed memory architectures*. PhD thesis, University of Tenessee, 2012.

[YKD] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels.

[YKLD17] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the plasma numerical library to the openmp standard. *International Journal of Parallel Programming*, 45(3):612–633, Jun 2017.

[Zaf18] Afshin Zafari. Taskuniverse: A task-based unified interface for versatile parallel execution. In Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 169–184, Cham, 2018. Springer International Publishing.

[ZKD+14] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. UPC++: a PGAS Extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1105–1114. IEEE, 2014.

[ZL18] Afshin Zafari and Elisabeth Larsson. Distributed dynamic load balancing for task parallel programming. *CoRR*, abs/1801.04582, 2018.

[ZLT16] Afshin Zafari, Elisabeth Larsson, and Martin Tillenius. DuctTeip: A task-based parallel programming framework for distributed memory architectures. Technical Report 2016-010, Department of Information Technology, Uppsala University, June 2016.

List of publications I am co-author of

Theses

[1] Samuel Thibault. *Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications*. PhD thesis, Université Bordeaux 1, December 2007.

International journals

[2] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[3] Paul-Antoine Arras, Didier Fuin, Emmanuel Jeannot, Arthur Stoutchinin, and Samuel Thibault. List Scheduling in Embedded Systems Under Memory Constraints. *International Journal of Parallel Programming*, November 2014.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[5] Vinicius Garcia Pinto, Lucas Mello Schnorr, Luka Stanisic, Arnaud Legrand, Samuel Thibault, and Vincent Danjean. A Visual Performance Analysis Framework for Task-based Parallel Applications running on Hybrid Clusters. *Concurrency and Computation: Practice and Experience*, April 2018.

[6] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. *Concurrency and Computation: Practice and Experience*, page 16, May 2015.

Books

[7] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

International conferences

[8] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *25th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2011)*, Anchorage, Alaska, USA, May 2011.

[9] Paul-Antoine Arras, Didier Fuin, Emmanuel Jeannot, Arthur Stoutchinin, and Samuel Thibault. List Scheduling in Embedded Systems under Memory Constraints. In Juan Guerrero, editor, *SBAC-PAD'2013 - 25th International Symposium on Computer Architecture and High-Performance Computing*, Porto de Galinhas, Brésil, October 2013. Federal University of Pernambuco & Federal University of Minas Gerais, IEEE Computer Society.

[10] Paul-Antoine Arras, Didier Fuin, Emmanuel Jeannot, and Samuel Thibault. DKPN: A Composite Dataflow/Kahn Process Networks Execution Model. In *24th Euromicro International Conference on Parallel, Distributed and Network-based processing*, Heraklion Crete, Greece, February 2016.

[11] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *The 16th International Conference on Parallel and Distributed Systems (ICPADS)*, Shanghai, China, December 2010.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874, Delft, The Netherlands, August 2009. Springer.

[13] Siegfried Benkner, Enes Bajrovic, Erich Marth, Martin Sandrieser, Raymond Namyst, and Samuel Thibault. High-Level Support for Pipeline Parallelism on Many-Core Architectures. In *Europar - International European Conference on Parallel and Distributed Computing - 2012*, Rhodes Island, Grèce, August 2012.

[14] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, pages 180–186, Pisa, Italia, February 2010. IEEE Computer Society Press.

[15] Usman Dastgeer, Christoph Kessler, and Samuel Thibault. Flexible runtime support for efficient skeleton programming on hybrid systems. In *Proceedings of the International Conference on Parallel Computing (ParCo), Applications, Tools and Techniques on the Road to Exascale Computing*, volume 22 of *Advances of Parallel Computing*, pages 159–166, Gent, Belgium, August 2011.

[16] Christoph Kessler, Usman Dastgeer, Samuel Thibault, Raymond Namyst, Andrew Richards, Uwe Dolinsky, Siegfried Benkner, Jesper Larsson Träff, and Sabri Pllana. Programmability and Performance Portability Aspects of Heterogeneous Multi-/Manycore Systems. In *Design, Automation and Test in Europe (DATE)*, Dresden, Allemagne, March 2012.

[17] Víctor Martínez, David Michéa, Fabrice Dupros, Olivier Aumage, Samuel Thibault, Hideo Aochi, and Philippe Olivier Alexandre Navaux. Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In *27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Florianopolis, Brazil, October 2015.

[18] Tetsuya Odajima, Taisuke Boku, Mitsuhisa Sato, Toshihiro Hanawa, Yuetsu Kodama, Raymond Namyst, Samuel Thibault, and Olivier Aumage. Adaptive Task Size Control on High Level Programming for GPU/CPU Work Sharing. In *The 2013 International Symposium on Advances of Distributed and Parallel Computing (ADPC 2013)*, Vietri sul Mare, Italie, December 2013.

[19] Satoshi Ohshima, Satoshi Katagiri, Kengo Nakajima, Samuel Thibault, and Raymond Namyst. Implementation of FEM Application on GPU with StarPU. In *SIAM CSE13 - SIAM Conference on Computational Science and Engineering 2013*, Boston, États-Unis, February 2013. SIAM.

[20] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. In *Euro-par - 20th International Conference on Parallel Processing*, Porto, Portugal, August 2014. Springer-Verlag.

International workshops

[21] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Jean Roman, Samuel Thibault, and Stanimire Tomov. Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, Knoxville, USA, July 2010.

[22] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. Harnessing clusters of hybrid nodes with a sequential task-based programming model. In *8th International Workshop on Parallel Matrix Algorithms and Applications*, July 2014.

[23] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, Julien Herrmann, Suraj Kumar, Loris Marchal, and Samuel Thibault. Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms. In *Heterogeneity in Computing Workshop 2015*, Hyderabad, India, May 2015.

[24] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *Proceedings of the International Euro-Par Workshops 2009, HPPC'09*, volume 6043 of *Lecture Notes in Computer Science*, pages 56–65, Delft, The Netherlands, August 2009. Springer.

[25] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Maik Nijhuis. Exploiting the Cell/BE architecture with the StarPU unified runtime system. In *SAMOS Workshop - International Workshop on Systems, Architectures, Modeling, and Simulation*, volume 5657 of *Lecture Notes in Computer Science*, Samos, Greece, July 2009.

[26] Vinicius Garcia Pinto, Luka Stanisic, Arnaud Legrand, Lucas Mello Schnorr, Samuel Thibault, and Vincent Danjean. Analyzing Dynamic Task-Based Applications on Hybrid Platforms: An Agile Scripting Approach. In *3rd Workshop on Visual Performance Analysis (VPA)*, Salt Lake City, United States, November 2016. Held in conjunction with SC16.

[27] Xavier Lacoste, Mathieu Faverge, Pierre Ramet, Samuel Thibault, and George Bosilca. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *HCW'2014 workshop of IPDPS*, Phoenix, États-Unis, May 2014. IEEE. RR-8446 RR-8446.

[28] Corentin Rossignon, Pascal Hénon, Olivier Aumage, and Samuel Thibault. A NUMA-aware fine grain parallelization framework for multi-core architecture. In *PDSEC - 14th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing - 2013*, Boston, États-Unis, May 2013.

[29] Marc Sergent, David Goudin, Samuel Thibault, and Olivier Aumage. Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System. In *21st International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Chicago, United States, May 2016.

[30] Samuel Thibault, François Broquedis, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. An Efficient OpenMP Runtime System for Hierarchical Architectures. In Barbara M. Chapman, Weimin Zheng, Guang R. Gao, Mitsuhisa Sato, Eduard Ayguadé, and Dongsheng Wang, editors, *A Practical Programming Model for the Multi-Core Era, 3rd International Workshop on OpenMP, IWOMP 2007, Beijing, China, June 3-7, 2007, Proceedings*, volume 4935 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2008.

[31] Philippe Virouleau, Pierrick BRUNET, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *10th International Workshop on OpenMP, IWOMP2014*, 10th International Workshop on OpenMP, IWOMP2014, pages 16 – 29, Salvador, Brazil, France, September 2014. Springer.

National journals

[32] Sidi Ahmed Mahmoudi, Pierre Manneback, Cédric Augonnet, and Samuel Thibault. Traitements d'images sur architectures parallèles et hétérogènes. *Technique et Science Informatiques*, 2012.

National conferences

[33] Paul-Antoine Arras, Didier Fuin, Emmanuel Jeannot, Arthur Stoutchinin, and Samuel Thibault. Ordonnancement de liste dans les systèmes embarqués sous contrainte de mémoire. In *21èmes Rencontres Francophones du Parallélisme (RenPar'21)*, Grenoble, France, January 2013. Inria Grenoble.

[34] Sidi Ahmed Mahmoudi, Pierre Manneback, Cédric Augonnet, and Samuel Thibault. Détection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicoeurs hétérogènes. In *20èmes Rencontres Francophones du Parallélisme (RenPar'20)*, Saint-Malo / France, May 2011.

[35] Vinicius Garcia Pinto, Lucas Mello Schnorr, Arnaud Legrand, Samuel Thibault, Luka Stanisic, and Vincent Danjean. Detecção de Anomalias de Desempenho em Aplicações de Alto Desempenho baseadas em Tarefas em Clusters Híbridos. In *17o Workshop em Desempenho de Sistemas Computacionais e de Comunicação (WPerformance)*, Natal, Brazil, July 2018.

Research reports

[36] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Luka Stanisic, and Samuel Thibault. Modeling Irregular Kernels of Task-based codes: Illustration with the Fast Multipole Method. Research Report RR-9036, Inria Bordeaux - Sud-Ouest, February 2017.

[37] Emmanuel Agullo, Alfredo Buttari, Mikko Byckling, Abdou Guermouche, and Ian Masliah. Achieving high-performance with a sparse direct solver on Intel KNL. Research

Report RR-9035, Inria Bordeaux Sud-Ouest ; CNRS-IRIT ; Intel corporation ; Université Bordeaux, February 2017.

[38] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Samuel Thibault, and Raymond Namyst. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. Research Report RR-8538, INRIA Bordeaux - Sud-Ouest, May 2014.

[39] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Technical Report 7240, Inria Bordeaux - Sud-Ouest, March 2010.

[40] Xavier Lacoste, Mathieu Faverge, Pierre Ramet, Samuel Thibault, and George Bosilca. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. Technical Report RR-8446, INRIA Bordeaux - Sud-Ouest, January 2014.

Posters

[41] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In Siegfried Benkner Jesper Larsson Träff and Jack Dongarra, editors, *EuroMPI 2012*, volume 7490 of *LNCS*. Springer, September 2012. Poster Session.

List of PhD theses I have co-advised

[42] Paul-Antoine Arras. *Scheduling of dynamic streaming applications on hybrid embedded MPSoCs comprising programmable computing units and hardware accelerators*. PhD thesis, Université de Bordeaux, February 2015.

[43] Cédric Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective*. PhD thesis, Université Bordeaux 1, December 2011.

[44] Suraj Kumar. *Scheduling of Dense Linear Algebra Kernels on Heterogeneous Resources*. PhD thesis, Université de Bordeaux, April 2017.

[45] Corentin Rossignon. *A fine grain model programming for parallelization of sparse linear solver*. PhD thesis, Université de Bordeaux, July 2015.

[46] Marc Sergent. *Scalability of a task-based runtime system for dense linear algebra applications*. PhD thesis, Université de Bordeaux, December 2016.

List of other publications around StarPU

[AAB+16] Emmanuel Agullo, Olivier Aumage, Berenger Bramas, Olivier Coulaud, and Samuel Pitoiset. Bridging the gap between OpenMP 4.0 and native runtime systems for the fast multipole method. Research Report RR-8953, Inria Bordeaux - Sud-Ouest, March 2016.

[AAB+17] Emmanuel Agullo, Olivier Aumage, Berenger Bramas, Olivier Coulaud, and Samuel Pitoiset. Bridging the gap between OpenMP and task-based runtime systems for the fast multipole method. *IEEE Transactions on Parallel and Distributed Systems*, April 2017.

[AAD+11] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU factorization for accelerator-based systems. In *9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11)*, Sharm El-Sheikh, Egypt, June 2011.

[ABC+14a] Emmanuel Agullo, Berenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based FMM for heterogeneous architectures. Research Report RR-8513, Inria Bordeaux - Sud-Ouest, April 2014.

[ABC+14b] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-Based FMM for Multicore Architectures. *SIAM Journal on Scientific Computing*, 36(1):66–93, 2014.

[ABC+16] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Martin Khannouz, and Luka Stanisic. Task-based fast multipole method for clusters of multicore processors. Research Report RR-8970, Inria Bordeaux Sud-Ouest, October 2016.

[ABEDK16] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. Are Static Schedules so Bad ? A Case Study on Cholesky Factorization. In *Proceedings of the 30th IEEE International Parallel & Distributed Processing Symposium, IPDPS'16*, Chicago, IL, United States, May 2016. IEEE.

[AGG+16] E Agullo, L Giraud, A Guermouche, S Nakov, and Jean Roman. Task-based Conjugate Gradient: from multi-GPU towards heterogeneous architectures. Research Report 8912, Inria Bordeaux Sud-Ouest, May 2016.

[AN08] Cédric Augonnet and Raymond Namyst. A unified runtime system for heterogeneous multicore architectures. In *Proceedings of the International Euro-Par Workshops 2008, HPPC'08*, volume 5415 of *Lecture Notes in Computer Science*, pages 174–183, Las Palmas de Gran Canaria, Spain, August 2008. Springer.

[Aug08] Cédric Augonnet. Vers des supports d'exécution capables d'exploiter les machines multicoeurs hétérogènes. Mémoire de DEA, Université Bordeaux 1, June 2008.

[Aug09] Cédric Augonnet. StarPU: un support exécutif unifié pour les architectures multicœurs hétérogènes. In *19èmes Rencontres Francophones du Parallélisme (RenPar'19)*, Toulouse / France, September 2009. Best Paper Award.

[BCED+16] Olivier Beaumont, Terry Cojean, Lionel Eyraud-Dubois, Abdou Guermouche, and Suraj Kumar. Scheduling of Linear Algebra Kernels on Multiple Heterogeneous Resources. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*, Hyderabad, India, December 2016.

[BEDK17] O. Beaumont, L. Eyraud-Dubois, and S. Kumar. Approximation proofs of a fast and efficient list scheduling algorithm for task-based runtime systems on multicores and gpus. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 768–777, May 2017.

[Bor13] Cyril Bordage. *Ordonnancement dynamique, adapté aux architectures hétérogènes, de la méthode multipôle pour les équations de Maxwell, en électromagnétisme*. PhD thesis, Université Bordeaux 1, December 2013.

[BPT+11] Siegfried Benkner, Sabri Pllana, Jesper Larsson Träff, Philippas Tsigas, Uwe Dolinsky, Cédric Augonnet, Beverly Bachmayer, Christoph Kessler, David Moloney, and Vitaly Osipov. PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems. *IEEE Micro*, 31(5):28–41, September 2011.

[CCRB17] Jean Marie Couteyen Carpaye, Jean Roman, and Pierre Brenner. Design and Analysis of a Task-based Parallelization over a Runtime System of an Explicit Finite-Volume CFD Code with Adaptive Time Stepping. *International Journal of Computational Science and Engineering*, pages 1 – 22, 2017.

[CGH+16] Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, and Pierre-André Wacrenier. Resource aggregation for task-based Cholesky Factorization on top of heterogeneous machines. In *HeteroPar'2016 workshop of Euro-Par*, Grenoble, France, August 2016.

[Che17] Arthur Chevalier. Critical resources management and scheduling under StarPU. Master's thesis, Université de Bordeaux, September 2017.

[Coj18] Terry Cojean. *Programmation of heterogeneous architectures using moldable tasks*. PhD thesis, Université de Bordeaux, March 2018.

[Cou13] Ludovic Courtès. C Language Extensions for Hybrid CPU/GPU Programming with StarPU. Research Report RR-8278, Inria Bordeaux - Sud-Ouest, April 2013.

[HDB12] Sylvain Henry, Alexandre Denis, and Denis Barthou. Programmation unifiée multi-accélérateur OpenCL. *Techniques et Sciences Informatiques*, (8-9-10):1233–1249, 2012.

[HDB+14] Sylvain Henry, Alexandre Denis, Denis Barthou, Marie-Christine Counilh, and Raymond Namyst. Toward OpenCL Automatic Multi-Device Support. In Fernando Silva, Ines Dutra, and Vitor Santos Costa, editors, *Euro-Par 2014*, Porto, Portugal, August 2014. Springer.

[Hen11] Sylvain Henry. Programmation multi-accélérateurs unifiée en OpenCL. In *20èmes Rencontres Francophones du Parallélisme (RenPar'20)*, Saint Malo, France, May 2011.

[Hen13a] Sylvain Henry. *Modèles de programmation et supports exécutifs pour architectures hétérogènes*. PhD thesis, Université Bordeaux 1, November 2013.

[Hen13b] Sylvain Henry. ViperVM: a Runtime System for Parallel Functional High-Performance Computing on Heterogeneous Architectures. In *2nd Workshop on Functional High-Performance Computing (FHPC'13)*, Boston, États-Unis, September 2013.

[HGNW13] Andra Hugo, Abdou Guermouche, Raymond Namyst, and Pierre-André Wacrenier. Composing multiple StarPU applications over heterogeneous machines: a supervised approach. In *Third International Workshop on Accelerators and Hybrid Exascale Systems*, Boston, USA, May 2013.

[Hug11] Andra Hugo. Composabilité de codes parallèles sur architectures hétérogènes. Mémoire de master, Université Bordeaux 1, June 2011.

[Hug13] Andra Hugo. Le problème de la composition parallèle : une approche supervisée. In *21èmes Rencontres Francophones du Parallélisme (RenPar'21)*, Grenoble, France, January 2013.

[Hug14] Andra-Ecaterina Hugo. *Composability of parallel codes on heterogeneous architectures*. PhD thesis, Université de Bordeaux, December 2014.

[JBSH16] Johan Janzén, David Black-Schaffer, and Andra Hugo. Partitioning GPUs for Improved Scalability. In *IEEE 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, October 2016.

[SA14] Marc Sergent and Simon Archipoff. Modulariser les ordonnanceurs de tâches : une approche structurelle. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS'2014)*, Neuchâtel, Suisse, April 2014.

[SAB+15] Luka Stanisic, Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Arnaud Legrand, Florent Lopez, and Brice Videau. Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers. In *The 21st IEEE International Conference on Parallel and Distributed Systems*, Melbourne, Australia, December 2015.

-=-=-=-