7th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2014)
Amsterdam, Netherlands
July 3-4, 2014

Automatic Task-based Code Generation for High Performance Domain Specific Embedded Language

Antoine Tran Tan · Joel Falcou · Daniel Etiemble · Hartmut Kaiser

Abstract Providing high level tools for parallel programming while sustaining a high level of performance has been a challenge that techniques like **Domain Specific Embedded Languages** try to solve. In previous works, we investigated the design of such a *DSEL* – NT^2 – providing a MATLAB -like syntax for parallel numerical computations inside a C++ library. In this paper, we show how NT^2 has been redesigned for shared memory systems in an extensible and portable way. The new NT^2 design relies on a tiered **Parallel Skeleton** system built using asynchronous task management and automatic compile-time *taskification* of user level code. We describe how this system can operate various shared memory runtimes and evaluate the design by using several benchmarks implementing linear algebra algorithms.

Keywords C++, parallel skeletons, asynchronous programming, generative programming

1 Introduction

As parallel architecture complexity increases, writing scientific applications with low-level languages (for example Fortran or C) becomes more difficult. It encompasses the difficulty for domain experts to read computer programs and the difficulty for software developers to maintain them efficiently. For sequential applications, the efficiency of *Domain Specific Languages* is well established: high-level semantics of the application domain is preserved and portability is given for any kind of computing systems. Despite their results in productivity and time savings, *DSL*s may involve significant costs including language learning and implementation costs. To limit these constraints, *Domain Specific Embedded Languages* [21,32] have been proposed.

LRI, Université Paris-Sud XI, INRIA - Orsay, France · CCT, Louisiana State University - Baton Rouge, USA
E-mail: antoine.trantan@lri.fr joel.falcou@lri.fr de@lri.fr hkaiser@cct.lsu.edu

*DSEL*s are languages nested inside a host language; they are compiled or interpreted according to the host language ecosystem.

In [19], a C++ library that uses such a solution has been proposed. This library called NT^2 – the Numerical Template Toolbox – combines template meta-programming techniques [1] and generative programming [14] inside a Matlab inspired DSEL. But while NT^2 provided a simple shared memory system and proper exploitation of SIMD instruction sets, problems remained. First, the extensibility of NT^2 was limited by the hard-coding of each architectural component. Secondly, its shared memory system only manages thread-based loop parallelization. As the execution of an arbitrary algorithm containing multiple independent matrix operations can be significantly improved if these operations were performed at the same time, using a task-based runtime may improve performance over a trivial data parallel implementation of such a library.

In this work, we describe how NT^2 was redesigned to take into account task based parallelism in shared memory environments and how this shared memory support has been upgraded to use a tiered *parallel skeletons* system for handling task parallelism in a generic way[1]. The rest of this paper is organized as follow: section 2 will quickly describe the NT^2 API and implementation techniques; section 3 will describe the parallel skeleton system that has been implemented and how it interacts with the NT^2 compile-time expression system. Section 4 describes how those skeletons can support asynchronous taskification. Finally, section 5 evaluates performance on some benchmarks and we discuss related works, our results and future perspectives in section sections 6 and 7.

2 The NT^2 library

NT^2 – the Numerical Template Toolbox – is a C++ library designed to help non-specialists from various fields to develop high performance applications [18]. To provide a general purpose API, NT^2 implements a sub set of the Matlab language as a DSEL within C++ while providing a high level of performance. The main data structure in NT^2 is the template class `table` representing the equivalent of a matrix in Matlab . Specificities like one-based indexing, dimensions handling and reshaping operations are preserved.

[1] as defined by Czarnecki et al. [14]

Listing 1 presents an example code that calculates the root mean square deviation between two arrays.

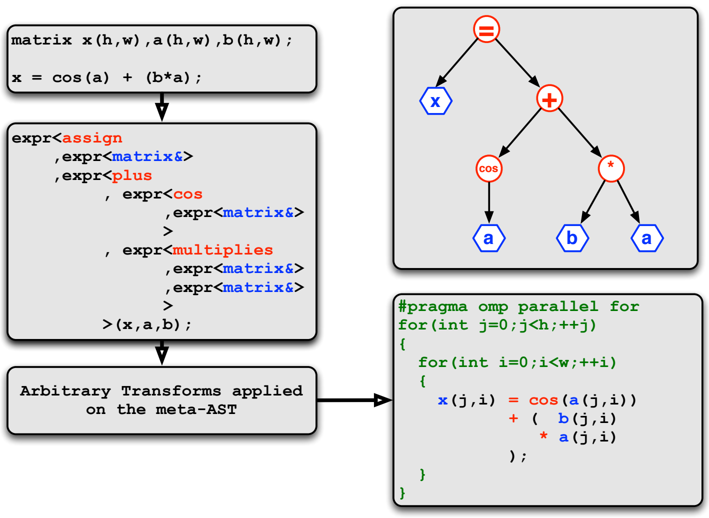
Listing 1: NT^2 RMSD Computation

```
// Matlab: A1 = 1:1000;
table<double> A1 = _(1.,1000.)

// Matlab: A2 = A1 + randn( size(A1) );
table<double> A2 = A1 + randn(size(A1));

// Matlab: rms = sqrt( sum((A1(:) - A2(:)).^2) / numel(A1) );
double rms = sqrt( sum(sqr(A1(_) - A2(_))) / numel(A1) );
```

In this sample code, note the first-class semantics of the table object on which global operations can be applied, the use of _ as a replacement of Matlab colon (:) operator and the availability of utility functions like `numel` and `size` behaving as their Matlab counterpart. More than 80% of core Matlab functionalities are provided along with an additional C++ based interface for compatibility with standard algorithms and major Boost libraries [15]. NT^2 relies on three kinds of optimizations: instruction level parallelism via the implicit usage of SIMD extensions with Boost.SIMD [17], thread-level parallelism using OpenMP or TBB and *Expression Templates*.

Expression Templates [34,33] is a well known technique implementing a form of delayed evaluation in C++ [29]. This idiom allows the construction at compile-time of a C++ type representing the abstract syntax tree associated with an arbitrary statement. This is done by overloading functions and operators according to those types so they return a lightweight object which type represents the current operation in the Abstract Syntax Tree (AST) being built instead of performing any kind of computation. Once reconstructed, functions can be used to transform this AST into arbitrary code fragments (see Figure 1).

While Expression Templates should not be limited to the sole purpose of removing temporaries and memory allocations from C++ code, few projects actually go further. The complexity of the boilerplate code is usually as big as the actual library code, making such tools hard to maintain and extend. To reduce those difficulties, Niebler proposed a C++ compiler construction toolkit for embedded languages called `Boost.Proto` [25]. It allows developers to specify grammars and semantic actions for DSELs and provides a semi-automatic generation of all the template structures needed to perform the AST capture. Compared to hand-written Expressions Templates-based DSELs, designing a new embedded language with `Boost.Proto` is done at a higher level of abstraction by designing and applying *transforms* (functions operating on the DSEL statements using *pattern matching*). NT^2 uses `Boost.Proto` as its expression template engine and replaces the classical direct walk-through of the compile-time AST by the execution of a mixed compile-time/runtime algorithm over a `Boost.Proto` standardized AST structure.



```
matrix x(h,w),a(h,w),b(h,w);

x = cos(a) + (b*a);
```

```
expr<assign
    ,expr<matrix&>
    ,expr<plus
         , expr<cos
                ,expr<matrix&>
                >
         , expr<multiplies
                ,expr<matrix&>
                ,expr<matrix&>
                >
         >(x,a,b);
```

```
Arbitrary Transforms applied
     on the meta-AST
```

```
#pragma omp parallel for
for(int j=0;j<h;++j)
{
  for(int i=0;i<w;++i)
  {
    x(j,i) = cos(a(j,i))
           + (  b(j,i)
              * a(j,i)
             );
  }
}
```

Fig. 1: General principles of *Expression Templates* – C++ *operator overloading allows us to reconstruct a recursive type encoding the original expression AST. This AST is then restructured at compile time.*

3 Parallel Skeletons in NT^2

An *Algorithmic Skeleton* (or *Parallel Skeleton*) [13] is defined as a recurrent design pattern associated with parallel programming. They usually behave as higher order functions, *i.e.* functions parametrized by other functions, including other skeletons. This composability reduces the difficulty of designing complex parallel programs as any combination of skeletons is viable by design. The other main advantage of skeletons is the fact that the actual synchronization and scheduling of a skeleton's parallel task is encapsulated within the skeleton. Once a skeleton semantics is defined, programmers do not have to specify how synchronizations and scheduling happen. This has two implications: first, skeletons can be specified in an abstract manner and encapsulate architecture specific implementation; second, the communications/computations patterns are known in advance and can be optimized [2, 16].

Even if a large number of skeletons has been proposed in the litterature [24, 12], NT^2 focuses on three data-oriented skeletons:

- **transform** that applies an arbitrary operation to each (or certain) element(s) of an input `table` and stores the result in an output `table`.
- **fold** that applies a partial reduction of the elements of an input `table` to a given table dimension and stores the result in an output `table`.
- **scan** that applies a prefix scan of the elements of an input `table` to a given table dimension and stores the result in an output `table`.

Those skeletons are tied to families of loop-nest that can or can not be nested. Those families are :

- **elementwise loop nests** that represent loop nests implementable via a call to **transform** and which can only be nested with other elementwise operations.
- **reduction loop nests** that represent loop nests implementable via a call to **fold**. Successive reductions are not generally nestable as they can operate on different dimensions but can contain a nested elementwise loop nest.
- **prefix loop nests** that represent loop nests implementable via a call to **scan**. Successive prefix scans, like reductions, are not nestable but can contain nested elementwise loop nests.

Those families of loop nests are used to tag functions provided by NT^2 so that the type of the operation itself can be introspected to determine its loop nest family. As the AST of an arbitrary expression containing at least one NT^2 custom terminal (mainly `table` or `_`) is being built at compile-time, the AST construction function has to take care of separating expressions requiring non-nestable loop nests by fetching the loop nest family associated with the current top-most AST node. As an example, figure 2 shows how the expression `A = B / sum(C+D)` is built and split into sub-ASTs handled by a single type of skeleton.

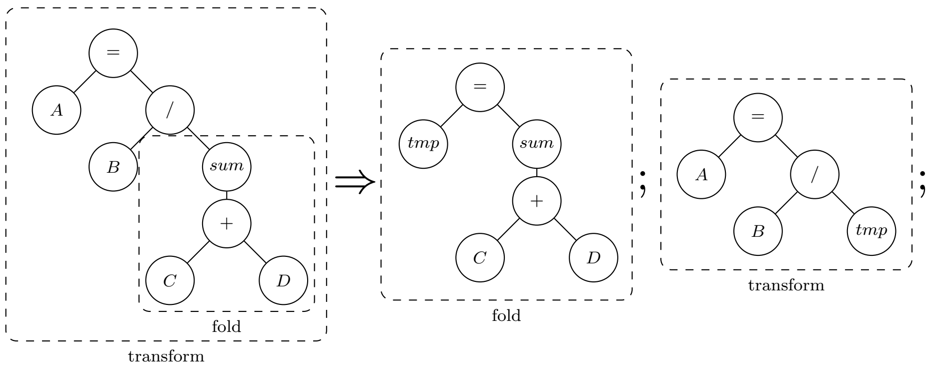


Fig. 2: Parallel Skeletons extraction process – *Nesting of different kinds of skeletons into a single statement is automatically unwrapped at compile time as a sequence of single skeleton statements.*

The split ASTs are logically chained by the extra temporary variable inserted in the right-hand side of the first AST and as the left-hand size of the second. The life-cycle management of this temporary is handled by a C++ shared pointer and ensures that the data computed when crossing AST barrier lives long enough. Notice that, as the `C+D` AST is an elementwise operation, it stays nested inside the `sum` node. NT^2 then uses the nestability of parallel skeletons to call the SIMD and/or scalar version of each skeleton involved in a serie of statements to recursively and hierarchically exploit the target hardware. At the end of the compilation, each NT^2 expression has been turned into the proper serie of nested loop nests using combinations of OpenMP, SIMD and scalar code.

4 Automatic taskification

As sketched above, once a DSEL statement has been issued either by the user or as a list of temporary statements generated by an AST split, they are executed following the simple fork-join model enforced by OpenMP and TBB. As the number of statements grows, the cost of synchronization, temporary allocation and cache misses due to poor locality handling lower the performance.

Our proposal is to use the automatic AST splitting system to derive a dependency graph between those statements and turn this graph into a runtime managed list of coarse grain tasks. To do so, we need to investigate which kind of runtime support fits those requirements and how to integrate this runtime in the current NT^2 skeleton based code.

4.1 Task-based runtime for shared memory systems

In order to exploit inter-statement parallelism, NT^2 requires a runtime that allows a proper level of performance, supports nestability and limits the cost of synchronization. *Tasking* [4] or *asynchronous programming* is a such a model. Available in several projects relating to task runtimes such as TBB [28], OmpSs [4], HPX [22], Quark [36] or OpenMP (3.0 and 4.0 specifications) [27], this model is able to generate and process an arbitrary task graph on various architectures while minimizing synchronization.

The second point is the nestability. To keep the NT^2 skeleton high level model, we need to use an implementation of tasking supporting such composable calls. Traditionally, low-level thread-based parallelism often suffers from a lack of composability as it relies on procedural calls that only work with a global view of the program. Another interface for such a tasking model is the *Future* programming model [20, 5] that has been integrated by the 2011 C++ Standard [30]. A *Future* is an object holding a value which may become available at some point in the future. This value is usually the result of some other computation but is usually created without waiting for the completion of the computation. Futures allow for composable parallel programs as they can be passed around parallel function calls as simple value semantic objects and have their actual contents be requested in a non-blocking or blocking way depending on context.

4.2 HPX - A Parallel Runtime System for Applications of any Scale

HPX [22] is a general purpose C++ runtime system for parallel and distributed applications of any scale. For the purpose of the described work, HPX was integrated as a backend for NT^2, providing the task-based runtime used for the presented results. HPX represents an innovative mixture of a global system-wide address space, fine grain parallelism, and lightweight synchronization combined with implicit, work queue based, message driven computation, full semantic equivalence of local and remote execution, and explicit support for hardware accelerators through percolation.

The design of the API exposed by HPX is aligned as much as possible with the latest C++11 Standard [30], the (draft) C++14 Standard [31], and related proposals to the standardization committee [26,35,11]. HPX implements all interfaces defined by the C++ Standard related to multi-threading (such as `future`, `thread`, `mutex`, or `async`) in a fully conforming way on top of its own user-level threading system. These interfaces were accepted for ISO standardization after a wide community based discussion and since then have proven to be effective tools for managing asynchrony. HPX seeks to extend these concepts, interfaces, and ideas embodied in the C++11 threading package to distributed and data-flow programming use cases. Nevertheless, HPX makes every possible effort to keep all of the implementation fully conforming to C++, which ensures a high degree of code and performance portability.

The AST generated during the automatic taskification step explicitly describes the data dependencies of the original expression. We use the asynchronous threading API of HPX to execute all tasks in proper sequence as defined by the AST. Each of the tasks is launched as a separate (lightweight) HPX thread using `hpx::async` generating a `hpx::future` which represents the expected result of each of the tasks. HPX additionally exposes facilities allowing to compose Futures sequentially and in parallel.

- **Sequential composition** is achieved by calling a Future's member function f`.then(`g`)` which attaches a given function g to the Future object f. Here, this member function returns a new Future object representing the result of the attached continuation function g. The function will be (asynchronously) invoked whenever the Future f becomes ready. Sequential composition is the main mechanism for sequentially executing several tasks, where this sequence of tasks can still run in parallel with any other task.
- **Parallel composition** is implemented using the utility function `when_all(` $f1$`,` $f2$`, ...)` which returns yet another Future object. The returned Future object becomes ready whenever all argument Future objects $f1$, $f2$, etc. have become ready. Parallel composition is the main building block for fork-join style task execution, where several tasks are executed in parallel but all of them must finish running before other tasks could be scheduled.

We use these composition facilities to create task dependencies which mirror the data dependencies described by the generated AST. Here, the Future objects represent the terminal nodes and their combination represents the edges and the intermediate nodes of the AST.

HPX' lightweight threading system imposes relative low overhead and allows to create and schedule a large number of tasks (up to several million concurrent tasks). This efficiency combined with the semantics of Futures which allow to directly express the generated AST as an execution tree generated at runtime, provides a solid base for a highly efficient auto-parallelization.

4.3 Integration in NT^2

The NT^2 integration is done by:

- **Making a generic implementation of Futures**. Although NT^2 uses HPX as a prime backend for task parallelism, most systems tend to use runtimes like OpenMP or TBB. Thus we implement a Future class template that acts as a generic template wrapper which maps the current runtime choice to its proper task implementation and related functions.
- **Adapting current skeletons for taskification**. NT^2 skeletons have been modified so their internal implementation rely on Futures and asynchronous calls. To do so, NT^2 skeletons now use a task-oriented implementation by using a *worker/spawner* model. The `worker` is a function object containing a function call operator that takes a range as parameter and processes it with all the possible optimizations. The `spawner` is a function template which acts as the parallel skeleton: it invokes multiple workers by binding them appropriately to tasks depending on the kind of skeleton required in a given statement.
- **Adding task management to NT^2**. Last part of this implementation lies in the process of chaining the asynchronous tasks generated by the various skeletons spawned from a given set of ASTs. This is done by implementing a Future-based **pipeline** skeleton that explicits the different dependencies required for the evaluation of expressions. Pipelines are then created between temporary ASTs and between sub-slices of pre-existing arrays.

Figure 3 shows the final task layout of the simple `A = B / sum(C+D)` expression. The expression is parallelized using both the *worker/spawner* model to take advantage of the data-oriented parallelism inside the array evaluation, and *pipelines* between the two sub-AST generated by the split skeleton.

Instruction Level Parallelism is maintained by using SIMD-optimized workers when it is possible, thus delivering proper performance from the data-parallel layer.

Optimization across statement boundaries is the main benefit of our task generation system. They enable us to preserve data locality, thus ensuring optimal memory accesses. Such optimizations are often difficult to perform with classical Expression Templates as they can only statically access the statement's structure.

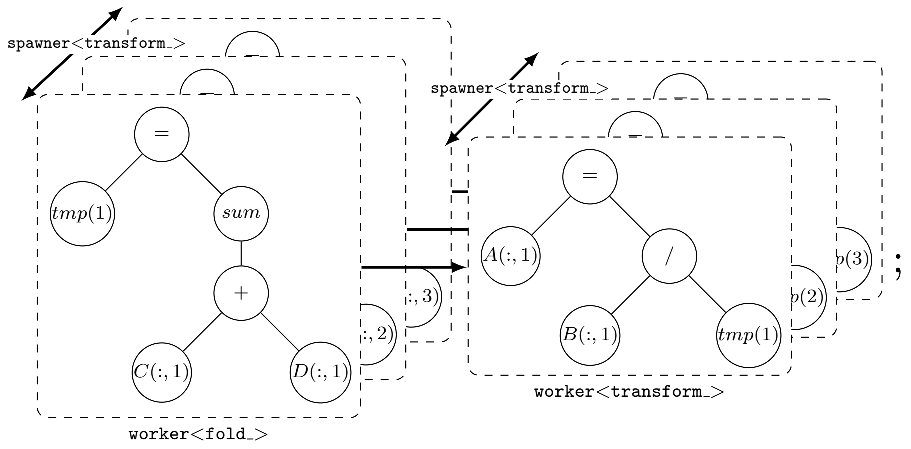


Fig. 3: Taskification of an AST – *Previous compile decomposition into multiple statement is augmented with the insertion of an asynchronous pipeline between the auto-generated statements.*

5 Performance results

This section presents two benchmarks to give an idea of the performance of NT^2. Those benchmarks were run over multiple executions (around 50 for each `table` size) from which the median execution time has been kept as the end result. Those tests were run on different machines:

- **Mini-Titan** composed of 2 sockets of Intel Core Westmere processors with 6 cores, 2 x 24GB of RAM and a 12MB L3 Cache. Code is compiled using g++-4.7 and SSE4.2 instructions
- **Lyra** composed of 8 sockets of AMD Istanbul processors with 6 cores, 128 GB of RAM and a 5MB L3 cache. Code is compiled using g++-4.7 and SSE4a instructions

5.1 Inter-statement Optimization

The Black and Scholes algorithm [6] represents a mathematical model able to give a theoretical estimate of the price of European-style options. The code is defined in listing 2. The Black & Scholes algorithm involves multiple high latency and high register count operations. The SIMD version of `log`, `exp` and `normcdf` uses polynoms and precision refinement steps that consume a large amount of registers. This implementation also uses multiple statements in which locality is important.

Listing 2: Black & Scholes NT^2 implementation

```
table<float> blackscholes ( table<float> const& S, table<float> const& X
                          , table<float> const& T, table<float> const& r
                          , table<float> const& v
                          )
{
  table<float> d  = sqrt(T);
  table<float> d1 = log(S/X)+(fma(sqr(v),0.5f,r)*T)/(v*d);
  table<float> d2 = fma(-v,d,d1));

  return S*normcdf(d1) - X*exp(-r*T)*normcdf(d2);
}
```

Since the Black & Scholes algorithm is a sequence of **transforms** encapsulating elementary operations, a minimum `table` size is required to get some efficiency. This benchmark is then evaluated using single-precision floating-point `tables` with out-of-cache sizes ranging from 64M to 1024M. As the performance by element changes slowly with the problem size (in the order of one cycle per element), we used cycles/element as measurement unit and kept the medium value over samples corresponding to one implementation.

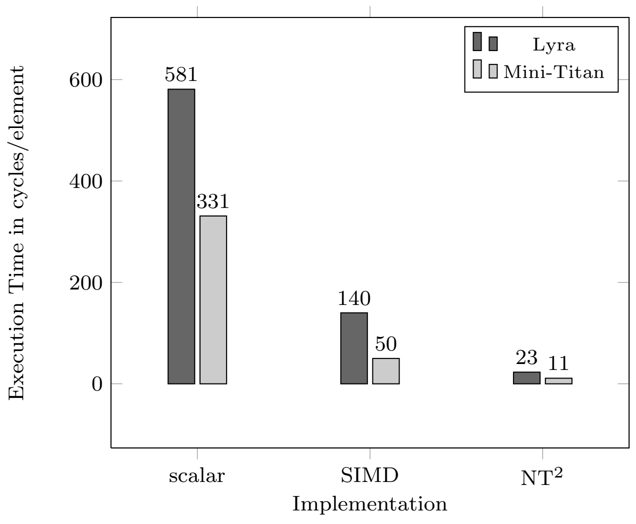


Fig. 4: Black & Scholes execution time comparaison

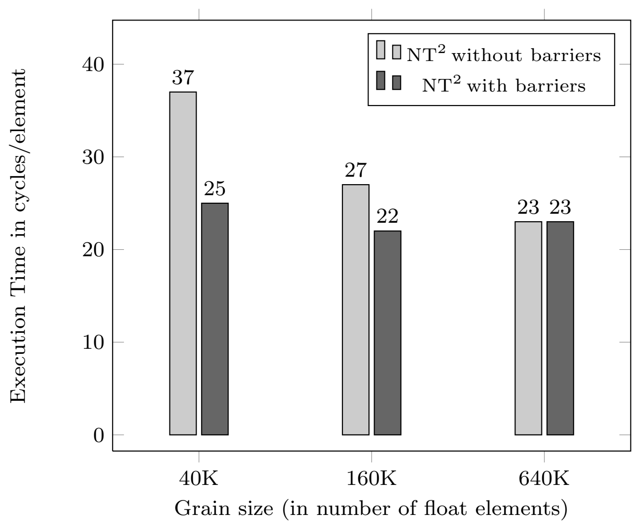


Fig. 5: Black & Scholes execution time in function of the grainsize - Lyra

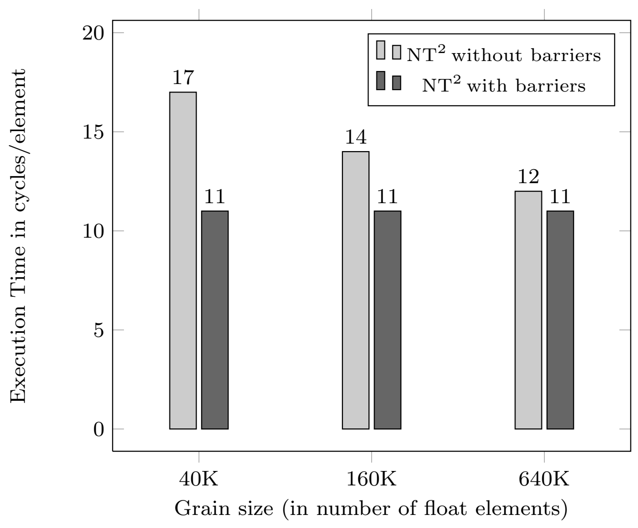


Fig. 6: Black & Scholes execution time in function of the grainsize - Mini-Titan

Figure 4 shows that the NT^2 implementation is better than the SIMD version with performance factor gains of **4** in **Mini-Titan** and **6** in **Lyra**. As we used all the processing units, the theoretical performance factor gain should be respectively 12 and 48 when ignoring the communication and synchronization latencies. Thus we can outline a real lack of scalability due to the implicit inter-statement barriers.

We then integrate the pipeline optimization and compare this NT^2 version with the one that keeps the barriers. Figures 6 and 5 show that the version with barriers is still better than the new version but its scalability doesn't progress with the grain size parameter. The pipelined version is not optimal for small grain sizes but progresses relatively well when this parameter increases. Since HPX uses the *First Come First Served* rule as a thread scheduling policy, the data locality is not preserved when walking through a pipeline of **transforms**.

Thus, the one-dependency continuation (`.then()` method) is optimized by integrating the following conditions:

- The Future is *not ready* - a callback is attached to the Future, so that the thread solving the Future executes its corresponding continuation.
- The Future is *ready* - the thread instantiating the continuation executes it immediately

However some problems can still arise while building a task graph during the execution. For example, the execution of a task can be completed before its continuation is instantiated. In this situation, the thread which has executed the task will choose a new one from the work queue. The continuation of the first task will then be executed by the wrong thread losing some memory locality. To keep this locality, the computational complexity of a task must be sufficient (ex: matrix product) to ensure an optimal scheduling.

5.2 Task-oriented Skeletons

We assess the efficiency of NT^2 Future-based task management by implementing the *tiled* version of LU factorization. Inspired from PLASMA [9], this algorithm adapts the regular LU factorization to fit in multicore architectures. It enables fine granularity by decomposing the whole calculation into dependent units of work that operate on square portions of a matrix. This leads naturally to a *directed acyclic graph* (DAG) representation that is easily translatable into a Future implementation.

As a first step, we perform the benchmark on Mini-Titan. Figures 7, 8 and 9 show that the regular version of LU (Intel MKL) is better than the tiled version (Plasma and NT^2). The key reason is that Mini-Titan is composed of only 2 NUMA domains and thus offers a perfect environment where the effect of synchronization points can be considered as negligible.

As a second step, we perform the benchmark on Lyra. Figures 10, 11 and 12 show that the tiled version of LU is now better than the regular version. Since the benchmark is performed in a machine composed of multiple 6 core NUMA domains, the performance of the MKL version drops around 20 cores because of memory transactions involving more than 3 NUMA domains (18 cores). By conception, the tiled version uses asynchronism to hide these transactions and then scales better. In those two cases, the scalability of the NT^2 implementation is very close to the PLASMA version regardless of the size of the problem.

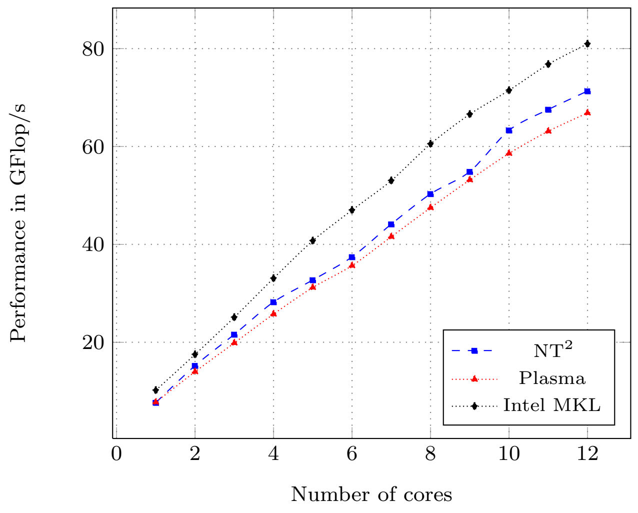


Fig. 7: LU performance for Problem Size 4000 - Mini-Titan

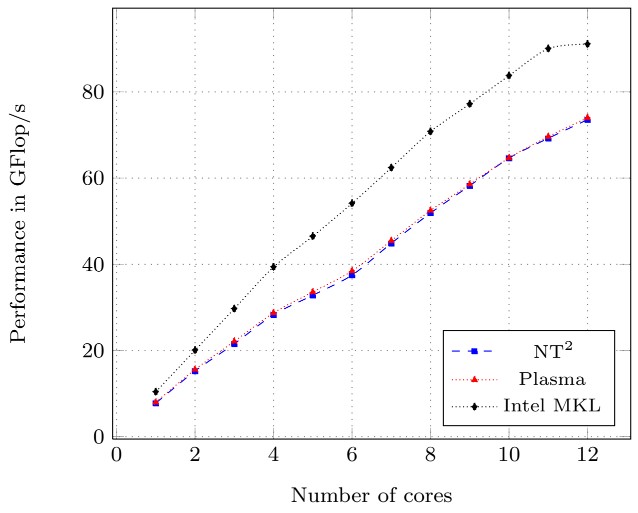


Fig. 8: LU performance for Problem Size 8000 - Mini-Titan

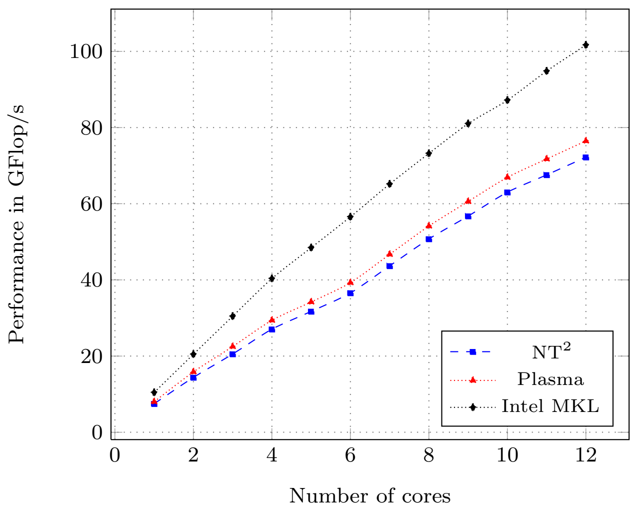


Fig. 9: LU performance for Problem Size 12000 - Mini-Titan

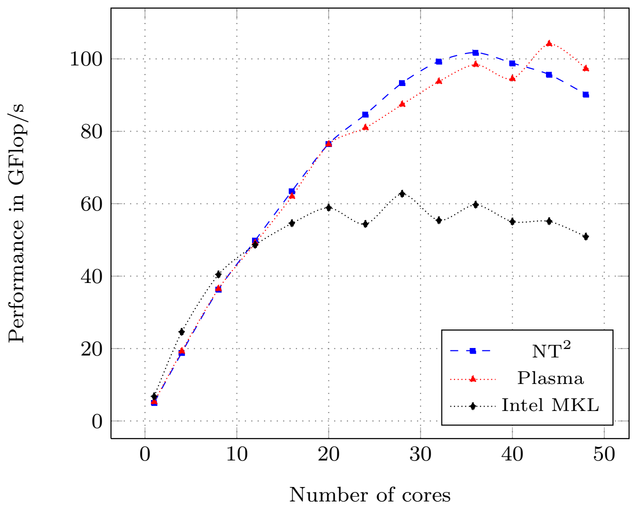


Fig. 10: LU performance for Problem Size 4000 - Lyra

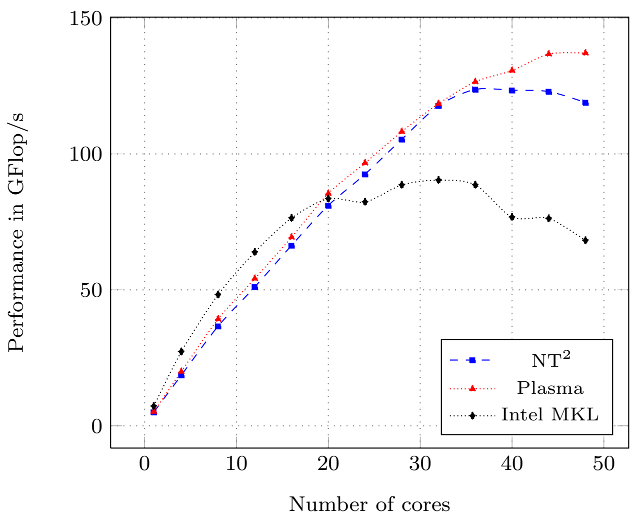


Fig. 11: LU performance for Problem Size 8000 - Lyra

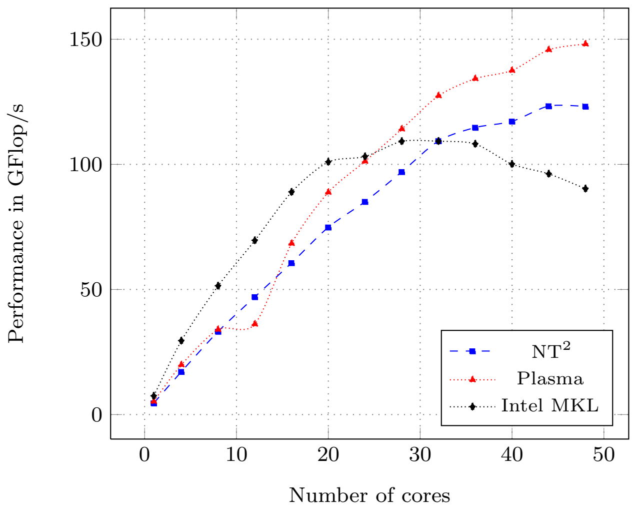


Fig. 12: LU performance for Problem Size 12000 - Lyra

6 Related Works

Various systems and libraries have been proposed to solve this kind of issues. The most notable are:

- **STAPL** [3] (Standard Template Adaptive Parallel Library) is a C++ library based on ISO Standard C++ components similar to the "sequential" ISO C++ Standard library. The library works with parallel equivalents of C++ containers (pContainers) and algorithms (pAlgorithms) that interacts through ranges (pRange). It provides support for shared and distributed memory and includes a complete runtime system, rules to easily extend the library and optimization tools. STAPL internally uses a system similar to HPX' Futures but based on a specific runtime. Also, STAPL focus on standard algorithms and containers use cases instead of providing a more high-level API.
- **Chapel** [10] is a full-fledged programming language developed by Cray to ease parallel programming. Like NT^2, Chapel uses abstractions for data-parallelism with objects named *Arrays* (equivalent of Tables), and abstractions for task-parallelism with objects named *Synchronization variables* (equivalent of Futures).
- **PaRSEC** [7] (or DAGuE) is a generic framework developed by the ICL (University of Tennessee) that can be used to extract data-flow patterns from a sequential C Code, generating a DAG representation at compile-time. PaRSEC uses a proper runtime to instantiate this DAG in the form of computation tasks.

7 Conclusion

In this paper, we proposed an implementation of an automatic taskification system which enables the extraction of asynchronism from high level DSEL statements. This system is then used to implement an efficient shared memory support inside a C++ numerical library called NT^2. Results show that the implementation of skeletons allows inter-statement optimizations and thus reduces one of the usual limitation of expression templates based systems. Current works aim to provide a larger support of shared memory runtimes by using wrappers for systems like OmpSs. On a broader scope, we also want to extent this system to distributed memory machines by keeping the *Future* based implementation and using different asynchronous runtimes for large scale systems like Charm++ [23] or STAPL [8].

References

1. D. Abrahams and A. Gurtovoy. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond.* Pearson Education, 2004.

2. M. Aldinucci, M. Danelutto, and J. Dnnweber. Optimization techniques for implementing parallel skeletons in grid environments. In S. Gorlatch, editor, *Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 35–47, Stirling, Scotland, UK, July 2004. Universitat Munster, Germany.
3. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. Stapl: An adaptive, generic parallel c++ library. In *Languages and Compilers for Parallel Computing*, pages 193–208. Springer, 2003.
4. E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009.
5. H. C. Baker Jr and C. Hewitt. The incremental garbage collection of processes. In *ACM SIGART Bulletin*, volume 12, pages 55–59. ACM, 1977.
6. F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.
7. G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, and J. Dongarra. From serial loops to parallel execution on distributed systems. In *Euro-Par 2012 Parallel Processing*, pages 246–257. Springer, 2012.
8. A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. Stapl: Standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10, New York, NY, USA, 2010. ACM.
9. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.
10. B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
11. Chris Mysen and Niklas Gustafsson and Matt Austern and Jeffrey Yasskin. N3785: Executors and schedulers, revision 3. Technical report, 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3785.pdf.
12. P. Ciechanowicz and H. Kuchen. Enhancing muesli's data parallel skeletons for multi-core computer architectures. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 108–113. IEEE, 2010.
13. M. I. Cole. *Algorithmic skeletons: structured management of parallel computation.* Pitman London, 1989.
14. K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In *Generic Programming*, pages 25–39, 1998.
15. B. Dawes, D. Abrahams, and R. Rivera. Boost C++ libraries. *URL http://www. boost. org*, 35:36, 2009.
16. K. Emoto, K. Matsuzaki, Z. Hu, and M. Takeichi. Domain-specific optimization strategy for skeleton programs. In A.-M. Kermarrec, L. Boug, and T. Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 705–714. Springer Berlin Heidelberg, 2007.
17. P. Estérie, M. Gaunard, J. Falcou, J.-T. Lapresté, and B. Rozoy. Boost. simd: generic programming for portable simdization. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 431–432. ACM, 2012.
18. J. Falcou, M. Gaunard, and J.-T. Lapresté. The numerical template toolbox, 2013. `http://www.github.com/MetaScale/nt2`.
19. J. Falcou, J. Sérot, L. Pech, and J.-T. Lapresté. Meta-programming applied to automatic smp parallelization of linear algebra code. In *Euro-Par 2008–Parallel Processing*, pages 729–738. Springer Berlin Heidelberg, 2008.
20. D. P. Friedman and D. S. Wise. *The impact of applicative programming on multiprocessing.* Indiana University, Computer Science Department, 1976.
21. P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.
22. H. Kaiser, M. Brodowicz, and T. Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 394–401. IEEE, 2009.

23. L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28-10. ACM, 1993.
24. H. Kuchen. *A skeleton library.* Springer, 2002.
25. E. Niebler. Proto : A compiler construction toolkit for DSELs. In *Proceedings of ACM SIGPLAN Symposium on Library-Centric Software Design*, 2007.
26. Niklas Gustafsson and Artur Laksberg and Herb Sutter and Sana Mithani. N3857: Improvements to std::future<T> and Related APIs. Technical report, 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3857.pdf.
27. OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013.
28. J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc., 2010.
29. D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91 – 99, 2001.
30. The C++ Standards Committee. ISO/IEC 14882:2011, Standard for Programming Language C++. Technical report, 2011. http://www.open-std.org/jtc1/sc22/wg21.
31. The C++ Standards Committee. N3797: Working Draft, Standard for Programming Language C++. Technical report, 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.
32. L. Tratt. Model transformations and tool integration. *Software & Systems Modeling*, 4(2):112–122, 2005.
33. D. Vandevoorde and N. M. Josuttis. *C++ Templates.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
34. T. Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.
35. Vicente J. Botet Escriba. N3865: More Improvements to std::future<T>. Technical report, 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3865.pdf.
36. A. Yarkhan, J. Kurzak, and J. Dongarra. Quark users guide. Technical report, Technical Report April, Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tenessee, 2011.