TaskUniVerse: A Task-Based Unified Interface for Versatile Parallel Execution

Afshin Zafari

Uppsala University,
Department of Information Technology,
Division of Scientific Computing
Lägerhyddsvägen 2, 752 37 Uppsala, Sweden
afshin.zafari@it.uu.se

Abstract. Task based parallel programming has shown competitive outcomes in many aspects of parallel programming such as efficiency, performance, productivity and scalability. Different approaches are used by different software development frameworks to provide these outcomes to the programmer, while making the underlying hardware architecture transparent to her. However, since programs are not portable between these frameworks, using one framework or the other is still a vital decision by the programmer whose concerns are expandability, adaptivity, maintainability and interoperability of the programs. In this work, we propose a unified programming interface that a programmer can use for working with different task based parallel frameworks transparently. In this approach we abstract the common concepts of task based parallel programming and provide them to the programmer in a single programming interface uniformly for all frameworks. We have tested the interface by running programs which implement matrix operations within frameworks that are optimized for shared and distributed memory architectures and accelerators, while the cooperation between frameworks is configured externally with no need to modify the programs. Further possible extensions of the interface and future potential research are also described.

Keywords: High Performance Computing, Task Based Programming, Parallel Programming, Unified Interface

1 Introduction

In the last decade there were many attempts to simplify parallel programming techniques by relieving the programmer of thinking about where and how to use the concurrency controls in a sequential program. One desired outcome is to minimize the required modification of a sequential program to enable it to run in parallel, as, e.g., in OpenMP [23] where a sequential program is annotated with compiler directives and the resulting compiled code can run in parallel on multiple threads. Along this minimal change paradigm, there are frameworks that provide parallel design patterns in object oriented programming languages

by which the parallelization efforts are made transparent to the programmer. The Intel Threading Building Blocks (Intel TBB) [18], is a C++ template library for parallel programming on multi-core processors that provides parallel constructs like algorithms, containers and tasks that the programmer can use to implement an algorithm and run it in parallel. The FastFlow and SkePU C++ template libraries [5, 16], abstract the most frequent parallel patterns of programs, such as map, reduce, pipeline and farm, as skeletons that programmers can instantiate with extended and custom operations and actions. Like for Intel TBB, all the synchronizations, parallelizations, communication and memory managements are done transparently by different platform-specific and low level back-end concrete implementations of the templates.

Task based parallel programming has experienced a great acceptance increase in the past decade due to its competitive outcomes in performance and productivity. The key to success for task based approaches is the abstract view of a program as a set of operations and data. This abstraction allows for programs to be written sequentially using tasks and data whenever an operation is to be performed on program variables/identifiers. When such a program runs, the tasks corresponding to the operations in the program are submitted to the background task-based framework run-time system where they are scheduled for parallel execution. This separation of a written program and its underlying tasks, enables the providers of the task-based frameworks to use different approaches for finding the optimal solution to the scheduling problem of running the tasks on the available computing resources.

Different techniques are provided by task based programming frameworks to the programmer for writing programs. The StarPU [6] and OmpSs [11] frameworks extend the C compiler and allow the programmer to use compiler directives to define C functions as tasks kernels and describe their data dependencies. The PaRSEC [10, 13] framework provides tools and utilities to analyze a program written in a special language that describes tasks and data dependencies and uses a source to source compiler to translate the optimal solution into a C code for compilation. The DuctTeip [29], Chunks and Tasks [24] and also StarPU [3] frameworks provide Application Programming Interfaces (API) for defining data and tasks to run in a distributed memory environment. The SuperGlue framework [26] provides a header-only C++ portable library for creating tasks and running them on multi-core processors.

These frameworks have individually shown very good results in terms of performance, scalability and productivity and have been used in a wide spectrum of scientific applications such as solving partial differential equations (PDE) [27], N-Body problems using Fast Multipole Method (FMM) [4, 17, 28], simulating stochastic discrete events [7, 8], Conjugate Gradient method [1], Finite Element Method (FEM) applications [22], chemistry applications [14], seismic applications [21] and image processing [9]. They have also shown the feasibility and benefits of using task based approaches for sparse data structures [20, 25].

There are also attempts to join pairs of the task based frameworks to combine benefits of both. The StarPU and PaRSEC frameworks joined to provide task

parallelism in clusters of heterogeneous processors [2, 20]. The DuctTeip and SuperGlue frameworks joined [29] to implement hierarchical task submission and execution in hybrid distributed and shared memory environments.

While these achievements seem promising for using task based programming models, the choice of a proper task based framework may still be risky because the impacts on legacy software could be great to adapt to future changes in the frameworks.

In this paper, we address these issues by proposing a unified task based programming TaskUniVersre (TUV) model in which any number of task based frameworks can be used to run a single application in many parallel environments. The idea of having one front-end with multiple back-ends for shared memory, heterogeneous computing and distributed memory has been explored before both at the language level, as done in the Chapel language [12], and at the library level as in HPX [19] and Generic Parallel Programming Interface (GrPPI) [15]. The main focus of this paper, however, is enabling an application to mix different available frameworks while avoiding rewriting the code in a new syntax.

This section continues with explaining the motivation for designing such a programming model. The overview, implementation and programming of the TUV model are described in Sections 2.1–2.3. Section 3 shows the performance results of executing a Cholesky factorization program in different parallel computing environments. The last section is devoted to discussion and conclusions.

1.1 Motivation

The number of applications that use task based programming approaches is increasing and more attempts to join two or more task based frameworks to exploit different advantages can be foreseen. These achievements for the task based parallel programming make it interesting for application scientists to try it on their codes to efficiently scale to thousands of processors. However, choosing frameworks for implementing solutions for a specific application domain is still a vital decision for the expert end users in that domain.

The basic factors influencing the choice of framework(s) are richness and flexibility. When needs of a scientific application span a wide spectrum of software and hardware varieties, it becomes hard or impossible to find a single framework to address them. Also the investment of developing an application even on top of a mixture of frameworks has to be secured against probable risks of future changes in underlying software and hardware. In this paper we show how these issues can be addressed by the TUV model through a unified task programming interface. In the experiments section, we show that if an application (e.g., Cholesky dense matrix factorization) expected to run hierarchically in cluster of CPU-GPU computing nodes, there is no choice of a single framework to satisfy this requirement. It is also shown that even if two frameworks support common features but use different methods, it can also be advantageous to gain the best performance of each framework while combining them together.

2 The TUV Model

2.1 Overview of the TUV Model

The TUV programming model is designed to provide an abstraction for common structures and behaviors of the task based frameworks mentioned above. This abstraction generalizes the task based frameworks as black boxes which get a set of sequential tasks and detects when they are ready to execute in parallel. Since all the frameworks use data dependencies for finding ready-to-run tasks and partition data for data locality concerns, data definitions and partitions are also included in the abstraction. We found these abstractions the most influential factors in achieving high performance in the frameworks and consider them as a boundary for the abstraction, without sacrificing any generality or degrading the frameworks performance.

To avoid losing any richness of frameworks by imposing this boundary, the abstraction can (be extended to) include interfaces for setting or getting framework specific parameters, for example, through environment variables or specific function calls (as it is in many BLAS and MPI libraries). This boundary is sufficient for demonstrating the idea and usefulness of a unified interface. A rich, thorough and high performing standard interface that covers a broad range of software and hardware requires much more amount of work (like the HiHAT[1] framework which is in its early phases of development) and is beyond the scope of this paper.

In the TUV abstract view, all the operations performed by a program on its data are replaced by tasks and special data types (e.g., handles or descriptors) representing the program data. Instead of running the operations immediately, the corresponding tasks are submitted to the frameworks' runtime where their dependencies are tracked and ready tasks are executed in parallel. Actual kernel computations of a ready task are performed through call back mechanisms which are introduced to the frameworks at task creation.

In order to have a single interface for cooperation, the TUV model requires the frameworks to implement predefined interfaces for data definition and task creation, submission, execution and completion. These interfaces unify the cooperation of TUV with any other compliant framework via conversations of *generic* data and tasks regardless of the concrete instances inside the framework. These generic data and task definitions will be mapped to internal data by every framework to extract task-data dependencies and find ready-to-run tasks. The memory management of the data contents is left to the application program and the frameworks only get access to the memory using specific attributes or member methods of the generic data objects. A central *dispatcher* in the TUV model orchestrates the flow of tasks and data between the program and frameworks by connecting the interfaces of one framework to another. The dispatcher submits tasks to the frameworks and they notify the dispatcher when the tasks are ready to execute or when the tasks are finished.

[1] https://xstackwiki.modelado.org/HiHAT_SW_Stack

The TUV model divides the software stack into three layers, as shown in Fig. 1(b), where the TUV interface in the middle decouples the application layer at the top from the task based frameworks at the bottom which in turn hide the technical details of parallel programming for the underlying hardware. In the TUV model, the taskified versions of the operations are provided by the middle layer to the application layer via ordinary subroutines while on the other side of the middle layer, the generic tasks move back and forth to the frameworks or their TUV compliant wrappers. Therefore, TUV at the middle layer translates program operations to tasks and data and distributes them properly to available task based frameworks through a generic interface.

Different task flows between the dispatcher and the task based frameworks can be *configured* in the TUV model by specifying which two frameworks interfaces are to be interconnected via the dispatcher. For example, the *task-execution* interface of one framework can be connected to the *task-creation* (submission) interface of another to enable hierarchical task management in which, when tasks get ready to execute at higher levels, they split into child tasks and are submitted to the framework at the next level of the hierarchy. This configurable task flow graph between the *program* and the *wrappers* allows a single program to run in different parallel computing environments by different task based frameworks.

2.2 Implementation of the TUV Model

The TUV model provides the necessary data structures and interfaces to the application layer for defining data and performing operations on them. These interfaces also enable *partitioning* the data into parts and *splitting* an operation into child tasks. Usages of data definition and partitioning interfaces by the program are propagated to all the task based frameworks to manage their own internal data types.

For running a program using various frameworks, the TUV model also contains implemented wrappers around the SuperGlue, StarPU and DuctTeip task based frameworks, Fig. 1(b). There are also cpuBLAS and cuBLAS wrappers around Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) libraries for CPU and GPU devices, respectively, that can be used in the task flow graph for running the actual computations of tasks on the corresponding devices. The tasks submitted to these two wrappers are immediately executed and their completions are reported back to the dispatcher.

Figure 1(a) shows some examples of possible and practical configurations of task flow graphs G_1–G_4 for running a program whose operations on data are totally decomposable into BLAS subroutines.

The edge from the program to the dispatcher D is common for all the graphs and is not shown in the figure. The flow of tasks and data between the dispatcher D and the wrappers can be configured to determine the hierarchy of data and tasks and the corresponding responsible frameworks.

The nodes in the graphs shown in Fig. 1(a) are the dispatcher D and the wrappers around the task based frameworks. A directed edge from the dispatcher D to node w denotes that the tasks coming from the program to the dispatcher

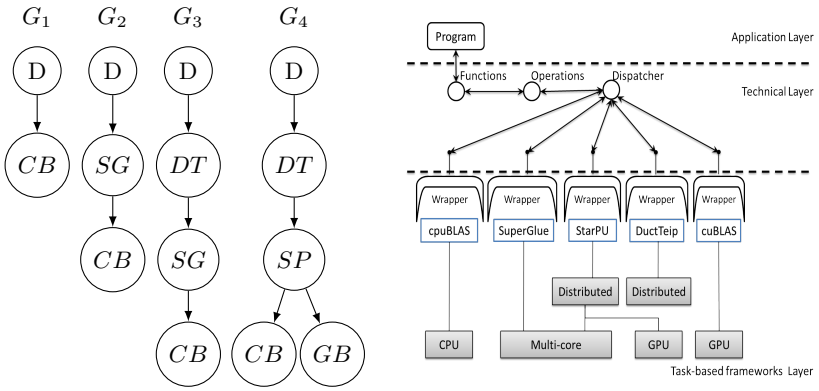
are forwarded to the wrapper w. For example, in graph G_1 the cpuBLAS wrapper (CB) is the only one connected to the dispatcher D, meaning that all the tasks coming from the program are delivered to the CB wrapper. Since the cpuBLAS wrapper is a single core implementation of the BLAS library, the tasks received by CB are run immediately and hence this configuration can be used to run the program sequentially. The edges between wrappers w_1 and w_2 show that ready tasks at w_1 are split into sub tasks by the dispatcher and are submitted to w_2, and task completions at w_2 are reported back to w_1 via the dispatcher. For example, in graph G_3, the DT, SG and CB wrappers are connected together by directed edges that show the flow of tasks and sub tasks between pairs of frameworks. These three wrappers are used to hierarchically break down the tasks at different levels for distributed memory and shared memory and kernel computations. Tasks are first submitted to DT for scheduling in the distributed memory environment. When DT notifies the dispatcher that a task is ready to run, the dispatcher splits the task into sub-tasks and submits them to the next wrapper, which is SG in the G_3 graph. When sub-tasks get ready at the SG wrapper, the dispatcher is notified from where they are forwarded to the next wrapper, CB in the graph G_3.

Using configurations G_1–G_4, the program can run in distributed/shared memory, and in heterogeneous (with accelerators) computing environments. Using the G_1 configuration, it can run sequentially on one CPU since tasks submitted to the dispatcher are forwarded to and immediately executed by the cpuBLAS wrapper. In the G_2 configuration, the same program can run on multi-core systems using the SuperGlue wrapper for managing submitted tasks on available cores and using the BLAS library wrapper for running the tasks on individual cores. The G_3 configuration is constructed by adding the DuctTeip wrapper on top, enables the program to run in a cluster of computing nodes where DuctTeip is managing data and tasks in the distributed memory environments. In the G_4 configuration, the program can run in a cluster of computing nodes with heterogeneous CPU/GPU processors using StarPU for managing tasks on both CPU and GPU.

2.3 Programming in the TUV Model

In the three layer view of the TUV model, the required programming at the bottom layer is already done in the TUV framework and provided as wrappers around some existing and frequently used task based frameworks. Further development of wrappers can be performed by framework providers or users by implementing the unified interface.

The programming at the middle layer, consists of implementing the translation and splitting of operations into tasks or sub tasks. This can be done by implementing predefined interfaces (e.g., the `split` method of an `operation` object) which will be used by the dispatcher during the program execution. User friendly functions can hide the technical details of the task and operations from the programmer at the application layer. The programming at the application



(a) Possible task flow graphs G_1–G_4. D is the *dispatcher* and DT, SG, SP are wrappers around the DuctTeip, SuperGlue and StarPU frameworks, respectively; CB and GB are wrappers around BLAS libraries on CPU and GPU, respectively.

(b) The overview of the TUV model. The TUV interface with task-based frameworks unifies the cooperation of the Dispatcher and any framework run-time. A single program in the Application Layer can be executed in various parallel environments using combinations of frameworks (or their wrappers).

Fig. 1. Cooperation between TUV and the task based frameworks.

layer consists of defining data and their partitions and calling functions provided by the technical layer to manipulate the data.

The programming in the TUV model is exemplified by implementing a block Cholesky matrix factorization called POTRF (POsitive definite matrix TRiangular Factorization) in BLAS/LAPACK terminology. The program at the application layer (shown in Fig. 2, lines 1–16) defines the input/output matrix `A` and its partitioning in two subsequent hierarchical levels (`b1` and `b2`) with parameters read from the command line and passes it to the `tuv_cholesky` function which is implemented in the technical layer (Fig. 2, lines 18–23).

The `<name>Task` objects in Figs. 2 and 3 are subclasses of a generic task class in the TUV model whose constructors accept an `Operation` object, a pointer to the parent task and data arguments of the task. The created `POTRFTask` in the `tuv_cholesky` function (Fig. 2, line 21) corresponds to the operation object `upotrfo` with no parent task.

The `Operation` objects in the TUV model are responsible for splitting an operation into child tasks that manipulate the partitions of the parent task's data arguments. Figure 3 shows the `split` method of the `upotrfo` operation which in nested loops manipulates the partitions of input argument `A` using the indexing interface `A(r,c)` to access the partition at row `r` and column `c` of `A`.

Figure 4 shows a possible implementation of a dispatcher with two cascaded wrappers, like SG and CB of G_2 in Fig. 1(a). An `Edge` type is defined to dis-

```
1  // unified_cholesky.cpp
2  #include "tuv.hpp"
3  int main(int argc, char **argv){
4    // TUV start
5    tuv_initialize(argc,argv);
6
7    int N, b1, b2;
8    // Get dimensions and partitions  from command line
9    tuv_get_parameters(N,b1,b2);
10
11   GData A(N, N, b1, b2);
12   tuv_cholesky(A);
13
14   // TUV waits for all tasks finished
15   tuv_finalize();
16 }
17 /*-----------------------------*/
18 //tuv_chol.cpp
19 #include "tuv.hpp"
20 void tuv_cholesky(GData &A){
21   POTRFTask *potrf = new POTRFTask(upotrfo,NULL,A);
22   dispatcher->submit_task(potrf);
23 }
24 /*-----------------------------*/
25 // cpuBLAS_wrapper.cpp
26 #include "tuv.hpp"
27 ...
28 upotrfo::run(GTask *t){
29   GData &A = *t->args[0];
30   double *mem = A.get_memory();
31   int info, N = A.get_rows_count();
32
33   dpotrf('L',N,mem,N,&info);
34   dispatcher->task_finished(t);
35 }
```

Fig. 2. The main program in the TUV model for implementing a Cholesky factorization of input matrix `A` (lines 1–16), the `tuv_cholesky` function provided by the TUV technical layer (lines 18–23), and the run method of the `upotrfo` operation (lines 25–35).

```
void upotrfo::split(GTask *p){
  //unpack arguments of t to A
  GData &A = p->args[0];

  int n = A.row_part_num();
  for(int i = 0; i<n; i++){
    for(int j = 0; j<i; j++){
      // submit task for A_ii = A_ij A_ij^T
      SYRKTask *syrk = new SYRKTask(usyrko,p,A(i,j),A(i,i));
      dispatcher->submit_task(syrk);
      for(int k = i+1; k<n; k++){
        // submit task for A_ki = A_ki + A_kj A_ij
        GEMMTask *gemm = new
            GEMMTask(ugemmo,p,A(k,j),A(i,j),A(k,i));
        dispatcher->submit_task(gemm);
      }
    }
    // submit task for A_ii -> LL^T
    POTRFTask *potrf = new POTRFTask(upotrfo,p,A(i,i));
    dispatcher->submit_task(potrf);
    for(int j = i+1; j<n; j++){
      // submit task for A_ji = A_ii^-1 A_ji
      TRSMTask *trsm = new TRSMTask(utrsmo,p,A(i,i),A(j,i));
      dispatcher->submit_task(trsm);
    }
  }
}
```

Fig. 3. The splitting method of the POTRF operation object in the TUV programming model where child tasks (SYRK, GEMM, POTRF and TRSM) for the parent task `p` are created and submitted to the `dispatcher` with their corresponding `u<name>o` operation objects.

```
1   /*-----------Edge ----------------------------------*/
2   template < typename T, typename U >
3   class Edge{
4   public:
5     typedef T First;
6     typedef U Second;
7   };
8   /*----------Edge Dispatch---------------------------*/
9   template < typename E >
10  class EdgeDispatch{
11  public:
12    typedef typename Edge<typename E::First,
13                          typename E::Second>::First  first;
14    typedef typename Edge<typename E::First,
15                          typename E::Second>::Second second;
16    /*-------------------------------------------------*/
17    template <typename T,typename P>
18    static void submit(UserProgram &, Task<T,P>*t){
19      //Tasks from the user-program are forwarded
20      //to the first wrapper
21      E::First::submit(t);
22    }
23    /*-------------------------------------------------*/
24    template <typename T,typename P>
25    static void submit(first &f, Task<T,P>*t){
26      //Tasks from the first wrapper are forwarded
27      //to the second  wrapper
28      E::Second::submit(t);
29    }
30    /*-------------------------------------------------*/
31    template <typename T,typename P>
32    static void ready(first &f,Task<T,P>*t){
33      // Ready-to-run tasks are split into sub-tasks
34      t->operation->split(t);
35    }
36    /*-------------------------------------------------*/
37    template <typename T,typename P>
38    static void finished(second &s,Task<T,P>*t){
39      // Finishing a task at the second wrapper may
40      // result in finishing a parent task in the
41      // first wrapper, if all of its children are
42      // finished
43      if ( t->allChildrenFinished() )
44        E::First::finished(t->get_parent());
45    }
46  };
```

Fig. 4. A C++ source code that shows one possible implementation of the Dispatcher (`EdgeDispatch`) when two wrappers are cascaded, as in graph G_2 in Fig. 1(a). An `Edge` type is used for distinguishing first and second wrappers. The first argument of each method from the dispatcher is the type of the caller. The workflow between wrappers can be implemented by connecting one call from a wrapper to a call to the other one.

tinguish two wrappers of `T` and `U`. An `EdgeDispatch` type is a dispatcher with generic parameter of type `Edge`. When objects in the program calls any method of the dispatcher, they pass in their type as the first argument. The figure shows how calls to dispatcher's `submit` method from the user program (line 18) or from the first wrapper (line 25) can be distinguished using the type of the first argument. The lines 32–35 of the code show how a ready to run task at the first wrapper can be split into sub-tasks by calling the `split` method of the `operation` object of the task. Similarly, the lines 38–45 show how a parent task at the level of the first wrapper is notified when all its children at the level of the second wrapper are finished. Therefore combining frameworks (or their wrappers) is simply providing a dispatcher object that forwards calls and notifications in this way. Thanks to the template programming in C++ and using static binding methods at compilation time, there is no run-time performance cost in using this approach for dispatching tasks between frameworks.

At the lowest level of the task hierarchy, when a task is submitted by the dispatcher to the `<cpu/cu>BLAS` wrappers, the `run` method of the task's operation is invoked where the BLAS/LAPACK routine is immediately called and task completion is reported back to the dispatcher, as shown in Fig. 2 lines 25–35.

3 Experiments

To demonstrate the productivity gained by the TUV programming model, the Cholesky matrix factorization algorithm is implemented and executed with different matrix sizes on different parallel computing resources. In these experiments the program at the application layer is written once and executed in different configurations for different underlying parallel hardware.

These programs were executed in the HPC2N computer cluster Kebnekaise using 32 nodes, each with two Intel Xeon E5-2690v4 CPU with 14 cores and with two NVIDIA K80 with 4992 cores. The programs are all written in C++, compiled with Intel compiler 17.0.1 and Intel MPI version 2017 Update 1 and use Intel MKL for BLAS/LAPACK routines.

The Cholesky factorization program is executed in multi-core, with or without GPUs, and multi-node computing environments. The SuperGlue, DuctTeip and StarPU frameworks are used with the TUV and the non-TUV models to compare performance when running the program in these environments, see Figs. 5 and 6. The StarPU implementation of the Cholesky factorization is taken from the version 1.2.0 of the installation package source code where explicit dependencies are used for tasks (by explicitly setting a task dependency to specific tags instead of data handles) and the input matrix is partitioned at two hierarchical levels. The `dmdar` scheduler is used and the experimental results are gathered after executing a few calibration runs. When StarPU is used within the TUV interface, the implicit dependencies and flat data (no hierarchy) is used. The DuctTeip implementation, uses implicit dependencies between tasks by tracking the accesses to the input and output data arguments of each task, and the data is partitioned at two hierarchical levels.

The DuctTeip, SuperGlue and StarPU frameworks are selected for experiments because they use different approaches and implement different types of parallelisms. DuctTeip has no support for multi-core and GPU systems, SuperGlue only supports multi-core environments and StarPU supports multi-core, distributed and GPU parallelism but uses different approaches.

In Fig. 5 different configurations (C_1–C_6) of frameworks are used for running the Cholesky factorization program in one computing node of multi-cores with/without GPUs. In the configurations C_1–C_3 matrices up to 30000×30000 elements are factorized using the StarPU framework without TUV interface (C_1), the StarPU wrapper within TUV (C_2) and the SuperGlue wrapper within TUV (C_3). For factorizing larger matrices in one computing node, parts of the computations are executed in the GPUs by using the StarPU framework without TUV interface (C_4), the StarPU wrapper within TUV (C_5) and the DuctTeip wrapper and StarPU wrapper within TUV (C_6).

The differences in performance of the configurations in the experiments can be explained in this way. In C_1, explicit dependencies are used and the matrix is partitioned recursively during the runtime. In C_2, StarPU within TUV uses implicit dependencies and fixed partitioning of the matrix. The SuperGlue framework was shown in [26] to have a low overhead compared with other frameworks, hence the performance results using SuperGlue (C_3) at the shared memory level tend to be competitive compared with other configurations. The C_4 configuration is the same as C_1 but with GPU enabled. Configuration C_4 can factorize much larger matrices than C_5 thanks to the recursive and hierarchical partitioning of the matrix. By adding hierarchical partitioning through DuctTeip to C_5, C_6 can produce the same throughput as C_4.

In Fig. 6, the performance of different frameworks is compared for Cholesky factorization of matrices in a distributed memory environment. The factorization is performed by the StarPU framework (C_7), the DuctTeip and StarPU wrappers within TUV (C_8) and the DuctTeip and SuperGlue wrappers within TUV (C_9). In configuration C_7, the StarPU framework does not employ hierarchical partitioning of the data. The MPI support is used for submitting and running tasks in the distributed memory environment. In C_8, DuctTeip is used with hierarchical data partitioning for the distributed memory environment and StarPU is used for running the sub tasks on the multiple cores within a computing node. In C_9, SuperGlue is used instead of StarPU in C_8 for running sub tasks. The hierarchical data partitioning and the efficient communication techniques used in DuctTeip [29], explain the performance difference between C_7 and C_8. Using the low overhead task scheduling of SuperGlue in C_9, higher performance than C_8 is obtained.

These experiments demonstrate that using the TUV model, a single program at the application layer written once can run in several parallel computing environments. Not only is it independent of any individual framework, but it can also attain the most favorable throughput from a customizable mixture of available frameworks. In other words, a program can always attain the highest

achievable performance by different mixtures of frameworks which individually are improving their efficiency from time to time.

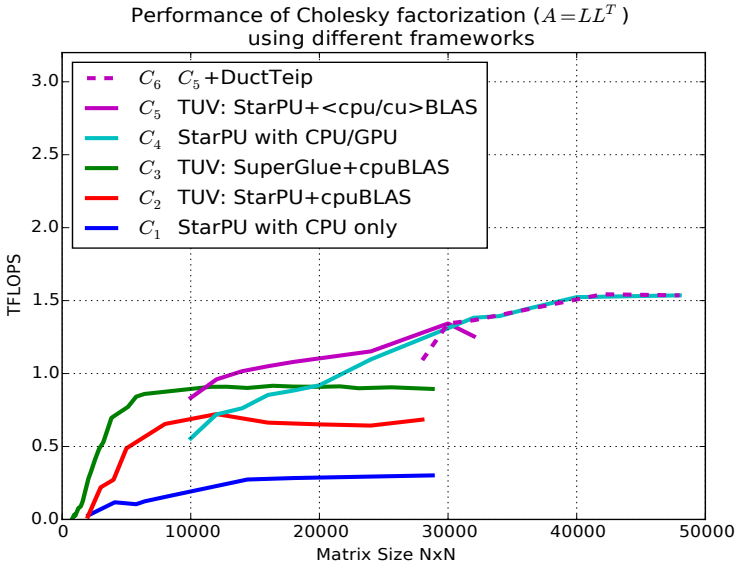


Fig. 5. Executing the Cholesky factorization in one computing node with or without GPU.

4 Conclusions

We have designed, implemented and verified the TUV model that unifies the cooperation interface between task based frameworks. This interface decouples the frameworks from the program that uses them which enables the program to run in different parallel computing environment, allows independent software development at technical layers and makes the program tolerant to the future changes in the underlying hardware. The configurable task flows in the TUV model allows a program to use a mixture of different frameworks to meet various needs of computations on different computing resources.

We have shown by experiments that when the performance requirements of a program are not satisfied with a single framework, the desired functionalities can be picked up from different frameworks and provided to the program. This enables a program to always achieve the best performance when the frameworks change during the time due to new features, improved performance, supporting a new hardware or implementing new approaches. Decoupling the program from the frameworks, like any other software library, makes the framework develop-

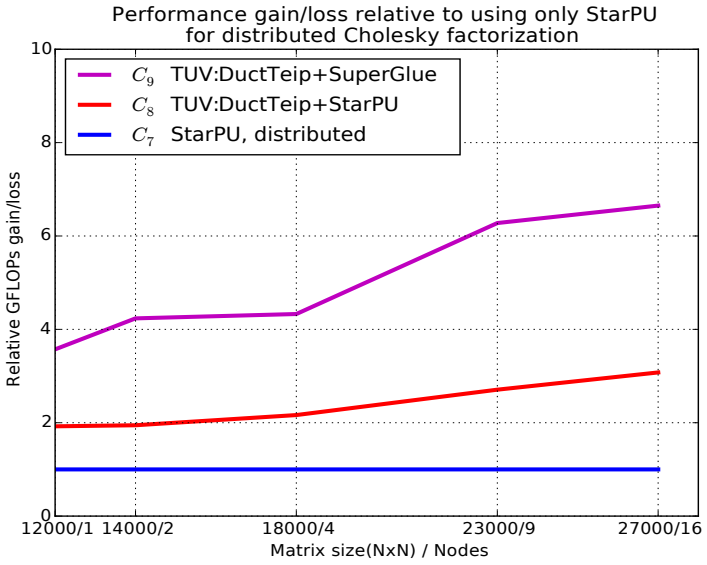


Fig. 6. Relative performance of distributed Cholesky factorization using TUV implementations vs StarPU only.

ment independent of the application programs that enables them to freely use new techniques and rapidly adapt to new technologies.

Acknowledgments

Thanks to Assoc. Prof. Elisabeth Larsson[2] for her valuable comments on improving the quality of this paper. The computations were performed on resources provided by SNIC through the resources provided by High Performance Computing Center North (HPC2N) under project SNIC2016-7-34.

References

1. Agullo, E., Giraud, L., Guermouche, A., Nakov, S., Roman, J.: Task-based Conjugate Gradient: from multi-GPU towards heterogeneous architectures. Research Report RR-8912, Inria (May 2016)
2. Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Ltaief, H., Thibault, S., Tomov, S.: QR factorization on a multicore node enhanced with multiple GPU accelerators. In: Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, IEEE (2011) 932–943

[2] http://www.it.uu.se/katalog/bette

3. Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., Thibault, S.: Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. Research Report RR-8927, Inria Bordeaux Sud-Ouest ; Bordeaux INP ; CNRS ; Université de Bordeaux ; CEA (June 2016)
4. Agullo, E., Bramas, B., Coulaud, O., Khannouz, M., Stanisic, L.: Task-based fast multipole method for clusters of multicore processors. Research Report RR-8970, Inria Bordeaux Sud-Ouest (October 2016)
5. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. Programming multi-core and many-core computing systems, parallel and distributed computing (2014)
6. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: 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., Delft, The Netherlands, Springer (August 2009) 863–874
7. Bauer, P., Engblom, S., Widgren, S.: Fast event-based epidemiological simulations on national scales. The International Journal of High Performance Computing Applications **30**(4) (2016) 438–453
8. Bauer, P., Engblom, S., Widgren, S.: Fast event-based epidemiological simulations on national scales. The International Journal of High Performance Computing Applications **30** (2016) 438–453
9. Boillot, L., Bosilca, G., Agullo, E., Calandra, H.: Task-based programming for seismic imaging: Preliminary results. In: High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on, IEEE (2014) 1259–1266
10. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., Dongarra, J.J.: PaRSEC: Exploiting heterogeneity to enhance scalability. Computing in Science & Engineering **15**(6) (2013) 36–45
11. Bueno, J., Martinell, L., Duran, A., Farreras, M., Martorell, X., Badia, R.M., Ayguade, E., Labarta, J.: Productive cluster programming with OmpSs. In: European Conference on Parallel Processing, Springer (2011) 555–566
12. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. The International Journal of High Performance Computing Applications **21**(3) (2007) 291–312
13. Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., Dongarra, J.: PTG: an abstraction for unhindered parallelism. In: Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on, IEEE (2014) 21–30
14. Danalis, A., Jagode, H., Bosilca, G., Dongarra, J.: PaRSEC in Practice: Optimizing a legacy Chemistry application through distributed task-based execution. In: 2015 IEEE International Conference on Cluster Computing, IEEE (2015) 304–313
15. del Rio Astorga, D., Dolz, M.F., Sanchez, L.M., Blas, J.G., García, J.D.: A c++ generic parallel pattern interface for stream processing. In: Algorithms and Architectures for Parallel Processing. Springer (2016) 74–87
16. Ernstsson, A., Li, L., Kessler, C.: Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. International Journal of Parallel Programming (2017) 1–19
17. Goude, A., Engblom, S.: Adaptive fast multipole methods on the GPU. The Journal of Supercomputing **63**(3) (2013) 897–918
18. Intel: Intel Threading Building Blocks. `https://www.threadingbuildingblocks.org` 2017.

19. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: 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, ACM (2014) 6
20. Lacoste, X., Faverge, M., Ramet, P., Thibault, S., Bosilca, G.: Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. Research Report RR-8446, INRIA (January 2014)
21. Martínez, V., Michéa, D., Dupros, F., Aumage, O., Thibault, S., Aochi, H., Navaux, P.O.: Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In: Computer Architecture and High Performance Computing (SBAC-PAD), 2015 27th International Symposium on, IEEE (2015) 1–8
22. Ohshima, S., Katagiri, S., Nakajima, K., Thibault, S., Namyst, R.: Implementation of FEM Application on GPU with StarPU. In: SIAM CSE13-SIAM Conference on Computational Science and Engineering 2013. (2013)
23. OpenMP-ARB: OpenMP 4.5 Specifications. `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf` 2017.
24. Rubensson, E.H., Rudberg, E.: Chunks and Tasks: A programming model for parallelization of dynamic algorithms. Parallel Computing **40**(7) (2014) 328–343
25. Rubensson, E.H., Rudberg, E.: Locality-aware parallel block-sparse matrix-matrix multiplication using the Chunks and Tasks programming model. arXiv preprint arXiv:1501.07800 (2015)
26. Tillenius, M.: SuperGlue: A shared memory framework using data versioning for dependency-aware task-based parallelization. SIAM Journal on Scientific Computing **37**(6) (2015) C617–C642
27. Tillenius, M., Larsson, E., Lehto, E., Flyer, N.: A scalable rbf–fd method for atmospheric flow. Journal of Computational Physics **298** (2015) 406–422
28. Zafari, A., Larsson, E., Righero, M., Francavilla, M.A., Giordanengo, G., Vipiana, F., Vecchi, G.: Task parallel implementation of a solver for electromagnetic scattering problems. Technical Report 2016-015, Uppsala University, Division of Scientific Computing (2016)
29. Zafari, A., Larsson, E., Tillenius, M.: DuctTeip : A task-based parallel programming framework for distributed memory architectures. Technical Report 2016-010, Uppsala University, Division of Scientific Computing (2016)