Dynamic Task Discovery in PaRSEC- A data-flow task-based Runtime

Reazul Hoque, Thomas Herault, George Bosilca
The University of Tennessee
Knoxville, USA

Jack Dongarra
The University of Tennessee, Knoxville, USA
Oak Ridge National Laboratory, Oak Ridge, USA
University of Manchester, Manchester, UK

ABSTRACT

Successfully exploiting distributed collections of heterogeneous many-cores architectures with complex memory hierarchy through a portable programming model is a challenge for application developers. The literature is not short of proposals addressing this problem, including many evolutionary solutions that seek to extend the capabilities of current message passing paradigms with intra-node features (MPI+X). A different, more revolutionary, solution explores data-flow task-based runtime systems as a substitute to both local and distributed data dependencies management. The solution explored in this paper, PaRSEC, is based on such a programming paradigm, supported by a highly efficient task-based runtime. This paper compares two programming paradigms present in PaRSEC, Parameterized Task Graph (PTG) and Dynamic Task Discovery (DTD) in terms of capabilities, overhead and potential benefits.

CCS CONCEPTS

• **Theory of computation → Distributed computing models**;

KEYWORDS

PaRSEC, task-based runtime, data-flow, dynamic task-graph

ACM Reference Format:
Reazul Hoque, Thomas Herault, George Bosilca and Jack Dongarra. 2017. Dynamic Task Discovery in PaRSEC- A data-flow task-based Runtime. In *ScalA17: 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3148226.3148233

1 INTRODUCTION

The future of high performance computing is shifting towards increasingly hybrid machines with many fat nodes with deep memory hierarchies and augmented with different types of accelerators (GPUs, APUs etc.). After hitting the physical frequency barrier the need to transition to an increase in parallelism in order to improve performance becomes unquestionable. This trend demands finer granularity of parallelism, but as long as we don't have the tools to automatically extract it from sequentially described source code, the burden to expose the intrinsic algorithmic parallelism remains on software developers. In a traditional programming environment, software developers are required in addition of exposing parallelism from the algorithms, to manage the resources and also decompose and express their computations in a way that is portable among shared and distributed memory machines with widely varying configurations. To address the challenges of efficiently utilizing this type of heterogeneous resources we need programming paradigms that provide the ability to express parallelism in flexible and productive manners. MPI and OpenMP are two of the most popular programming models for parallel applications. They both encourage a practice of parallel programming for hero-programmers, where the developers perform multiple jobs: express parallelism, manage the computational resources and communications, and programmatically provide the mapping between these two. These burdens become heavier with the increase in core and node count, in heterogeneity of computational resources and application size.

At the opposite of the spectrum, task based runtime systems have become popular in tackling such challenges and making it easier to write parallel HPC applications. Runtimes relieve the users from managing low-level resources and gives them the opportunity to focus on writing parallel applications by describing the potential parallelism in a way that is comprehensible by the runtime. Any task-based runtime expects the users to express their computations and the data on which the computations will be performed, in a way where computations become entities (aka. tasks) and the data flowing among them are the dependencies. Runtimes then create a complete, or in some cases partial, directed acyclic graph (DAG) of tasks based on these dependencies to figure out a correct execution. Thus, the major challenge of using a runtime is not only on the capabilities of the runtime, but also on the expressivity of the language or API the runtime provides for expressing the task-graph.

In this paper we propose a dynamic feature of one such runtime: Parallel Runtime Scheduling and Execution Controller (PaRSEC) and compare two different methods of expressing parallel computation in PaRSEC. PaRSEC has been previously proposed as a runtime for heterogeneous architecture where users would use a parameterized expression of task dependencies with PaRSEC implicitly inferring the communication between nodes and the accelerators. This programming model is called Parameterized Task Graph (PTG) [12]. In this paradigm, users provide a concise description of all data flow, the tasks that are the source of such data and also those that are the destination. This creates a compressed algebraic representation of the task graph, which is then transformed into *C* code using a pre-compiler. Users need to know all the data-flow of all the types of task in this model, but data-dependent algorithms are possible.

ScalA17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5125-6/17/11…$15.00
https://doi.org/10.1145/3148226.3148233

We propose to augment the capabilities of the PaRSEC runtime with another task-based programming paradigm, Dynamic Task Discovery (DTD), to provide an alternative way to express task dependency in PaRSEC that achieves a similar purpose, hopefully with comparable performance. Using this new paradigm users can write sequential constructs (ifs, loops, etc.) to insert task in PaRSEC instead of expressing them in a parameterized manner. Each task is expressed to the runtime with the data it will use and the mode of usage, based on which dependencies are created. This mode is somewhat similar to the `task` directive in OpenMP, in the sense that dependencies will be automatically computed by the runtime, out of data pointers used by the tasks. However, unlike OpenMP, exposing the data as independent entities instead of mere hash keys opens the opportunity to derive distributed version of the algorithms in a simple way. The data movement among the nodes in a distributed system become completely implicit in this paradigm. A similar programming paradigm has been previously proposed by other runtimes such as QUARK[20] and StarPU[4]. In this paper:

- We propose an alternative way to express dependency in PaRSEC, that blends into the runtime and interoperates with all the other PaRSEC programming paradigms;
- We describe the API for inserting task that allow online distributed building of the dependency DAG, and highlight how dependencies between tasks are built and maintained;
- We discuss the optimizations required to minimize the overhead of building dynamic task-graph dynamically during runtime;
- We model the overhead of dynamically building a task-graph compared to the PTG's compressed representation using a mathematical model and validate the model using experimental data;
- We present the performance of dense linear algebra algorithms in shared and distributed memory systems. We compare the results with several other runtimes (where similar capabilities are available).

The organization of the rest of the paper is as follows: Section 2 gives an overview of the state-of-the-art of task-based runtimes. Section 3 describes the proposed programming model, the required optimizations and a look at the theoretical overhead. Section 4 explains the various experiments performed and their results on shared and distributed memory. Finally, Section 5 summarizes the programming model and its performance with some discussion about future work.

2 RELATED WORK

In this context we refer to task-based runtimes that are designed specifically to handle fine granularities tasks below the millisecond. Workflow systems where the usual granularities are in the order of tens of seconds, share some similitudes with task-based runtime but are outside the scope of this paper. There are multiple task-based runtime systems that allow developers to express their application in a way that takes away the burden of mapping computational tasks to the underlying hardware. Some of the recent task-based runtimes such as Legion [5], StarPU [4], QUARK [20], HPX [16], OCR [15], OmpSs[10], SuperGlue[18], OpenMP [2] and PaRSEC [8] abstract the available resources to simplify the process of writing massive parallel scientific application.

Legion describes logical regions of data and uses those regions to express the data flow and dependencies between tasks. It uses a low level runtime, Realm[19], to schedule and execute tasks and uses GASNet as the underlying communication layer. It supports heterogeneous architecture and works in both shared and distributed memory systems.

QUARK and StarPU each lets users submit tasks using their API and dynamically builds the task-graph. QUARK does not support heterogeneous architecture and works only in shared memory systems whereas StarPU has support for heterogeneous systems, and a nascent support for distributed memory. Both runtimes manage threading internally and has their own scheduler.

Open Community Runtime (OCR) currently supports homogeneous architecture in distributed systems and uses Intel Threading Building Blocks to manage threading. It is still in very early development stage.

OmpSs uses Nanos++ runtime to manage tasks and follows a master-slave model. It allows nesting of tasks in individual node to relieve the master, however the master-slave model may suffer from scalability issues as the scale of the underlying execution platform increases or the scale of the application.

SuperGlue employs data versioning to represent dependencies. This means that a task depends on data rather than on another task and as of now it supports shared memory systems.

OpenMP introduced *depend* clause in standard 4.0 which allows users to express the computation in task form which indicates the popularity and potential of task based runtime systems. OpenMP is widely used and supports homogeneous shared memory systems, but extensions in the heterogeneous environments exists and are currently investigated by the OpenMP standardization body.

The common point between all these runtimes is the fact that they all use some codified description of dependencies to build the task graph during execution, and then distribute the work on the available resources. Their capability of using heterogeneous computing resources varies, as well as what is the definition of a task (in the sense of what types of operations are allowed to be executed). The proposed extension to PaRSEC, DTD, while looking similar to many of them, differs in many subtle ways, providing more opportunities for efficient scheduling over heterogeneous resources, and overlapping between communications and computations.

3 PaRSEC AND DYNAMIC TASK DISCOVERY

This section describes PaRSEC runtime and introduces a new feature - Dynamic Task Discovery. We discuss the advantages and disadvantages of the existing and proposed programming paradigm and explain the optimizations required to reach comparable performance with the new interface.

3.1 PaRSEC

PaRSEC [8] is a task-based runtime for distributed heterogeneous architectures, capable of tracking (and when necessary moving) data between different memory (in and between nodes) and scheduling tasks on heterogeneous resources. It employs several Domain Specific Languages (DSL) to provide flexible domain specific programming models to application developers. These DSLs create

a data-flow model to create dependencies between tasks and exploits the available parallelism present in application. PaRSEC is rich with many features aimed at helping developers express their application to the runtime correctly and efficiently. Certainly the most exposed DSL, PTG, allow users to use a parameterized task graph (PTG) [13] known as Job Data Flow (JDF) which handles the dependencies between tasks. To enhance the productivity of the application developers, PaRSEC implicitly infers all the communication from the expression of the tasks, supporting one-to-many and many-to-many types of communications. The runtime has been designed to excel in distributed systems and has been extensively tested for performance yielding excellent results [13] in comparison to widely used library, ScaLAPACK [6], or currently state-of-the-art computational chemistry applications [14, 17].

Multiple components constitute PaRSEC runtime: programming interfaces (DSL), schedulers, communication engines and data interfaces. The runtime uses a modular component architecture, allowing different modules to be selected, providing different capabilities to different instances of the runtime (such as scheduling policies, or support for heterogeneity). A clear API for these modules allows interested developers or users to implement their own, application specific, policies. The different DSL share the same runtime, data representation, communication engine, scheduler, allowing them to seamlessly inter-operate in the context of the same application.

Traditionally, application developers have a propensity to write sequential code. PaRSEC, with the help of a pre-compiler, transforms some form of sequential code to PTG, with the limitation that the sequential code must be affine [9]. In the remaining of this paper, we propose a different PaRSEC programming model, Dynamic Task Discovery, that removes the need of a pre-compiler, and therefore abolish the loop-affine limitation.

3.2 Dynamic Task Discovery in PaRSEC

Dynamic Task Discovery (DTD) is a new PaRSEC DSL (or in this particular instance low-level task interface) proposed in this work. This interface allows users to write sequential-looking code, including conditionals, *for* loops, code blocks, to insert task using PaRSECs API. There are three main concepts to express a task graph in PaRSEC using DTD: task, dependency and data. A *task* is any kind of computation that will not block, *data* are pieces of memory on which the computations will be performed and *dependencies* are the ordering relationship between tasks. To insert a task in PaRSEC, users indicate the data and the mode of operation that will be performed on the data by a task (read, write or read/write). Dependencies between tasks are created based on the operation-type on the data, a task performing a *write* before a task performing a *read* on the same data will create a read-after-write (*RAW*) dependency between the write-task and read-task, such that the read-task will only execute after the write-task is completed. The sequential expression guarantees the correct ordering of tasks. In distributed memory systems all the participating processes need to have a consistent view of the DAG for DTD to maintain the correct sequential order of tasks and this require the whole DAG to be discovered by all the processes.

Applications operating in distributed memory have data distributed among participant processes. PaRSEC has its own data description interface that allows users to express how the data should be distributed (block-cyclic, 2d-block-cyclic, completely irregular etc). PaRSEC then abstracts the distribution information in a consistent global structure called *data-collection*. Each piece of data is represented by a *data_t* structure that can hold references of multiple *data-copy(s)* in circulation and each *data_t* in turn will belong to a certain *data-collection*. PaRSEC manages the *data_t* and *data-copy* internally. Both the DSL in PaRSEC shares the same data interface.

To illustrate PaRSEC's API to insert task we provide in Listing 1 an example of tiled Cholesky factorization, composed of three nested levels of affine loops, and contains four operations (tasks), namely: POTRF, TRSM, HERK and GEMM.

```
1  for( k = 0; k < total; k++ ) {
2   parsec_insert_task (POTRF,
3           tile_of(A, k, k), INOUT | AFFINITY);
4    for( m = k+1; m < total; m++ )
5      parsec_insert_task (TRSM,
6              tile_of(A, k, k), INPUT,
7              tile_of(A, m, k), INOUT | AFFINITY);
8    for( m = k+1; m < total; m++ ) {
9      parsec_insert_task (HERK,
10             tile_of(A, m, k), INPUT,
11             tile_of(A, m, m), INOUT | AFFINITY);
12
13     for( n = m+1; n < total; n++ )
14       parsec_insert_task (GEMM,
15               tile_of(A, n, k), INPUT,
16               tile_of(A, m, k), INPUT,
17               tile_of(A, n, m), INOUT | AFFINITY);
18   }
19 }
```

Listing 1: Cholesky Factorization

Each operation takes a number of data as input and performs a specific mathematical operations on the input data. Line 2 of listing 1 shows the API to insert task in PaRSEC. Each data, the tasks take as input, has an 'operation-type' flag associated with it. The operation-type flags currently supported are INPUT - specifying the data is read-only, INOUT - the data will be read and written on and OUTPUT - the task will write on the data. The AFFINITY flag indicates the placement of the task in a distributed environment, on the rank where the data corresponding to the AFFINITY flag resides, e.g. line 3 of listing 1 shows that POTRF tasks will be placed in the rank where data *A(k, k)* resides. Task placement in a distributed environment depends on the initial data distribution. The DTD interface builds a DAG of all the tasks inserted in the runtime depending on the data and the way each task consumes and produces them. All the communications required to carry out a deterministic and coherent execution of any application are implicitly inferred, depending on the affinity of the tasks that update and consume a data. In a more general context, assuming *Task 1* updates *Data A* in rank 0 and has a successor *Task B* in rank 1, the necessary data transfer from rank 0 to rank 1 is automatically inferred from the dependency between the tasks and is completed asynchronously by PaRSEC. This eliminates the cumbersome and error-prone requirement of expressing explicit communication, and makes the algorithm itself independent of the data distribution, with each task instance affinity dependent. For the sake of understanding the difference with the original PaRSEC programming model listing 2

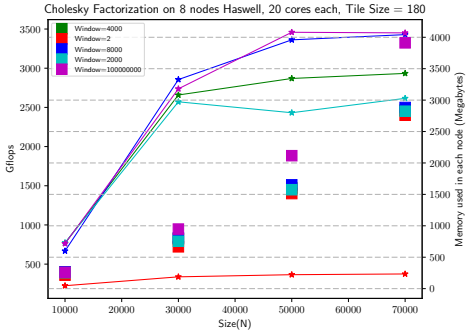

Figure 1: Performance and Memory-footprint per node for different window size

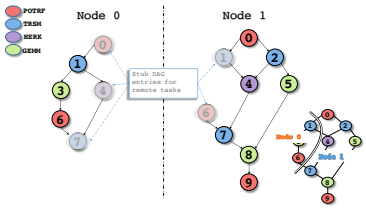

Figure 2: Sample Trimmed DAG

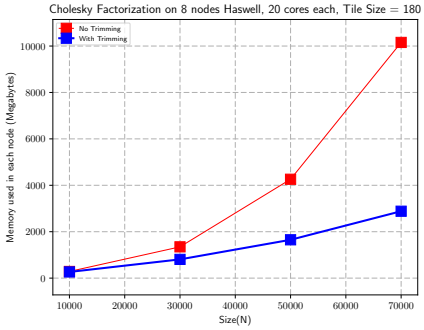

Figure 3: Memory-footprint in each node with and without Trimming

shows a sample code for one of the task class, *POTRF*, using the PTG interface of the same tiled Cholesky factorization used above to illustrate the proposed interface.

```
1  POTRF(k)
2  k = 0 .. total // Execution space
3  : A(k, k)      // Task affinity
4  RW T <- (k == 0) ? A(k, k) : T HERK(k-1, k)
5       -> T TRSM(k+1 .. total, k)
6       -> A(k, k)
7  BODY { POTRF( ... ); } END
```

Listing 2: JDF of Cholesky Factorization

Line 2 of listing 2 defines the range of the parameter k, which in turn defines how many tasks belonging to task class *POTRF* will be there. Line 3, defines the task placement in a distributed environment, where each *POTRF*(k) task will be placed where data A(k, k) resides (equivalent to the AFFINITY flag described previously). Line 4-6 specifies the parameterized data dependency of this task class. Each *POTRF* task will consume and produce one data *T*. *RW* on line 4 specifies that the data will be read and written by each task of this task class. Arrow pointing to the left indicates input dependency for that data and arrow pointing right indicates output dependency. Line 4-6, in words would read *POTRF*(k) will consume data *T* in *RW* mode, where *T* will either come from memory(A(k, k)) or from task *HERK*(k-1, k) depending on parameter *k* and the data produced will be consumed by a bunch of tasks belonging to the same task class *TRSM* ranging from (k+1 .. total, k) and also that *POTRF*(k) will be the final writer of data A(k, k) (line 6).

3.2.1 Challenges and Optimization. Dynamic Task Discovery has the advantage of being able to dynamically discover tasks, but poses challenges that need to be tackled in order to obtain scalability and performance. Some of the features like, untying task insertion from a specific thread, independent task insertion from multiple threads, DAG trimming in distributed environment are novel to DTD. We list some of those challenges, including the ones mentioned earlier, and discuss how we address them.

Unrolling the DAG. The whole DAG of tasks needs to be *unrolled* in memory in order to progress and schedule them in DTD. Saving the task graph in memory is defined as unrolling. Looking ahead in the task-graph gives the runtime opportunity to improve the scheduling decisions and the occupancy of the computational units, and in extreme cases might result in unrolling the whole DAG. The other paradigm, PTG, does not incur this overhead. PTG does not need to unroll a DAG as the whole DAG has been compressed by the parameterized expression given by the user. In case of DTD, the memory requirement for the DAG is $O(|V| + |E|)$ where $|V|$ is the number of tasks discovered, and $|E|$ the number of dependencies. In PTG, the memory requirement is $O(|TC| + |DC|)$, where $|TC|$ is the number of types of task, and $|DC|$ the number of data each type of task refers. Building a DAG is an operation that is at least $O(N)$ where N is the size of the representation. This theoretically puts PTG ahead in terms of performance and less memory overhead. The memory requirement of unrolling the whole DAG can limit the size of the problem we want to solve using DTD. We solve this challenge by implementing a throttling mechanism for task insertion in the system. It works like a sliding window of DAG that user can control with environment variables read by PaRSEC. This bounds the memory usage of any problem to the size of the window. DTD reuses the task structure of completed tasks to keep the memory footprint at a minimum. Figure 1 shows the performance of Cholesky factorization with different window sizes and the memory footprint of the application at those window sizes. We can see, with a bounded sliding window we achieve the same performance with substantially less memory overhead. The window size is a performance tuning parameter and it dictates how many tasks the task-inserting thread will insert before joining the other threads in doing useful work. So, a window of 2k means the responsible thread will insert 2k tasks (local tasks in case of distributed memory) before it stops to join the others. If the window size is too small not enough tasks are discovered which results in less parallel work and bad performance. If the window size is too large, more memory is used to store the DAG and this results in large memory overhead. Another important variable is the threshold parameter, which dictates the lower watermark after which the responsible thread will start inserting task again. For the experiment we show in Figure 1, the threshold is 1 for window size 100 million and half the corresponding window size for the rest. Setting a small threshold will result in starvation as the runtime will be delayed in inserting tasks. We see in Figure 1 that window size of 2 shows dramatic performance loss with less memory overhead, while a window size of 100 million has the same performance as a more reasonable window size of 4k or 8k with significantly higher memory overhead. At window size 4k and 8k

we see as good a performance as we would if the runtime had a large look ahead and observe much better performance compared to performance of small window sizes like 2k or smaller. We also observe the memory overhead at those window sizes to be almost similar to storing a very small DAG like we would at window size 2. At window size 100 million we are storing the whole DAG for each process in memory and we see for large size (70k) that occupies almost a gigabyte per process more than storing window (4k, 8k) of DAG.

Untying task insertion from specific thread. User can select to insert task using a specific thread, where the thread blocks after a certain number of task is inserted to maintain a sliding window. This results in having only one specific thread inserting task and creates dependency on that single thread for task insertion (poor performance in this case) and reduces parallelism. Alternatively, the user can insert a task that will generate other tasks and untie the tasks insertion from a specific thread. In the latter case PaRSEC provides mechanism for a task to de-schedule itself without completion. Tasks can return a special 'schedule-me-later' flag to the runtime signaling that the task is not complete and needs to be rescheduled later. The untied scheme eliminates performance degradation in the case of the responsible thread being de-scheduled by the operating system, and permits a sliding window of tasks, as described in the previous section. It also enables users to generate independent tasks simultaneously and provides a mechanism to insert task recursively in DTD.

DAG Trimming. In distributed environment DTD improves its memory footprint by trimming the DAG. We keep track of all the local tasks and all the remote tasks that are either a direct successor or a predecessor of a local task and trim the rest of the DAG. Trimming the DAG is not as simple as storing a task if it is local and ignoring it if it is not, as we have to keep related remote tasks. To achieve this, we need to keep track of all the data, local or remote, that has been discovered by the runtime so far to identify the remote tasks that are related to local ones and then decide on keeping or ignoring them. Figure 2 shows a sample trimmed DAG in each node in a distributed run. We can see the advantage of trimming the DAG of remote not-related tasks in Figure 2. We are successfully able to restrict the memory overhead of large distributed problems using this technique. Figure 3 shows the memory usage of distributed Cholesky factorization on 8 nodes in double precision with and without trimming. Factorization of a large matrix involves a lot more tasks compared to smaller one and we see by trimming the DAG and reusing the task structures we require almost five times less memory in the case of size 70000.

Communication. Communication in DTD is accomplished using the communication engine present in PaRSEC. In DTD paradigm each individual process in a distributed environment will discover the entire task-graph independently and at its own speed. This results in situations where processes are out of sync, and process P_i is trying to send a message about a task $T(k)$ to process P_j, and P_j did not yet discover task $T(k)$. The task is then known, but not yet locally discovered, and in order to minimize the local impact on memory usage we delay the communication between process P_i and process P_j until process P_j discovers $T(k)$, and knows how the data should be fetched. This situation is unique in DTD and adds more overhead in the communication side compared to PTG.

Moreover, as we discover that due to the tasks dependencies, the same data version would need to be transfered multiple time between 2 processes, we optimize the number of communication necessary by marking the data accordingly with the local knowledge, and avoiding to send it more than once. This avoids redundant communications. We have also carefully engineered the communication engine, to maximize the memory reuse by recycling buffers allocated for remote data, reducing the number of calls to costly memory allocation/deallocation and pinning/unpinning functions. A local copy of a remote data becomes reusable once all the local uses of the data version are completed and the local process has also discovered the next writer.

3.3 Overhead of DTD compared to PTG

In this section we present a mathematical model to represent the overhead present in Dynamic Task Discovery compared to Parameterized Task Graph. We start by defining the notations to represent the different performance tuning parameters of any task based runtime. Let's define N as the total number of tasks, C_T as the cost/duration of each task, P as total number of process and n as the number of actual cores in each process. Let us also define C_D as the cost/duration of discovering each task during execution and C_R as the cost/duration of building the DAG/relationship. Given these definitions we can very simply express the overall execution time of a PTG run as :

$$T_{PTG} = \frac{N \times C_T}{P \times n} \tag{1}$$

and DTD's overall time as:

$$T_{DTD} = \frac{N \times C_T}{P \times n} + N \times C_D + \frac{N \times C_R}{P} \tag{2}$$

Here, we consider both PTG and DTD to have the same scheduling overhead as they share the scheduler and the communication engine, and do not include that in the total time. For PTG we show the total useful computation time as the total time. DTD's total time include the computation time plus the time to discover and build the task graph. Having the advantage of the compressed representation of the task-graph, PTG does not incur this cost. The total overhead of DTD paradigm is the time needed to discover N tasks in all P processes and the linking of tasks that are dependent. Since, all the processes will have to discover all the tasks to maintain the same coherent view of any DAG, the cost is $\frac{N \times C_D \times P}{P} = N \times C_D$. We assume a perfectly balanced task-graph where each process is concurrently building $\frac{N}{P}$ part of the DAG and hence the total DAG building cost is $\frac{N \times C_R}{P}$.

To validate the overhead of DTD, we performed Cholesky factorization in distributed setting of 8 nodes. By tuning the tile size, we can vary the cost of each task, C_T, and the total number of tasks in the system, N, two principal overhead tuning parameters. In Figure 4, we see the performance of both the paradigm drops as we move to finer granularity resulting in smaller tasks. Important thing to notice is the drop of DTD is significantly higher than PTG. Altering the matrix size varies one of the tuning parameters, N, while the other two: P and C_T are constant. By looking at Equation 2, we

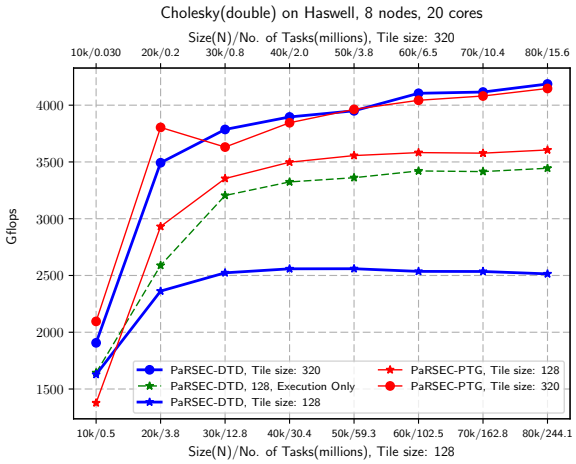

Figure 4: Overhead test in PaRSEC

can identify the cases where the overhead of DTD paradigm will be visible compared to PTG.

Let us start with the parameter N. For varying matrix sizes, the performance of DTD and PTG are comparable in the case of tile size 320. Here, as we grow the matrix size, only N increases. We do not see any performance drop in either case, as when N alone grows it only increases the shared scheduling overhead and the ratio of cost parameters in DTD remains constant in this case as P is fixed. As we move from 320 to a lower tile size we see performance drop in both PTG and DTD. We change two parameters as we lower the tile size, the C_T and N, we decrease the cost of each task and we increase the number of tasks. In the previous example we see that varying only N does not effect the performance when the other parameters are constant, so we can attribute the performance drop in this case to the decrease in C_T. In Equation 2 as we decrease C_T the ratio $\frac{C_T}{C_D+C_R}$ follows. This affects only DTD as this overhead is not present in PTG. To validate this relationship we repeated the same experiment for DTD, but this time we build the whole DAG before and excluded this building time. We see that the performance improves drastically and is only 4% lower than PTG.

From Equation 2 we can identify the cases where DTD will not scale and perform well. If the ratio $\frac{C_T}{C_D+C_R}$ is small the overhead of discovering and building DAG will be significant. For large distributed execution involving numerous processes and billions of task the middle part of Equation 2 ($N \times C_D$) will be a bottleneck. All processes needs to at least discover all the tasks in the DAG for correctness and for large P the computation time as well as the partial DAG building time (last part of Equation 2) will be lower, but as the middle part is not a function of P, it will not be affected at all. Given, N remains constant and we keep adding P up to the point where there is enough parallel work for all the processes, DTD will stop performing because of the bottleneck of discovering all the tasks. Given, N and P both increases, the discovery part will grow much faster(depending on P) than the computation time, and will eventually not perform. The one possible solution to this problem is pruning the task-graph, where user takes more responsibility and does not submit all the tasks in all processes, which in ideal case will make $N \times C_D$ go down to $\frac{N \times C_D}{P}$.

Table 1: Summary of Softwares used

Software	Version	Software	Version
Compiler	GCC 5.1	BLAS	MKL
StarPU	1.2.0	Chameleon	0.9.1
Quark	0.9.0	PaRSEC	2.0 rc
Open MPI	2.1		

4 EXPERIMENTS AND PERFORMANCE

In this section we describe the different experiments we have performed to assess the performance of DTD and the environment and parameters of each experiment. We present the performance of each experiment and discuss the outcome.

4.1 Experiment Details

Dense Linear Algebra Routines. We test tiled Cholesky and QR factorization (double precision) on both shared and distributed memory system and compare with other runtimes. QR factorization uses multiple data where as Cholesky uses one and both factorizations create DAGs with multiple task types. These tests should allow us to assess the performance of both the interfaces in PaRSEC and the other runtimes. It is important to note that the tile size in these tiled algorithms determine the total number of tasks that will be generated and that in turn determines the stress on the runtime. With all these experiments the target is not to show the percentage of machine peak we are reaching but rather how the runtimes are performing compared to each other given the same stress. PTG [13] has already shown to scale better than the alternatives, so achieving comparable results to PTG with obvious more runtime-overhead should be an indication of good performance.

We have used Chameleon to test all the runtime systems other than PTG in shared memory. Chameleon [3] provides the PLASMA [11] library with an option to choose the underlying runtime. Currently supported runtimes are QUARK, StarPU and PaRSEC. For distributed system we have used DPLASMA [7] for PaRSEC and ScaLAPACK [6]. Runtimes and libraries compared in the dense linear algebra routines for shared memory are QUARK [20] and StarPU [4] and for distributed memory, ScaLAPACK [6]. For PaRSEC we have used the local flat queue (LFQ) scheduler and for StarPU we have used local work stealing (LWS) scheduler. GCC 5.1 was used to compile all the libraries and OpenMPI 2.1 was used as the communication library.

For shared memory tests we have used Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz machine with 20 physical cores. We also report result of Cholesky on Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz with 68 physical cores.

For distributed memory system we have used TACC [1] cluster - Stampede. Each node is equipped with 2 8-core Xeon E5 processors and 32GB of memory and connected with Infiniband FDR interconnect.

4.2 Performance Evaluation

Shared Memory. We present result of Cholesky factorization on two architectures, Intel Haswell and Intel KNL. Figure 5 shows the performance on Haswell, where the peak performance of the GEMM kernel is 645 GFlops. The top plot shows the performance of the runtime systems for a tile size of 320x320. At this size there is a clear convergence in terms of performance between all runtimes,

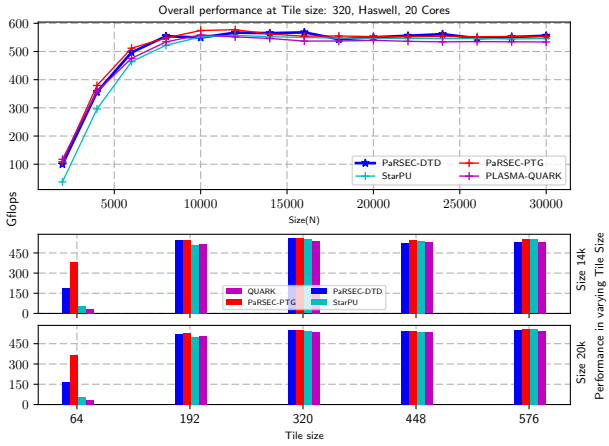

Figure 5: Performance of Cholesky factorization on Haswell (shared memory)

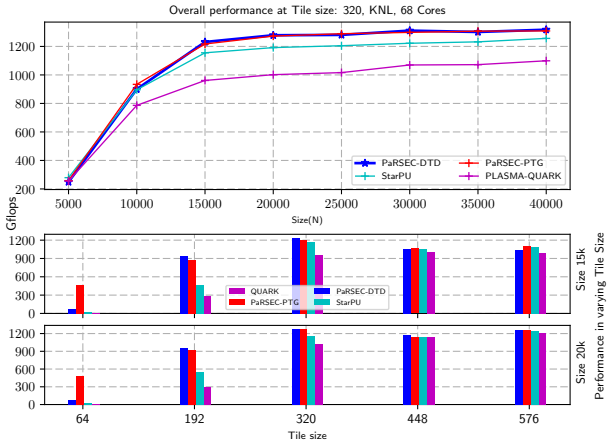

Figure 6: Performance of Cholesky factorization on KNL (shared memory)

the computational intensity of the target kernel (matrix-matrix multiplication) tolerate a lot of overheads in the runtimes. In the bottom figure, we fix the size of the matrices (14k and 20k) and investigate the impact of the tile size on the performance. For each chosen matrix size we varied the tile size from 64 to 576 incrementing by 128 at each step. The results is that without modifying the total amount of computations needed to solve the problem, we are decreasing the granularity of each task and therefore increase the number of tasks (for Cholesky there is a cube relationship), and as a result we increase the task-management stress on the different runtimes. We see that all runtime based on PaRSEC, PTG and DTD, perform better for small tasks, certainly due to a more careful implementation of the base runtime. In addition, PTG is favored by the fact that it does not build a task-graph at runtime unlike the others and as a result has a lower task management overhead.

Figure 6 shows the performance of Cholesky on KNL, where the core count is 3 times higher and the frequency of each processor is almost half of Haswell. The peak performance of GEMM kernel on this machine is 2 TFlops. In the top plot we see that PTG and DTD both perform similar at tile size 320 where StarPU does slightly lower and QUARK seems to suffer a little. If instead of increasing the problem size, we fix it and do a tile tuning experiment to assess the behavior of the different runtimes under stress, a different picture emerges. Clearly, larger tile sizes (and directly tasks execution duration) lead to similar results for all runtimes. However, when the tile size decrease, we see a similar result to the Haswell experiment, both PaRSEC DSL, PTG and DTD, outperform all the other runtimes. For information at tile size of 64, PaRSEC DTD is 3x faster than the other runtimes, where PTG is about 6x faster than DTD, due to it's efficient task handling, a smaller number of known tasks, and a more streamlined scheduling.

Distributed Memory. We used a QR factorization on 128 nodes, 2048 cores, on Stampede, to perform a problem scaling test (the total number of computing resource remains constant while the problem size increases). Out of the 4 runtimes only 2 provide a distributed QR, but we have added ScaLAPACK, which represent the current state-of-art algorithm on this setup. Figure 7 show that for matrix size of up to $120kx120k$ both PaRSEC DSL, PTG and DTD, outperform ScaLAPACK by a significant factor (up to $3x$ for small matrices). Once the problem size reaches saturation ($280kx280k$), ScaLAPACK catches up with PaRSEC, and both asymptotically converge toward the machine peak. These results show that, at least up this number of processes, the QR performance of PTG and DTD are equivalent, highlighting a similar scalability for both PaRSEC-DSL.

To further assess the scalability of DTD, we performed a weak scaling test on Cholesky and QR varying the number of cores from 16 to 2304 cores. Here, the problem size is determined by the number of process taking part in the execution while the workload per process/node is kept constant. The workload per node for both factorizations is kept constant at a matrix size of $20kx20k$ and the final total size in both case was $240kx240k$. The data was distributed in a block-cyclic way across a PxQ processors grid. From 1 to 4 processes we see a drop in performance as communications will introduce latencies. However, as P increases its impact on the ratio of computation/communication becomes negligible, which is why the performance stabilizes. The QR factorization is less impacted by the process grid, and we observe good scalability for both PTG and DTD. For Cholesky factorization DTD is 5% slower than PTG at 144 nodes. PTG is able to extract and realize collective communication patterns, where DTD lack such capability due to the way the task graph is discovered by different nodes (each process might have discovered only a portion of the entire task graph).

5 CONCLUSION

In this paper, we presented a new task insertion extension for PaRSEC, Dynamic Task Discovery, supporting shared and distributed memory environments. We highlighted the differences with existing task insertion paradigms, described and implemented several automatic runtime-level optimizations, and analyzed the new paradigm's performance using a set of widely-used dense linear algebra algorithms. The result shows good scalability and comparable result to PTG in most cases and, where comparable benchmarks exist, consistently better performance compared to other runtime. We also discussed the benefits and drawbacks of DTD programming

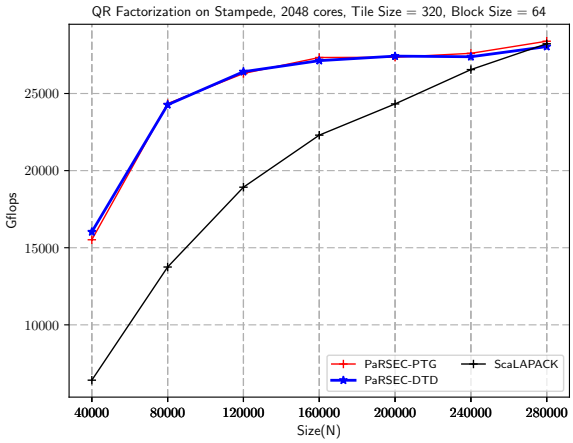

Figure 7: QR factorization on 128 nodes (2048 cores)

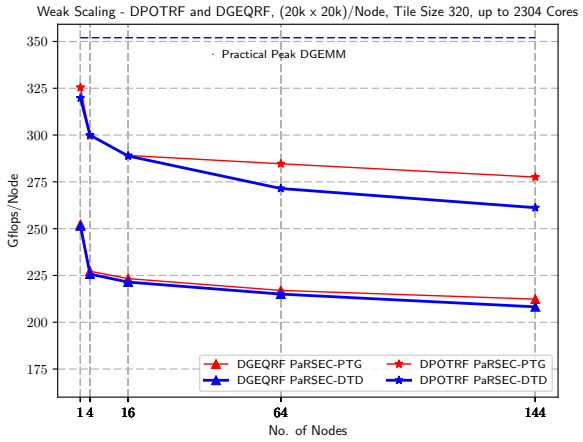

Figure 8: Weak Scaling of Cholesky and QR, up to 2304 Cores

model compared to PTG. This feature opens up the opportunity for application developers to build applications using multiple programming API (currently DTD and PTG) over the same runtime, merging in the same application multiple programming models with complementary capabilities. It also highlights the opportunity to develop specialized DSL over the PaRSEC runtime, without making compromises regarding the performance of the resulting applications.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] [n. d.]. TEXAS ADVANCED COMPUTING CENTER. ([n. d.]). https://www.tacc.utexas.edu/

[2] 2013. OpenMP 4.0 Complete Specifications. (2013). http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[3] Emmanuel Agullo, CÃĪdric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. 2012. A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. *GPU Computing Gems Jade Edition* (2012), 473âĂŞ484. https://doi.org/10.1016/b978-0-12-385963-1.00034-4

[4] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. 2014. Harnessing Supercomputers with a Sequential Task-based Runtime System. 13, 9 (2014), 1–14.

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. https://doi.org/10.1109/SC.2012.71

[6] L Blackford, J Choi, A Cleary, E D'Azevedo, J Demmel, I Dhillon, J Dongarra, S Hammarling, G Henry, A Petitet, K Stanley, D Walker, and R Whaley. 1997. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719642

[7] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, and et al. 2011. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (2011). https://doi.org/10.1109/ipdps.2011.299

[8] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack Dongarra. 2013. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering* 99 (2013), 1. https://doi.org/10.1109/MCSE.2013.98

[9] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, and Jack Dongarra. 2012. *From Serial Loops to Parallel Execution on Distributed Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 246–257. https://doi.org/10.1007/978-3-642-32820-6_25

[10] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. 2012. Productive programming of GPU clusters with OmpSs. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012* (2012), 557–568. https://doi.org/10.1109/IPDPS.2012.58

[11] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* 35, 1 (2009), 38 – 53. https://doi.org/10.1016/j.parco.2008.10.002

[12] M. Cosnard, E. Jeannot, and T. Yang. 1999. SLC: Symbolic scheduling for executing parameterized task graphs on multiprocessors. In *Proceedings of the 1999 International Conference on Parallel Processing*. 413–421. https://doi.org/10.1109/ICPP.1999.797429

[13] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. 2014. PTG: An abstraction for unhindered parallelism. *Proceedings of WOLFHPC 2014: 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Stor* (2014), 21–30. https://doi.org/10.1109/WOLFHPC.2014.8

[14] A. Danalis, H. Jagode, G. Bosilca, and J. Dongarra. 2015. PaRSEC in Practice: Optimizing a Legacy Chemistry Application through Distributed Task-Based Execution. In *2015 IEEE International Conference on Cluster Computing*. 304–313. https://doi.org/10.1109/CLUSTER.2015.50

[15] Jiri Dokulil, Martin Sandrieser, and Siegfried Benkner. 2016. Implementing the Open Community Runtime for Shared-Memory and Distributed-Memory Systems. *Proceedings - 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016* (2016), 364–368. https://doi.org/10.1109/PDP.2016.81

[16] T. Heller, H. Kaiser, and K. Iglberger. 2013. Application of the ParalleX execution model to stencil-based problems. *Computer Science - Research and Development* 28, 2-3 (2013), 253–261. https://doi.org/10.1007/s00450-012-0217-1

[17] H. Jagode, A. Danalis, G. Bosilca, and J. Dongarra. 2016. *Accelerating NWChem Coupled Cluster Through Dataflow-Based Execution*. Springer International Publishing, Cham, 366–376. https://doi.org/10.1007/978-3-319-32149-3_35

[18] Martin Tillenius. 2015. SuperGlue: A Shared Memory Framework Using Data Versioning for Dependency-Aware Task-Based Parallelization. *SIAM Journal on Scientific Computing* 37, 6 (2015), C617–C642. https://doi.org/10.1137/140989716

[19] Sean Treichler, Michael Bauer, and Alex Aiken. 2014. Realm: An Event-based Low-level Runtime for Distributed Memory Architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 263–276. https://doi.org/10.1145/2628071.2628084

[20] Asim Yarkhan. 2012. Dynamic Task Execution on Shared and Distributed Memory Architectures. December (2012). http://trace.tennessee.edu/utk