

Dynamic Task Discovery in a Data-Flow, Task-Based Runtime System

A Dissertation Presented for the
Doctor of Philosophy
Degree

The University of Tennessee, Knoxville

Reazul Hoque

December 2019

© by Reazul Hoque, 2019
All Rights Reserved.

To my parents Prof. Dr. Md. Gofranul Hoque and Ferdousi Rezwan, my wife Sadika Amreen and my siblings Rehnuma Sharmin, Tazmina Sharmin and Izazul Hoque for their relentless support, continuous encouragement and constant love.

Acknowledgments

I will start by thanking my advisor Dr. Jack Dongarra for giving me this great and rare opportunity to work at Innovative Computing Lab. I'm greatly indebted to my co-advisor, Dr. George Bosilca. I would like to thank him for introducing me to High Performance Computing and teaching me all about it. He has been a friend, a great mentor and a very patient guide. It would not have been possible to complete this journey without him. I would like to acknowledge my other mentors Dr. Anthony Danalis, Dr. Thomas Herault, Dr. Aurelien Bouteiller and Dr. Damien Genet for their support and guidance throughout the whole program. My first year, which is the toughest, was made manageable by Dr. Danalis and I would like to specially express my gratitude to him.

Innovative Computing Lab is full of wonderful and competent people and without the support of each of them it is impossible to have a journey as great as the one I had. I would like to thank Tracy Rafferty for all her help and for making my day in 2013 with an offer of research assistantship at ICL, Teresa Finchum and Leighanne Sisk for taking care of all and any issues I had with the university, Terry Moore for providing all the wisdom and Sam Crawford for scheduling all the Friday lunch talks and providing all the guidance regarding IGMCS.

I am grateful to my committee members Dr. Michlea Taufer, Dr. Michael Berry and Dr. Dmitry Liakh to agree to serve in my PhD committee. I appreciate their time and all their guidance with the dissertation.

Thananon Patinyasakdikul (Arm) has been a great friend and my partner in crime throughout my tenure at University of Tennessee. We started at the same time and shared a lot of ups and downs together. He has been a constant good company that aided me greatly. I would like to acknowledge my "senpai" Dr. Chongxiao Cao, Dr. Wei Wu and Dr. Khairul

Kabir. I learned a great deal from them. I would like to thank my office mates Xi Luo and Yu Pei for always being awesome and helpful. I would like to thank my friends from ICL and UTK, Dong Zong, Qinglei Cao, Kapil Agrawal and Mohit Shukla for always being a good friend.

My wife, Sadika Amreen has been a constant source of inspiration for me. We met each other during our PhD journey and I couldn't be more grateful to the almighty for making me so lucky. This journey was so much fun because she was there by my side and because we did it together.

Finally, I would like to express my heartfelt gratitude towards my whole family for always being there and for always inspiring and believing in me. My father Gofranul Hoque and my mother Ferdousi Rezwan has been my greatest inspiration for the inception and completion of my PhD. My siblings Rehnuma Sharmin, Tazmina Sharmin and Izazul Hoque has always been the biggest support system throughout my life. I am forever grateful to my family for their love and to have been part of such a wonderful family.

Abstract

The successful utilization of the modern configuration of the heterogeneous many-core architectures with complex memory hierarchies is a challenge for many application developers. Portability and performance of existing and new applications are the key challenges scientific application developers are continuously facing. Many evolutionary solutions have been proposed, including ones that seek to extend the capabilities of the current message passing paradigm 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 method of programming such a **Runtime** is important, as that directly affects the productivity of the developers and the performance of the applications. This work extends the capability of one of such runtime, the Parallel Runtime Scheduling and Execution Controller (**PaRSEC**), to the novel programming approach of allowing users to insert task in the **Runtime** by writing sequential code. This programming model is called Dynamic Task Discovery (DTD), which discovers tasks dynamically at runtime and uses optimized graph unrolling techniques to accommodate applications with large task graphs.

In this work, **PaRSEC**'s capability is extended by providing a new programming model, DTD. Bottlenecks of this programming model are identified and solutions to overcome its limitations are proposed. The performance of the implementation of DTD on top of dense linear algebra workload is analyzed at scale, where DTD has shown excellent results in distributed memory: $2.3\times$ – $1.3\times$ better performance at 128 nodes for QR factorization compared to ScaLAPACK and in shared memory, $4\times$ – $5\times$ better performance for Cholesky factorization compared to other runtimes, StarPU and QUARK. DTD was also evaluated via the coupled-cluster method of state of the art quantum chemistry application NWChem,

where it performed remarkably well among all considered **Runtimes** at scale of 128 nodes. The hope is that the concept and the development of DTD, the detailed evaluation of its practical performance at scale, the analysis of the theoretical limitations of it, the thorough study and classification of various task-based **Runtime** systems, and the design, implementation and evaluations of the chosen **Runtimes** on micro-benchmarks will help the broad scientific application developer community.

Table of Contents

1	Introduction	1
1.1	Overview	1
1.2	Motivation	3
1.2.1	Challenges of Parallel and Distributed Computing	3
1.2.2	State of Current Task-Based Runtime Systems	8
1.3	Thesis Statement	10
1.4	Contributions	10
1.5	Dissertation Organization	12
2	Background	13
2.1	Dataflow Programming Model	13
2.2	Parallel Runtime Scheduling and Execution Controller	14
2.3	Tile-Based Dense Linear Algebra Routines	16
2.4	Related Work	17
2.4.1	Shared-Memory Runtime Systems	18
2.4.2	Distributed-Memory Runtime Systems	20
2.4.3	Landscape of Task-Based Runtime Systems	22
3	Dynamic Task Discovery	24
3.1	Overview	24
3.2	Parameterized Task Graph	25
3.3	Dynamic Task Discovery in ParSEC	27
3.3.1	Challenges and Optimization of DTD	30

3.4	Evaluation of DTD	37
3.4.1	Evaluation of generic task management cost	37
3.4.2	Evaluation on Shared Memory	39
3.4.3	Evaluation on Distributed Memory	41
3.4.4	Evaluation on Heterogeneous Architecture	44
3.5	Overhead of DTD	46
3.5.1	Overhead of DTD compared to PTG	46
3.5.2	Upper Bound of DTD's Performance	49
3.5.3	Owner-Tracks-Future Scheme	50
3.6	Integration of DTD with Existing Applications	57
3.6.1	Chameleon	57
3.6.2	SLATE	58
3.7	Collaborative Efforts Using DTD	62
3.7.1	Electronic Structure Methods	62
3.7.2	INTERTWinE	63
3.8	Conclusion	63
4	Current State and Performance of Runtimes	64
4.1	Introduction	64
4.2	Related Work	66
4.3	Terminology	67
4.4	Runtime Features and Characteristics	68
4.5	The Survey	69
4.5.1	PaRSEC	70
4.5.2	Legion	71
4.5.3	StarPU	72
4.5.4	High Performance ParalleX (HPX)	74
4.5.5	Charm++	74
4.5.6	Uintah	76
4.5.7	STAPL	77

4.6	Benchmark	78
4.6.1	Embarrassingly Parallel Tasks	79
4.6.2	Chain of Tasks	80
4.6.3	Stencil Operation	81
4.7	Productivity	82
4.8	Conclusion	84
5	Conclusion and Future Work	86
5.1	Overview	86
5.2	Summary of Contributions	87
5.3	Future Work	90
	Bibliography	91
	Appendices	101
A	PaRSEC DTD API and Users Guide	102
B	Publications	114
	Vita	115

List of Tables

3.1	Summary of Softwares used	39
4.1	Summary of the features and characteristics of each Runtime	70
4.2	Software Specification	79

List of Figures

1.1	Supercomputer Performance Trend	2
1.2	Summit’s Heterogeneous Architecture	6
1.3	Example of dependency between tasks	8
2.1	PaRSECs Framework	15
2.2	Tile based Cholesky Factorization	18
3.1	Sample Cholesky DAG from Listing 3.2 for <i>total</i> = 2	31
3.2	Performance and memory footprint per node for different window size	33
3.3	Sample Trimmed DAG	36
3.4	Memory-footprint in each node with and without Trimming	36
3.5	Cost of each task in DTD	38
3.6	Performance of Cholesky factorization on Haswell (shared memory)	40
3.7	Performance of Cholesky factorization on KNL (shared memory)	42
3.8	QR factorization on 128 nodes (2048 cores)	43
3.9	Weak Scaling of Cholesky and QR, up to 2304 Cores	45
3.10	Performance of Tile-based GEMM on top of P-100 Nvidia GPU	46
3.11	Overhead test in PaRSEC	48
3.12	Scalability test of DTD	51
3.13	Alternate Scheme for DTD	52
3.14	Sample Task-Graphs	55
3.15	Panel-by-Panel Cholesky Factorization	59
3.16	Multicast by POTRF task	60
3.17	Performance Result of Cholesky Factorization	61

4.1	Sample Task Graph	68
4.2	Performance result of Embarassingly Parallel Tasks Benchmark	80
4.3	Chain of Tasks, each task in different machine	81
4.4	Performance result of Chain of Tasks Benchmark	82
4.5	Task dependencies in Stencil Operation	83
A.1	PaRSEC Data Collection at different granularity	106
A.2	Example of a chain of tasks	108
A.3	Ping-Pong Example	111

Chapter 1

Introduction

1.1 Overview

Simulations of various extremely complex real life systems are required to assess their feasibility and analyze their behavior. For example, applications such as weather forecasting to predict the condition of the atmosphere for a given location and time, climate research to understand weather condition averaged over a period of time, and fluid dynamics to model air flow around airplanes or automobiles, have vast numbers of variables and equations that need to be solved through a large number of steps. Simulation also helps in cases where real systems cannot be engaged, perhaps due to accessibility or due to safety concerns, or simply because the systems do not exist. Results of such simulations push the scientific boundaries and allow us to have a better understanding of our surroundings.

Compute- and data-intensive simulations require high-performing hardware that can deliver results accurately and efficiently. Therefore, supercomputers have been traditionally used for various scientific simulation purposes. The early supercomputers, introduced in the 1960s, could perform around 200×10^6 floating point operations per second (FLOP/s). Supercomputers have consistently improved in performance since then. Figure [1.1](#) shows how the peak performance of supercomputers has improved between 1940 to 2019. This performance gain was made possible with continuous improvement in hardware coupled with adaptation in software.

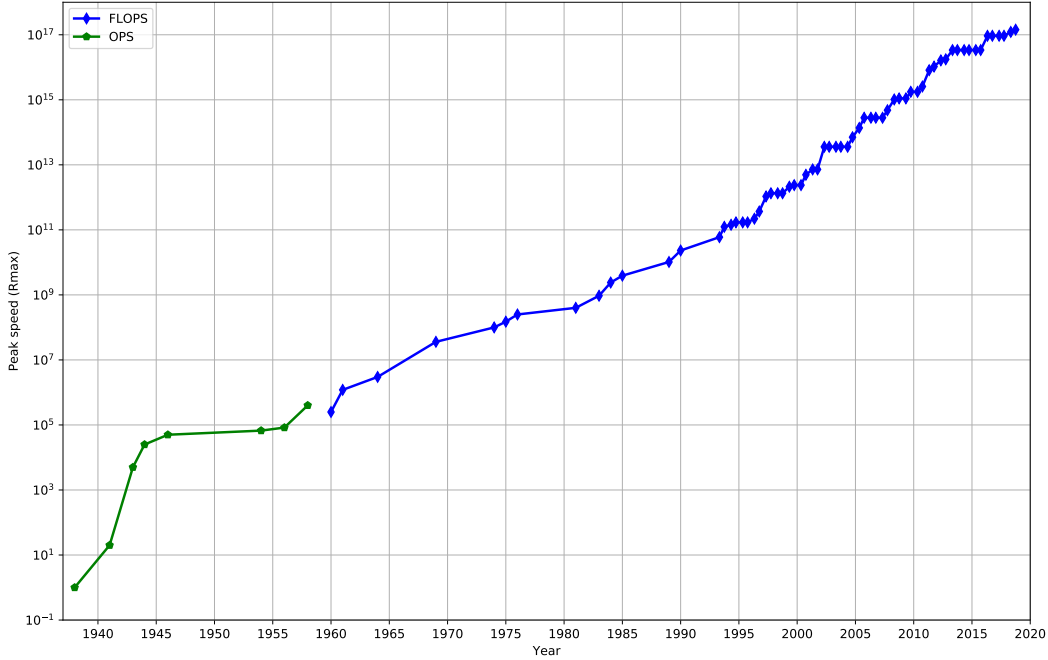


Figure 1.1: Supercomputer Performance Trend*

Computing hardware, also known as processors, are made out of transistors. The advancements we witnessed were made possible by making these transistors smaller and by discovering new materials to improve the electric flow, resulting in faster switch time. Around the early 2000s we reached the physical limit of making smaller transistors, which left us with the only option of increasing processor counts to gain performance Kindratenko and Trancoso (2011), HPCWIRE (2013). This trend made it mandatory for software to be rewritten/developed in a way that allows us to exploit the parallelism offered by the processors. As writing parallel applications is an extremely challenging task compared to more traditional sequential programming, it acts as a barrier to users who are working to improve scientific applications. Therefore, introducing middlewares (task-based Runtime systems) that relieve application developers from programming details such as expressing parallelism, managing data locality, managing communication between processes, managing data transfers between devices etc., greatly boosts the development of high-performing

*Figure taken from <https://en.wikipedia.org/wiki/Supercomputer>

scientific applications. Ideally, we want developers to focus on domain science—and not on the computer science aspect of the code development—and this requires an interface to the middleware that does not make the application developers shift from sequential programming but still retains enough information about the parallelism available in the algorithms to be able to deliver a significant percentage of the execution platform performance. This dissertation’s work is based on one of such middleware, **PaRSEC**, allowing a highly productive and easy to use programming interface and providing developers with the means to achieve code portability and performance. The resulting programming paradigm, the Dynamic Task Discovery (DTD), as will be explained in detail later in this dissertation, allows scientific application developers to express their application sequentially using a paradigm that can then be automatically unfolded on massively parallel supercomputers.

In the following sections I elaborate on what task-based **Runtime** systems are, why they are important, and how DTD helps users program them. The goals that were set to achieve through this research are also outlined.

1.2 Motivation

This section discusses in detail the current challenges of writing parallel applications capable of securing most of the performance offered by modern supercomputers, and provides a comprehensive list of existing—both research and production quality—solutions and describes how their limitations provided the underlying motivation to search for a better, more portable, efficient, and developer-friendly solution. The landscape of task-based **Runtime** systems and motivations behind developing benchmarks to start a process of comparing various task-based **Runtime** systems are also discussed.

1.2.1 Challenges of Parallel and Distributed Computing

High-performance computing (HPC) is moving towards processors with increasing numbers of cores and deeper hierarchies of memory. Use of graphics processing units (GPUs) for general-purpose computing is becoming more mainstream to get as much performance as possible. This trend is evident in the list of top 500 supercomputers in the world, where

the highest core count in a supercomputer has gone up from around 100 thousand in 2005 to around 10 million in 2018, according to the Top 500 [Top-500 \(2018\)](#). Theoretically, these advances allow scientific application developers to get better performance. However, attaining the practical performance gain offered by these heterogeneous hardware is extremely challenging, as discussed below.

- **Expressing Parallelism:** More computing resources to solve a single problem entails dividing the problem into smaller independent sub-problems, when possible, for efficient use of the available resources. By efficiently using multiple processors at the same time on the same problem, we can reduce the total time needed to come to a solution. This introduces sections in the program that might be independent of some operations but dependent on others. This is called dependency, where, to safely execute a section of the program, some other section(s) needs to be completed. To identify instructions/operations in a sequential block of code that can be executed in parallel without altering the outcome of the sequential execution, we have to make sure they satisfy the Bernstein conditions [Bernstein \(1966\)](#). The possible dependency types between two instructions/operations $s1$ and $s2$ can be categorized as:

– Flow Dependency:

$$s1 : \boxed{A} = 1 + 2;$$

$$s2 : B = \boxed{A} * 3;$$

In the above example, instruction $s2$ is flow dependent on $s1$ because the variable A used in $s2$ is the output of $s1$.

– Output Dependency:

$$s1 : \boxed{A} = x + 2;$$

$$s2 : \boxed{A} = 3;$$

In the above example, instruction $s2$ is output dependent on $s1$, as the variable A is the output of both $s1$ and $s2$ instructions.

– Anti Dependency:

$$s1 : B = \boxed{A} + 2;$$

$s2 : \boxed{A} = 3;$

In the above example instruction, $s2$ has its output, A , overwrite an input variable of $s1$, $s2$ therefore has dependency on $s1$.

Two instructions $s1$ and $s2$ can only be executed in parallel if $s1$ and $s2$ do not have any of the above dependencies. Expressing and managing these dependencies explicitly is cumbersome and error-prone [McKenney et al. \(2009\)](#). Failing to ensure correct expression of parallelism will thus result in computation on incoherent or the wrong version of the data.

- **Communication Management:** Heterogeneous machines include hardware that have multi-core general-purpose computing processors with one or more accelerators like GPUs and/or Accelerated Processing Units (APUs). These accelerators are connected to the central processing unit (CPU, the general purpose processor) with high-speed buses. Each of the heterogeneous machines can be considered as one unit of resource (also known as a node). Supercomputers today are made of thousands of these nodes, where each of these nodes are connected with high-speed interconnects. Figure 1.2 shows the architecture of the current fastest supercomputer in the world, Summit[†]. Each socket is composed of a POWER9 processor and 3 NVIDIA V100 GPUs—the figure shows one socket and each node has 2 of those sockets. Scientific application developers intending to run simulations or experiments at a large scale will need to write programs that are distributed. This means that the nodes do not share memory and, thus, will need to communicate to share data among themselves. This increases the challenge of writing programs for large scale simulations, as now users also need to manage communication among these nodes explicitly. The distance between the accelerators and CPU inside each node are limited, but the distance between nodes can vary; since the speed at which we can transfer information is limited, an increase in distance means an increase in latency. Given this limitation, users will have to be very careful about when to communicate and how much to communicate, which only adds to the difficulty.

[†]<https://www.top500.org/lists/2019/06/>

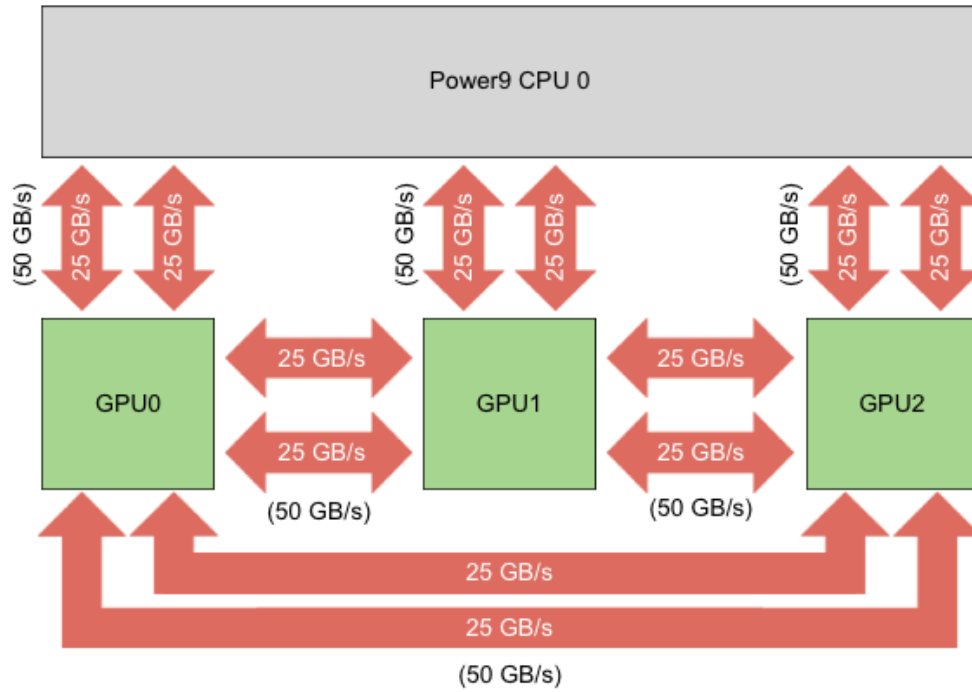


Figure 1.2: Summit’s Heterogeneous Architecture[‡]

- **Portability:** Computing resources are produced by different vendors, each of them providing software for basic operations on that hardware. If application developers want to run their application on a machine (supercomputer) that has new hardware (maybe a new accelerator), they have to augment their application to get optimal performance. This adds the challenge of performance portability across multiple generations of hardware, where major effort needs to be put into porting the existing application to new hardware for optimal performance.

[‡]Figure taken from <https://www.olcf.ornl.gov/for-users/system-user-guides/summit/summit-user-guide/>

Need for a High-Performing Expression of Parallelism

By the 1990s, application programming interfaces for massively parallel processors (MPPs) and clusters converged to a single standard known as the Message Passing Interface (MPI) [Forum \(1993\)](#). For shared memory multi-core computing, a similar process followed and converged around two main standards—Open Multi-Processing (OpenMP) [Dagum and Menon \(1998\)](#) and Pthreads [Butenhof \(1997\)](#). From then on $MPI + X$, where X can be OpenMP, Pthread or other shared memory manager, became a widely adopted strategy for programming clusters. In a $MPI + X$ paradigm users need to express parallelism, manage computation and communication (resources), and provide a mapping between these two which in other words mean that the parallelization of an application is being explicitly done by the developer. The burden of this responsibility only increases with an increase in core count, heterogeneity, and application size, and it ($MPI + X$) does not address the challenges described above.

Task-based **Runtime** systems offer solutions to the problems of resource management, communication management, and portability by abstracting the computation and communication at varying levels. However, all these are contingent upon the effective expression of parallelism. For instance, basic **Runtime** systems allow users to abstract computations and communications in the form of tasks and ask the user to explicitly manage their ordering. Advanced **Runtimes** offer programming paradigms to users to allow them to express their application in the form of computations (tasks) that produce and consume data. This allows the **Runtime** to analyze the data flow between tasks and form a directed acyclic graph (DAG), where the nodes are tasks and each edge a dependency between two tasks, as shown in Figure 1.3a. This makes dependency resolution and communication in distributed-memory systems implicit, relieving the users from explicit task ordering and communication, as shown in Figure 1.3b. My motivation behind this research is to build a programming model that simplifies the difficult task of expressing parallelism for the end scientific application developers while preserving performance and portability.

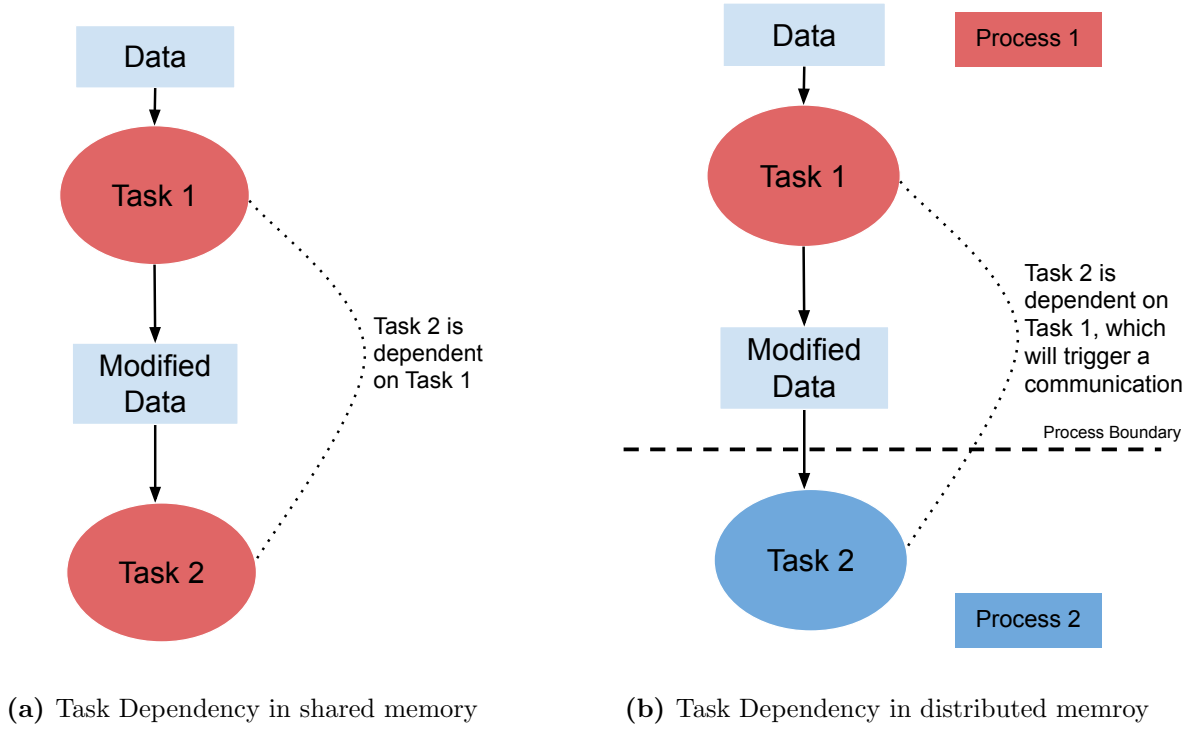


Figure 1.3: Example of dependency between tasks

1.2.2 State of Current Task-Based Runtime Systems

Even though the last decade has seen a sharp increase in interest and development in task-based **Runtime** systems, the standardization of terminology and concepts are missing from its landscape. The work presented in this dissertation aggregates terminologies used by different **Runtimes** that describe similar concepts, and presents a comprehensive study of the available task-based **Runtime** systems supporting distributed memory, their differences and their particular features. The different overheads shared by all task-based **Runtimes** are also characterized as follows:

- **Management of Tasks:** Task-based **Runtime** systems allow users to insert tasks to perform computations. Each task has a certain lifetime, and in certain cases multiple states (ready, not-ready, sleeping etc.). **Runtimes** need to manage meta-structures for different states of a task and for the overall book-keeping. Depending on the implementation, different **Runtimes** will have different overheads of managing tasks and thus can support different granularity of tasks. Fine-grained tasks will have shorter

duration, and performance of applications with smaller tasks will be more sensitive to such overhead.

- **Resource Management:** Runtime systems abstracts the underlying computing resource to the application developers. One of the fundamental feature to assess the performance of a Runtime is the efficiency by which the Runtime manages the computing resource.
- **Ordering of Tasks:** There are two broad kinds of task-based Runtime systems. The first kind is a low-level Runtime, where users need to explicitly manage the ordering of tasks. The other kind is a high-level Runtime, where the ordering of tasks is inferred from the expression of parallelism. The inference introduces an overhead for high-level Runtimes, making task management more costly for the Runtime but in turn makes building parallel applications much easier.
- **Data Transfers:** For computation in heterogeneous and distributed environments, all Runtimes need to transfer data from one device to the other. In certain cases these communications are explicitly performed by users; in other cases these communications are also inferred like the task ordering. In both cases, the management cost of these data transfers and the efficiency at which they are performed are an important metric with which to assess the performance of a Runtime.

Need for Generic Assessment of Task-Based Runtimes

To assess the overheads described above, an experimental setup needs to be designed that can test multiple aspects of the Runtimes, such as task management, task ordering, and data transfers. While there are established benchmark suites [Bienia \(2011\)](#) designed for measuring some aspects of task-based Runtime systems supporting shared memory, a comprehensive benchmark suite does not exist for task-based Runtime systems supporting distributed memory [Hoque and Shamis \(2018\)](#). Therefore, generic micro-benchmarks to assess the same overheads of different Runtime systems are critical and useful for application developers to choose a Runtime for new or existing applications.

1.3 Thesis Statement

The main objective of this dissertation is to demonstrate a programming model that makes programming task-based `Runtime` systems simple while not sacrificing their performance. The resulting programming paradigm allows users to write sequential code using provided API, offers portability and efficiency, and facilitates exposing parallelism present in scientific applications. This dissertation addresses the challenges of having an implementation supporting applications of all scales. Theoretical limitations of this programming model are also thoroughly studied and presented.

The other objective of this dissertation is to present a detailed study identifying key features of task-based `Runtime` systems and a classification of them based on the identified features. Design, implementation, and performance evaluation of several simple benchmarks to assess the chosen `Runtimes` are also examined.

1.4 Contributions

Dynamic Task Discovery (DTD): A simple interface for developing parallel scientific applications

An extension to a task-based `Runtime` system is proposed by offering an interface that makes it easy to describe the parallelism available in scientific algorithms, increasing the productivity of scientific application developers without sacrificing the resulting parallel applications' performance. The programming model is designed, implemented, theoretically assessed and thoroughly tested on a wide variety of distributed hardware.

- **Design, Implement and Test DTD:** In this research, a new programming model to program task-based `Runtime` systems, Dynamic Task Discovery (DTD), is proposed and designed. DTD allows users to express their application using sequential code while keeping communication transparent. This interface, based on the data flow among tasks, determines the correct ordering of tasks and allows the underlying task-based `Runtime` system to manage device-specific workload and scheduling of

these tasks. Details of required optimizations which makes this programming model suitable for massively parallel applications—such as tile-based dense linear algebra algorithms—are also presented. Furthermore, the implementation details of a dense linear algebra routine, Cholesky factorization, is provided to demonstrate utilization of the proposed interface built on top of a data flow task-based `Runtime` system, the Parallel Runtime Scheduling and Execution Controller (PaRSEC) [Bosilca et al. \(2013\)](#). A thorough evaluation of the performance of DTD in both shared- and distributed-memory systems is presented. The necessary `Runtime`-level optimizations required to enable the resulting programming paradigm to scale are also described. The performance of DTD with other `Runtime` systems offering similar capabilities is then compared.

- **Analyze Theoretical Limitations of DTD:** Another contribution of this study is the analysis of the limiting features of this programming paradigm. A detailed inspection of the various schemes using which DTD can be implemented is presented, along with comparisons between them providing examples such as the advantages and disadvantages of both schemes of implementation, and discussion of scenarios where one would be better than the other.

A generic benchmark suite to assess performance of task-based `Runtime` systems

A study was conducted on existing benchmarks and skeleton assessors to identify a set of features allowing the classification of task-based `Runtime` systems with support for distributed memory. As a result of this study a micro-benchmark suite was designed and implemented to enable assessment of common features of `Runtimes` by application developers.

- **Study of Distributed Task-Based Runtime Systems:** A thorough study of past and current task-based `Runtimes` with and without support for distributed-memory systems was conducted. Their features important to application developers (because of performance, scalability, portability, and productivity aspects) were identified, and then used to classify a number of `Runtime` systems with an active developer base.

- **Design, Implement, and Test Micro-Benchmarks:** Overheads incurred during normal execution by any task-based `Runtime` system were identified, and a set of assessors were designed and implemented to evaluate the cost of these overheads. The resulting benchmark was used on the selected `Runtime` systems and the performance result of the benchmarks is presented.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows:

- Chapter 2 introduces a historical background on dataflow programming. I discuss `PaRSEC`, upon which this work is based, as well as tile-based dense linear algebra algorithms, such as Cholesky Factorization, using which this work of DTD is evaluated. A literature review of recent and related work in the same field is then presented.
- Chapter 3 describes the design of the DTD programming model, provides details on different optimizations required to make this interface work and perform optimally in distributed systems, and shows the various experimental designs and detailed analysis of the performance results. Finally, this chapter presents an in-depth analysis of the theoretical limitations of this paradigm.
- Chapter 4 presents a survey of different task-based `Runtimes` supporting distributed-memory systems. A classification of the chosen `Runtimes` is discussed and their performance on micro-benchmarks shown.
- In Chapter 5 the contributions of this work are summarized, its limitations are outlined, and a discussion of the potential future directions of this research is presented.

Chapter 2

Background

This chapter introduces the dataflow programming model and provides some history on its evolution. This chapter further discusses why dataflow programming models are important in providing solutions to current challenges in high-performance computing. The discussion then shifts to the dataflow task-based **Runtime** system, **PaRSEC**, on which this work is based, and how it employs the dataflow programming model. Tile-based dense linear algebra routines, which is the main workload for assessing performance and correctness of this work, are also introduced, and finally, recent work in task-based **Runtime** systems supporting both shared- and distributed-memory machines are discussed in detail.

2.1 Dataflow Programming Model

Data-flow architecture is different from the more traditional control-flow (Von Neumann) architecture, where the executability of an instruction is solely dependant on the availability of the input data of that instruction. The principal motivation behind this architecture is to exploit large-scale asynchronous parallelism. The research on dataflow architecture dates back to 1960s when it was introduced and formalized by the work of **Karp and Miller (1969)** and **Rodriguez (1969)**. These efforts were mainly focused on coming up with alternative hardware architectures to classical Von Neuman architecture, as it was thought to be unsuitable to dataflow execution, but a survey of the experimental dataflow machines by **Veen (1986)** found out several difficulties of those machines to deal with fine-grained

parallelism like managing parallelism, load balancing, and the management of required data structures.

After an unsuccessful attempt to come up with a viable machine that uses the dataflow architecture, the research focus moved to dataflow-based algorithms on top of the Von Neumann architecture. This shift was accelerated as we hit the physical limit on the increase in frequency of a single processor, and the only option was to move to machines with many cores. This naturally meant we needed more parallel algorithms, which made the use of the dataflow programming model more widespread and prominent.

A dataflow programming model allows the movement of data from one instruction to the next and models a program as a series of connected instructions. Here, an instruction or a set of instructions can be considered as a task. Each task will have its input and output explicitly defined by the application developer. A directed acyclic graph (DAG) can be constructed depending on the defined input and output. A task becomes ready for execution when all its input data becomes ready. This model is inherently parallel, as parallel sections of a program can be easily figured out from the DAG modelling the instructions of the program, and is well suited for large distributed systems. A DAG is a set of vertices and edges, $D = (V, E)$, where each vertex, $v \in V$, represents a task and each edge, $(v_i, v_j) \in E$, a dependency between two vertices or tasks. If an edge (v_i, v_j) exists in E , then there exists a dependency between task v_i and task v_j . In this dissertation, DAG and task-graph will be interchangeably used.

2.2 Parallel Runtime Scheduling and Execution Controller

PaRSEC [Bosilca et al. \(2013\)](#) is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures, developed at the Innovative Computing Laboratory at the University of Tennessee, Knoxville. This work is implemented on PaRSEC which employs the dataflow programming model to schedule tasks efficiently on computing resources. Figure [2.1](#) shows the detailed framework of PaRSEC,

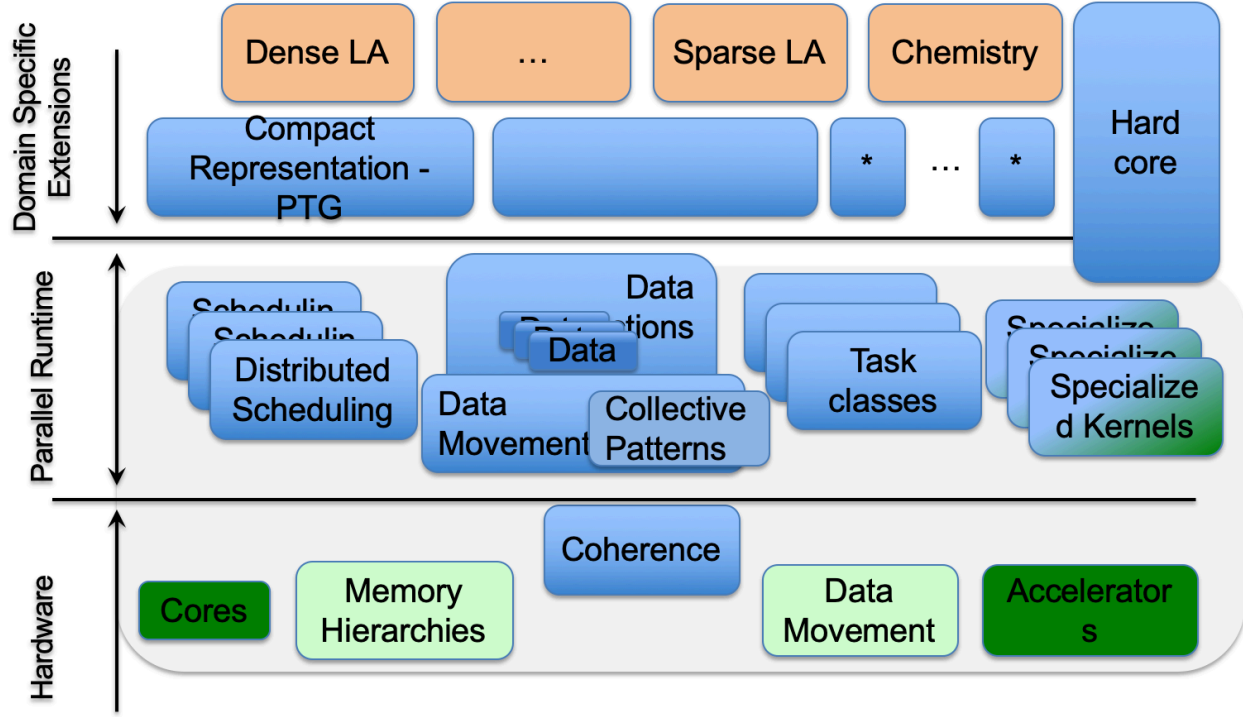


Figure 2.1: PaRSECs Framework

which consists of three layers. The bottom layer, which deals with the hardware, interacts with different kinds of architectures like the multi-core CPUs, deep memory hierarchies, and accelerators. The middle layer, which is the parallel Runtime, includes all the management required for abstracting a computing resource to an application developer like scheduler, communication engine, etc. The top layer, which includes domain-specific extensions, specifies the interfaces PaRSEC provides for developers to express their applications. The work being presented in this dissertation is on the top layer.

PaRSEC employs several domain-specific languages (DSLs) and interfaces to provide flexible domain-specific programming models to application developers. These DSLs create a dataflow model to create dependencies between tasks and exploits the available parallelism present in applications. PaRSEC is rich with many features aimed at helping developers describe their application to the Runtime correctly and efficiently.

Certainly the most exposed DSL, PTG, allows users to use a parameterized task graph (PTG) [Danalis et al. \(2014\)](#) 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 [Danalis et al. \(2014\)](#) in comparison to the widely used library, ScaLAPACK [Blackford et al. \(1997\)](#), or current state-of-the-art computational chemistry applications [Danalis et al. \(2015\)](#), [Jagode et al. \(2016\)](#). A complete example of a JDF is demonstrated in Chapter 3.

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, and 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 [Bosilca et al. \(2012\)](#). The main goal of my work is to investigate alternative approaches to express the parallelism available at the algorithmic level, in a format that can be handled with ease by a `Runtime`. Dynamic Task Discovery (DTD), is one such approach that removes the need of a pre-compiler, and therefore abolishes the loop-affine limitation.

2.3 Tile-Based Dense Linear Algebra Routines

In tile-based dense linear algebra [Buttari et al. \(2009\)](#) a $N \times N$ matrix is split into $NT \times NT$ tiles, where each tile is of size $B(N/B = NT)$. This essentially divides one matrix factorization into many smaller matrix factorization, some of which have to be executed sequentially and some can be executed concurrently. Here, each of the factorizations performed on a tile is considered a kernel or task. By varying the tile size we can easily tune the parallelism and the overhead to execute a task. A classic, widely used and

easy to understand example of a tile-based algorithm is the Cholesky Factorization. This factorization is used to solve a linear system $Ax = b$. This factorization has four different types of kernel/tasks: POTRF, TRSM, SYRK and GEMM, which are successively applied on the trailing sub-matrix at each step of the algorithm.

- POTRF performs an untiled version of Cholesky factorization of a diagonal tile of the input matrix and overrides it with the final elements of the output matrix.
- SYRK is a symmetric rank-k update, which updates to the diagonal tile of the input matrix.
- TRSM is a triangular system solve, which applies transformation computed by POTRF to an off-diagonal tile below the diagonal tile operated by the last POTRF of the same column.
- GEMM is a matrix-matrix multiplication, used to update tiles in trailing matrix.

An example of a Cholesky factorization on a matrix of 3×3 with a tile size of 1×1 is shown in Figure 2.2, where three steps of the tile-based algorithm are shown successively. Tile-based dense linear algebra routines have been extensively ported on top of task-based **Runtime** systems. Generally, the location (CPU, GPU, other devices) where a kernel will execute is dynamic and is chosen by the **Runtime**. For a task-based **Runtime** system like **PaRSEC**, application developers only need to focus on expressing the algorithm to the **Runtime**. **PaRSEC** will take care of the task-scheduling. In a distributed Cholesky factorization, users will need to distribute the data, and based on the data distribution **PaRSEC** will take care of the task placement. It automatically deploys tasks to the proper unit based on load balance and data locality. These tile-based algorithms are the main workload on which the implementation of this work is tested for scalability and performance.

2.4 Related Work

In the context of this work, only task-based **Runtime** systems that can efficiently handle fine-grained tasks with duration below the millisecond are referred. While there are similarities

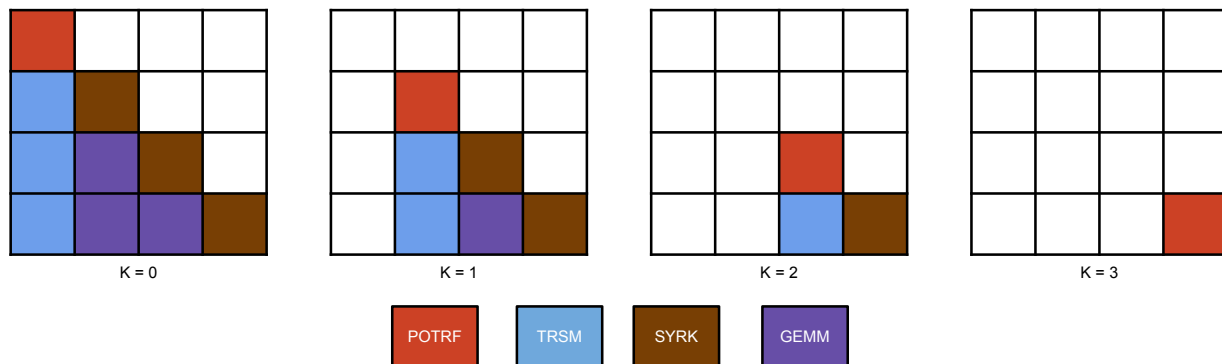


Figure 2.2: Tile based Cholesky Factorization

to workflow systems, task granularity is one of the most important metrics that sets them apart, imposing strict limits on what the runtime can and can't do with the time spent on the estimation and the scheduling of tasks. Thus, workflow systems where the usual task granularities are in the order of tens of seconds address a different type of problem and are beyond the scope of this dissertation.

2.4.1 Shared-Memory Runtime Systems

OpenMP [Dagum and Menon \(1998\)](#) is a widely used standard for writing shared-memory parallel applications. The standard allows users to define private and shared variables for each thread of a processor and provide pragmas to parallelize sequential constructs like *for*. It is supported by most mainstream *C* compilers. In version 3.0 of the OpenMP standard, task-level parallelism was first introduced and the task concept was extended with the introduction of the *depend* clause in the ensuing version. The *depend* clause allows applications to create tasks and express dependencies among them. More recent versions of OpenMP support heterogeneous memory systems, though users are responsible for data management. Users use *pragmas* instead of API to port current applications and to develop new ones. OpenMP does not support distributed-memory systems yet. Transparent data movement between devices and support for distributed systems are among the many advantage ParSEC DTD has over OpenMP.

OmpSs [Bueno et al. \(2012\)](#) is an effort from the Barcelona Supercomputing Center. It uses Nanos++ runtime to manage ordering of tasks. They had the initial proposals for task based directives in OpenMP and are a primary driver for the inclusion of advanced concepts that allows task-based parallelism. OmpSs uses different execution model from OpenMP and does not implement fork-join parallelism like OpenMP. It supports heterogeneous memory systems through leveraging native kernel implementations. Currently, they only support shared memory systems.

Cilk [Blumofe et al. \(1995\)](#) project provides a *C*-based system for multi-threaded parallel programming targeting shared-memory systems. Its implementation is compiler directed and provides a divide-and-conquer style parallelism that induces a fork-join style execution path. It employs work stealing to balance loads between the different threads on multi-core systems. The compiler-directed approach makes porting existing applications to Cilk rather simple and easy, but the parallelism is limited to the granularity that is exposed when functions are defined, and the fork-join style parallelism hinders better utilization of the available resources.

SuperGlue [Tillenius \(2015\)](#) employs data versioning to represent task dependencies. In this model, tasks depend on data rather than on other tasks. SuperGlue has support for shared-memory systems only. All the cores in the systems are assumed to be identical and to increase system throughput fairness or task deadlines are not considered by SuperGlue.

QUARK [Yarkhan \(2012\)](#), developed at the Innovative Computing Laboratory, is a task-based dynamic **Runtime** for the tile-based dense linear algebra library PLASMA [Buttari et al. \(2009\)](#). It exposes a simple API for users to insert tasks in the **Runtime** while expressing how each task will use each data. The **Runtime** then examines the data dependencies between the tasks and executes them in a way that preserves the sequential consistency of the insertion. QUARK only supports shared-memory systems.

Kokkos [Edwards et al. \(2014\)](#) is a *C++* library that provides a programming model to application developers targeting heterogeneous shared-memory architectures. It supports

fine-grained data parallelism within an execution space. The data movement among different devices are transparent and provides polymorphic data layout. It provides support for spawning task at runtime and creating explicit dependency between tasks as well. It is mainly developed at Sandia National Laboratories.

RAJA [RAJA](#) is a collection of *C++* abstractions being developed at Lawrence Livermore National Laboratory (LLNL). It mainly enables architecture portability for HPC applications. It is primarily focused on production mesh-based multi-physics application at LLNL. RAJA currently does not support the abstraction of task.

All the above **Runtime** systems are developed for HPC applications—but they only support shared-memory systems and this puts a lot of responsibility on application developers in case they want to run their application on distributed-memory systems. DTD offers task-based parallelism through dataflow and also supports distributed-memory systems.

2.4.2 Distributed-Memory Runtime Systems

Most applications running on distributed-memory systems use communication libraries to move data across node boundaries. The necessary communication support can be provided by standards like the Message Passing Interface (MPI) [Forum \(1993\)](#). This standard is most widely used as a two-sided process (both sides are involved)—meaning for a successful communication, the sender needs to perform a send operation and the receiver needs to perform a pairing receive operation. When using MPI alone for multi-process parallelism, the developers have a lot more responsibility as everything (data movement, synchronizations, etc.) has to be explicitly managed. This induces users to naturally develop bulk-synchronous programs as they are much simpler to write correctly. These limitations are the driving force behind the increase in efforts to find alternate programming models, like DTD.

Legion [Bauer et al. \(2012\)](#) is developed at Stanford University. It uses logical regions which represent partitioned real data to infer task dependency and uses GASNet [Yelick et al. \(2007\)](#) for inter-node communication. They expose APIs like DTD and are powered by the low-level event-based **Runtime Realm** [Treichler et al. \(2014b\)](#) underneath. It supports

heterogeneous architectures and works in both shared- and distributed-memory systems. It has shown good performance for the applications that were ported to it, but critical performance features like analysis on granularity of tasks at which the overhead of the `Runtime` would be evident are not available.

StarPU [Augonnet et al. \(2011\)](#) lets users submit tasks using its API and dynamically builds task-graph at runtime. It is developed by STORM research team of Inria, France. StarPU supports heterogeneous systems and has nascent support for distributed-memory systems. It profiles task duration and builds a model to be later used by smarter "HEFT" [Topcuoglu et al. \(2002\)](#) like schedulers to select appropriate processors on heterogeneous systems. StarPU and DTD closely resemble each other, where DTD offers features like recursive tasking, untying task insertion from specific thread, etc. that StarPU lacks.

Open Community Runtime (OCR) [OCR \(2016\)](#) is mainly a joint effort between Rice University and the Intel Corporation and was unveiled in the year 2012. It is an event-driven `Runtime`. OCR is fairly new and currently supports homogeneous architectures in distributed systems and uses Intel Threading Building Blocks to manage shared-memory parallelism. This is an event-driven `Runtime` which is intended to be used as a low-level `Runtime` by higher-level programming models. It does not support the dataflow model and thus lacks the capabilities DTD offers.

HPX The first implementation of ParallelX execution model is High Performance ParallelX (HPX) [Heller et al. \(2013\)](#). It is developed at Louisiana State University. This is a low-level runtime intended to be used by high-level programming models. This implies the task dependencies have to be explicitly created by the application developer. It does not support the dataflow model and lacks the capabilities DTD offers. It supports distributed-memory systems but does not yet support heterogeneous architectures.

Charm++ [Acun et al. \(2014\)](#) is one of the earliest `Runtime` systems. Its inception precedes MPI. It is based on concepts like fine-grained tasks to have a big number of

tasks per processing element, synchronous message-driven execution, and migrations of tasks to balance load. It is a low-level **Runtime** where the ordering of tasks has to be explicitly managed by the application developers; this makes it harder to program than DTD. Developers also have to use an API that is not based on any standard compiled language like *C*, *C++* to program Charm++ and this increases the difficulty to use it.

Uintah [de St. Germain et al. \(2000\)](#) is a domain-specific **Runtime**. It supports distributed-memory systems and is focused on adaptive mesh refinement (AMR) applications. It is limited to applications related to structured meshes and cannot deal with data of any shape and structure. DTD is not domain-specific and is more general than Uintah.

STAPL [Papadopoulos et al.](#) started as a generic **Runtime** with focus on graph algorithms on shared memory. It closely follows *C++* Standard Template Library directives. It does not support heterogeneous architectures yet, but does support distributed-memory systems. It is focused on irregular problems to allow the use of parallelism for applications that use dynamically linked data structures. STAPL provides a pre-processing phase that can perform automatic translation of some standard template library of *C++*.

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 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.

2.4.3 Landscape of Task-Based Runtime Systems

With the increase in interest in task-based **Runtime** systems, different research groups have initiated efforts to understand and implement their own **Runtimes**. In 2015, Sandia National Lab published a report [Bennett et al. \(2015\)](#) presenting a qualitative and quantitative examination of three asynchronous many-task **Runtime** system. The chosen **Runtimes** were

Charm++, Legion, and Uintah. They identified the increase in interest in task-based **Runtime** systems and a lack of comprehensive comparison between them. They ported a mini-application to the three chosen **Runtimes** and measured the performance and the level of difficulty (subjective) to port it. The effort was extensive and deep but the focus was on the overall performance on a specific mini application. The study presented in this study on task-based **Runtime** systems identified the features important to applications of various kinds and then classified seven **Runtimes** based on the chosen features. Simple benchmarks targeting the core overheads of task-based **Runtimes** were designed instead of one mini-application that does not characterize the basic costs of the **Runtime** systems. This study was less deep than the study from Sandia National Lab but covered more **Runtime** systems at more basic levels.

Another report [Wilke et al.](#) from Sandia National Lab presented a study on various **Runtimes** but covered them from a very high level. Concepts driving the development of each **Runtime** are discussed with different descriptions of various terminologies being used by each. No effort to assess the performance of the **Runtimes** were undertaken in this study. The study in this dissertation attempts to unify these different terminologies being used by each **Runtime**, going deeper than previous studies, to understand the practical performance of the chosen **Runtimes**.

Chapter 3

Dynamic Task Discovery *

3.1 Overview

Writing parallel code to harness the power of modern supercomputers is undoubtedly a challenging task for many application developers. It requires significant effort by domain experts to be able to write parallel code that can efficiently utilize systems that have multi-core, heterogeneous, and distributed architectures—code that usually has low portability across different hardware instances and generations. This steep learning curve can, therefore, act as a barrier to domain scientists and deter them from taking the steps to write parallel versions of their algorithms, limiting the opportunities to investigate larger and more demanding scientific problems. This chapter discusses a possible solution, a first step to introduce an abstraction that can take away the grilling and tiresome job of writing parallel code so that domain scientists can focus mostly on the science aspect of their work.

PaRSEC is a task-based **Runtime** system that abstracts the underlying computing resource to application developers. This means that application developers can relieve themselves from writing complicated parallel code and yet achieve high performance on contemporary machines with distributed architectures. **PaRSEC** allows users to program it using a concept called Parameterized Task Graph (PTG). This meant that application developers had to familiarize themselves with a new language, specific to **PaRSEC** only, called Job Data Flow

*Material in this chapter has been published by R. Hoque, T. Herault, G. Bosilca, and J. Dongarra. Dynamic task discovery in PaRSEC: A data-flow task-based runtime. In Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA 17

(JDF), in order to be able to program **PaRSEC**. This added overhead to the application development process makes it needlessly costly, in terms of lost hours to learn a very specific language that can only be used to program a specific **Runtime**.

This overhead in the development process can be avoided if application developers were provided with application programming interfaces (APIs), built using popular programming languages (such as *C*), having similar functionalities to PTG. This chapter introduces one such interface, Dynamic Task Discovery (DTD), that is API-based and is, therefore, akin to using libraries that many application developers are accustomed to. DTD allows users to write sequential code to program task-based **Runtime** systems like **PaRSEC** while building a task graph at runtime. This introduces a number of challenges unique to this programming style, especially when running the resulting applications at scale—challenges that are also addressed in this chapter. Furthermore, this chapter also discusses DTD’s performance at scale using various dense linear algebra algorithms, provides a thorough analysis of overhead causing limitations, and then a proposed possible solution to the researched problem. The last part of this chapter focuses on the in-house integration of DTD to existing applications and also details a number of collaboration efforts where DTD was used by external teams to implement more efficient version of their applications.

3.2 Parameterized Task Graph

The PTG is the first domain-specific language (DSL) of **PaRSEC**. **PaRSEC**’s format of PTG is known as Job Data Flow. The main idea is to remove the control flow that is imposed by sequential instructions and instead exploit the dataflow that is inherently more parallel but preserves the semantic of the algorithm. This DSL is based on true data-flow, where a user expresses an algorithm as a collection of task classes. Each task will be an instance of one of the task classes. Listing 3.1 shows a complete example of Cholesky Factorization written in PTG.

Line 1 shows the name of the task class, *POTRF*, and this task class uses a single parameter called *k*. The same format is true for the other task classes—*TRSM*, *HERK*, *GEMM*—and they use different numbers of class parameters respectively. Line 2 shows the

```

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
8  TRSM(m, k)
9  m = 1 .. total // Execution space
10 k = 0 .. m-1   // Execution space
11 : A(m, k)      // Task affinity
12 READ T <- T POTRF(k)
13 RW C <- (k == 0) ? A(m, k) : C GEMM(m, k, k-1)
14     -> A HERK(k, m)
15     -> A GEMM(m, k+1 .. m-1, k)
16     -> B GEMM(m+1 .. total, m, k)
17     -> A(m, k)
18 BODY { TRSM( ... ); } END
19 HERK(k, m)
20 k = 0 .. total-1 // Execution space
21 m = k+1 .. total // Execution space
22 : A(m, m)        // Task affinity
23 READ A <- C TRSM(m, k)
24 RW T <- (k == 0) ? A(m, m) : T HERK(k-1, m)
25     -> (m == k+1) ? T POTRF(m) : T HERK(k+1, m)
26 BODY { HERK( ... ); } END
27 GEMM(m, n, k)
28 k = 0 .. total-2 // Execution space
29 m = k+2 .. total // Execution space
30 n = k+1 .. m-1   // Execution space
31 : A(m, n)
32 READ A <- C TRSM(m, k)
33 READ B <- C TRSM(n, k)
34 RW C <- (k == 0) ? A(m, n) : C GEMM(m, n, k-1)
35     -> (n == k+1) ? C TRSM(m, n) : C GEMM(m, n, k+1)
36 BODY { GEMM( ... ); } END

```

Listing 3.1: JDF of Cholesky Factorization

execution space of *POTRF*: this defines the valid range for each parameter of a task class, which in this case is from 0 to value of *total*. Here, *total* represents the total iteration count. Line 3 defines the affinity of each instance of *POTRF* task class in a distributed run. Each task instance of the *POTRF* task class will be placed in the process that contains the data $A(k, k)$ in this case. All task classes in PTG will contain a set of precedence constraints, which define the constraints that need to be satisfied for a task instance of that class to be ready to be scheduled by ParSEC. For *POTRF*, the set of precedence constraints are stated from line 4–6. The *RW* in line 4 states all task instances will *READ and WRITE* this data (represented by flow **T**). The arrow pointing to the left defines the input constraint on data (of flow **T**) and the arrow pointing to the right defines the output constraints. *POTRF* tasks will either read data $A(k, k)$ from memory if value of k is equal to 0 or the input data will come from task $(k-1, k)$ of task class *HERK* as shown in line 4. The data, after being modified by a task belonging to *POTRF* task class, will be used by multiple tasks of task class *TRSM* ranging from $(k+1 .. total, k)$ (line 5). The data $A(k, k)$ will not be further modified by any other tasks and will be written to memory (line 6). Each task class also contains a set of code regions where the actual computation is expressed. Line 7 shows the code region, which is marked by *BODY* and *END*.

The same format is true for each task class shown in our example. The PTG description is independent of the problem size that will be solved by the algorithm. The main feature that makes PTG especially lucrative is the fact that the representation of the task graph is concise and is not discovered at runtime. However, to program ParSEC, users need to **learn how to write JDF** and need to know **all dependencies among task classes a priori**. Thinking in terms of operations to be applied on data is not a simple task. All these factors combined motivated us to explore alternate interfaces that make programming ParSEC easier and intuitive while not losing performance.

3.3 Dynamic Task Discovery in ParSEC

DTD is a new ParSEC Interface (or in this particular instance, a low-level task interface) proposed in this work. This interface allows users to write sequential-looking code, including

conditionals, *for* loops, and code blocks, to insert tasks using **PaRSEC**'s API. Though the code looks sequential, since users are inserting tasks, traditional control flow enforced by sequential code does not apply in this case. Instead, users insert tasks and the ordering of instructions (tasks) depends on how users are describing each task. 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 relationships 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, read and 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. A task performing a *write* before a task performing a *write* on the same data will create a write-after-write (*WAW*) dependency, and a task performing a *read* before a task performing a *write* on the same data will create write-after-read (*WAR*) dependency between the two tasks.

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 requires 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 is 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 DSLs in **PaRSEC** share the same data interface.

To illustrate **PaRSEC**'s API to insert tasks we provide in Listing 3.2 the same tiled Cholesky factorization we demonstrated using the PTG interface. This algorithm is composed of three

```

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    for( n = m+1; n < total; n++ )
13      parsec_insert_task (GEMM,
14        tile_of(A, n, k), INPUT,
15        tile_of(A, m, k), INPUT,
16        tile_of(A, n, m), INOUT | AFFINITY);
17  }
18 }

```

Listing 3.2: Cholesky Factorization

nested levels of affine loops, and contains four operations (tasks), namely: POTRF, TRSM, HERK and GEMM.

Each operation takes a number of data as input and performs a specific mathematical operation on the input data. To get the meta-data of each data DTD provides the "tile_of" method. This method takes a *data-collection*, as described earlier, and indices that specifies which piece of *data* from that *data-collection* this task will be using. Here, a *data-collection* can represent any type and shape of data. This gives DTD the capability to handle task-based workload of any kind and not only task data that is a matrix or a vector tile. Line 2 of listing 3.2 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
- 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; for example, line 3 of listing 3.2 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 as users will specify the placement of a task by assigning the AFFINITY flag. 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. We show a sample DAG in Figure 3.1. 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.

3.3.1 Challenges and Optimization of DTD

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, and DAG trimming in distributed environments, are novel to this programming paradigm of sequential task insertion. We list some of those challenges, including the ones mentioned earlier, and discuss how they are addressed.

Unrolling the DAG

DTD works by deducing the correct ordering of tasks from a sequentially described application. It then builds a DAG at runtime from this description. As the DAG is built at

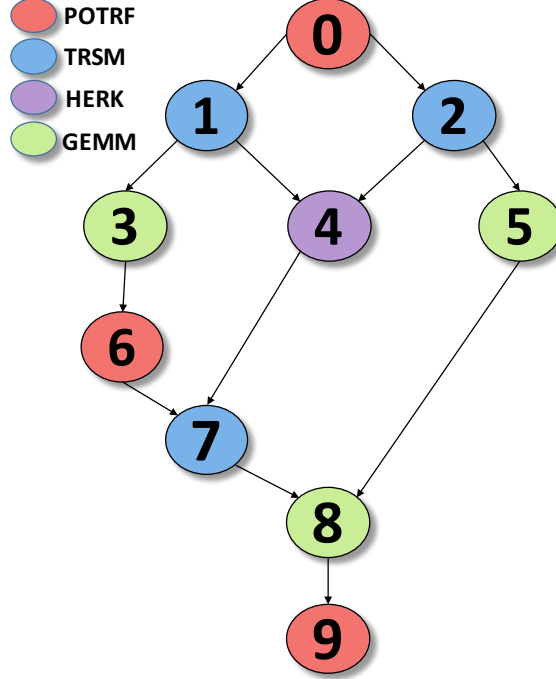


Figure 3.1: Sample Cholesky DAG from Listing 3.2 for $total = 2$

runtime, the DAG needs to be *unrolled* in memory in order to progress and schedule tasks 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 efficient 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 DTD’s case, the **memory requirement** for storing the entire DAG is $\mathcal{O}(|V| + |E|)$, where $|V|$ is the number of tasks discovered, and $|E|$ the number of dependencies. In PTG, the memory requirement is $\mathcal{O}(|TC| + |DC|)$, where $|TC|$ is the number of types of task, and $|DC|$ the number of data to which each type of task refers. Building a DAG is an operation that is at least $\mathcal{O}(N)$, 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.

To address this challenge a throttling mechanism for task insertion in the system can be implemented, allowing only a limited number of tasks to be inserted into the runtime at any

moment. It works like a sliding window of DAGs that the 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 3.2 shows the performance of Cholesky Factorization with different window sizes and the memory footprint of the application at those window sizes. We can see that with a bounded sliding window the same performance is achieved with substantially less memory overhead. The window size is a performance tuning parameter. The window size 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 tasks again. For the experiment shown in Figure 3.2, 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. Figure 3.2 shows 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 lookahead 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, the whole DAG is being stored 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. The window size will determine how far the `Runtime` can see in future of an algorithm. In a distributed environment this window is determined by the number of local tasks in each rank/process.

Cholesky Factorization on 8 nodes Haswell, 20 cores each, Tile Size = 180

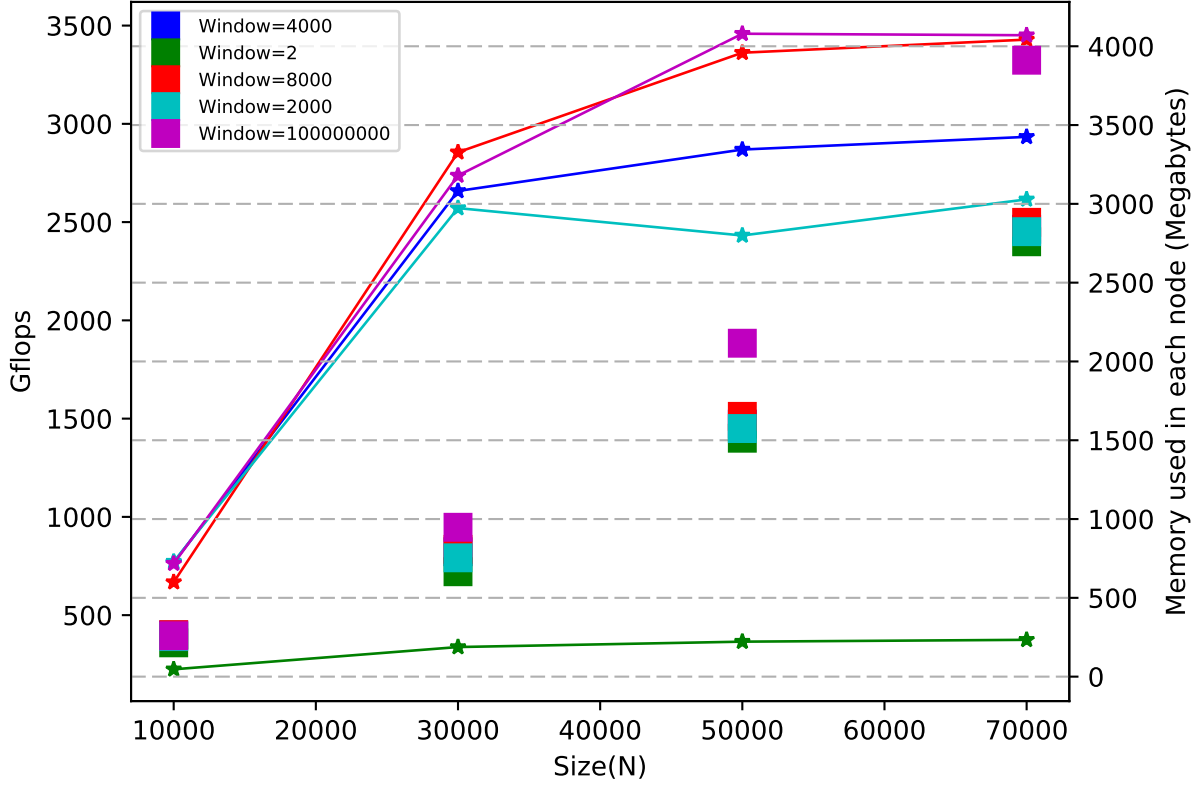


Figure 3.2: Performance and memory footprint per node for different window size

Untying task insertion from a specific thread

The user can choose to insert tasks using a specific thread, where the thread blocks after a certain number of tasks 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, DTD allows users to have tasks that can yield, thus offering a way to have "generator" tasks that will insert other tasks. This method unties the job of task insertion from one specific thread, and allows any thread to insert tasks in the `Runtime` and de-localize the task insertion into another context outside the critical path. To allow tasks to yield, `ParSEC` provides a 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 return from the task with the

special flag to yield has to be performed by the users explicitly inside the task and the maintenance of the state of the task also has to be performed by the user. 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 tasks recursively in DTD. The correctness of this programming model is enforced by the sequential order in which tasks are inserted. When multiple threads are inserting tasks simultaneously no guarantee of any such ordering can be promised, given this constraint users can only insert independent tasks using multiple threads simultaneously in DTD. Listing 3.3 shows an example of recursive tasking in DTD. In that Listing, a DTD program is shown that will insert a task that will in turn insert other tasks. The generator task, in this example, is responsible for inserting 10 tasks that actually compute, or are *actual tasks*, and while inserting them—if a certain condition is true—the generator task can simply return a "schedule-me-later" flag to the scheduler as shown in Line 10 of Listing 3.3.

```

1 int Real_Task(..) {
2     /* Execution of Real Task */
3     return PARSECHOOK_RETURN_DONE;
4 }
5
6 int Task_inserting_Task(..) {
7     for( i = 0; i < 10; i++) {
8         if( some_condition ) {
9             /* Return some kind of rescheduling signal */
10            return PARSECHOOK_RETURN_AGAIN;
11        }
12        /* Inserting real task */
13        parsec_insert_task (Real_Task);
14    }
15    return PARSECHOOK_RETURN_DONE;
16 }
17
18 int main(int argc, char **argv) {
19     /* Inserting task that will generate other tasks */
20     parsec_insert_task (Task_inserting_Task);
21 }

```

Listing 3.3: Recursive Task Insertion using DTD

DAG Trimming

Using DTD, applications are expressed as tasks which are inserted sequentially at runtime via DTD’s API. In an execution of multiple machines, each machine will need to discover and examine all the tasks to determine the correct order and relationships of tasks. This means each process will discover both local and remote tasks. There might be remote tasks that are not related to any local task of a process but for correct ordering each process needs to discover all the tasks gradually during runtime. This will result in a large memory footprint for DTD applications. In distributed environments, DTD improves its memory footprint by trimming the DAG. It tracks of all the local tasks and all the remote tasks that are either a direct successor or a predecessor of a local task are kept, and it trims 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 all related remote tasks need to be stored. To achieve this, all the data needs to be tracked, 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 to keep or ignore them. Figure 3.3 shows a sample trimmed DAG on each node in a distributed run. We can see the advantage of trimming the DAG of remote not-related tasks in Figure 3.3. We are successfully able to restrict the memory overhead of large distributed problems using this technique. Figure 3.4 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 a smaller one, and we see that by trimming the DAG and reusing the task structures we require almost five times less memory in the case of matrix size 70000.

Communication

Communication in DTD is accomplished using the communication engine present in `PaRSEC`. In the 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; for example, 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 $T(k)$ is then known

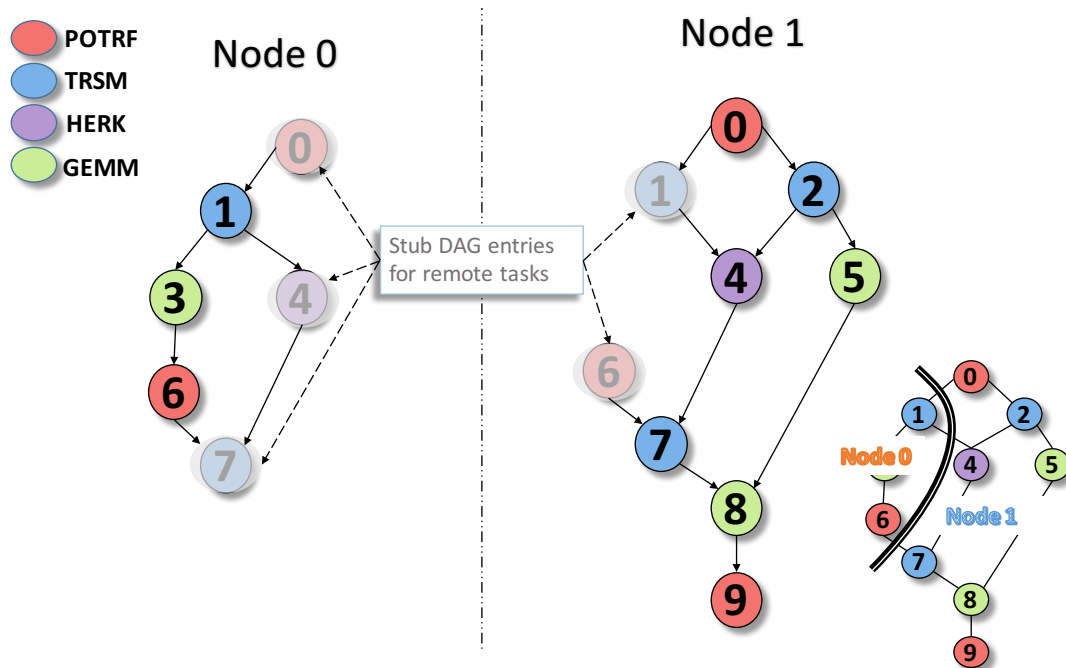


Figure 3.3: Sample Trimmed DAG

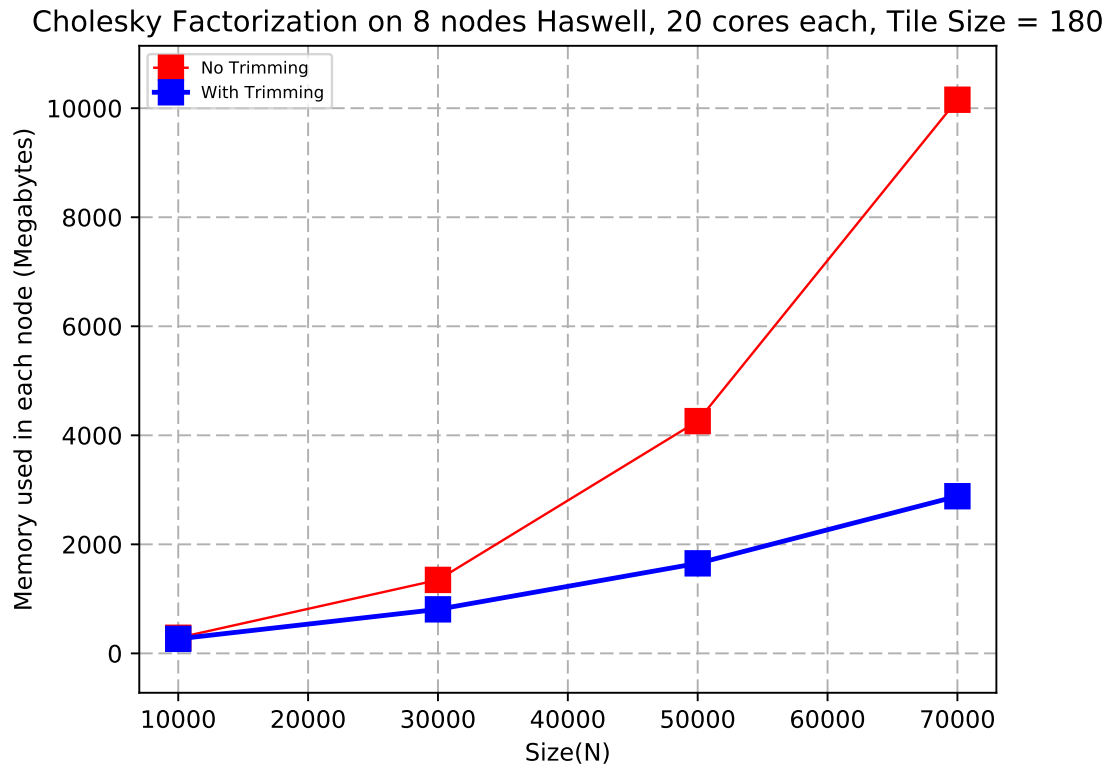


Figure 3.4: Memory-footprint in each node with and without Trimming

in process P_j , but not yet locally discovered, so in order to minimize the local impact on memory usage we delay the communication between process P_i and 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 on the communication side compared to PTG.

Moreover, as it is discovered that due to the tasks' dependencies, the same data version would need to be transferred multiple time between 2 processes, the number of necessary communications is optimized by marking the data accordingly with the local knowledge, and by avoiding to send it more than once. This avoids redundant communications. The communication engine has also been carefully engineered 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 discovered the next remote writer.

3.4 Evaluation of DTD

To test the performance of the implementation of DTD on top of ParSEC, evaluations of certain overheads of this programming paradigm have to be performed. These evaluations include **the cost of managing tasks, the efficiency of building a task graph at runtime, the efficiency of pushing ready tasks in the scheduler, and the efficiency of pushing communication when required**. Dense linear algebra routines allow the tuning of task granularity and build moderately complex task graphs involving multiple data usage on the same task. This makes them ideal to test the performance of DTD in terms of the above mentioned overheads in both shared- and distributed-memory systems.

3.4.1 Evaluation of generic task management cost

This experiment was performed to get an estimate of the cost of managing one task using varying number of data in ParSEC using the DTD interface. The machine configuration for this experiment was Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz. The cost is represented in units of time and was measured by recording time once at the beginning of task insertion

and once at the end of the experiment. A total of one hundred thousand *empty* (no computations were performed) tasks were inserted using a single thread and the total time was divided by the total number of tasks to calculate the cost of managing a single task. Figure 3.5 shows the cost of managing a task with varying data. The cost is shown in the y-axis while the number of data (No. of Flows) each task is using is shown on the x-axis. The best fit line indicates a linear relationship of cost with number of data a task uses. For the given architecture, a task using a single data would incur a cost of around 2.5 microseconds. This is a mitigated cost and is achieved by using a window and reusing task structure of already finished tasks as described above.

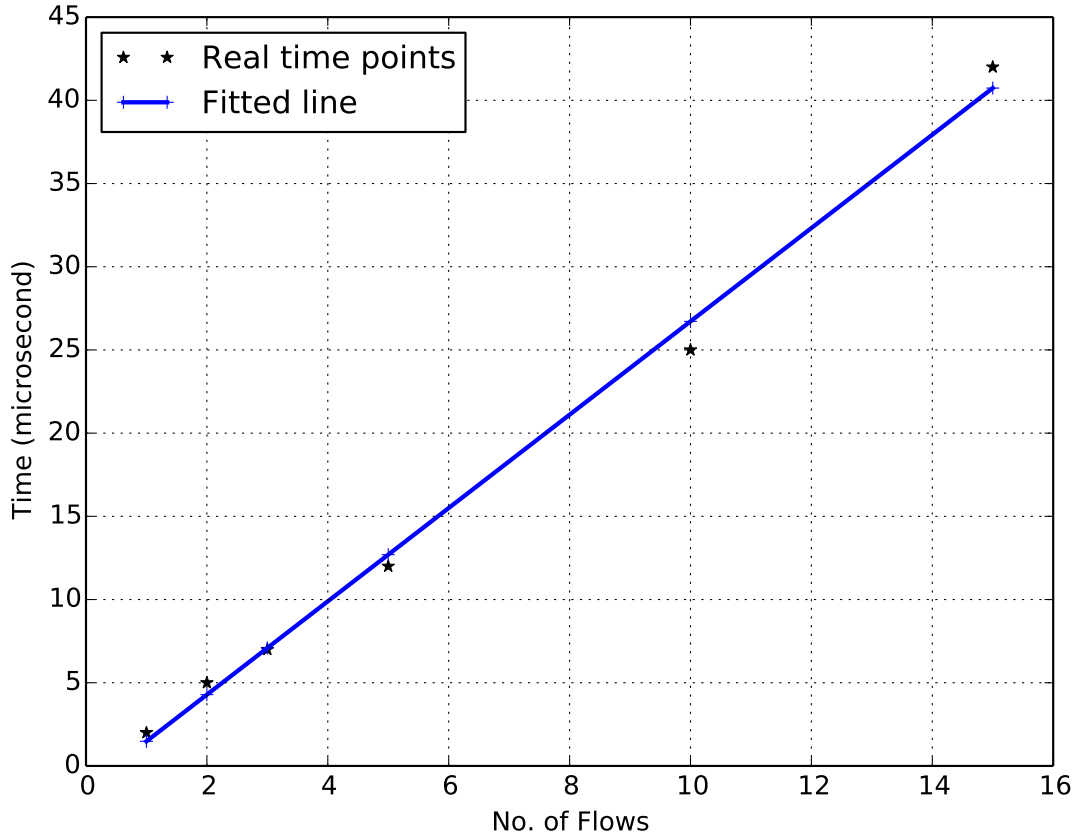


Figure 3.5: Cost of each task in DTD

3.4.2 Evaluation on Shared Memory

For assessing DTD’s performance on a shared-memory tile-based dense linear algebra routine, a DTD version of Cholesky Factorization was used. It is important to note that the tile size in these tiled algorithms determines not only the performance each task can achieve, but also the total number of tasks that will be generated, and that in turn determines the stress on the `Runtime` of the task management. With all these experiments, the goal is not to show the percentage of machine peak these algorithms are reaching, but rather how the different `Runtimes` are performing compared to each other given the same stress. PTG [Danalis et al. \(2014\)](#) has already been shown to scale better than most of the alternative runtimes, thus achieving comparable results to PTG—where more runtime overhead should be an indication of good runtime efficiency and an unhindered description of the available parallelism.

Chameleon [Agullo et al. \(2012\)](#) was used to test all the `Runtime` systems other than PaRSEC PTG in shared memory. Chameleon provides functionality like the PLASMA [Buttari et al. \(2009\)](#) library with an option to choose the underlying `Runtime`. Currently supported `Runtimes` are QUARK [Yarkhan \(2012\)](#), StarPU [Agullo et al. \(2014\)](#), and PaRSEC. StarPU and QUARK both allow users to express their application sequentially in the form of tasks, builds a DAG, and tries to efficiently schedule the tasks. Their programming paradigm is similar to DTD and they share the same overheads as DTD. They also provide dense linear algebra routines supporting mainly shared-memory systems, and this makes them ideal for comparison with DTD. For PaRSEC, the local flat queue (LFQ) scheduler was used and for StarPU the local work stealing (LWS) scheduler. GCC 5.1 was used to compile all the libraries. A summary of the different software and their corresponding versions are provided in Table [3.1](#).

Table 3.1: 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		

For shared-memory experiments an Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz machine with 20 physical cores was used as well as Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz with 68 physical cores. All experiments were run 5 times and the average performance of the 5 runs are reported here.

Result of Cholesky factorization on two architectures, Intel Haswell and Intel KNL, are presented. Figure 3.6 shows the performance on Haswell, where the peak performance of the GEMM kernel is 645 gigaFLOP/s. The top plot shows the performance of the **Runtime** systems for a tile size of 320×320 . At this size, there is a clear convergence in terms of performance between all **Runtimes**; the computational intensity of the target kernel (matrix-matrix multiplication) tolerates a lot of overhead in the **Runtimes**. In the bottom figure, the size of the matrices (14k and 20k) are fixed and the impact of the tile size on the performance

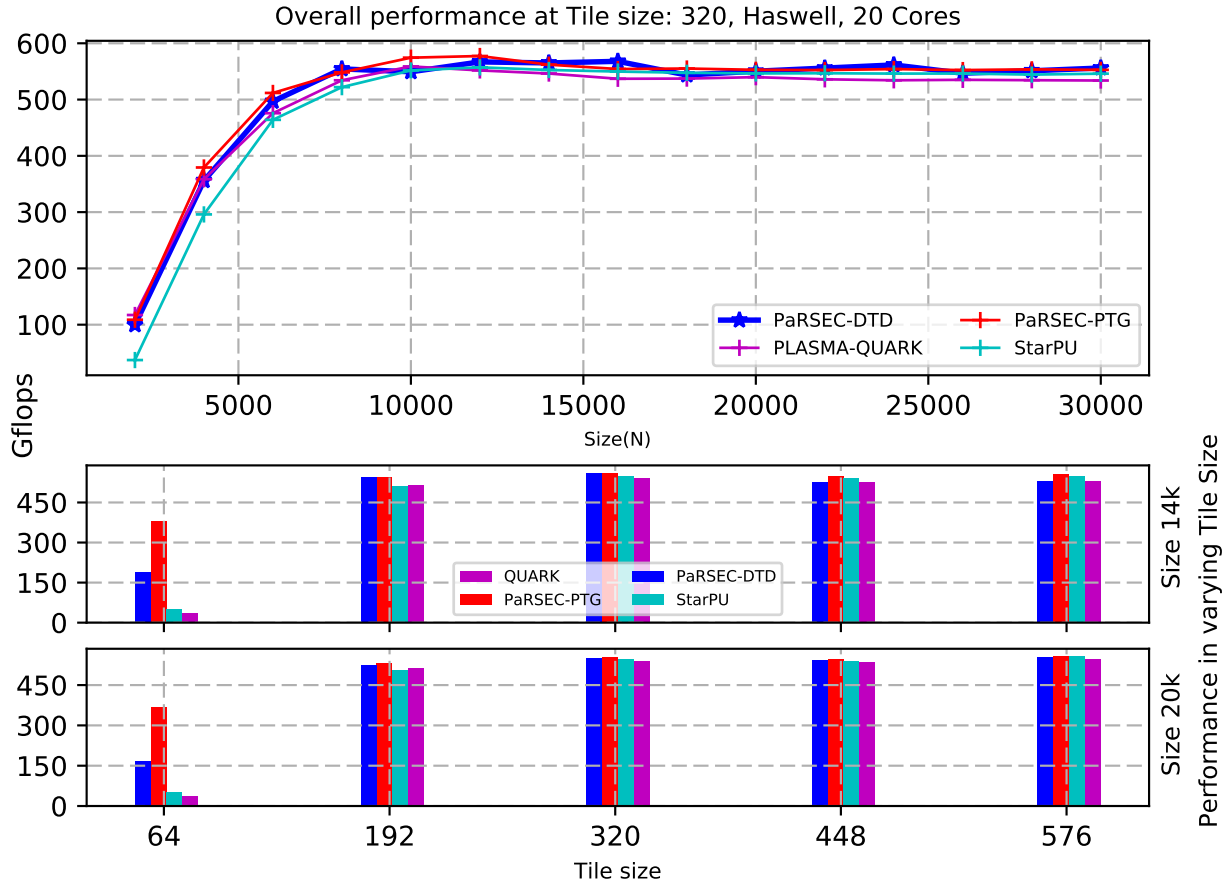


Figure 3.6: Performance of Cholesky factorization on Haswell (shared memory)

is investigated. For each chosen matrix size, the tile size is varied from 64 to 576 with a step size of 128. The result is that without modifying the total amount of computations needed to solve the problem, we are decreasing the granularity of each task and therefore increasing the number of tasks (for Cholesky there is a cube relationship), and as a result the task management stress is increased on the different **Runtimes**. We see that of all interfaces 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. For tile size 64 and matrix size 14k, PaRSEC DTD has $4\times$ the performance compared to StarPU and $5\times$ the performance compared to QUARK.

Figure 3.7 shows the performance of Cholesky Factorization 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 the GEMM kernel on this machine is 2 teraFLOP/s. In the top plot we see that PTG and DTD both perform similarly 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 decreases, we see a similar result to the Haswell experiment, both PaRSEC DSLs, PTG and DTD, outperform all the other **Runtimes**. For information at tile size of 64, PaRSEC DTD is $3\times$ faster than the other **Runtimes**, where PTG is about $6\times$ faster than DTD, due to its efficient task handling, a smaller number of known tasks, and a more streamlined scheduling.

3.4.3 Evaluation on Distributed Memory

For assessing DTD’s performance on a distributed-memory tile-based dense linear algebra routine, QR Factorization was used. QR factorization contains tasks that use multiple data and configurations where there may exist writers of data not residing in the same rank/process as the data—also known as remote writer. For this reasons it is more complex to get a version of QR factorization than Cholesky in a distributed-memory environment. From the perspective of **Runtime**, each of these algorithms represents a DAG, but the complexity of

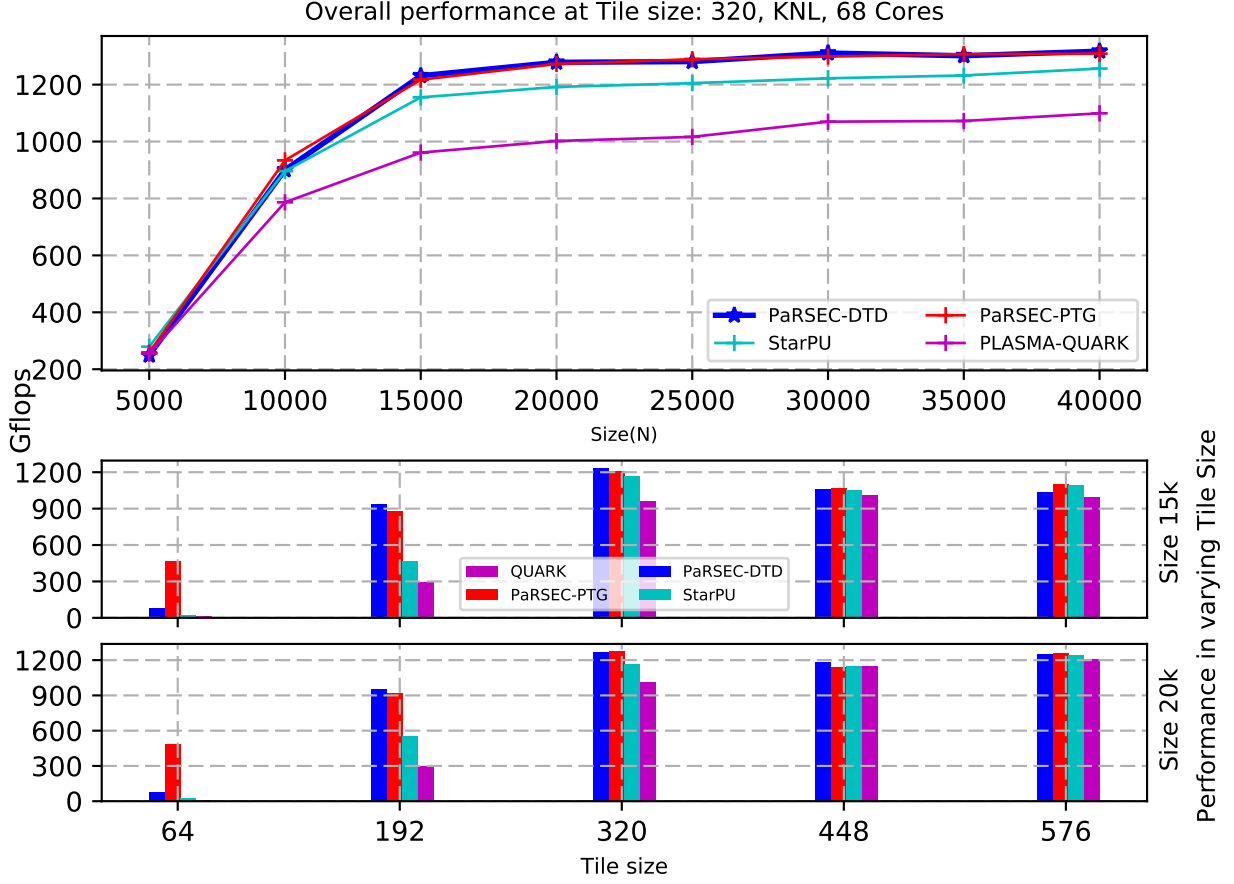


Figure 3.7: Performance of Cholesky factorization on KNL (shared memory)

building the DAG varies in each case. At the time of performing these experiments, Quark provided none and StarPU provided only Cholesky Factorization for distributed-memory systems, thus they were included in the experiments assessing performance on distributed systems. For distributed systems we have used DPLASMA [Bosilca et al. \(2011\)](#) for PaRSEC and ScaLAPACK [Blackford et al. \(1997\)](#). Open MPI 2.1 was used as the communication library. A summary of the different software and their corresponding versions are provided in Table 3.1.

For the distributed-memory system, the TACC [TACC](#) cluster Stampede was used, where each node is equipped with 2 8-core Xeon E5 processors and 32GB of memory and connected with InfiniBand FDR interconnect. All experiments were run 5 times and the average performance of the 5 runs are reported here.

The first experiment performed was a problem scaling (the total number of computing resource remains constant while the problem size increases) of QR Factorization. Out of the 3 `Runtimes` only 1 provided a distributed QR, but ScaLAPACK was added, which represent the current state-of-art algorithm on this setup. The highest node count was 128 where the total core count was 2048. Figure 3.8 show that for matrix size of up to $120k \times 120k$, both PaRSEC DSLs, PTG and DTD, outperform ScaLAPACK by a significant factor (up to $3x$ for small matrices). Once the problem size reaches saturation ($280k \times 280k$), ScaLAPACK catches up with PaRSEC, and both asymptotically converge toward the machine peak. These results show that, at least up to this number of processes, the QR performance of PTG and DTD are equivalent, highlighting a similar scalability for both PaRSEC DSLs.

To further assess the scalability of DTD, a weak scaling test was performed on Cholesky and QR Factorization by varying the number of cores from 16 to 2304 cores as shown in

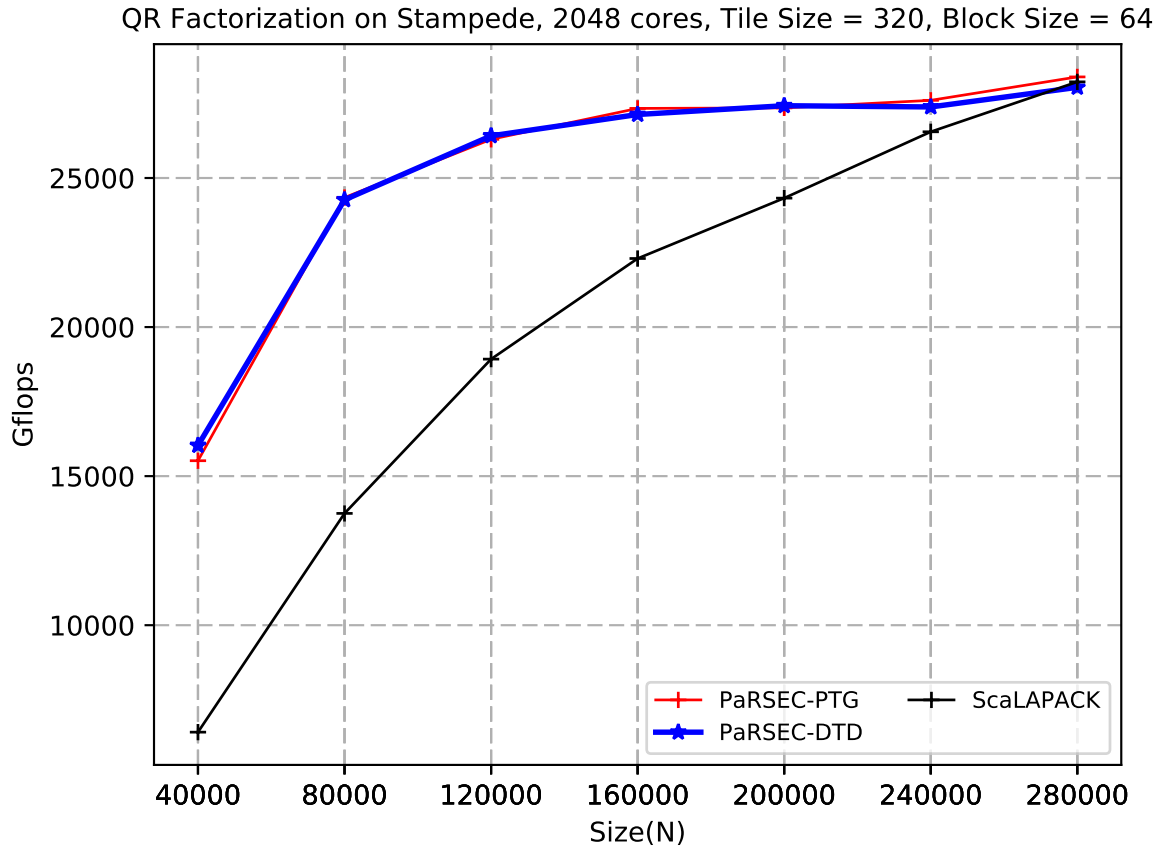


Figure 3.8: QR factorization on 128 nodes (2048 cores)

Figure 3.9. Here, the problem size is determined by the number of processes 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 $20k \times 20k$ and the final total size in both cases was $240k \times 240k$. The data was distributed in a block-cyclic way across a $P \times Q$ processors grid, where P and Q are the same in all cases. From 1 to 4 processes we see a drop in performance as communications will introduce latency. 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 than Cholesky. Both the interfaces of ParSEC, DTD and PTG, has almost a straight line as we scale up, which indicates they are scaling well.

For Cholesky factorization the impact of introducing communications is higher than for QR. Both the interfaces have almost a straight line in this case as well but 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).

3.4.4 Evaluation on Heterogeneous Architecture

A prototype was built of heterogeneous execution of tile-based General Matrix Multiply (GeMM) algorithm using DTD. The goal was to keep the requirements to enable heterogeneous execution with DTD at a minimum. With that goal in mind no additional option to specify special kernels for accelerators was added in the API of DTD; instead, the decision was to allow users to write a separate kernel for each accelerator following a certain pattern in the name of the function of the CPU kernel. The users then need to compile the new kernel for each accelerator into a library and simply add the path to that library when executing existing algorithm. For example, the GEMM kernel is used in multiple algorithms (Cholesky, GEMM, etc.), instead of adding a special accelerator kernel to each of these algorithms for GEMM, users can write the GEMM kernel for each accelerator once and provide the path (DTD’s assigned environment variable) to the library containing the corresponding method following DTD’s function signature. ParSEC can then figure out that for a certain task the

Weak Scaling - DPOTRF and DGEQRF, (20k x 20k)/Node, Tile Size 320, up to 2304 Cores

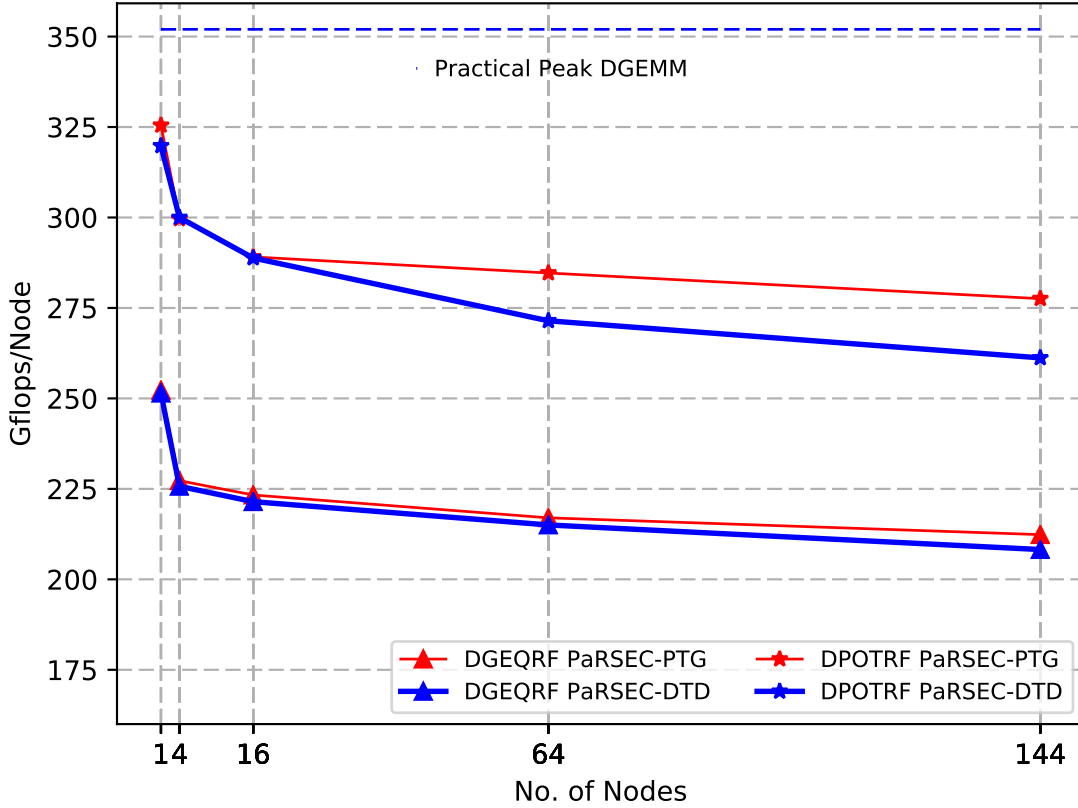


Figure 3.9: Weak Scaling of Cholesky and QR, up to 2304 Cores

users have provided a kernel for a certain accelerator and can then decide to offload work to that additional device.

As a prototype, a CUDA [Nickolls et al. \(2008\)](#) kernel for GEMM was developed and the path to the library containing that CUDA kernel was provided while executing the tile-based GEMM algorithm in DPLASMA. The experiment was run on a machine containing an Intel Haswell with 20 cores and a P100 NVIDIA GPU. Figure 3.10 shows the performance result of the experiment. The y-axis shows the performance (higher is better) and the x-axis shows the matrix size. The tile size is 1024: a bigger tile size than usual was chosen as bigger computation performs better on the GPU, which is the more powerful device between the CPU and the GPU. The experiment showed the performance of both PaRSEC PTG and DTD to be comparable.

Cholesky Factorization on Haswell - 20 cores and P100, Tile Size = 1024

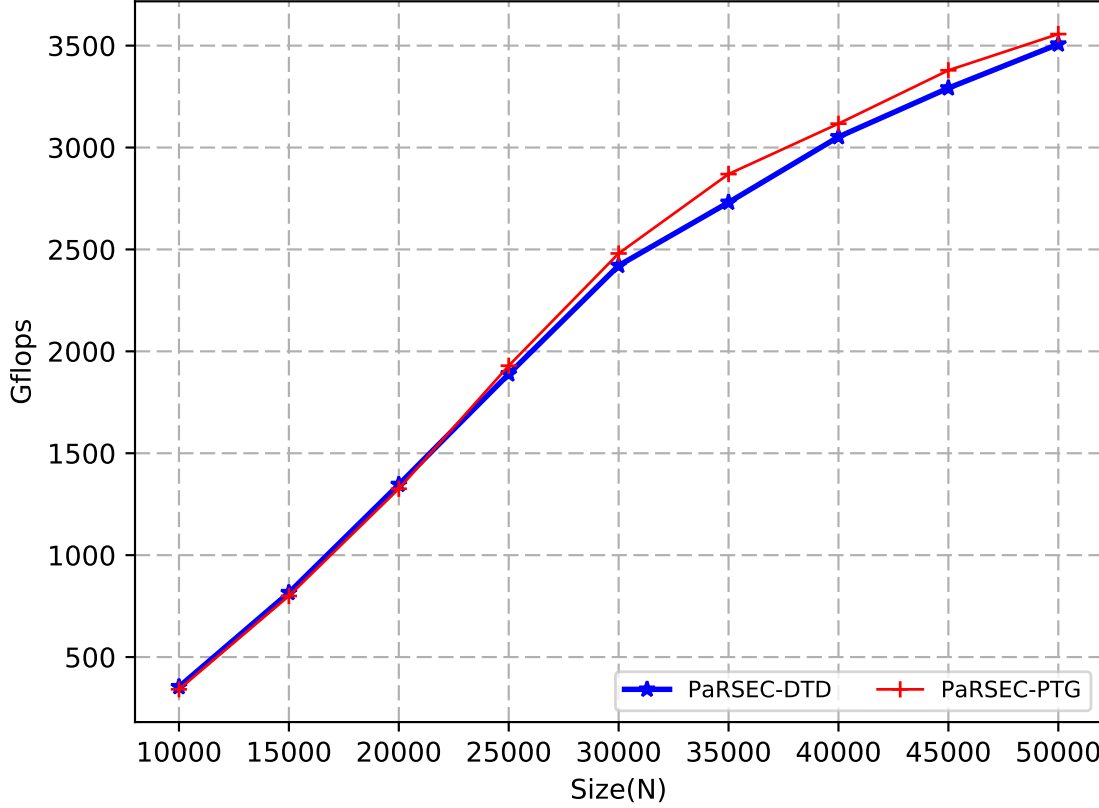


Figure 3.10: Performance of Tile-based GEMM on top of P-100 Nvidia GPU

3.5 Overhead of DTD

In this section, first, a mathematical model is provided to compare the overhead of DTD and PTG at runtime. The upper bound of DTD's performance is then presented with a description of a different scheme using which DTD can be implemented. Examples are shown of cases in which a scheme of implementation would be beneficial, and the section concludes with the theoretical limit of sequential task insertion.

3.5.1 Overhead of DTD compared to PTG

A mathematical model is discussed in this section that represents the overhead of Dynamic Task Discovery compared to Parameterized Task Graph. This section begins 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:

$$T_{PTG} = \frac{N \times C_T}{P \times n} \quad (3.1)$$

and DTD's overall time as:

$$T_{DTD} = \frac{N \times C_T}{P \times n} + N \times C_D + \frac{N \times C_R}{P} \quad (3.2)$$

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 includes 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, Cholesky factorization was performed in a distributed setting of 8 nodes. By tuning the tile size, the cost of each task can be varied, C_T , and the total number of tasks in the system, N , two principal overhead tuning parameters. In Figure 3.11, we see the performance of both the paradigms drop as we move to finer granularity, resulting in smaller tasks. An important thing to notice is the drop of DTD is significantly higher than PTG. Altering the tile size varies one of the tuning parameters, N , while the other two, P and C_T , are constant. By looking at Equation 3.2, we can identify the cases where the overhead of the DTD paradigm will be visible compared to PTG.

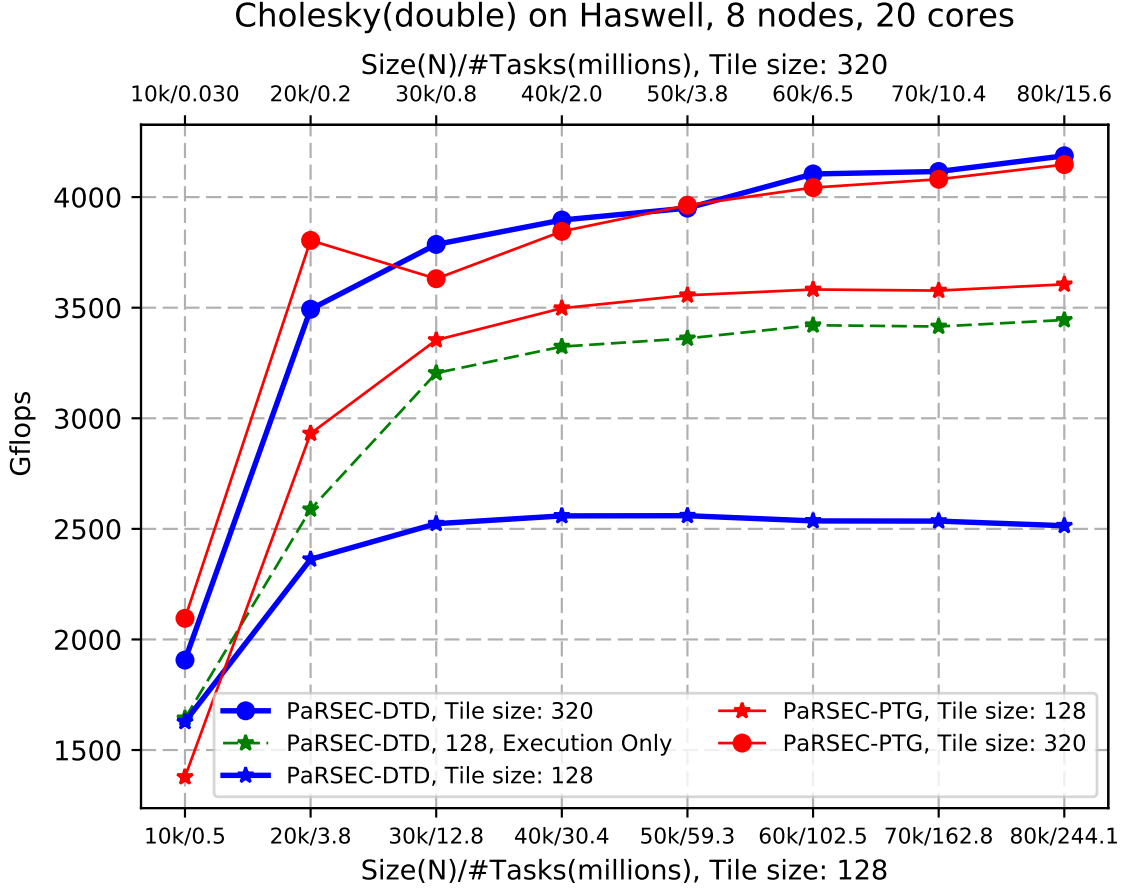


Figure 3.11: Overhead test in PaRSEC

Let us start with the parameter N . For varying matrix sizes, the performance of DTD and PTG are comparable in the case of 320 as the tile size. 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 affect 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 3.2, 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 the same experiment was repeated for DTD, but this time the whole DAG was built before

and was excluded from the total time of execution. We see that the performance improves drastically and is only 4% lower than PTG.

From Equation 3.2 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 the DAG will be significant. For large distributed executions involving numerous processes and billions of tasks the middle part of Equation 3.2 ($N \times C_D$) will be a bottleneck. All processes need 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 3.2) 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. Provided that N and P both increase, 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 an ideal case will make $N \times C_D$ go down to $\frac{N \times C_D}{P}$.

3.5.2 Upper Bound of DTD's Performance

DTD will have the best performance and will be scalable if the following cost for an execution can be achieved:

$$T_{DTD} = \frac{N \times C_T}{P \times n} + \frac{N \times C_D}{P} + \frac{N \times C_R}{P} \quad (3.3)$$

Users will need to partition the task graph perfectly to realize this. This would give us an upper bound for the best DTD can do. To verify this and to test if our implementation of DTD supports the above equation, experiments were run in a distributed environment. The experiment consisted of running multiple independent chains of tasks, where tasks in the chain would be dependent but there would be no dependency among the chains. Furthermore, the number of participating process were increased while the amount of work per process was kept constant. There were three versions of the test:

- All tasks are submitted in all processes (current implementation) - Insert All
- Only local tasks are submitted but users explicitly filter tasks - Insert Local
- Only local tasks are submitted but the filter to skip inserting tasks is inside the `Runtime- Select Insert`

The experiment was run on Stampede 2, where each machine is an Intel Skylake with 48 cores and they are connected with Omnipath interconnect with 100 Gbps bandwidth. The duration of each task was fixed to $160\mu\text{s}$. Each node had to execute 100k tasks. Each process was allocated 3 cores, which resulted in 16 process per real node. Figure 3.12 shows the result of the experiment. The y-axis shows the time—the lower the better. The x-axis shows the number of participating processes. We would ideally want the line to be as straight as possible to show good scalability. In the Insert All (current implementation) case we see that equation 3.2 holds true, as DTD is discovering more and more remote tasks, as it has to discover all tasks; the time to complete execution keeps growing starting from 256 process and keeps on increasing. For the other two cases, where either users are selectively inserting tasks or the `Runtime` is only selecting the local tasks, we see the finish time remains constant as the number of participating processes are increased, showing good scaling. This experiment shows that DTD would scale in an ideal case where we can somehow have a correct algorithm without DTD discovering all tasks. This experiment also shows that the increase in execution time with an increase in count of ranks/processes is not due to poor implementation of DTD.

3.5.3 Owner-Tracks-Future Scheme

As we have seen in the previous experiment, the current implementation of DTD—where all processes discover all tasks in order to determine the correct order and relationships between tasks—will not scale. This led to efforts in finding out an alternate approach of implementation of DTD that might not suffer from this scalability bottleneck. Such an alternative approach to DTD’s implementation could be a scheme where each rank discovers all the local tasks and the current *owner* rank of a data (process in a distributed system in

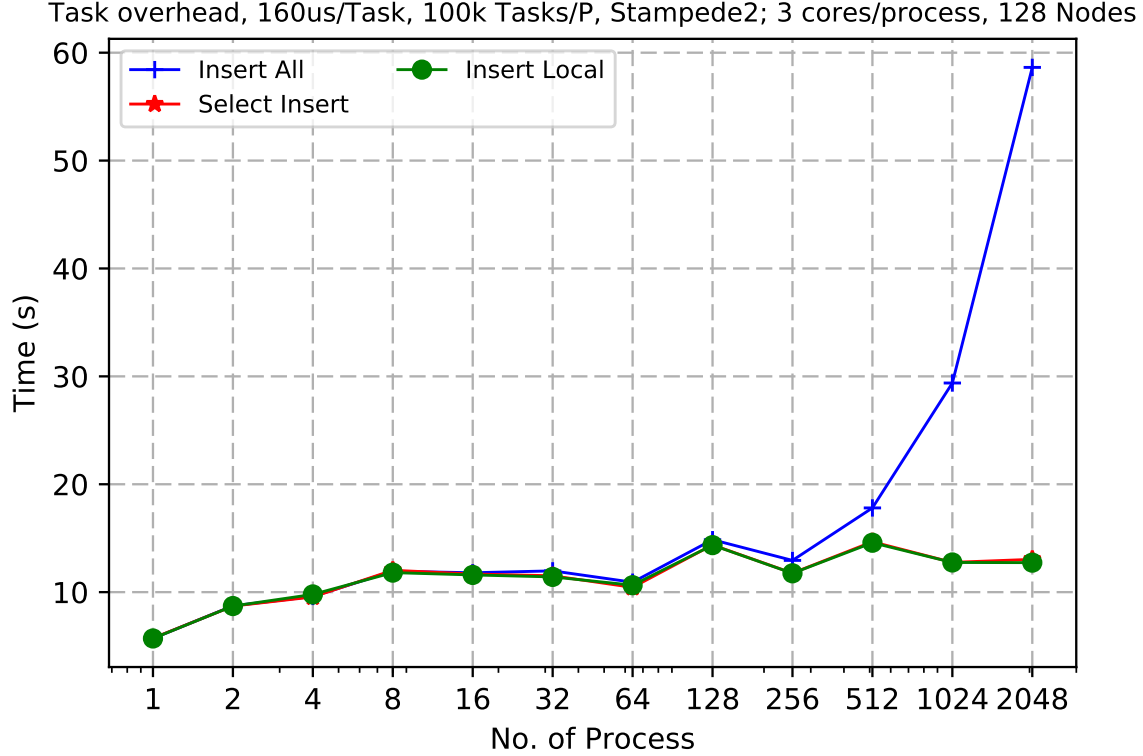


Figure 3.12: Scalability test of DTD

which that data resides according to the initial data distribution) tracks all the remote tasks using the data until the ownership of the data changes. We are calling this approach Owner-Tracks-Future, as the owner is responsible for tracking what happens with a particular data until the ownership changes. Figure 3.13 shows a sample task graph and how the current and alternate scheme would discover it. In this example we have 3 data, A, B, and C, each residing in Rank 1, 2, and 3 respectively. So, in this case, Rank 1 is the owner of data A, Rank 2 the owner of data B, and Rank 3 the owner of data C. In the left of the figure we have the full task graph showing all the tasks and their corresponding dependencies, if any. The top right of the figure shows the current scheme; to figure out the correct ordering of tasks, DTD discovers all tasks and then decides to keep the ones necessary in each rank. The tasks DTD trims in each rank are shown as faded. The bottom right shows the alternate scheme (Owner-Tracks-Future) where the owner of a data will track all the tasks using that particular data and will drive the correct ordering of those tasks in all necessary ranks. In

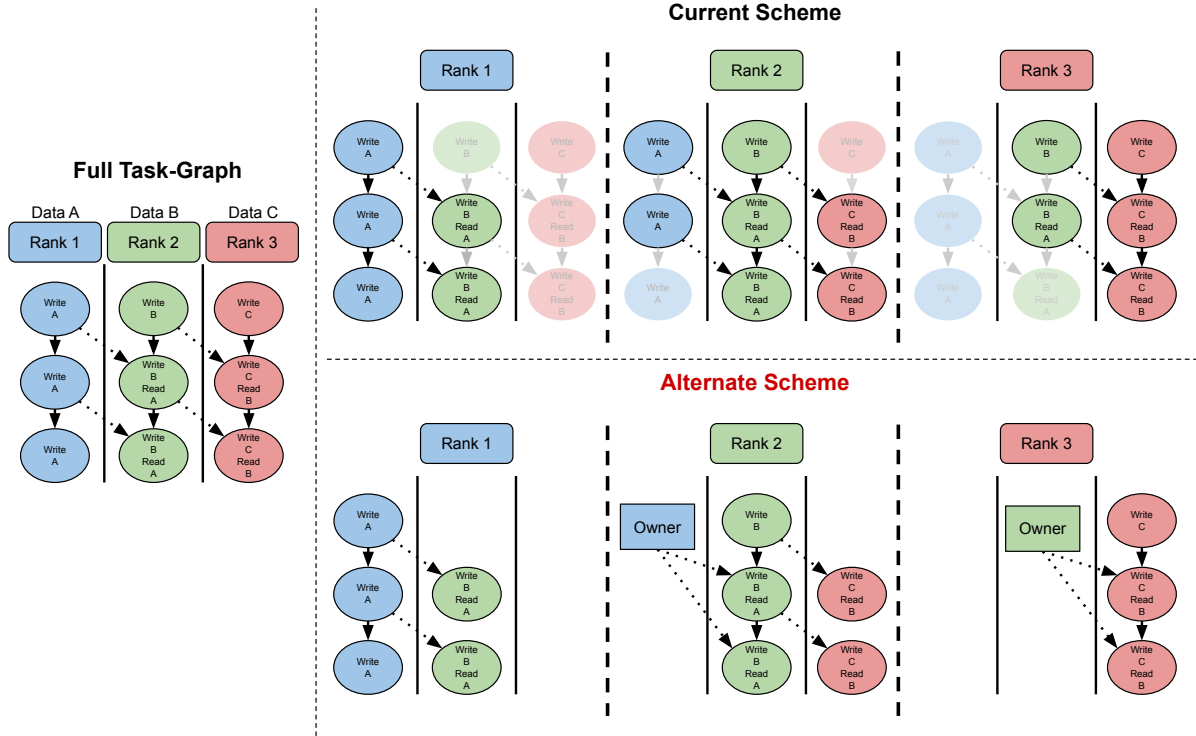


Figure 3.13: Alternate Scheme for DTD

this scheme in case of rank 1 it discovers all the local tasks and the remote tasks of Rank 2 that uses a data Rank 1 *owns* (data A). Rank 2 only discovers the local tasks and the tasks local to Rank 3 that uses data owned by Rank 2 (data B); note that Rank 2 in this case does not discover any tasks of Rank 1, as none of those tasks are local to Rank 2 or uses any data owned by Rank 2. The same is followed for Rank 3.

Listing 3.4 shows code that would generate a DAG for the current implementation of DTD. In Listing 3.4, at first a task is inserted that will execute in Rank 1 and will write data A. Then a task is inserted that will execute in Rank 2 and will write data B; the later task of this class will also read data A and the *if* in Line 4 is to distinguish between the first task and the later tasks of this class. The same process is repeated for the third kind of task that will execute in Rank 3. Listing 3.5 shows sample code of what the Owner-Tracks-Future scheme would look like, to generate the DAG in Figure 3.13. The big difference with the current implementation is the *if* statements, in Line 7, 11, 16, and 21, to conditionally insert tasks in each rank. The conditions can be added by the user or can be implemented inside the `Runtime` (Select Insert and Insert Local from Section 3.5.2). In this scheme, as mentioned

```

1  for( k = 0; k < 3; k++ ) {
2      parsec_insert_task(RANK_1_TASK,
3                          tile_of(A), INOUT | AFFINITY);
4      if(k == 0) {
5          parsec_insert_task(RANK_2_TASK,
6                              tile_of(B), INOUT | AFFINITY);
7          parsec_insert_task(RANK_3_TASK,
8                              tile_of(C), INOUT | AFFINITY);
9      } else {
10         parsec_insert_task(RANK_2_TASK,
11                             tile_of(A), INPUT,
12                             tile_of(B), INOUT | AFFINITY);
13         parsec_insert_task(RANK_3_TASK,
14                             tile_of(B), INPUT,
15                             tile_of(C), INOUT | AFFINITY);
16     }
17 }

```

Listing 3.4: Current Scheme

before, each rank will test if any of the data being used by the task is local to it and will only insert a task if that is true.

Overhead analysis of Owner-Tracks-Future

One of the major questions we need to clarify about the new Owner-Tracks-Future scheme is whether this scheme will result in an optimal discovery of tasks in all cases. To answer this question, we will analyze two sample task graphs shown in Figure 3.14. In both cases there will be two participating processes (ranks) and one data that will be used by all tasks.

Figure 3.14a shows a sample task graph (DAG) consisting of one data and only two types of tasks—tasks that *Write* on the data and tasks that *Read* the data. The red tasks are *Write* tasks and the yellow tasks are *Read* tasks. The left shows the complete task graph. In Rank 1 we have one *Write* task followed by 1 million *Read* tasks. In Rank 2 we have 1 *Write* task following the million *Read* tasks in Rank 1 and 1 million *Read* tasks after

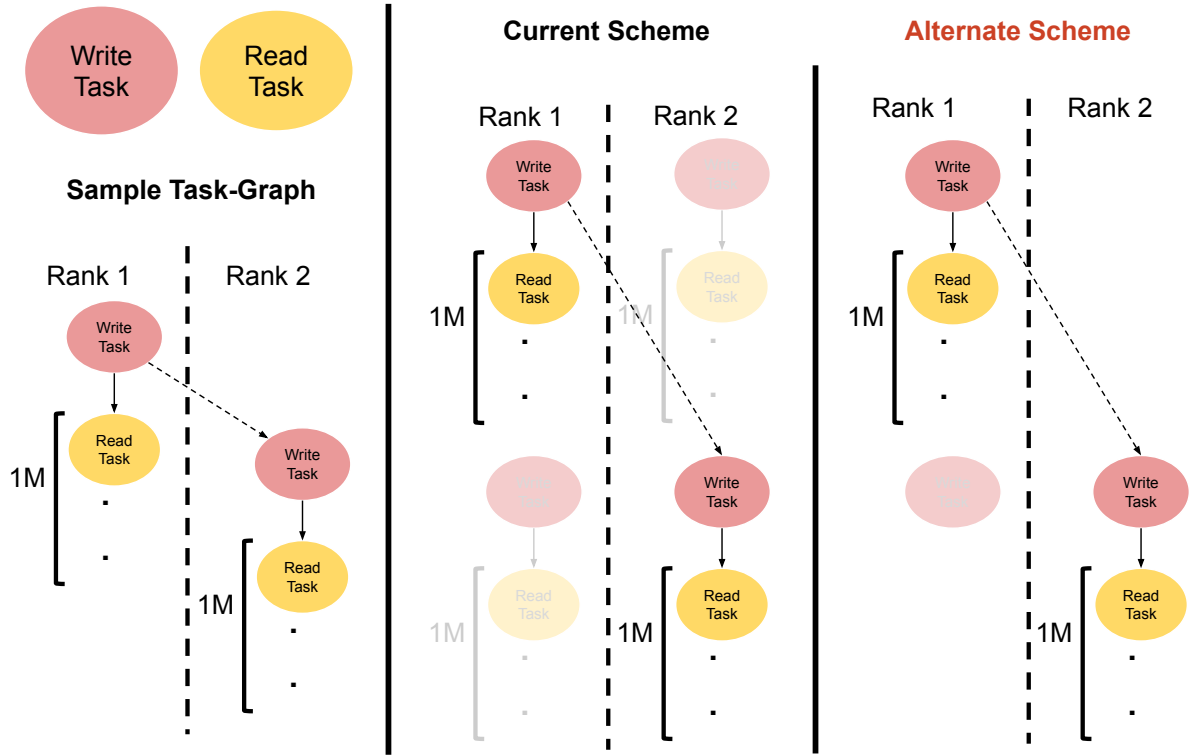
```

1  for( k = 0; k < 3; k++ ) {
2      if( data_is_local ( tile_of(A)) ) {
3          parsec_insert_task(RANK_1_TASK,
4                              tile_of(A), INOUT | AFFINITY);
5      }
6      if(k == 0) {
7          if( data_is_local ( tile_of(B)) ) {
8              parsec_insert_task(RANK_2_TASK,
9                                  tile_of(B), INOUT | AFFINITY);
10         }
11         if( data_is_local ( tile_of(C)) ) {
12             parsec_insert_task(RANK_3_TASK,
13                                 tile_of(C), INOUT | AFFINITY);
14         }
15     } else {
16         if( data_is_local ( tile_of(A)) || data_is_local ( tile_of(B)) ) {
17             parsec_insert_task(RANK_2_TASK,
18                                 tile_of(A), INPUT,
19                                 tile_of(B), INOUT | AFFINITY);
20         }
21         if( data_is_local ( tile_of(B)) || data_is_local ( tile_of(C)) ) {
22             parsec_insert_task(RANK_3_TASK,
23                                 tile_of(B), INPUT,
24                                 tile_of(C), INOUT | AFFINITY);
25         }
26     }
27 }

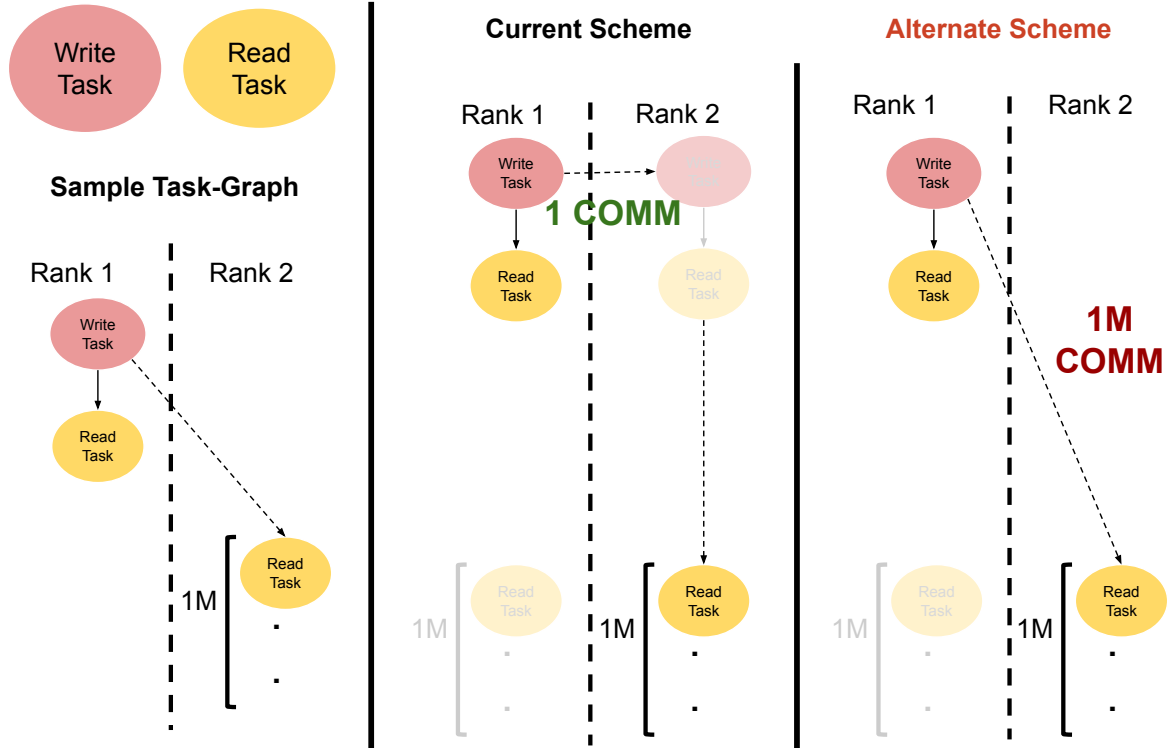
```

Listing 3.5: Alternate Scheme

that. The next part of the diagram shows the current DTD implementation, where both ranks will discover all tasks (2 Million and 2 tasks). After the first *Write* task (Rank 1) is complete, DTD will activate all its local successors in Rank 1 and will send a message to Rank 2 to activate the next *Write* task local to Rank 2. To summarize, we will incur a cost of discovering all 2 Million and 2 tasks and 1 communication in each rank. In the alternate



(a) First Task-Graph



(b) Second Task-Graph

Figure 3.14: Sample Task-Graphs

scheme, Rank 1 will discover all the local tasks and the second *Write* task in Rank 2 (1 million and 2 tasks). Rank 1 will have to send data to Rank 1 and Rank 2 will only discover all the local tasks. In this strategy each rank will incur a cost of discovering 1 million and 2 tasks and 1 communication. Owner-Tracks-Future is a clear winner in this case.

The second sample task graph is depicted in Figure 3.14b. In this example, we have a *Write* task in Rank 1 followed by a *Read* task. In Rank 2 we have 1 million *Read* tasks following the *Read* task of Rank 1. In our current DTD scheme, Rank 1 will discover all tasks but will send only 1 message to Rank 2. Rank 2 will discover all tasks, and hence 1 message from Rank 1 is enough to know when to safely activate the million *Read* tasks. In this case the total cost incurred by each rank will be discovering all tasks (1 million and 2) and 1 communication. In the other scheme Rank 1 will discover all tasks, as it is the owner of the data. Rank 2 will discover only its local tasks, so 1 million *Read* tasks. In this case, Rank 1 will need to send 1 million messages (in the worst case) to Rank 2 for activating each *Read* task, as Rank 2 did not discover the *Write* task of Rank 1, Rank 2 has no way of knowing the correct ordering of the 1 million *Read* tasks. The owner in this scheme needs to drive the ordering of the remote tasks that uses a data it owns, and thus Rank 1 will need to send 1 million messages. The total cost for each rank will be discovering an almost equal number of tasks (1 Million) and 1 million communications. Given the latency of sending a message, it is not clear if the Owner-Tracks-Future is a winner or not. It is obviously performing more work than the current scheme in this case. None of the implementation schemes is foolproof, and our theoretical analysis shows cases where each of them would not perform optimally.

Efforts presented in this subsection to analyze various schemes to implement DTD is mainly exploratory. The aim is to provide a theoretical analysis to aid future research directions. The Owner-Tracks-Future scheme will perform better than the current implementation of DTD on average and the example showing a possible worse performance of Owner-Tracks-Future scheme can be considered an extreme case.

In order to not submit all tasks in each rank, the users either need to explicitly test some conditions or the interface needs to provide that functionality. In any case, submitting tasks sequentially in a distributed environment will require testing some property of all tasks in

each process in order to figure out whether that process should care about that task or not. In the current DTD implementation we chose to care about all tasks for later filtration; in the alternate approach we save cost on this step but we have to check for some property of each task in all ranks—in this case that is whether a task is local or if we are the current owner of a data a task will use. As long as we are testing for some property of **all** tasks in a paradigm, that paradigm will not scale, as we cannot still make the middle part of equation 3.2 a function of P . With the alternate approach, the best we can hope for is to push the explosion of overhead as shown in Figure 3.12 further to the right.

3.6 Integration of DTD with Existing Applications

This section describes integration of DTD with dense linear algebra libraries Chameleon and the Software for Linear Algebra Targeting Exascale (SLATE). Chameleon, which is a joint effort between Inria* and ICL†, offers tile-based dense linear algebra routines where users can choose from various **Runtime** as the underlying driver. SLATE, which is being developed at ICL, is a new library under development that envisions to replace ScaLAPACK. It also provides dense linear algebra routines but moved away from the tile-based algorithms.

3.6.1 Chameleon

Chameleon Agullo et al. (2012) is a dense linear algebra software relying on sequential task-based algorithms where sub-tasks of the overall algorithms are submitted to a **Runtime** system. It is like PLASMA Buttari et al. (2009) library with an option to choose the underlying **Runtime**. The **Runtimes** supported were StarPU and QUARK, where StarPU had support for one algorithm in distributed system and QUARK being limited to shared memory systems. ParSEC was integrated with Chamleon library using the DTD interface and currently has support for shared memory systems. All the experiments and performance result of shared memory systems we present in section 3.4 are collected through Chameleon for all **Runtimes** including ParSEC DTD. This effort allowed ParSEC DTD to have access to

*<https://www.inria.fr/en/>

†<http://www.icl.utk.edu/>

a complete library of dense linear algebra routines, which made checking the correctness of implementation and analysis of performance possible for a wide range of algorithms.

3.6.2 SLATE

The objective of SLATE [SLATE \(2019\)](#) (Software for Linear Algebra Targeting Exascale) is to replace, de-facto library providing dense linear algebra kernels in distributed systems, Scalable Linear Algebra PACKage (ScaLAPACK). SLATE provides shared-memory parallelism through OpenMP and performs explicit communications through MPI. `PaRSEC` was plugged in as an underlying `Runtime` replacing OpenMP and MPI through the DTD interface. SLATE decided to move away from the full tiled approach of previous dense linear algebra library PLASMA. As the kernels have been taskified and data dependencies are being utilized through OpenMP, a full tile-based algorithm would result in resolving too many dependencies. They moved to an algorithm where there will be levels of tasks, and dependencies will only exist between high-level tasks and the low-level tasks will be embarrassingly parallel. This model moves away from the dataflow model and relies on explicit communication to tackle the bottlenecks of fine-grained parallelism in distributed environments.

Cholesky Factorization was taken as the target algorithm to be ported to test the feasibility of using a `Runtime` instead of the MPI+X paradigm for SLATE. As mentioned above, this factorization has 4 different kernels: POTRF, TRSM, HERK and GEMM. The tile-based algorithm inserts each kernel and creates the dependencies among all tasks; here, there is no notion of level of tasks—all tasks are of the same level. In SLATE for Cholesky factorization, the algorithm is designed to progress "panel by panel," where a panel is a column in the matrix. The idea is to cluster all the TRSM tasks following a POTRF task as one big *PANEL* task and all the HERK and GEMM tasks as one big *UPDATE* task. Figure 3.15 shows the high-level overview of cluster of tasks. Previously, each task would have the same granularity, but in this scheme fine-grained tasks are grouped into one large task, in turn making the grain coarser, to save overhead of dependency resolution.

To realize this algorithm in DTD, existing features were used, and DTD's current capabilities were extended. Some of the utilized features are:

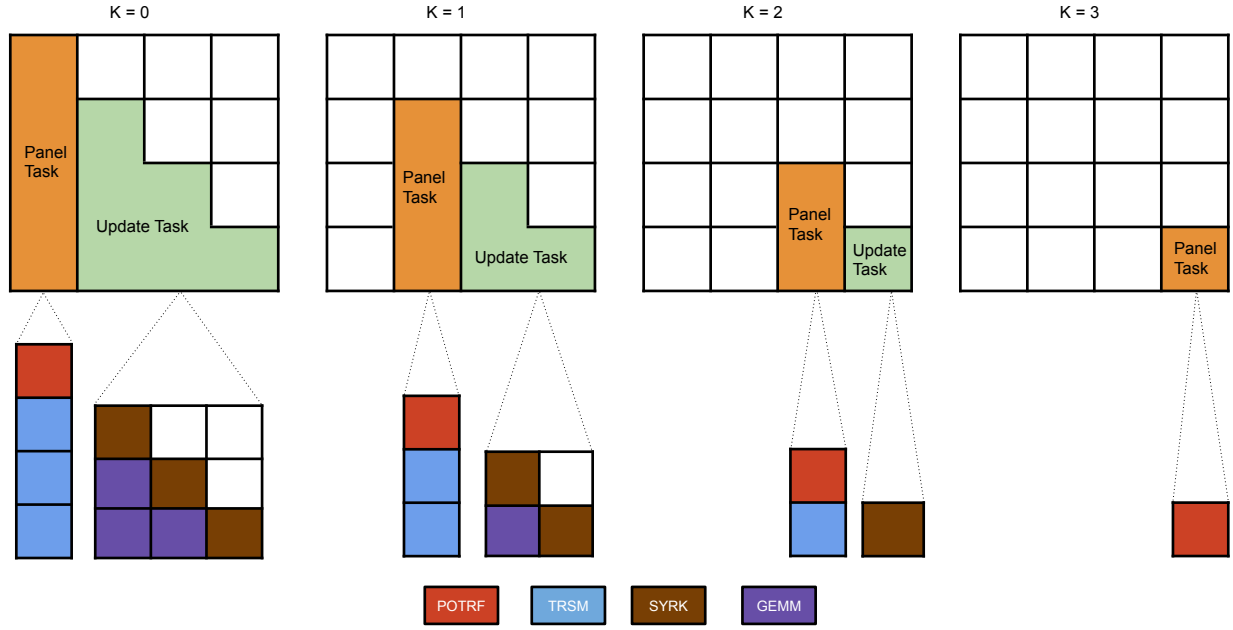


Figure 3.15: Panel-by-Panel Cholesky Factorization

- DTD taskpool (collection of tasks) having the capability to trigger an event when all tasks inserted so far are completed, by invoking a callback.
- DTD tasks being able to create taskpools to offload work.
- Recursively inserting tasks.
- Wait on a taskpool to complete all the tasks inserted so far in that taskpool.

There were different levels of taskpools, where the main thread of each process would only insert high-level tasks in the high-level taskpool. The dependencies between high-level tasks were expressed using a 1-dimensional array as the progression was "column-by-column": each cell in the array could represent a column. Communication would be inserted by the task themselves as needed (explicit) and the high-level tasks would insert smaller fine-grained tasks which are embarrassingly parallel.

Here, the communications are explicit and hence collective operations are possible using DTD. DTD implicitly inferring communication sacrifices the capability to sense collective communications, as different participating processes will have a different views of the task graph at the same instant. After a POTRF task is done it can send the data it modified to all

the ranks that have a TRSM task. This communication is a broadcast communication except all the participating process will not participate. We call this a multicast communication. After the big task representing the smaller TRSM task is complete, the data can be sent to the *UPDATE* tasks and that results in another multicast communication. SLATE originally performed this multicast by creating separate MPI communicators for each multicast. PaRSEC has its own communication-ordering layer that supports collective communications and the communication layer of PaRSEC was augmented to have support in DTD for these multicasts. Figure 3.16 shows an example of multicast by a high-level POTRF task.

To test the implementation of SLATE on top of DTD, experiments were run on a cluster of 64 Intel Westmere CPU with 12 cores each connected by InfiniBand QDR interconnect. Figure 3.17 shows the performance of each library. The y-axis shows the performance; the higher the line for a library the better the performance. The x-axis shows the size of the matrix being factorizing. Cholesky Factorization was performed, and to test scalability problem scaling was followed, where the resources are kept the same while problem size is

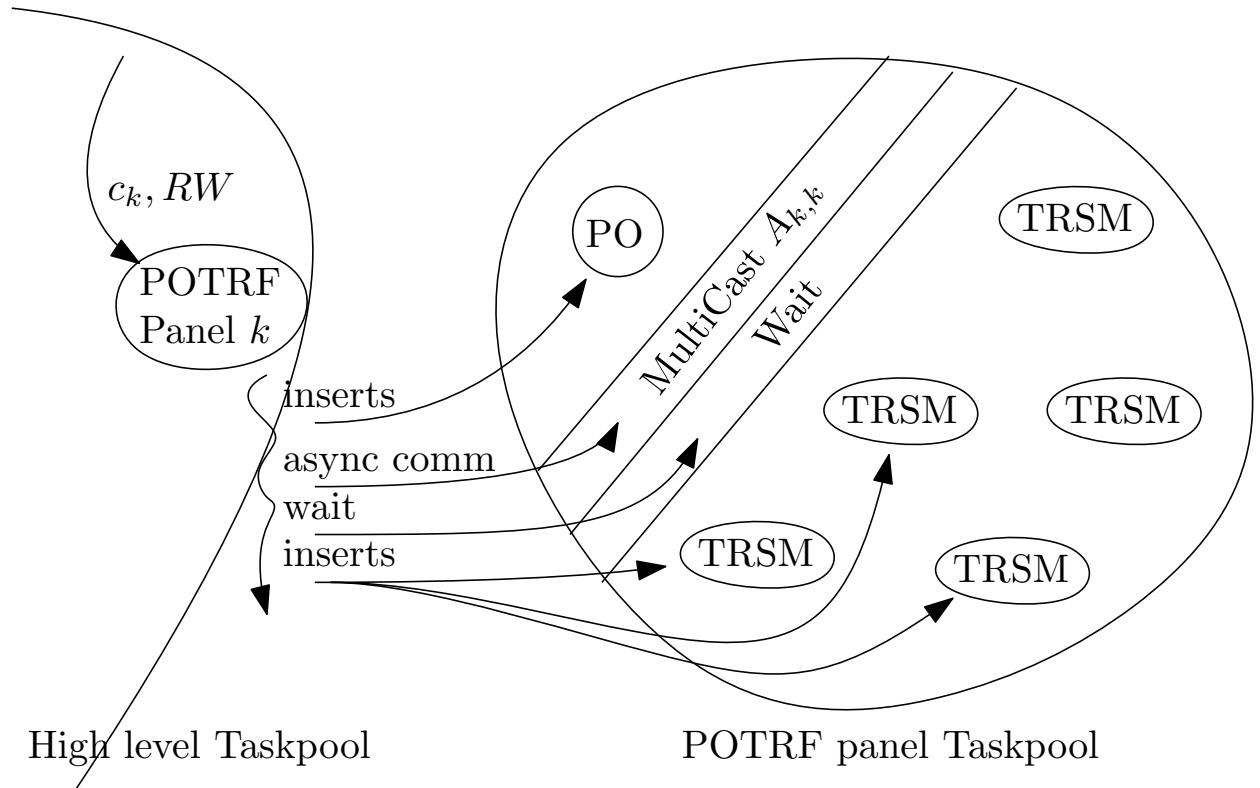


Figure 3.16: Multicast by POTRF task

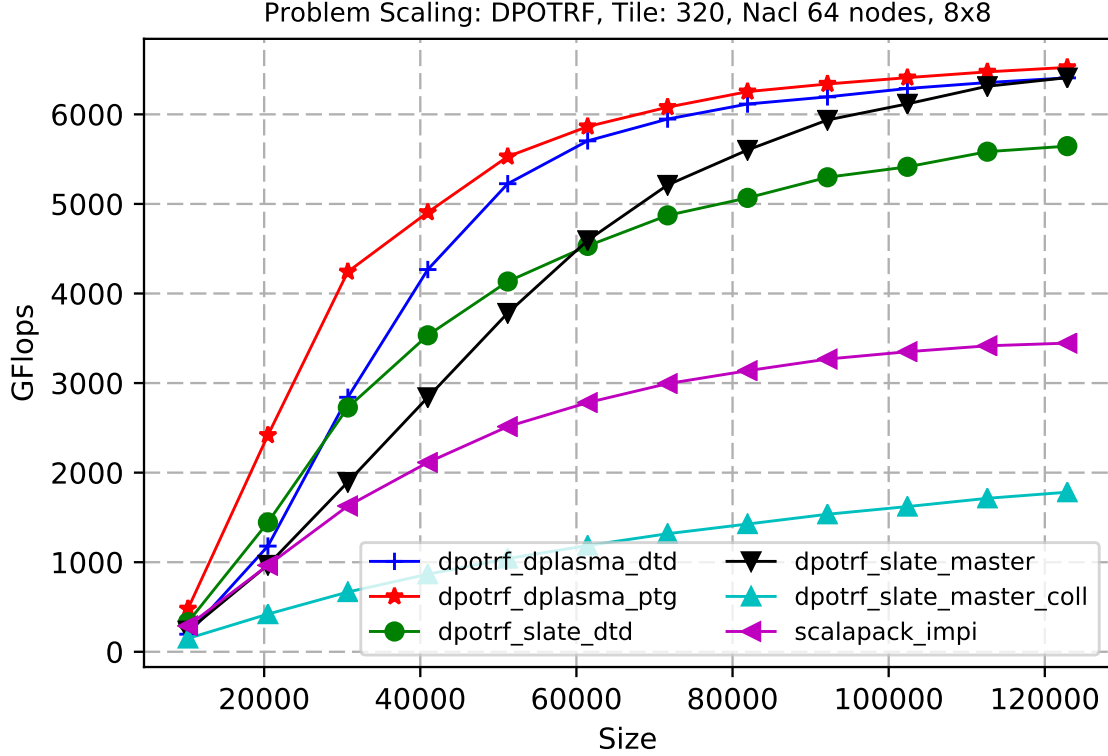


Figure 3.17: Performance Result of Cholesky Factorization

increased. Performance of tile-based Cholesky Factorization from the DPLASMA [Bosilca et al. \(2011\)](#) library—which is on top of PaRSEC—is also shown for comparison. Two versions of the original implementation of SLATE’s performance is presented here, one that performs multicasts and one that performs point-to-point communications. Performance of ScaLAPACK and the implementation of SLATE on top of PaRSEC DTD is also shown.

The performance of DPLASMA PTG is the best, with DTD closely following in this case. Performance of SLATE with point-to-point communication reaches the top performance of DPLASMA at very large matrix sizes at around matrix size $110K \times 110K$. For SLATE with collective communication it was forming a new communicator in MPI for every multicast; forming new communicators is a very expensive operation which resulted in the low performance we observe in this experiment. SLATE with PaRSEC DTD matches the performance of DPLASAMA DTD but reaches a plateau early. After investigation, the

cause of the early capping of performance was found to be caused by an implementation of internal hash table in `PaRSEC`.

3.7 Collaborative Efforts Using DTD

Applications on top of `PaRSEC` DTD that have been discussed so far are developed majorly by the `PaRSEC` team. However, DTD has been used by other research groups as well to program `PaRSEC`. Such efforts are discussed in this section.

3.7.1 Electronic Structure Methods

To test the feasibility of dataflow programming for electronic structure methods, `PaRSEC` DTD was used along with StarPU and OpenMP in a study by Jagode et al. (2018). In this work the main objective was to move away from programming for a specific architecture and move to dataflow programming for portability and performance. Real life application like electronic structure theory was chosen as the workload, as most of computational chemistry applications are unable to exploit the performance offered by the modern supercomputers. Two major types of dataflow forms, implicit (DTD) and explicit (PTG), were tried and tested for programmability, resource utilization and scalability. For the evaluation, state-of-the-art NWChem coupled cluster methods were ported to dataflow programming model. Three `Runtimes` were used—`PaRSEC`, StarPU, and OpenMP—where both interfaces of `PaRSEC` were used: DTD for implicit dataflow and PTG for explicit dataflow. Coupled cluster computation was performed on Beta-Carotene (tile size 45) and the computation was performed at 3 scales—32, 64, and 128 nodes. Performance of all `Runtimes` and all forms of dataflow were compared against the original implementation of NWChem. At the highest scale of 128 nodes, `PaRSEC` DTD performed the best among the implicit dataflow paradigms (`PaRSEC`, StarPU and OpenMP) and achieved a speedup of $1.5\times$ compared to the original implementation.

3.7.2 INTERTWinE

INTERTWinE is a joint European project that addresses the problem of programming models at exascale. Its objective is to investigate hybrid programming models by combining existing ones; the focus is especially on task-based programming models. Three task-based **Runtime** systems were part of this project, OmpSs, StarPU and PaRSEC, and two communication libraries, MPI and GASPI [GASPI \(2017\)](#). PaRSEC’s main collaborator in this project was the University of Manchester. As an effort to test the performance of dense linear algebra routines in a distributed environment, Cholesky Factorization was developed from scratch using both PaRSEC DTD and StarPU. The performance was then tested on the Archer cluster containing Intel Ivy Bridge CPUs with 24 cores. The experiment was run at a scale of 100 nodes with increasing matrix size. PaRSEC DTD, StarPU, and ScaLAPACK were tested in this experiment and PaRSEC had the best performance for all matrix sizes compared to the other libraries.

3.8 Conclusion

This chapter presents an intuitive and easy to program interface for any dataflow task-based **Runtime** system, Dynamic Task Discovery, supporting both shared- and distributed-memory environments. The differences between existing task insertion paradigms were highlighted, several automatic **Runtime**-level optimizations were described and implemented, and the new paradigm’s performance was analyzed 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 **Runtimes**. Discussions were also presented on the benefits and drawbacks of the DTD programming model compared to PTG. Analysis of theoretical limitation of the DTD programming paradigm and different implementation schemes have been discussed with clear examples showing advantages and disadvantages of each scheme. Finally, description of integration of DTD with existing applications and external collaborations that utilize DTD to extract excellent performance have been presented.

Chapter 4

Current State and performance of Runtimes *

4.1 Introduction

High-performance computing (HPC) has been evolving in the multi-core direction for the last two decades. Given the limitation on performance of one processor, fat nodes (nodes with many-core count) have been the solution to increase performance. The only way to extract performance out of a multi-core machine is through parallelism existing in an application, but expressing the parallelism efficiently while managing the underlying physical resources is a tough challenge. Many solutions to this problem are proposed, the most prominent being OpenMP [OpenMP \(2013\)](#). OpenMP lets users define shared and private variables for each core (thread) and provide pragmas to parallelize sequential constructs like *for*. OpenMP has included support for tasks and dependencies between tasks from standard 4.0, but currently supports shared-memory architecture only. Along with multi-core processors, accelerators like GPUs have also become mainstream for general-purpose computing. This puts additional burden of off-loading computations to the accelerator and managing the data movement to and from the host on the application developer. Scaling up any problem

*Material in this chapter has been published in R. Hoque and P. Shamis. Distributed task-based runtime systems - current state and micro-benchmark performance. In 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems(HPCC/SmartCity/DSS)

results in using multiple hosts with accelerators connected to each other through high-speed interconnects to accommodate the increased requirement of memory and computation. This adds yet another burden: communication between the distributed-memory machines. So, the challenge now includes writing parallel application for many-core heterogeneous architecture with distributed memory. Out of the many proposed solutions, the most widely used for communication in distributed memory system is the Message Passing Interface (MPI) [Forum \(1993\)](#). Users need to use a combination of a shared-memory parallelism manager like OpenMP with MPI or use MPI alone where communication in shared memory has more overhead. Alternative to this solution are task-based **Runtime** systems that have recently gained popularity.

In MPI, users need to explicitly express the communications among the different nodes and need to insert synchronization points to ensure correct program ordering with communication. This induces bulk synchronizations in the resulting application. Task-based **Runtimes** alleviate this problem by allowing the users to express their computation in the form of tasks which are asynchronous. The support of asynchronous computation and overlapped communication in task-based **Runtime** comes with the overhead of managing and bookkeeping the tasks and related communications. Users need to express different kinds of computations in form of tasks and express the correct ordering of the tasks through dependency between them. **Runtimes** through this expression of tasks and dependencies create a directed acyclic graph (DAG), also called task graph, where each node in the graph represents a task and the edges, dependency among the tasks. From such DAG, the **Runtime** can safely identify the correct ordering of tasks and the tasks that can execute in parallel.

There are numerous task-based **Runtimes** offering different features and giving control to users at different levels. All the **Runtimes** leverage libraries like MPI or GASNet [Yelick et al. \(2007\)](#) for communication. For this survey, we have selected the **Runtimes** that:

1. Support distributed-memory systems.
2. Have active researchers and developers.

The study of considered **Runtimes** on which we present this paper are: PaRSEC [Bosilca et al. \(2013\)](#), Legion [Bauer et al. \(2012\)](#), StarPU [Augonnet et al. \(2011\)](#), HPX [Heller et al. \(2013\)](#),

Charm++ [Acun et al. \(2014\)](#), Uintah [de St. Germain et al. \(2000\)](#), and STAPL [Papadopoulos et al.](#). Runtime systems such as OCR [OCR \(2016\)](#), CnC [Burke \(2010\)](#), and others were also studied but we excluded them from the scope of this work as they are in a very early stages of development, lacking most of the features considered important in this study. The main contribution of this work is a comprehensive study of the current Runtime systems with support for distributed memory, a micro-benchmark suite and their evaluation. The subjective and objective assessment presented here is based on the firsthand experience of programming each one of them for the benchmark development.

The organization of the rest of this Chapter is as follows: In Section [4.2](#), previous work surveying Runtime systems are discussed and Section [4.3](#) explains the terms that will be used throughout this literature. Section [4.4](#) discusses the challenges faced by application developers trying to use task-based Runtimes and the explanation of how the Runtimes are classified. Section [4.5](#) discusses the Runtimes, categorizes them, and presents the features offered by each. Section [4.6](#) explains the simple benchmarks designed to assess the Runtimes and presents the results. Section [4.7](#) shares our experience of developing the benchmarks. Finally, Section [4.8](#) concludes on the state of task-based Runtimes supporting distributed-memory machines and future work.

4.2 Related Work

A report [Bennett et al. \(2015\)](#) from Sandia National Lab provides an in-depth analysis of three Runtimes: Legion, Uintah, and Charm++. This report dives deep in exploring the differences of each of the Runtime from subjective and objective points of view. They selected a miniapp called miniAero and tried porting it to different Runtimes. Code snippets of each port are presented with results of surveys to assess the subjective metrics. Performance results of the miniapp for each Runtime are also presented in that report.

Another report [Wilke et al.](#) from Sandia National Lab covers more Runtimes on a conceptual level, where different terminologies are defined and different models followed by each are discussed. Implementation of benchmarks or porting of any miniapp for the considered Runtimes were not attempted in that study.

This study aims to be in between these two reports, going deeper than the latter and more on the surface than the former. Through this study, attempts were made to make a performance perspective for the considered **Runtimes** from simpler benchmarks focusing on common key parameters that will affect any task-based **Runtime**.

4.3 Terminology

There are multiple concepts and terms used by each **Runtime** to describe the different parts and features; we will start by defining the very basic terms that are common to all of them. A **task** is any form of computation that does not block and takes some data as input and produces output. A **dependency** can be described as the ordering mechanism for tasks (e.g., *Task B* dependent on *Task A* enforces that *Task A* will have to complete before *Task B* can start executing). Data consistency and correct program order is maintained using dependency between tasks in task-based **Runtime** systems. **Data** is input to a task and output from it. **Events** are entities to connect multiple tasks in a producer-consumer relationship.

A **task graph** or DAG is the graph that is created by the **Runtime** from the expression of the algorithm. Each task is represented by a vertex and each edge in the graph represents a dependency between two tasks. Figure 4.1a shows a sample task graph. Figure 4.1b shows a sample task graph that crosses node boundaries. The dashed line represents the boundary and the edge crossing the boundary is a communication for triggering a dependency activation and moving data. Communication takes place when data produced by a task need to move to a remote node where a consumer task will execute. **Dataflow** describes how the data is flowing through the tasks and can be used to identify all the dependencies automatically by a **Runtime**.

Task-based Runtime is a system that takes the expression of user’s “*taskified*” algorithm and executes it efficiently in modern HPC systems, without involving the users to manage and map the computations to the hardware. Data-flow based programming model allow users to express the data each task will be using and based on the flow, infers the dependencies

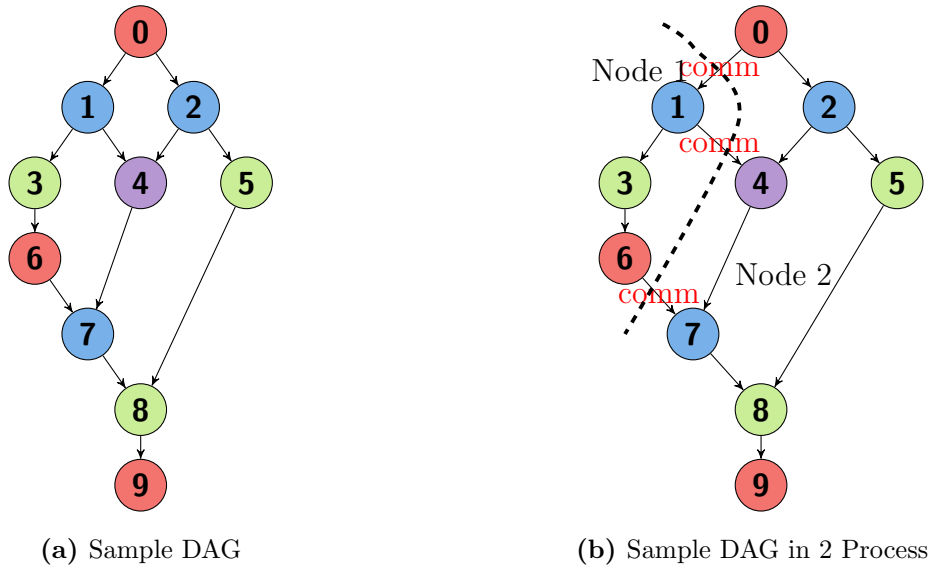


Figure 4.1: Sample Task Graph

and orders the task accordingly. For example, *Task A* uses *data D* and *Task B* will use *data D* after *Task A*, the **Runtime** seeing the flow of *data D* will order *Task B* after *Task A*.

4.4 Runtime Features and Characteristics

A task-based **Runtime** abstracts the computational resources to the application developer, but this advantage comes with a cost—users are required to express their application to the **Runtime**. This cost is determined by the programmability of each **Runtime**. This can be measured using the interfaces of each **Runtime**, which may be API based in selective languages or a language specific to a **Runtime**. This measure is subjective, but given that developers are already comfortable in one of the popular programming languages (*C*, *C++*, *Python*, *etc*), using the API of one such language is less challenging than learning a new language only related to a specific **Runtime**. The ease of porting existing parallel scientific application is a determining factor for the users.

Runtimes are categorized based on the following aspects:

High-level and/or low-level Runtime: High level indicates having support for a high-level programming model to identify dependencies. Low level indicates that the dependencies

are explicitly expressed by the application in the form of events or tags to connect related tasks.

Static and/or dynamic task graph: This describes the kind and the time when the task-graph is built. Some **Runtimes** have the capability to build the task graph at compile time, reducing significant overhead at runtime. Some build it during runtime, having the advantage of accommodating dynamically changing task graphs.

Load Balancing mechanisms: There are two forms, intra-node and inter-node. Intra-node deals with balancing load in shared memory and inter-node deals with distributed memory. Balancing intra-node load is simpler and less expensive than inter-node, as it does not involve communication or transferring data.

Fault Tolerance support: With more cores and complex architecture the mean time to failure (MTTF) will be decreasing, making fault tolerance an essential part of any **Runtime** for large-scale (exascale) computing. There are two types of faults, soft error and hard error, where soft error is corruption of data and hard error is a computing resource failing completely.

Tools for profiling applications: Making an application port or writing a new application on top of any **Runtime** correctly can be assisted and hence accelerated by profiling tools. **Runtimes** having support for inspection of the task graph and tools to profile different aspects related to an application like communication, task priority, etc., can help identify bottlenecks and in turn improve performance.

Modular: Having the facility to add custom components to a **Runtime** is a desirable feature, e.g., being able to employ a custom scheduling strategy for an application or being able to add a counter to measure an event custom to an application.

4.5 The Survey

In this section, each feature described in Section 4.4 for each **Runtime** is discussed. A summary of the **Runtimes** is presented in Table 4.1.

Table 4.1: Summary of the features and characteristics of each Runtime

Runtime	Runtime Level		Task-Graph		Load Balancer		Fault-Tolerance		Profiling Tools	Modular	Heterogeneous
	Low	High	Static	Dynamic	Intra-node	Inter-node	Soft	Hard			
PaRSEC	✓	✓	✓	✓	✓	×	✓	×	✓	✓	✓
Legion	✓	✓	×	✓	User	User	×	×	✓	✓	✓
StarPU	✓	✓	×	✓	✓	×	×	×	✓	×	✓
HPX	✓	×	×	✓	✓	×	×	×	✓	✓	×
Charm++	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓
Uintah	×	✓	×	✓	✓	✓	×	×	✓	✓	✓
STAPL	✓	✓	✓	✓	✓	×	×	×	✓	✓	×

4.5.1 PaRSEC

PaRSEC is a dataflow task-based Runtime that supports distributed heterogeneous architectures. It is developed at the Innovative Computing Laboratory at the University of Tennessee, Knoxville. PaRSEC started as the underlying Runtime for the DPLASMA [Bosilca et al. \(2011\)](#) library, which is an extension of PLASMA for distributed memory, and later evolved as a generic Runtime. PaRSEC allows users to distribute data using its data interface, and infers and performs all communications among participating nodes through the dataflow. This relieves the users of expressing communication explicitly. PaRSEC manages both shared and distributed resources and uses MPI to manage inter-node communication. Furthermore, dense linear algebra routines implemented on top of PaRSEC help it to outperform [Danalis et al. \(2014\)](#) the de facto library for distributed dense linear algebra, ScaLAPACK. Some recent experiments [Jagode et al. \(2016\)](#) on computational chemistry applications have shown improved performance as well. PaRSEC, first published in 2010, is written in C and has interfaces for Fortran and C++.

High-level, Low-level PaRSEC is a high-level Runtime which employs a dataflow model to infer task dependencies and communications required to execute an algorithm in a distributed environment. It currently supports two high-level programming models—Parameterized Task Graph (PTG) [Cosnard et al. \(1999\)](#) and Dynamic Task Discovery (DTD) [Hoque et al. \(2017\)](#).

Static, Dynamic Task Graph PaRSEC has support for both a static task graph (PTG) and a dynamic task graph (DTD). In the PTG programming model, the user needs to express their algorithm in a parameterized manner, where they indicate the different types of tasks

and the flows of each type of task using the Job Data Flow (JDF) [Danalis et al. \(2014\)](#). The JDF, which is PaRSEC’s PTG language, is then pre-compiled into *C* code that contains the static task graph. The DTD interface is an API-based interface that allows users to write sequential code to insert task in PaRSEC at runtime.

Load Balance For intra-node load balancing, PaRSEC uses work stealing among local threads. PaRSEC does not have any strategy implemented for inter-node load balancing.

Fault Tolerance PaRSEC supports resilience for soft errors [Cao et al. \(2015\)](#) and work is underway to provide support for hard errors. To recover soft errors, PaRSEC re-executes the minimum required sub-DAG and checkpoints data for re-execution.

Profiling Tools and Modularity PaRSEC provides rich profiling tools to examine system and user code. It has a profiling interface with accessor to Python and R and uses PAPI [Mucci et al. \(1999\)](#) to extract hardware counters. It allows users to visualize the task graph with task and data identifiers to check the correctness of the expression. It also allows measurements such as the cost of a task, communication payload, and total communications. PaRSEC follows a module component architecture, allowing users to add their own scheduling policy, events to profile, etc.

4.5.2 Legion

Legion is a Runtime developed at Stanford University. It introduces a concept of logical regions to virtually represent partitioned real data to infer task dependencies. For each task using any logical regions, users provide coherence and privilege information, and Legion extracts the parallelism present depending on the provided constraints. Communication among remote nodes are not required to be expressed explicitly as it is managed by Legion. It uses GASNet for inter-node communications and has support for heterogeneous architectures. Legion has shown good performance on benchmarks [Bauer et al. \(2012\)](#) and a port of S3D [Chen et al. \(2009\)](#) on it has shown good speedup [Bauer \(2014\)](#) at scale compared to the base implementation. Legion provides a strong and flexible interface to decompose any data into logical regions to help port applications exhibiting different behaviors. Legion, first published in 2012, is written in C++.

High-level, Low-level Legion provides high-level programming model through Legion and Regent [Slaughter et al. \(2015\)](#) and low level through Realm [Treichler et al. \(2014a\)](#). Realm uses events to connect and order tasks—events in this case are completely asynchronous and does not require any central management. Regent, Legion’s own programming language, lets a user write sequential programs which it then translates into Legion code. This minimizes complexity during mapping physical regions into logical ones and, with the help of static analysis and optimization at compile time, generates programs with performance similar to hand written Legion code.

Static, Dynamic Task Graph Legion builds a dynamic task graph at runtime, using both the high-level and low-level interface. It does not have any mechanisms to build a static task graph at compile time.

Load Balance Legion provides hooks for stealing tasks in the form of pushing or pulling to balance load in shared or distributed memory. Users are responsible for explicitly calling these hooks. There is no implicit mechanism in the `Runtime` to automatically balance the load.

Fault Tolerance Currently, there is no fault tolerance mechanism in Legion. It is reported that active development is going on to add support for resilience [Bennett et al. \(2015\)](#).

Profiling Tools and Modularity Legion has support for profiling tools that can measure important parameters such as scheduling cost, data movement, etc. It also has utilities that can generate a visual representation of the task graph to help users check correctness. Legion uses a mapper interface that isolates the correctness of an application from the performance. Users are responsible to express the task mapping and distribution using this mapper interface. This gives advanced users the flexibility to decide the scheduling policy and other aspects they want to control. A default mapper is also provided for beginners.

4.5.3 StarPU

StarPU is developed by STORM research team at Inria, located in Bordeaux, France. It started as a high-level `Runtime` for shared-memory systems with support for accelerators,

and later on added support for distributed memory as well. This was achieved in a two step process: (1) allow users to put communication explicitly, and (2) infer communication from dataflow (implicit). StarPU has its own data interface to manage data movement and versioning. For inter-node communication it uses MPI and has support for heterogeneous architectures. Multiple applications such as Explicit Finite-volume CFD [Carpaye et al. \(2017\)](#), Seismic wave modeling [Martínez et al. \(2015\)](#), and dense and sparse linear algebra kernels have had better performance with StarPU as the underlying **Runtime**. StarPU, first published in 2008, is written in C and has interfaces for other programming languages.

High-level, Low-level StarPU allows users to both implicitly and explicitly express dependencies between tasks, making it a high-level and low-level **Runtime**. It provides a dataflow programming model allowing users to insert tasks. The **Runtime** infers the dependencies to order tasks from the information provided by the users regarding the data required by each task the their privilege. It also allows more flexibility to insert tasks with explicit dependencies using tags. The interface for inserting tasks in the system is by writing sequential code using the provided API.

Static, Dynamic Task Graph StarPU builds the task graph dynamically during runtime. For distributed memory, a global view of the task graph is maintained to connect the local and remote tasks. The global view can be pruned but needs user input. There is no support for building a static task graph.

Load Balance StarPU provides task-stealing mechanism for intra-node load balancing. However, no support is mentioned in the literature for the distributed-memory environment.

Fault Tolerance There is no support for fault tolerance.

Profiling Tools and Modularity StarPU has rich profiling tools to track task graphs, generate traces, and measure different events which users can utilize to analyze performance bottlenecks. There are multiple scheduling strategies present in StarPU; some of the notable ones are: Heterogeneous Early Finish Time (HEFT) and Local Work Stealing (LWS). It is not mentioned in the literature if users can add custom scheduling schemes or other components.

4.5.4 High Performance ParalleX (HPX)

The first implementation of the ParalleX execution model is HPX, which is mainly developed at Louisiana State University. Currently, it supports distributed memory but does not support heterogeneous architecture. HPX is based on the Active Global Address Space (AGAS) programming model, where the distributed memory is abstracted as a global address space, and HPX has its own implementation of AGAS. It strictly conforms to the C++ standard to form its API and has support for all the multithreading interfaces like future, thread, async, etc. It has been shown to perform well for benchmarks and applications like Jacobi factorization and N-body [Khatami et al. \(2017\)](#). HPX, first published in 2009, is written in C++.

High-level, Low-level HPX is designed to be a low-level **Runtime** where there would be multiple domain-specific languages (DSLs) on top to facilitate application developers from different domains [Kaiser et al. \(2014\)](#). It uses future objects, on which future tasks can wait, and dataflow objects which can wait on multiple future objects and provide support for high-level expression. In both cases users are responsible to create the dependencies.

Static, Dynamic Task Graph HPX builds task graphs dynamically at runtime. It does not support static task graphs.

Load Balance Work stealing strategies are implemented to support intra-node load balancing. No inter-node support has been implemented yet.

Fault Tolerance There is no support for fault tolerance.

Profiling Tools and Modularity The modular design of the system enables each module to deal with different responsibilities; however, it is not clear if users can add custom modules. HPX has a performance counter framework allowing developers access to different hardware and software counters.

4.5.5 Charm++

Charm++ is one of the most mature **Runtime** systems, and is developed at the University of Illinois at Urbana-Champaign. It is based on concepts like over-decomposition, migratability, and asynchronous message-driven execution. Over-decomposition is to break down each

computation to smaller ones to increase parallelism, migratability is to allow tasks and data to migrate inter-node, and asynchronous message-driven execution is to transfer control from one unit to the other through messages. Charm++ has support for heterogeneous architectures. Charm++ uses asynchronous method invocation for communication and supports a number of backends like UDP, TCP, and MPI. Charm++ has multiple real applications such as NAMD [Kale et al. \(2011\)](#) and applications in chemistry, material science, and others, as well as numerous miniapps. Charm++ can be used in composition with any MPI application as well [Jain et al. \(2015\)](#). Its development started from around early 1990 and is written in C++.

High-level, Low-level Runtime Charm++ is a low-level Runtime, very similar to MPI. The basic object of Charm++ is known as Chares and represents every piece of computation and data in the system. The fundamental principle of Charm++ is to have migrate-able Chares. For message passing, Charm++ uses remote method invocation. For each Chare, the user needs to specify a method—the entry method, which can be invoked remotely. Users need to express the parallelism using Charm++’s own language in the *.ci* files. Charm++ does not have a high-level programming model and therefore does not support dataflow inspection. The communication as well as the dependencies have to be manually managed by the user.

Static, Dynamic Task Graph A dynamic task graph is built by Charm++ from the users’ expression at runtime. Charm++ does not support building static task graphs.

Load Balance The portability of Chares allows Charm++ to have both intra-node and inter-node load balancing. There is a suite of load balancing strategies implemented in Charm++ providing support like centralized, hierarchical, and distributed. All the load balancing takes place dynamically by observing the behavior of the application.

Fault Tolerance Charm++ is resilient to both hard and soft errors by checkpointing and restarting once failure occurs.

Profiling Tools and Modularity The load balancers are modular in Charm++, allowing users to write their own. It also provides a suite of performance tools such as projections and a debugger with GUI support to analyze an application.

4.5.6 Uintah

Uintah is developed by the Center for Simulation of Accidental Fires and Explosions in Utah. It is a domain-specific, task-based **Runtime**. It supports distributed memory with heterogeneous architectures where communication is performed using MPI and is inferred by the **Runtime** from the dependencies between tasks. Uintah has the concept of a data warehouse, which controls all the data movement between CPU cores, GPU cores, and in between nodes; one can view this as a data server controlling all aspects related to data. Uintah strictly describes data as uniform patches and is limited to applications pertaining to structured meshes. It cannot deal with data of any shape and structure. Tasks are expressed for an algorithmic step and then repeated on patches. Uintah describes this type of task graph as a tensor product taskgraph [Parker et al. \(2006\)](#). It is primarily application focused and has multi-physics modeling including fluid dynamics, chemical reaction, and turbulence. It is shipped with these applications as components, making it readily available for use. They have shown scaling of up to 98k cores for AMR applications [Luitjens and Berzins \(2010\)](#). It started in 1997 and is written in C++.

High-level, Low-level Uintah is a high-level **Runtime** highly suited for problems with structured meshes. It employs a dataflow model to infer dependencies between tasks to order them correctly. It uses multiple notations to define the meshes like patch, cell, node, face, ghost cell, etc. to express the data and the dependencies. All communications are implicitly handled by the **Runtime**.

Static, Dynamic Task Graph In almost all of the literature, Uintah uses the term *compiled taskgraph* to express the DAG, and this creates confusion in identifying whether it builds the task graph at runtime or compiles it as a static task graph. In practice, Uintah creates the task graph dynamically during runtime and does not support building a static task graph at compile time.

Load Balance For load balancing, Uintah has support of its own highly parallel load balancing algorithm, as well as the Zoltan load balancing package, to offer easy access to different load balancing algorithms.

Fault Tolerance There is no support for fault tolerance.

Profiling Tools and Modularity Uintah is built on the DOE's common component architecture (CCA) which makes it highly modular. All the aspects that are disjoint are developed as separate components facilitating the development process of future applications. It provides tools to check the correctness of simulations and analyze the output, as well as support tools designed to analyze MPI programs.

4.5.7 STAPL

The Standard Adaptive Parallel Library (STAPL), developed at Texas A&M University in 1998, started as a generic **Runtime** with focus on graph algorithms for shared memory. It then progressed to support distributed-memory systems with its Adaptive Remote Method Invocation (ARMI) communication library, which is built on top of MPI. STAPL's design is based on C++ STL and is written in C++. It provides support for standard C++ directives for parallel programming like `move`, `future`, `promise`, etc. for shared and distributed memory. The development team is focused on making STAPL compliant with the C++ standard. A majority of the applications STAPL works with are based on graph algorithms. They have shown scalable performance on basic graph traversal algorithms like BFS, to more complicated graph algorithms like PageRank. Computational physics and biology applications have also been developed on top of STAPL. STAPL does not have support for heterogeneous architectures yet.

High-level, Low-level STAPL is a high-level **Runtime** providing support for dataflow execution. It has a module PARAGRAPH that resolves the data dependencies to the execution layer from the users' expression of the algorithm. It does not expose the low-level expression to the user.

Static, Dynamic Task Graph STAPL builds a static task graph at compile time. Users use STAPL's Skeleton framework to express their application in a parameterized dataflow form. The Skeleton framework passes it to PARAGRAPH to instantiate the task graph at runtime. STAPL allows creation of tasks dynamically through Task Factories, but has the limitation of assuming some knowledge about patterns present in the task graph. This dynamic strategy is low level given the task identifier of the predecessor needs to be provided during the creation of its successor.

Load Balance Intra-node load balancing is offered in STAPL through work stealing in shared memory. There is currently no support for inter-node load balancing.

Fault Tolerance There is no support for fault tolerance.

Profiling Tools and Modularity Instrumentation of different aspects of an application is provided by STAPL. It is a modular **Runtime** by design, where each component is separated. However, users are not allowed to add their custom component.

4.6 Benchmark

For an application developer, understanding the overheads and performance is as important as the features provided by a **Runtime**. Throughout this study, no efforts were found to develop simple and small benchmarks that assess the overhead of each **Runtime** and allows a comparison. This is the first attempt, to the best of my knowledge, to come up with a benchmark suite to see the overheads of common aspects all the **Runtimes** share. This effort also enabled gaining hands-on experience of developing simple programs on top of these **Runtimes**. For benchmarking, three overheads that are common to the **Runtimes** were decided to be assessed:

1. Spawning and maintaining tasks
2. Dependency resolution
3. Communication

Embarrassingly parallel tasks were launched to test (1). To test (2) and (3), chains of tasks in a distributed environment were run. The behavior of each benchmark is described in more detail later. The main focus while implementing these micro-benchmarks was to make sure a like-to-like comparison is being made (e.g., it was made sure that all the **Runtimes** are given the exact same task graph and have identical task placement in a distributed setting.

Benchmarks were written for 5 out of the 7 **Runtimes** we initially considered. The reason for not considering Uintah and STAPL are lack of publicly available tutorials and domain specificity. The software specifications are given in Table 4.2.

Table 4.2: Software Specification

Runtime	Version	Boost Lib	Comm. Lib
PaRSEC	master commit 03abf75	x	OpenMPI 3.0.0
StarPU	1.2.2	x	OpenMPI 3.0.0
Legion	Release Candidate legion-17.05.0	x	GASNET 1.30.0
HPX	master commit 52dd1e1	1.58.0	OpenMPI 3.0.0
Charm++	6.7.1	x	OpenMPI 3.0.0

Machine Configuration: The experiments were carried out on an Intel(R) Xeon(R) CPU X7550 @2.00GHz with 2 sockets and 10 physical cores on each socket. For distributed runs, a cluster of the same 9 machines were used which were connected by an InfiniBand EDR 100G interconnect.

4.6.1 Embarrassingly Parallel Tasks

By launching embarrassingly parallel tasks, the average duration a **Runtime** takes to generate and consume tasks can be measured. Consistent computation and memory allocations inside a task were performed to tune the duration of a task. The number of tasks, duration of each task, and the core count were varied to see how each **Runtime** performs. The same memory allocator (tcmalloc) was used for all **Runtimes**. The durations of a task written using both *C* and *C++* were measured to ensure all **Runtime** have uniform task duration. The granularity of a task is determined by its duration. The higher the granularity (smaller duration) the less opportunity a **Runtime** has to hide its overhead.

Figure 4.2 shows the performance result of the first benchmark. PaRSEC-DTD, StarPU, and HPX perform almost the same and show very little overhead per task, but it should be noted that HPX does not perform any dependency resolution between tasks like PaRSEC and StarPU. For tasks of duration 116, μ s Charm++ suffers as the number of tasks increases, and that trend is consistent as more cores are added. Charm++ shows the same trend for bigger tasks, with a declining performance for a million tasks on 20 threads. An SMP version

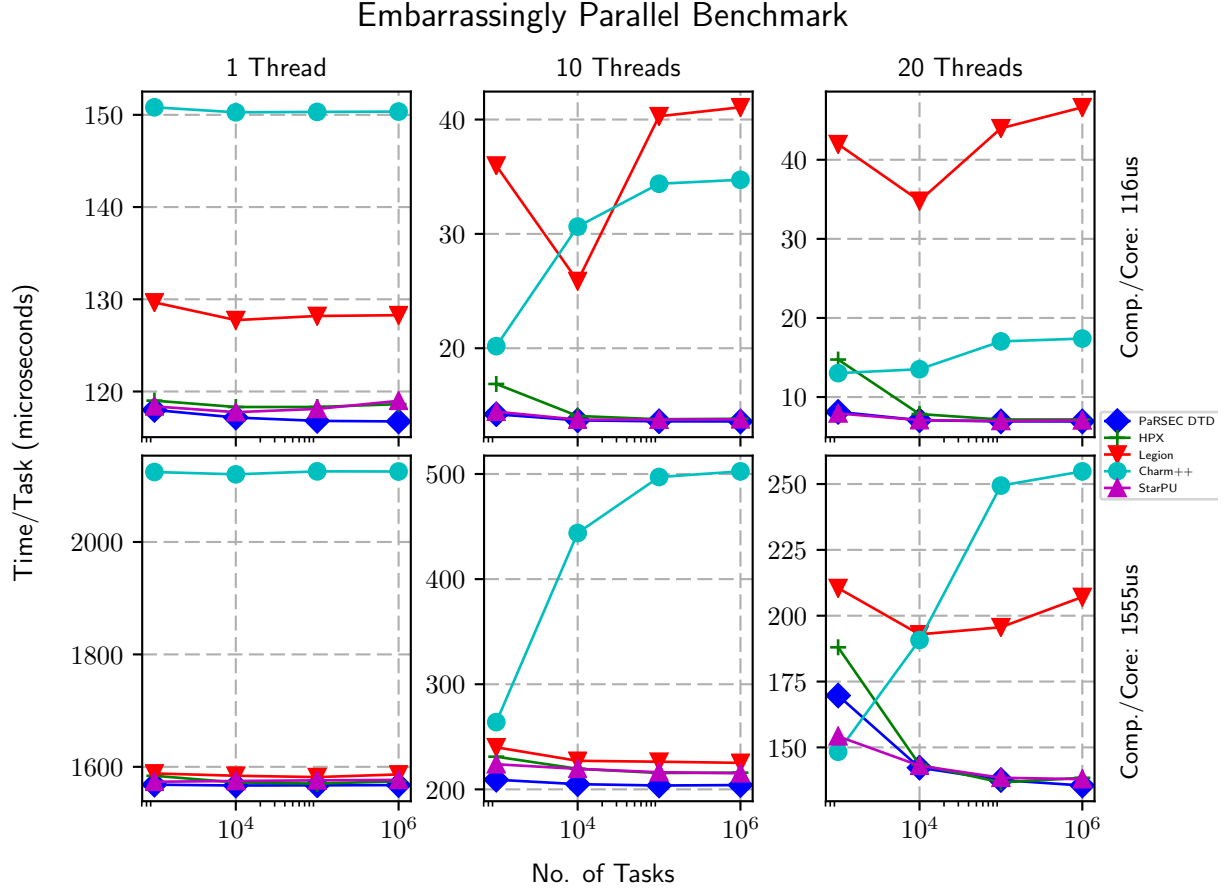


Figure 4.2: Performance result of Embarassingly Parallel Tasks Benchmark

of Charm++ was tried in hope of improving the performance but the result ended up being worse. For smaller tasks, Legion performs worse than the other `Runtimes` on 20 cores. The average time per task increases as the number of tasks was increased. The trend changes for bigger tasks for up to 10 cores but the overhead was observed to be increasing for a million tasks on 20 cores.

4.6.2 Chain of Tasks

With this benchmark the objective was to estimate the communication efficiency of each `Runtime` and the cost of simple dependency resolution in a distributed environment. Figure 4.3 shows how the tasks are distributed and the dependencies among them. Different color indicates that the task is in a different machine. Task in machine 2 depends on the

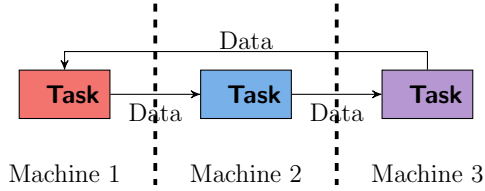


Figure 4.3: Chain of Tasks, each task in different machine

previous task in machine 1 and this pattern repeats for as many machines there are. This pattern is repeated creating a circle among all the machines. E.g. For example, for a total of 9 machines if we run a chain, where each chain would consist of 9 tasks, one in each machine, and this chain was then repeated for a number of times we could get an average time for a chain. Each task in the chain sends an array of integers to the next task, where the number of integers decides the data payload.

Here, the duration of each task and the data payload is varied. Figure 4.4 shows the performance of the different **Runtimes**. The x-axis shows the time per task while the y-axis represents the number of integers, which represents the size of payload. There were 9 tasks in a chain and it was repeated for that pattern for 1000 times, making the total tasks in the system 9000. The total running time of the execution (we only measure the task execution time and not the setting up and tear down time of any **Runtime**) was then divided by the total tasks in the system to get the time per task. One worker thread and one communication thread were used. For tasks with smaller duration, **PaRSEC**, **StarPU**, and **Charm++** perform the same while **Legion** exhibits higher cost and **HPX** suffers from large data payload. To make sure the right build of **GASNet** was being used, both **MPI** and **InfiniBand** conduits were tried, where **Legion** performed same in both cases. For **Charm++**, **MPI** was used underneath instead of using its own communication library on top of **Infiniband**.

Each row in the plot represents the task duration. The same trend is observed in the case of larger tasks except **Charm++** gets more costly than **PaRSEC** and **StarPU**.

4.6.3 Stencil Operation

This benchmark should be representative of general workload and hence should give an overall idea of how each **Runtime** is performing. The dependency resolution is more complex

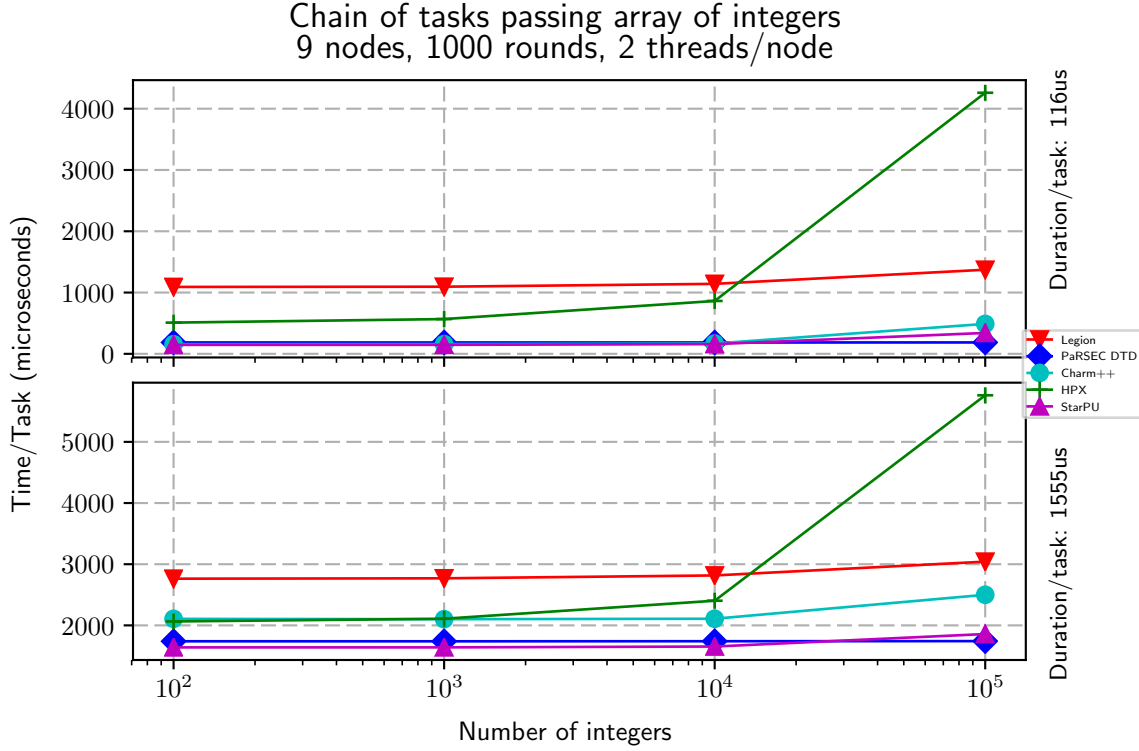


Figure 4.4: Performance result of Chain of Tasks Benchmark

in this benchmark and is similar to real-life applications. Figure 4.5 shows the dependency of a task on others. The whole grid shows a matrix of dimension 5x5. Tasks using the data on the blue square in the middle will use the data where each of the 4 red circles are around it. This resembles Jacobi factorization and the heat diffusion equation. The implementation of this benchmark is not complete yet.

4.7 Productivity

The main objective of this study was to gain hands-on experience and provide a report of the subjective views on productivity to future application developers. The benchmarks developed on top of the various `Runtimes` in this study are simple and trivial, so the findings might not accurately transcribe to most complex applications. However, it allowed me to have a good understanding of the difficulty of using the `Runtimes` for building very basic programs which can be mapped to difficulties of building a complex application.

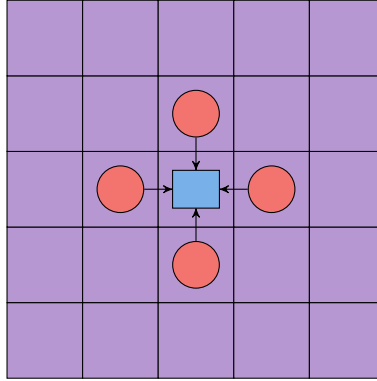


Figure 4.5: Task dependencies in Stencil Operation

PaRSEC: The newly developed DTD interface of PaRSEC was used and it is API based. PaRSEC provides some documentation but is inadequate. However, the development team is active and is willing to engage in collaboration and assisting users. Developers may be reached by creating issues in the public repository or by emails. Installation of this **Runtime** is straightforward. **Examples:** PaRSEC has adequate examples showing how to perform basic operations in a distributed environment.

Legion: Building a new application with Legion was easier compared to the rest, as the process is elaborately documented on their website. Legion's developers maintain a Google group and a mailing list for all issues. This **Runtime** has a prerequisite, GASNet (for distributed memory), which is not as widely adopted as MPI. Legion provides an API-based interface which relieves users from learning their own language, Regent. **Examples:** Basic operations in Legion are neatly demonstrated through the tutorial. Building an application in distributed memory proved to be more complicated than in shared memory. Legion has applications showing advanced usages, but to map work in different nodes requires deeper understanding of their mapper as they have the layers separated.

StarPU: StarPU, with an API based interface, has elaborate and comprehensible documentation, making the installation and development process easy. StarPU developers are very prompt at responding to any queries and they can be reached through emails. The data interface requires some level of understanding and manipulation from the users. **Examples:** A plethora of examples are available through various libraries like Chameleon. They provide examples showing basic operations that help achieve custom operations easily.

HPX: HPX, with an API-based interface, did not prove to be simple to install due to prerequisites like the BOOST library. The installation is also dependent on the *c++* standard during compilation. Fortunately, the installation guide is descriptive, giving ample information about the whole process. Extended documentation of the functionality of the interface is available, which makes building a custom application easy. The development team can be reached through an IRC and is responsive. **Examples:** Even though examples are available, they do not demonstrate all capabilities. Developing a shared-memory application is easier than distributed. The principle of HPX is similar to that of Legion, where they both ask developers to focus on the shared-memory implementation first. The data interface is non-trivial and requires understanding of their components used for communication.

Charm++: Charm++ has the most content, including a wide range of applications ported to it. The interface is a mixture of *c++* APIs and their own language. This makes developing an application on top of Charm++ harder than the rest. Charm++ is very well documented and its developers can be reached via email. Developing custom application in Charm++ is complex, as the language is very explicit and users are required to express a lot more than what is required by other **Runtimes**. **Examples:** Charm++ has plenty of examples showing how to accomplish custom operations, but as it provides dynamic load balancing, the control of mapping work is not given to the users. The tutorials are informative and well designed.

Overall, Legion’s tutorials are exemplary in terms of design and content. It was found that developing applications in distributed memory is non-trivial for all the **Runtimes** considered in this study. Users need to describe the data to the **Runtime** in case of a high-level **Runtime** or move data explicitly for a low-level **Runtime**; accomplishing both was found to be difficult. Task placement in a distributed environment is non trivial as well as understanding the different machine model chosen by each **Runtime**.

4.8 Conclusion

The objective of this study is to report on the current landscape of task-based **Runtime** systems and helping application developers to make informed choices that fit their

requirements. As the computer systems become increasingly hybrid and complex, the path to future high-performance computing will be through **Runtimes**. Given this trend, the hope is this study will help guide each **Runtime** to implement micro-benchmarks and provide examples demonstrating their use in distributed memory. A thorough study was executed on the features most important to users and benchmarking was tried on the selected **Runtimes** on some common overheads they share. The implementation of all the planned benchmarks is not complete and is left as future work.

Multiple research groups are trying to solve the challenge of scalable programming, but there is very little standardization in the **Runtime** ecosystem. The DARMA **DARMA** (2019) group from Sandia National Lab is working towards a standard interface to program each **Runtime**. I believe that this is the correct direction and there is need for more standardization in terminology and concepts to make it easier to use and adapt any **Runtime**.

Chapter 5

Conclusion and Future Work

5.1 Overview

Many-core processors with different types of accelerators are becoming increasingly pervasive in scientific computing. Given this trend, developers need abstractions that will enable them to harness the available compute power transparently, while also providing portability and efficiency. **Runtime** systems like **PaRSEC** promise to provide such abstractions. The next obvious question then becomes how users can program such **Runtime** systems efficiently without sacrifices on any other fronts, such as performance. In this dissertation, a sequential API-based programming paradigm (Dynamic Task Discovery [DTD]) is proposed to program dataflow task-based **Runtime** systems in both shared and distributed environments where communication is implicit. The ease of use of this programming model is also demonstrated using dense linear algebra applications on top of **PaRSEC**. The easiness is subjective and hard to measure but projects involving dense linear algebra in heterogeneous distributed machines such as Chameleon [Agullo et al. \(2012\)](#), HiCMA [Akbudak et al. \(2017\)](#), and others embracing interface similar to DTD indicates that developers have a propensity to write sequential code.

With supercomputers becoming more distributed and heterogeneous with deeper hierarchies of memory, the challenges of programming them is only increasing. As a result, various groups are investing in developing **Runtime** systems that can cater to the rise in demand. As this field is still emerging, it lacks standards such as common terminology of concepts. This

variation in terminology is difficult to interpret for scientists and researchers and therefore needs unification. To do this, an in-depth study was conducted that chose **Runtime** systems that are task based, support distributed-memory systems, and have an active developer base. Different features that are important to the users were also identified, and each **Runtime** was classified based on those features.

With the large pool of **Runtimes** available to application developers, it has become challenging to determine the appropriate **Runtime** suitable for the task at hand for a developer. A study on the current state of task-based **Runtime** systems with details of what features are offered by each and a suite of simple benchmarks to assess the performance of the implementations of each **Runtime** can help application developers make the right choice for their application. As no such effort was found, efforts were put in to designing micro-benchmarks to assess the efficiency of the chosen **Runtimes** with a detailed study and classification of them.

In the following sections, a list of contributions of this PhD is provided and work that can potentially be done is proposed to modify the existing programming model to overcome the discussed theoretical limitations.

5.2 Summary of Contributions

The DTD interface supports the dataflow programming model, where the parallelism of an application is expressed as a DAG. This model coupled with a task-based **Runtime** has proven to outperform legacy approaches like fork-join, as the fork-join model enforces more synchronization points, thereby causing more overhead. In this dissertation, the differences in existing task insertion paradigms like DTD are highlighted, and several automatic optimizations of DTD at the **Runtime** level are described and implemented.

1. **Clever DAG unrolling to restrict memory consumption:** One of the challenges of DTD is to discover and manage the DAG that represents an application at runtime. Clever unrolling of the DAG is designed and implemented that addresses the issue of limiting the memory usage at runtime, and in turn enabled applications of any size to be suitable for DTD. For example, without the clever DAG unrolling, the memory

consumption at each node in a 8-node Cholesky Factorization on a matrix of size $70,000 \times 70,000$ would be approximately 4 GB. This is reduced by a gigabyte when the DAG is unrolled using the implemented approach. The gain in reduction of the memory usage will increase with more tasks, which is dependent on the matrix size and the tile size in this case. This is discussed in Chapter 3, Figure 3.2.

2. **Automatic DAG trimming to restrict memory usage:** Automatic DAG trimming is applied in a distributed environment to restrict memory usage and perform optimal communication. This facilitates management and reuse of internal meta data structures to make ParSEC DTD suitable for high-performance applications. For example, without automatic DAG trimming, the memory consumption at each node in an 8-node Cholesky Factorization on a matrix of size $70,000 \times 70,000$ would be approximately 10 GB. This is reduced by 70% when the DAG is trimmed using the implemented approach. The gain in reduction of the memory usage will increase with more tasks. This is discussed in Chapter 3, Figure 3.4.
3. **Untied, Recursive Tasking and Shared Memory Evaluation:** DTD provides the unique facility of untying the task insertion from any specific thread, which allows users to have recursive tasking in DTD. The performance of DTD’s implementation is evaluated on several widely used dense linear algebra algorithms (Cholesky and QR Factorization). The performance of DTD was found to show good scalability and to be comparable to alternate interfaces like PTG—and to consistently outperform other *Runtimes* with similar capabilities. For example, DTD performed 4 times faster than the existing state-of-the-art *Runtimes* such as StarPU and 5 times faster than QUARK for Cholesky factorization at tile size 64 on an Intel Haswell architecture, as discussed in Chapter 3, Figure 3.6.
4. **Evaluation using multiple applications:** ParSEC has been integrated with libraries providing dense linear algebra algorithms such as Chameleon and DPLASMA through DTD. Prototypes of more explicit programming of dependency and communication for dense linear algebra algorithms were also tested with DTD, such as SLATE, and have shown to achieve increased performance. As a part of a collaborative effort, the Coupled

Cluster method of state-of-the-art quantum chemistry application, NWChem, was also evaluated on top of ParSEC through DTD. ParSEC DTD has shown to perform consistently better than StarPU and OpenMP for an experiment of Beta-Carotene on Coupled Cluster computation on 128 nodes [Jagode et al. \(2018\)](#).

5. **Analysis of DTD’s Theoretical Limits:** The theoretical limitation of this sequential task insertion paradigm is analyzed. An alternative scheme is proposed, Owner-Tracks-Future, which will result in overall decrease in runtime overhead in most cases. Examples are provided of when which implementation of DTD would perform optimally, and it is shown that the theoretical bottleneck of performing some operations for all tasks in each node in a distributed environment cannot be omitted in this paradigm.
6. **Study of Task-Based Runtime systems supporting distributed memory:** Task-based Runtime systems have a very promising path to providing a manageable paradigm to application developers for managing massive heterogeneous systems. As such, we have seen an increase in effort to develop Runtime systems from different groups. This field is still new and, hence, lacks standard. To unify the different concepts, each of the Runtimes are introduced, and to identify different features that are important to application developers—and features that have practical performance implications—an in-depth study was conducted. The study chose Runtime systems that are task-based, support distributed-memory systems, and have an active developer base. Features such as: if the Runtime is high level or low level, builds static or dynamic task graphs, has support for load balancing, has support for fault-tolerance, etc. were identified to be potentially important to the users. The results of the classification is described elaborately and summarized in Table [4.1](#).
7. **Design and Implementation of Micro-Benchmarks:** Micro-benchmarks were designed to assess the efficiency of 5 Runtimes. The first benchmark was to check the overhead of task management in each Runtime and the second was to test the communication overhead. For example, it was found that among the 5 Runtimes ParSEC performed the best on the average followed closely by StarPU, as determined

by the first benchmark. This is shown in Chapter 4, Figure 4.2. This work should aid future application developers to get a better understanding of features to look for and find the `Runtime` that is most suitable for their needs.

5.3 Future Work

This study of different task-based `Runtime` systems and the efforts to unify concepts should kick-start an effort from groups developing task-based `Runtimes` to push for more standardization in the terminology. A convergence of the various efforts in the `Runtime` community in a standard like MPI would be highly impactful, benefiting the scientific application developers tremendously while providing a clearly defined path to the framework developers. As part of the micro-benchmark effort, multiple benchmarks were designed, but only some of their implementation were complete. One of the future works is to complete the remaining micro-benchmarks to further assess the quality of each `Runtime` on the overhead task-based `Runtimes` share.

Application developers can now build applications using multiple programming APIs (currently DTD and PTG) over the same `Runtime`, `ParSEC`, merging in the same application using multiple programming models with complementary capabilities. This also highlights the opportunity to develop specialized DSLs over the `ParSEC Runtime`, without making compromises regarding the performance of the resulting applications. This opens up the opportunity to compose different programming models (DTD and PTG) in the future at varying levels.

The simple sequential task insertion model has a theoretical limitation that will hinder scalability. Given this constraint, we have to carefully think about the next interface to program task-based `Runtimes`. The future interface has to allow users to express their applications more easily than the current interfaces, like PTG, which requires learning a completely new programming language, and at the same time not having theoretical limitation like the simpler interfaces, like DTD.

Bibliography

- OCR The Open Community Runtime Interface. 2016. URL <https://xstack.exascale-tech.com/git/public?p=ocr.git;a=blob;f=ocr/spec/ocr-1.1.0.pdf>. 21, 66
- B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Toton, L. Wesolowski, and L. Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2014. ISBN 9781479955008. doi: 10.1109/SC.2014.58. 21, 66
- E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. A hybridization methodology for high-performance linear algebra software for gpus. *GPU Computing Gems Jade Edition*, page 473484, 2012. doi: 10.1016/b978-0-12-385963-1.00034-4. 39, 57, 86
- E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault. Harnessing Supercomputers with a Sequential Task-based Runtime System. 13(9):1–14, 2014. 39
- K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes. Tile low rank cholesky factorization for climate/weather modeling applications on manycore architectures. In *International Supercomputing Conference*, pages 22–40. Springer, 2017. 86
- C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 2011. ISSN 15320626. doi: 10.1002/cpe.1631. 21, 65
- M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012. ISBN 9781467308069. doi: 10.1109/SC.2012.71. 20, 65, 71
- M. E. Bauer. Legion: Programming distributed heterogeneous architectures with logical regions, 2014. 71

- J. Bennett, R. Clay, G. Baker, M. Gamell, D. Hollman, S. Knight, H. Kolla, G. Sjaardema, N. Slattengren, K. Teranishi, J. Wilke, M. Bettencourt, S. Bova, K. Franko, P. Lin, R. Grant, S. Hammond, S. Olivier, L. Kale, N. Jain, E. Mikida Charm, A. Aiken, M. Bauer, W. Lee, E. Slaughter, S. Treichler, M. Berzins, T. Harman, A. Humphrey, J. Schmidt, D. Sunderland, P. McCormick, S. Gutierrez, M. Schulz, A. Bhatele, D. Boehme, P.-T. Bremer, and T. Gamblin. ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms. 2015. [22](#), [66](#), [72](#)
- A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, Oct 1966. ISSN 0367-7508. doi: 10.1109/PGEC.1966.264565. [4](#)
- C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011. [9](#)
- 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. *ScaLAPACK Users’ Guide*. Society for Industrial and Applied Mathematics, 1997. doi: 10.1137/1.9780898719642. URL <http://epubs.siam.org/doi/abs/10.1137/1.9780898719642>. [16](#), [42](#)
- R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995. ISSN 0362-1340. doi: 10.1145/209937.209958. URL <http://doi.acm.org/10.1145/209937.209958>. [19](#)
- G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemariner, H. Ltaeif, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. pages 1432–1441, Anchorage, Alaska, USA, 00-2011 2011. IEEE. [42](#), [61](#), [70](#)
- G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, and J. Dongarra. *From Serial Loops to Parallel Execution on Distributed Systems*, pages 246–257. Springer Berlin Heidelberg,

- Berlin, Heidelberg, 2012. ISBN 978-3-642-32820-6. doi: 10.1007/978-3-642-32820-6_25. URL https://doi.org/10.1007/978-3-642-32820-6_25. 16
- G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. Dongarra. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering*, 99:1, 2013. doi: 10.1109/MCSE.2013.98. 11, 14, 65
- J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. Productive programming of GPU clusters with OmpSs. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012*, pages 557–568, 2012. ISSN 1530-2075. doi: 10.1109/IPDPS.2012.58. 19
- M. G. Burke. The Concurrent Collections Programming Model. 2010. 66
- D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-63392-2. 7
- 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. ISSN 0167-8191. doi: <https://doi.org/10.1016/j.parco.2008.10.002>. URL <http://www.sciencedirect.com/science/article/pii/S0167819108001117>. 16, 19, 39, 57
- C. Cao, T. Hérault, G. Bosilca, and J. Dongarra. Design for a soft error resilient dynamic task-based runtime. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 765–774, May 2015. doi: 10.1109/IPDPS.2015.81. 71
- J. M. C. Carpaye, J. Roman, and P. Brenner. Design and analysis of a task-based parallelization over a runtime system of an explicit finite-volume {CFD} code with adaptive time stepping. *Journal of Computational Science*, pages –, 2017. ISSN 1877-7503. 73
- J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, 2(1):015001, Jan. 2009. doi: 10.1088/1749-4699/2/1/015001. 71

- M. Cosnard, E. Jeannot, and T. Yang. Slc: Symbolic scheduling for executing parameterized task graphs on multiprocessors. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 413–421, 1999. doi: 10.1109/ICPP.1999.797429. 70
- L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. 7, 18
- A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra. 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*, pages 21–30, 2014. doi: 10.1109/WOLFHPC.2014.8. 15, 16, 39, 70, 71
- A. Danalis, H. Jagode, G. Bosilca, and J. Dongarra. Parsec in practice: Optimizing a legacy chemistry application through distributed task-based execution. In *2015 IEEE International Conference on Cluster Computing*, pages 304–313, Sept 2015. doi: 10.1109/CLUSTER.2015.50. 16
- DARMA. Darma. 2019. URL <https://share-ng.sandia.gov/darma/>. 85
- J. D. de St. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson. Uintah: a massively parallel problem solving environment. pages 33–41, 2000. ISSN 1082-8907. doi: 10.1109/HPDC.2000.868632. 22, 66
- H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. ISSN 0743-7315. doi: <https://doi.org/10.1016/j.jpdc.2014.07.003>. URL <http://www.sciencedirect.com/science/article/pii/S0743731514001257>. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. 19
- T. M. Forum. Mpi: A message passing interface, 1993. 7, 20, 65
- GASPI. Gaspi. 2017. URL <http://www.gaspi.de/gaspi>. 63

- T. Heller, H. Kaiser, and K. Iglberger. Application of the ParalleX execution model to stencil-based problems. *Computer Science - Research and Development*, 28(2-3):253–261, 2013. ISSN 18652034. doi: 10.1007/s00450-012-0217-1. 21, 65
- R. Hoque and P. Shamis. Distributed task-based runtime systems - current state and micro-benchmark performance. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 934–941, June 2018. doi: 10.1109/HPCC/SmartCity/DSS.2018.00155. 9
- R. Hoque, T. Herault, G. Bosilca, and J. Dongarra. Dynamic task discovery in parsec: A data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA '17*, pages 6:1–6:8. ACM, 2017. ISBN 978-1-4503-5125-6. 70
- HPCWIRE. Hpc progress: No more free lunch. 2013. URL <https://www.hpcwire.com/2013/12/11/hpc-progress-free-lunch/>. 2
- H. Jagode, A. Danalis, G. Bosilca, and J. Dongarra. *Accelerating NWChem Coupled Cluster Through Dataflow-Based Execution*, pages 366–376. Springer International Publishing, 2016. ISBN 978-3-319-32149-3. doi: 10.1007/978-3-319-32149-3_35. 16, 70
- H. Jagode, A. Danalis, R. Hoque, M. Faverge, and J. Dongarra. Evaluation of dataflow programming models for electronic structure theory. *Concurrency and Computation: Practice and Experience*, 30(17):e4490, 2018. 62, 89
- N. Jain, A. Bhatele, J. S. Yeom, M. F. Adams, F. Miniati, C. Mei, and L. V. Kale. Charm++ and MPI: Combining the Best of Both Worlds. In *Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015*, 2015. ISBN 9781479986484. doi: 10.1109/IPDPS.2015.102. 75
- H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International*

- Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11. ACM, 2014. ISBN 978-1-4503-3247-7. [74](#)
- L. V. Kale, A. Bhatele, E. J. Bohm, and J. C. Phillips. NANOScale Molecular Dynamics (NAMD). In D. Padua, editor, *Encyclopedia of Parallel Computing*. Springer Verlag, 2011. [75](#)
- R. M. Karp and R. E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2): 147–195, May 1969. ISSN 0022-0000. doi: 10.1016/S0022-0000(69)80011-5. URL [http://dx.doi.org/10.1016/S0022-0000\(69\)80011-5](http://dx.doi.org/10.1016/S0022-0000(69)80011-5). [13](#)
- Z. Khatami, H. Kaiser, P. Grubel, A. Serio, and J. Ramanujam. A massively parallel distributed N-body application implemented with HPX. In *Proceedings of Scala 2016: 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems - Held in conjunction with SC16: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017. ISBN 9781509052226. doi: 10.1109/Scala.2016.012. [74](#)
- V. Kindratenko and P. Trancoso. Trends in high-performance computing. *Computing in Science & Engineering*, 13(3):92, 2011. [2](#)
- J. Luitjens and M. Berzins. Improving the performance of Uintah: A large-scale adaptive meshing computational framework. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, 2010. ISBN 9781424464432. doi: 10.1109/IPDPS.2010.5470437. [76](#)
- V. Martínez, D. Michéa, F. Dupros, O. Aumage, S. Thibault, H. Aochi, and P. O. A. Navaux. Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In *27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Florianopolis, Brazil, Oct. 2015. IEEE. URL <https://hal.inria.fr/hal-01182746>. [73](#)

- P. E. McKenney, M. M. Michael, M. Gupta, P. W. Howard, J. Triplett, and J. Walpole. Is parallel programming hard, and if so, why? 2009. 5
- P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999. 71
- J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, Mar. 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL <http://doi.acm.org/10.1145/1365490.1365500>. 45
- OpenMP. Openmp 4.0 complete specifications. 2013. URL <http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>. 64
- I. Papadopoulos, N. Thomas, A. Fidel, N. M. Amato, and L. Rauchwerger. STAPL-RTS: An Application Driven Runtime System. doi: 10.1145/2751205.2751233. 22, 66
- S. G. Parker, J. Guilkey, and T. Harman. A component-based parallel infrastructure for the simulation of fluid–structure interaction. *Engineering with Computers*, 22(3):277–292, 2006. ISSN 1435-5663. doi: 10.1007/s00366-006-0047-5. 76
- RAJA. Raja performance portability layer. URL <https://github.com/LLNL/RAJA>. 20
- Rodriguez. A graph model for parallel computations. Technical report, Cambridge, MA, USA, 1969. 13
- SLATE. Slate. 2019. URL <http://icl.utk.edu/slate/>. 58
- E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken. Regent: A high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15*, pages 81:1–81:12. ACM, 2015. ISBN 978-1-4503-3723-6. doi: 10.1145/2807591.2807629. 72
- TACC. Texas advanced computing center. URL <https://www.tacc.utexas.edu/>. 42

- M. Tillenius. SuperGlue: A Shared Memory Framework Using Data Versioning for Dependency-Aware Task-Based Parallelization. *SIAM Journal on Scientific Computing*, 37(6):C617–C642, 2015. ISSN 1064-8275. doi: 10.1137/140989716. URL <http://epubs.siam.org/doi/10.1137/140989716>. 19
- Top-500. Top 500. 2018. URL <http://www.top500.org>. 4
- H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002. ISSN 1045-9219. doi: 10.1109/71.993206. 21
- S. Treichler, M. Bauer, and A. Aiken. Realm: An Event-based Low-level Runtime for Distributed Memory Architectures. *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 263–276, 2014a. ISSN 1089795X. doi: 10.1145/2628071.2628084. 72
- S. Treichler, M. Bauer, and A. Aiken. 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, pages 263–276, New York, NY, USA, 2014b. ACM. ISBN 978-1-4503-2809-8. doi: 10.1145/2628071.2628084. URL <http://doi.acm.org/10.1145/2628071.2628084>. 20
- A. H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, Dec. 1986. ISSN 0360-0300. doi: 10.1145/27633.28055. URL <http://doi.acm.org/10.1145/27633.28055>. 13
- J. J. Wilke, J. C. Bennett, and R. L. Clay. Enabling Runtime / Application Co-Design through Common Concurrency Concepts. 23, 66
- A. Yarkhan. Dynamic Task Execution on Shared and Distributed Memory Architectures. (December), 2012. URL <http://trace.tennessee.edu/utk-graddiss/1575>. 19, 39
- K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su,

M. Welcome, and T. Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07. ACM, 2007. ISBN 978-1-59593-741-4. [20](#), [65](#)

Appendices

Appendix A

PaRSEC DTD API and Users Guide

The goal of DTD in PaRSEC is to provide a simple and easy interface to application developers that allows them to program PaRSEC easily and to have an efficient implementation that ensures the application will get good performance.

DTD API through Simple Examples

This section will present and describe the API developers can use to program PaRSEC. With this interface PaRSEC intends to allow developers of all domains to build their parallel applications for massive parallel machines. We will list the basic APIs to write a complete PaRSEC DTD program and then we will demonstrate the usage of these APIs using simple examples. Listing [A.1](#) shows the complete list of APIs required for a complete DTD application.

Every PaRSEC application will have to create a PaRSEC context that abstracts the underlying computing resource. DTD is an interface to program PaRSEC by inserting tasks in it. A complete DTD program will allow users to insert task in PaRSEC while expressing dependencies between tasks for correct order of execution. To insert tasks in PaRSEC using DTD, we will need a PaRSEC *Taskpool* that can handle DTD tasks, a PaRSEC context can have multiple *Taskpool* at the same time. A *Taskpool* is a collection of tasks that performs the internal bookkeeping for those tasks. To get a *Taskpool* of DTD type users will need to use the function shown line 1-2 in Listing [A.1](#). After a DTD *Taskpool* is created it can be

```

1 parsec_taskpool_t *
2 parsec_dtd_taskpool_new();
3
4 int
5 parsec_dtd_taskpool_wait(parsec_context_t *parsec,
6                          parsec_taskpool_t *tp);
7
8 void
9 parsec_dtd_data_collection_init(parsec_data_collection_t *dc);
10
11 void
12 parsec_dtd_data_collection_fini(parsec_data_collection_t *dc);
13
14 parsec_dtd_tile_t *
15 parsec_dtd_tile_of(parsec_data_collection_t *dc, parsec_data_key_t key);
16
17 void
18 parsec_dtd_data_flush(parsec_taskpool_t *tp,
19                      parsec_dtd_tile_t *tile);
20
21 void
22 parsec_dtd_data_flush_all(parsec_taskpool_t *tp,
23                          parsec_data_collection_t *dc);
24
25 typedef int (parsec_dtd_funcptr_t)(parsec_execution_stream_t *,
26                                   parsec_task_t *);
27
28 void
29 parsec_dtd_taskpool_insert_task(parsec_taskpool_t *tp,
30                                parsec_dtd_funcptr_t *fpointer, int priority,
31                                const char *name_of_kernel, ...);
32
33 parsec_task_t *
34 parsec_dtd_taskpool_create_task(parsec_taskpool_t *tp,
35                                parsec_dtd_funcptr_t *fpointer, int priority,
36                                const char *name_of_kernel, ...);
37
38 void
39 parsec_insert_dtd_task(parsec_task_t *this_task);
40
41 void
42 parsec_dtd_unpack_args(parsec_task_t *this_task, ...);

```

Listing A.1: API provided by DTD

waited on multiple times. A wait on a *Taskpool* will ensure all the tasks (local) that have been inserted so far with this *Taskpool* have completed. Line 4-6, shows the API to wait on a DTD *Taskpool*.

Users can insert tasks in two ways using DTD. First is to use the API shown in Line 27-30, where the task is directly inserted or they can create a DTD task first using function in Line 32-35 and then insert the already created DTD task at a later time, using API in Line 37-38. Both the function, to insert task directly and to create a task has the same signature, where the arguments after the first 4 are variable. The first argument is the *Taskpool* the task will be associated with, the second argument is the function pointer to the actual task which has to have the function signature as shown in Line 25-26, the third argument defines the priority of the task and the fourth argument is the name of the task. We use flag "PARSEC_DTD_ARG_END" to indicate to the function that list of variable argument is over. Each variable argument has three components. The first component indicates the size of the argument the second component provides the reference of the argument and the third component consists of flags that indicate multiple attributes of an argument. Valid flags are:

- **VALUE:** This indicates users are passing an argument by value and we copy the size passed as the first component and save it as a parameter of the respective task.
- **SCRATCH:** This indicates users wants the **Runtime** to allocate a piece of memory for the task, the size of the memory is the first component. If the second component of the argument is not NULL the reference of the second argument is copied instead of allocating a new piece of memory.
- **INPUT:** This indicates the argument is being passed by reference and is a data that will be used by the task. This also tells DTD that the task will use the data (second component) in read only mode.
- **INOUT:** This indicates the argument is being passed by reference and is a data that will be used by the task. This tells DTD that the task will use the data (second component) in read and write mode.

- **OUTPUT:** This indicates the argument is being passed by reference and is a data that will be used by the task. This tells DTD that the task will use the data (second component) in write mode.
- **DONT_TRACK:** This indicates the argument is being passed by reference and instructs DTD to not track any dependency for this data.
- **AFFINITY:** This flag places a task in a process. This flag can be used with an argument of type value or reference (data). When used with an argument passed by value the value of the argument indicates the rank or process where the task will be executed. When used with an argument being passed by reference the task is placed in the process the data actually resides in. This flag is useful in distributed memory machines.

To access the arguments users passed to a task inserted in **PaRSEC**, developers will need to use the unpack function provided by DTD. Line 40-41 in Listing [A.1](#) shows the function to unpack the arguments inside the task during execution. This function takes variable number of arguments. Users will have to pass the correct type of variables that matches the type of arguments passed while inserting the task. The order of the arguments passed to this function has to be the exact same order in which this arguments were passed in the task insertion API.

Dependencies between DTD tasks can be created using the data the tasks will use. **PaRSEC** has its own data interface that allows users to describe the data to it. A global description of data is necessary for correct execution in distributed environment. In this section, we will only describe the part of the data interface that is required to program DTD and not the whole **PaRSEC** data interface. The structure describing data to **PaRSEC** is known as *Data Collection* or *dc* in short. For every **PaRSEC** *dc*, we have to initialize and finalize necessary DTD structures using functions shown in Line 8-9 and 11-12. A *dc* can describe any kind of data - regular, irregular etc., where a *tile* represents data at a finer granularity. Figure [A.1](#) shows 1-dimensional and 2-dimensional data, where the whole data is represented by a *dc* and smaller chunks of the data is represented by *tile*. DTD has the capability to track dependency at *tile* level. To get a *tile* of certain index/key users can use function *parsec_dtd_tile_of*, shown in Line 14-15 of Listing [A.1](#), here each index/key has to be unique.

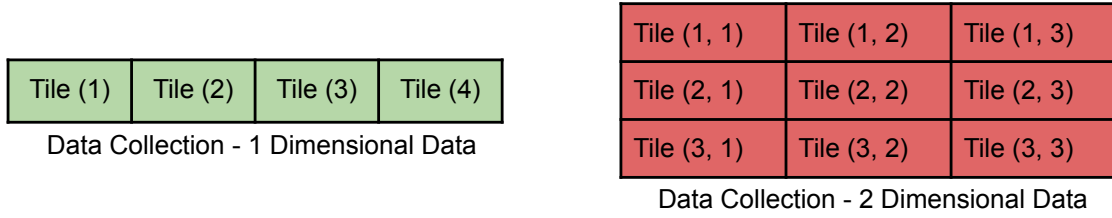


Figure A.1: ParSEC Data Collection at different granularity

Developers will use a flag described above combined with a piece of data to describe how a task will use that data/*tile* and depending on the usage DTD will automatically create dependencies between tasks. In distributed system, the application will start with an initial data distribution and when the application has ended the distribution should remain the same. This does not imply that data can not move during the execution. There might be cases when a data can be moved to a remote location and modified, ParSEC DTD provides automatic facility to bring the data back to the original owner Rank using the function *parsec_dtd_data_flush* (Line 17-19, Listing A.1). This function brings a piece of data - a *tile*, to its original location. Function in Line 21-23, Listing A.1, performs the same functionality but for all *tiles* belonging to a *dc*.

”Hello World!” Example

The first example we will use is a program that prints ”Hello World”. Listing A.2 shows a complete example of that application using the DTD interface of ParSEC. In this example we show all the function calls including the initialization and finalization of ParSEC and DTD. In the latter example we will omit the initialization and finalization parts. Function ”task_hello_world” that starts at line 4 of Listing A.2 is the actual ”task” or ”kernel” that ParSEC will execute. All DTD task functions are required to have the same signature as this function. What we actually do inside the function is very trivial. We print ”Hello World!” and return flag ”PARSEC_HOOK_RETURN_DONE” to ParSEC, this flag indicates to the ParSEC scheduler that the task has successfully executed.

For a successful ParSEC DTD program we need a ParSEC context and a *Taskpool* with which we will associate a ParSEC task. In the case of DTD we will require a *Taskpool* that

```

1 #include "parsec/interfaces/superscalar/insert_function.h"
2
3 /* Task that prints "Hello World" */
4 int task_hello_world(parsec_execution_stream_t *es,
5                      parsec_task_t *this_task) {
6     printf("Hello_World!\n");
7     return PARSECHOOK_RETURN_DONE;
8 }
9
10 int main(int argc, char **argv) {
11     parsec_context_t *parsec;
12     int cores = 1;
13     /* Initializing parsec context */
14     parsec = parsec_init(cores, &argc, &argv);
15     /* Initializing parsec handle(collection of tasks) */
16     parsec_taskpool_t *dtd_tp = parsec_dtd_taskpool_new();
17     /* Registering the dtd_handle with PARSEC context */
18     parsec_context_add_taskpool(parsec, dtd_tp);
19     /* Starting the parsec_context */
20     parsec_context_start(parsec);
21     /* Inserting task to print Hello World */
22     parsec_dtd_taskpool_insert_task(dtd_tp, task_hello_world,
23                                    0, "Hello_World_task",
24                                    PARSEC_DTD_ARG_END);
25     /* finishing all the tasks inserted */
26     parsec_dtd_taskpool_wait(parsec, dtd_tp);
27     /* Waiting on the parsec context */
28     parsec_context_wait(parsec);
29     /* Cleaning the parsec handle */
30     parsec_taskpool_free(dtd_tp);
31     /* Cleaning up parsec context */
32     parsec_fini(&parsec);
33     return 0;
34 }

```

Listing A.2: Hello World Example

supports DTD tasks aka. DTD *Taskpool*. Line 14 shows how we initialize a **PaRSEC** context, we get a new DTD type *Taskpool* in line 16. We add the DTD *Taskpool* to the **PaRSEC** context and start the **PaRSEC** context in line 18-20. Line 22 shows the API we use to insert task in **PaRSEC**. Line 26 and 28 makes sure we are waiting on all the tasks inserted so far to be completed. Line 30 and 32 finalizes and cleans up the **PaRSEC** context and the DTD *Taskpool*. Since, the task we insert does not have any prior dependency, it will be ready to be executed as soon as it is inserted and hence will be picked up by **PaRSEC** scheduler as soon as possible. This is a very simple example showing how to insert a task in **PaRSEC**. In the next example we will show a "Chain" program, where we will show how to create dependencies between tasks.

Creating Dependencies between Tasks

In this example we will demonstrate how to create dependencies between tasks using data in DTD. We will create a very simple chain of three tasks, where the current task will depend on the immediate previous task. All the tasks belong to the same task class and have the same behavior. The data used in this example is a single integer with initial value of 1. Each task will multiply the current value of the data by 2. Figure A.2 shows the relationship between the three tasks. The value of the data before the task modifies it given at the top of each task and the value after the modification is given below.

Listing A.3 shows the real code to achieve the chain of tasks. Line 1-7 of Listing A.3 shows the actual task, where unpack the data passed as argument of the task, print the

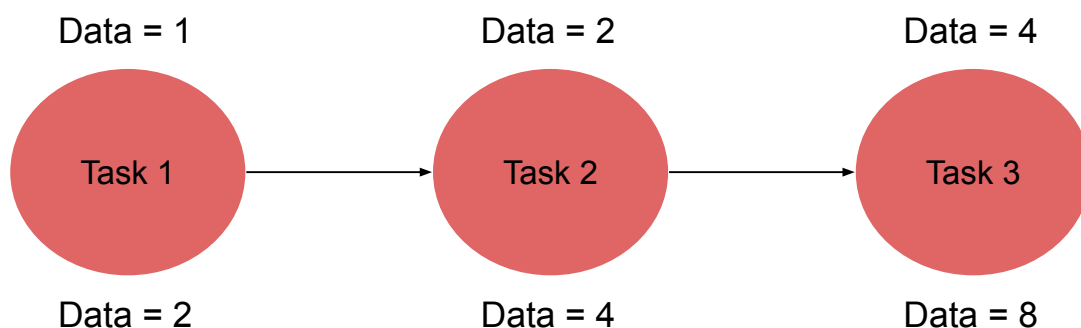


Figure A.2: Example of a chain of tasks

current value of the data and then multiply the current value of the data with 2. Line 2 of Listing A.3 shows the initialization of the data collection that has only one integer, so in this case only index/key 0 is valid. We insert 3 tasks, one after the other, using the direct task insertion API. We specify in all cases that the task will use data at index/key 0 of *dc* "data". We also specify the mode of the usage, "INOUT", signalling that the task will both Read and Write the data. As we insert the second task, DTD figures out that the same data will be used by the first task in "Read_Write" mode and creates a dependency between the two, making the second task not ready for execution until the first task completes. The same process takes place when we insert the third task and we end up getting a chain of three tasks. The output of the program is given in Listing A.4. In the next example, we will show a "Ping-Pong" example to show a program in distributed environment. The next example is an extension of this Chain program in two processes.

Example to show usage of DTD in Distributed Environment

With the help of a simple "Ping-Pong" program we will demonstrate the usage of ParSEC DTD API in distributed environment. The intended program will involve two processes, Process 1 as Rank 0 and Process 2 as Rank 1. Figure A.3 shows the sequence of events that will take place. We insert three tasks in both Ranks. The first and third task will execute in Rank 0 and the middle task will execute in Rank 1. First task (Rank 0) will send data to second task (Rank 1) and consequently second task will send data to the third task (Rank 0). Data in this case is a single integer with an initial value of 1. All tasks in Rank 0 will multiply the current value of data with 2 and tasks in Rank 1 will add 1 to the current value. The value of data before a task executes is given at the top of the task, the value of data after that task has completed execution is given below the task in Figure A.3.

Listing A.5 shows the sample code to program ParSEC to execute the above mentioned sequence. Both Ranks, in this case, will insert all the tasks. In line 20, we insert the first task. We pass one integer as the data and mark that the task will use this data in *Read_Write* mode. We pass variable *rank_0* as "VALUE" and append "AFFINITY" flag to it. This makes sure that we are placing the first task in Rank 0. Next, we insert the second task in line 25. We use variable *rank_1* to place this task in Rank 1. We pass the same data

```

1 int chain_task(..) {
2     int *data;
3     parsec_dtd_unpack_args(this_task, &data);
4     printf("Value_of_Data: %d\n", *data);
5     *data *= 2;
6     return PARSEC_HOOK_RETURN_DONE;
7 }
8
9 int main(int argc, char **argv) {
10     /* Boilerplate code */
11     /* data = 1 */
12     parsec_dtd_data_collection_init(data);
13     parsec_dtd_taskpool_insert_task(dtd_taskpool, chain_task,
14                                     priority, name_of_task,
15                                     PASSED_BY_REF, TILE_OF(data, 0), INOUT,
16                                     PARSEC_DTD_ARG_END);
17     parsec_dtd_taskpool_insert_task(dtd_taskpool, chain_task,
18                                     priority, name_of_task,
19                                     PASSED_BY_REF, TILE_OF(data, 0), INOUT,
20                                     PARSEC_DTD_ARG_END);
21     parsec_dtd_taskpool_insert_task(dtd_taskpool, chain_task,
22                                     priority, name_of_task,
23                                     PASSED_BY_REF, TILE_OF(data, 0), INOUT,
24                                     PARSEC_DTD_ARG_END);
25     parsec_dtd_data_flush_all(dtd_taskpool, data);
26     parsec_dtd_data_collection_fini(data);
27     /* Boilerplate code */
28     return 0;
29 }

```

Listing A.3: Example showing Chain of Tasks

```

1 Value of Data: 1
2 Value of Data: 2
3 Value of Data: 4

```

Listing A.4: Output of the Chain program

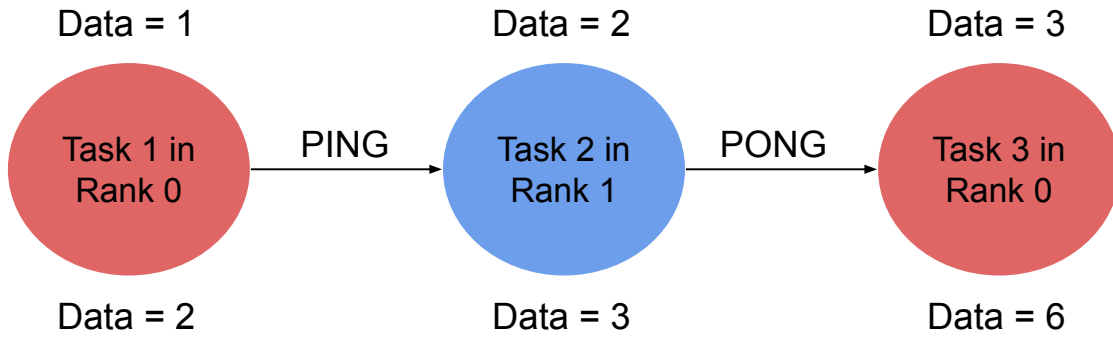


Figure A.3: Ping-Pong Example

being used in the same mode as the previous case. DTD will infer a dependency between these two tasks and a communication will be initiated by the `Runtime` implicitly. This first communication is the "Ping" from Rank 0 to Rank 1. Then, we insert the third and the final task, which will execute in Rank 0. This will again another communication between Rank 1 to Rank 0 as the third task (Rank 0) we inserted uses the same data being modified by the second task (Rank 1). This communication is the "PONG". Here, "AFFINITY" flag is being used to place tasks in a specific process. The output of the program is shown in Listing A.6.

PaRSEC DTD Variables

DTD allows users to tune multiple features according to the need of the application. The list of all the DTD variables is given below with description of what each does. All these variables can be passed to the PaRSEC application as command-line arguments or can be put in in a file called "mca-params.conf" inside a directory ".parsec" in the home of the user.

- For limiting the memory consumption DTD does optimized DAG unrolling as described in section 3.3.1. There are two variables that control the size of the sliding window of DAG. They are:
 - `dtd_window_size`: This determines the number of local tasks after which call to insert new tasks in PaRSEC will block. The default value is set to 8000 tasks.

```

1 int task_in_rank_0(..) {
2     int *data, *rank;
3     parsec_dtd_unpack_args(this_task, &data, &rank);
4     printf("Value_of_Data: %d_in_rank %d\n", *data, *rank);
5     *data *= 2;
6     return PARSEC_HOOK_RETURN_DONE;
7 }
8
9 int task_in_rank_1(..) {
10     int *data, *rank;
11     parsec_dtd_unpack_args(this_task, &data, &rank);
12     printf("Value_of_Data: %d_in_rank %d\n", *data, *rank);
13     *data += 1;
14     return PARSEC_HOOK_RETURN_DONE;
15 }
16
17 int main(int argc, char **argv) {
18     /* Boilerplate code */
19     /* data = 1; rank_0 = 0, rank_1 = 1 */
20     parsec_dtd_taskpool_insert_task(dtd_taskpool, task_in_rank_0,
21                                     priority, name_of_task,
22                                     PASSED_BY_REF, TILE_OF(data, 0), INOUT,
23                                     sizeof(int), &rank_0, VALUE | AFFINITY,
24                                     PARSEC_DTD_ARG_END);
25     parsec_dtd_taskpool_insert_task(dtd_taskpool, task_in_rank_1,
26                                     priority, name_of_task,
27                                     PASSED_BY_REF, TILE_OF(data, 0), INOUT,
28                                     sizeof(int), &rank_1, VALUE | AFFINITY,
29                                     PARSEC_DTD_ARG_END);
30     parsec_dtd_taskpool_insert_task(dtd_taskpool, task_in_rank_0,
31                                     priority, name_of_task,
32                                     PASSED_BY_REF, TILE_OF(data, 0), INOUT,
33                                     sizeof(int), &rank_0, VALUE | AFFINITY,
34                                     PARSEC_DTD_ARG_END);
35     /* Boilerplate code */
36     return 0;
37 }

```

Listing A.5: Example of Ping-Pong program using DTD

```
1 Value of Data: 1 in rank 0
2 Value of Data: 2 in rank 1
3 Value of Data: 3 in rank 0
```

Listing A.6: Output of the Ping-Pong program

- `dtd_threshold_size`: This determines the number of local tasks reaching which PaRSEC will allow users to insert new tasks. This number can be thought of as the lower water mark after which we resume inserting tasks again. The default value is 4000 tasks.
- PaRSEC DTD uses hash tables for managing meta-data about data and storing information about remote tasks. Users have the control to determine the size of these hash tables
 - `dtd_tile_hash_size`: This variable sets the size of the hash table used for managing meta-data about data the tasks will use. The default value is 1024.
 - `dtd_task_hash_size`: This variable sets the size of the hash table used for managing information about remote tasks. The default value is 1024.
- For debugging an application DTD provides useful information at runtime that might help figure out a unwanted behavior.
 - `dtd_debug_verbose`: This variable indicates the verbosity level of separate DTD output stream and also determines if we will be using a separate output stream for DTD or not. Valid values are "50" and "60". 50 will print relationship between different task classes and 60 will print the information we print with 50 plus additional traversal of the DAG at runtime.

Appendix B

Publications

1. R. Hoque, T. Herault, G. Bosilca, J. Dongarra (2017). Dynamic Task Discovery in PaRSEC- A data-flow task-based Runtime. In ScalA17: 8th Workshop on Latest Advances in Scalable Algorithms for LargeScale Systems, ACM, Denver, CO
2. R. Hoque, P. Shamis (2018). Distributed Task-based Runtime Systems - Current State and Micro-Benchmark Performance. In 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)
3. H. Jagode, A. Danalis, R. Hoque, M. Faverge, J. Dongarra (2018). Evaluation of dataflow programming models for electronic structure theory. In Concurrency and Computation: Practice and Experience, Volume 30, Issue 17, Page e4490

Vita

Reazul Hoque was born in Chittagong, Bangladesh on October 1989. He completed his Bachelor of Science degree in Computer Science and Information Technology from Islamic University of Technology, Bangladesh in 2011. After completing his Bachelor, he worked as a lecturer in Stamford University Bangladesh for 2 years before joining University of Tennessee, Knoxville in 2013 as a graduate student. During his graduate studies, he worked as a Graduate Research Assistant at Innovative Computing Lab (ICL) under the supervision of Dr. Jack Dongarra and Dr. George Bosilca. His major was Computer Science and his focus was in Distributed Computing. He mainly worked on a federally funded project called PaRSEC. His current research interest are High Performance Computing, Distributed Runtime Systems and Heterogeneous Computing. Reazul Hoque is expected to receive his Doctor of Philosophy in Computer Science in August 2019.