TOWARDS A NEW DIRECTIVE-BASED TASKING API FOR DISTRIBUTED SYSTEMS

A Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Abdulelah AlTuaimi

December 2016

TOWARDS A NEW DIRECTIVE-BASED TASKING API FOR DISTRIBUTED SYSTEMS

Abdulelah AlTuaimi

APPROVED:

Dr. Jaspal Subhlok, Chairperson
Department of Computer Science

Dr. Barbara Chapman
Stony Brook University

Dr. Edgar Gabriel
Department of Computer Science

Dr. Henryk Marcinkiewicz
Aramco Services Company

Dean, College of Natural Sciences and Mathematics

Acknowledgements

I would first like to express my gratitude to my advisor, Dr. Barbara Chapman, for her guidance and for creating an excellent environment for research. I would like to thank Dr. Jaspal Subhlok, Dr. Edgar Gabriel, and Dr. Henryk Marcinkiewicz for their support and for serving as members of my committee. I would also like to thank my mentor, Dr. Dounia Khaldi, for providing considerable input and direction in the work discussed in this thesis. It has been an exceptional learning experience working with them.

I sincerely thank my employer, Saudi Aramco, and Aramco Services Company for sponsoring my graduate study and for providing the support and guidance I needed during this assignment.

Lastly, and most importantly, I am forever so grateful to my parents, Badryiah and Abdullah. I would not be where I am today without their love and support.

TOWARDS A NEW DIRECTIVE-BASED TASKING API FOR DISTRIBUTED SYSTEMS

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Abdulelah AlTuaimi

December 2016

Abstract

Programming for large-scale computing requires programming models carefully designed for that purpose. MPI is often the model of choice for distributed systems, but writing MPI program is time-consuming and complicated to maintain and debug as the program size gets larger. Moreover, MPI does not exploit some of the potential benefits of shared memory systems. Using a hybrid model also requires a high level of programmer expertise. Designing algorithms in terms of tasks potentially reduces the development effort and has many performance-related advantages. In addition, directive-based programming styles have made parallel programming and migration of serial code to multicore chips easier than ever. Although directive-based tasking models have paved the way to distributed systems, they still lack capabilities necessary for efficient large-scale computing.

TagHit is an API proposed by the HPCTools group in the Department of Computer Science at the University of Houston. Targeted for exascale computing, TagHit combines the benefits of task-based programming models with the simplicity of directive-based programming styles. This thesis tackles task creation and scheduling in TagHit. First, I present an overview of six existing task-based programming models. Next, I propose an experimental runtime design of TagHit's task creation and scheduling modules and then describe in detail a prototype implementation of the runtime. The goal of this work is to guide the definition of TagHit's concept and semantics and to assess the implementation cost and challenges of creating and scheduling tasks in TagHit. Finally, I present two TagHit benchmarks with results that show the design and implementation have supported the general concept of TagHit with good speedup and scheduling behavior.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

Exascale computing will require programming models prepared for systems with hundreds of thousand nodes and thousands of cores per node [5, 26]. MPI is the de-facto standard for parallel programming in distributed systems. However, expressing parallelism in MPI is often time-consuming and complicated to maintain and debug. Furthermore, as the number of cores per node increase, MPI alone does not sufficiently exploit the potential benefits of share-memory parallelism. Thus, finding a software solution that exploits the computation power of such a massive scale is a critical task. One current popular solution is hybrid programming, which uses a combination of inter-node and intra-node programming models, such as MPI with OpenMP. However, this makes the programming task even more complex and requires a high-level of programmer expertise to adopt two programming styles to

solve a single problem.

Task-based programming allows programmers to divide computation into independent sets of asynchronous tasks to execute in parallel. This has many advantages over the conventional thread-based approach. Task-based programming allows forward scaling, as the number of tasks can scale with the size of the problem. Tasks can be created dynamically to allow a high degree of parallelism as if there are an infinite number of processors and then to scale when more resources are available [19]. Tasking also prevents deadlocks and data races among working threads, an essential advantage for efficient scalability. Furthermore, the runtime system of a tasking model is usually responsible for minimizing the overhead and executing tasks efficiently [19]. Generally, the runtime system creates a customizable number of threads, in which tasks are scheduled, which may be used over and over again to eliminate the overhead of creating and deleting a thread for each computation task. With a smart scheduling mechanism, such as work stealing, tasks are also distributed among threads for efficient, load-balanced execution. Although directive-based tasking models have paved the way to distributed systems, they still lack capabilities necessary for efficient large-scale computing.

TagHit is a directive-based tasking API proposed by the High Performance Computing Tools group (HPCTools) in the Department of Computer Science at the University of Houston. Targeted for exascale computing, TagHit combines the benefits of task-based programming models with the simplicity of directive-based programming style as found in OpenMP. TagHit programmers can create dynamic tasks using directives, and the TagHit runtime system will take care of the rest

by scheduling and executing tasks efficiently in a distributed environment. TagHit will ultimately support other features necessary for large-scale computing such as tasks with different levels of granularity, data distribution mechanisms, data flow dependencies, and heterogeneous systems support.

1.2 Contributions

The following are the main contributions of this thesis:

- I explored six existing task-based models and described their key features.
- I proposed an experimental runtime design of TagHit's task creation and scheduling modules and described its different aspects to guide the definition of TagHit concept and semantics.
- I implemented a prototype of the runtime to create, execute, and synchronize TagHit tasks. I also contributed to TagHit's future development by describing the compiler translation of TagHit task creation directive to its runtime code.
- I further improved the prototype implementation with a work-stealing scheduling mechanism to migrate and load-balance tasks between processing resources.

1.3 Organization

The rest of this thesis report is organized as follows:

Chapter 2 presents an overview of six existing tasking models. It describes their key features and some observations about each one. Chapter 3 describes the basic TagHit API and more specifically the syntax of the directives for creating and synchronizing tasks. Chapter 4 defines the general design goals of TagHit and accordingly proposes an experiential runtime design of TagHit's task creation and scheduling modules. Chapter 5 details a prototype implementation of the runtime to create, execute, and synchronize TagHit tasks. Additionally, Chapter 5 discusses task scheduling and the adopted work-stealing mechanism to migrate and load-balance tasks between processing resources. Chapter 6 presents two TagHit benchmarks. The first section of this chapter examines the performance of a TagHit program to compute PI and investigates the scheduler's behavior. The second section demonstrates our prototype's functional behavior for handling synchronized nested tasks. Finally, Chapter 7 summarizes this work and my conclusions.

Chapter 2

Related Work

Various distributed tasking models have been developed to achieve specific tasking goals. Although they are based on the same idea, they differ in many design choices and in the features they provide to users. In fact, this makes direct comparison hard. This section, however, describes six key task-based models, which are recent and have received the most focus. Other models, however, are older but still have similar execution models. It is worth mentioning now that all models described in this section are not directive-based, except for XcalableMP, which is described in section 2.5.

2.1 Open Community Runtime (OCR)

OCR [22] is a low-level runtime for writing task-based applications. The basic idea of OCR is to write applications as a directed acyclic graph of event-driven tasks.

The OCR programming model is based on three concepts: Event Driven Tasks (EDTs), events, and Data Blocks (DBs).

The current reference implementation of OCR was developed by Rice University and Intel. Although it supports the entire OCR standard, it was developed as a proof of concept and still focuses on correctness, rather than performance. OCR-Vx [14] is another implementation of OCR. Two versions of OCR-Vx are available; one for distributed memory systems and another for shared memory. This implementation only shows good scalability in a small-scale environment. P-OCR [20], however, is a more recent implementation, which is described as high performance, optimized, and more scalable than the first two.

2.1.1 Tasks Execution and Flow Control

In OCR, EDTs are data structures representing computational units (tasks). An EDT is defined as a template that is linked with a task function and is created by passing required arguments. EDTs can be associated with events, which are OCR objects used to signal the completion of a task or the availability of data. The execution of an EDT is controlled by its events and will only execute when all of its events are completed (satisfied). Thus, EDTs and events can be used together to compose a dynamic computational graph. DBs, however, are objects that hold data. DBs are the only way to share data between tasks in OCR. They are also used to satisfy events that indicate the availability of data.

Once all events associated with a task are satisfied, the task and any resources required for its execution are submitted to a worker, which will schedule and execute

the task. When a task completes execution, it will release any associated DBs and trigger its post event, which may enable other tasks.

2.1.2 Task Scheduling

The OCR standard does not specify how tasks are exactly scheduled across processing elements. In addition, it does not specify how waiting EDTs are stored within a processing element. However, the OCR runtime is responsible for dynamically managing tasks and data. EDTs can be migrated during runtime across processing elements for better load balancing, but before they are assigned resources and scheduled for execution [20]. P-OCR, for example, performs work-stealing to load-balance tasks within a node, and users can either specify the node on which a task must run, or leave the decision to the runtime to also perform work-stealing across nodes [20].

2.1.3 Observations

DBs can only be accessed by passing them to tasks. That is, there is no API call in OCR to access a data block by knowing its reference (i.e., GUID in OCR). This restriction would introduce complexity in terms of application implementation. Additionally, events in OCR are difficult to handle when they are compared to future objects as found in HPX. An event that represents a task completion is destroyed and cannot be used once triggered (satisfied), requiring the set up of all dependent tasks before running that task to ensure correct execution behavior [22].

This, in fact, further complicates implementing OCR applications as it requires additional synchronization when an immediately ready task is created before its dependent tasks. On the other hand, a future object represents data regardless of whether or not it is ready at the creation of a dependent task, which makes coding task dependencies easy to use. Finally, although OCR is low-level and intended to support higher-level programming models such as TagHit, it is evolving and available implementations are still in an early stage of development.

2.2 ParalleX (HPX)

ParalleX is an execution model available in two implementations: HPX-3, from Louisiana State University, and HPX-5, from Indiana University [17]. The basic semantic mechanisms of ParalleX are as follows [29]:

- Global Name Space: makes actions and data globally accessible
- Threading: threads are lightweight and exist within a single locality (computing element)
- Parcel: a data structure that is sent to another locality to trigger a thread to execute
- Local Control Objects: synchronization primitives managed by the runtime
- Parallel Processes: contain many threads on multiple localities

The ParalleX goal is to address the challenges of efficiency, scalability, and

sustained performance with conventional parallel programming such as MPI [17]. ParalleX improves efficiency by basically reducing scheduling and synchronization overhead. It improves scalability by using data-directed computing and lightweight synchronizations. The performance will consequently be improved as a result of improving efficiency and scalability.

2.2.1 Task Creation and Execution

A task in HPX executes a function along with its arguments and returns a future object (LCO) that represents the data returned by the function. Generally, a future object acts as a proxy for a result that will be available at some time during the program execution but is initially unknown [17]. A parcel, which represents a task during communication, is generated and sent whenever a task must be executed in a remote locality. When it reaches its destination, the parcel creates a PX-thread to work on the associated task. PX-threads are implemented as user threads, and a thread manager schedules them to operate on top of one OS thread [17]. A running PX-thread will not be preempted until it finishes execution. When a task is finished, the result is sent to its corresponding future because the parcel that represents it carries the ID of the future as its continuation.

2.2.2 Task Scheduling

HPX has several available scheduling policies. It also has an API to write custom schedulers. By default, each OS thread operates on a separate FIFO priority queue

where the thread manager inserts PX-threads (tasks). OS threads can steal tasks from queues of other OS threads to facilitate on-node load balancing. The thread manager also gathers both the machine topology information and the number of OS threads mapped to the allocated resources. As a default behavior, the thread manager will use all available cores, assigning one OS thread per core.

All scheduling policies in HPX use two queues to manage tasks [15]. A task is first created as a PX-thread description and then placed in a staged queue. Since these threads have not been allocated yet, they can easily be moved to other localities, if necessary. The thread manager will then allocate a stack for the staged PX-thread and place it in a pending queue. The PX-thread will be active when it is running. PX-threads can suspend waiting for resources or data. When they do, they will be placed back in the pending queue, allowing other threads to run. A PX-thread will be terminated after it completes execution.

2.2.3 Task Flow Control

Tasks in HPX archive synchronization and control of parallelization via local control objects (LCOs). LCOs organize the execution flow and are designed to replace global barriers, such as barriers in MPI that might impede parallelism and, consequently, performance [18]. HPX provides traditional concurrency controls but also provides both future objects and dataflow objects. All concurrency controls in HPX are exposed as LCOs.

Futures, on which threads can wait, suspend any PX-thread (task) that must access their values when the data is not available yet. The future itself resumes all

suspended PX-threads that are waiting. Dataflow objects, on the other hand, can be used to construct and manage data dependencies. A dataflow object waits for a set of futures launches the associated task when the data of all futures becomes available.

2.2.4 Observations

With today's architectures, even though PX-threads are lightweight, allocating, context-switching, and releasing them for each task adds overhead, especially in situations where millions of tasks must be executed. For future work, however, the latency hiding and data locality control [18] in HPX are potential mechanisms for TagHit to improve application scalability.

2.3 Legion

Legion is a dynamicly distributed tasking model [2]. Legion's key feature is its logical regions. Legion programmers should define how data is grouped into regions. They can also divide regions into subregions. Parallelism in Legion is expressed by specifying if these regions are shared or private so that tasks that operate on private regions are independent tasks [28].

2.3.1 Task Creation and Execution

Tasks are created by specifying the task function and passing the required arguments. When all dependencies of a task are satisfied and the task is safe to run, it will be placed on a ready queue. To identify whether the dependencies are satisfied, a dependence analysis process must be performed. Section 2.2.3 provides a high-level description of this complex process. While tasks are in the ready queue, a processor might permit remote processors to steal them. The Legion scheduler will then map the tasks to run locally or remotely based on the location of the associated regions. Finally, tasks on a processor group (i.e., a node) are placed in a queue from which processors pull when they are ready to execute a task [3].

2.3.2 Task Scheduling

A software out-of-order processor (SOOP) is used to schedule tasks [2]. The SOOP is distributed such that each processor runs an instance of it. The SOOP performs dependence analysis to determine which task to run. This process is divided into three stages. The first stage analyzes the task dependencies, identifying which tasks must run before the current task. When dependencies are satisfied, the task is placed in a ready queue. During this time, another processor might steal the task and place it in its ready queue. The second stage distributes the task and decides whether it should run locally or remotely on another processor. This stage basically depends on the target processor for the task (i.e., CPU, GPU, etc.) and location of its data. The third stage performs more analysis of the dependent tasks, ensuring that the data the task requires is valid and up to date. When all the above

operations have finished, the task is launched. When a running task launches a child task, the SOOP instance on the process where it is launched, will be used to perform the same scheduling steps as its parent.

2.3.3 Task Flow Control

To determine dependent and independent tasks in Legion, regions that are accessed by tasks provide initial but useful locality information. Because logical regions in Legion can be partitioned into separate or overlapping subregions, the way regions are partitioned provides information to identify dependencies. Furthermore, privileges in Legion (e.g., read-only, read-write, and atomic) specify how a task accesses its region [2]. These privileges provide data-dependence information as well. For example, if two tasks access a region with read-only, the tasks are independent and can execute in parallel.

Legion also supports future objects, which represent a return value of a task that will become available at some point during the program execution. Futures can be waited on or passed as arguments to other tasks [3]. However, Legion programmers should not rely on waiting on futures as they block tasks and prevent the runtime from exposing additional work. In addition, storing futures in logical regions is not permitted

2.3.4 Observations

We noticed a steep learning curve associated with Legion. Although suitable for special (non-trivial) problems, we attribute this to the complexity of using logical regions to manage application data. Thus, using Legion tasks in TagHit would cause an overhead in creating tasks using directives, requiring the user to always use logical region. Otherwise, the compiler would be responsible to map task data to logical regions, which would consequently add a layer of complexity. Similarly, the Legion SOOP scheduler depends heavily on logical regions and is constrained by their dependencies [2].

2.4 PaRSEC (DAGuE)

PaRSEC is the more pronounceable successor of DAGuE. DAGuE is a Direct Acyclic Graph (DAG) scheduling framework, where the edges are data and the nodes are tasks [8]. PaRSEC runtime can be invoked via parameterized task graphs (PTGs), which a programmer can manually write or generate via the compiler. PaRSEC has been used to build DPLASMA [7], which is an implementation of a dense linear-algebra package for distributed systems.

2.4.1 PaRSEC Environment

In addition to a runtime environment, PaRSEC consists of tools to build, analyze, and generate a DAG task-based program. The PaRSEC library includes the runtime

environment and the program-specific generated codes. The runtime library consists of a distributed scheduler, a communication engine, and a data dependency engine. PaRSEC application is essentially an MPI application. However, it combines with PaRSEC functionalities generated using the PaRSEC tool chain. Ultimately, the end user of a PaRSEC program does not have to use PaRSEC, because all of PaRSEC's functionalities are embedded in the program [8].

2.4.2 Tasks and Data Flow

The input format of the PaRSEC framework is job data flow (JDF), a representation of task(s) and the dataflow between them [11]. The task body is written in regular C code and the dependencies are separately defined by algebraic conditions. The programmer can add a task priority function, which hints to the PaRSEC scheduling engine to set task priorities.

The JDF is then compiled to C code. The generated code is basically a C function that the programmer can call to construct and execute the tasks. The code generated is totally separate and independent of the problem size. The programmer can package the generated code to set up the data required for commutation and then call the generated function to pass data to tasks. When running a PaRSEC program, the PaRSEC engine moves data between processors, if necessary, and a task can only be executed when all of its input data are local [8].

2.4.3 Task Scheduling

The PaRSEC scheduler is distributed such that all nodes run the scheduling engine, eliminating the need for a centralized scheduler [8]. The scheduling engine also allows work stealing to load balance tasks between threads. The work stealing implemented with the PaRSEC engine focuses on improving cache locality.

PaRSEC binds threads to cores. Each thread alternates between executing tasks and executing calls to the scheduler. Tasks are stored in queues local to threads. When a thread completes a task, it invokes the scheduler to decide which task to run next.

To execute a new task by thread, the scheduler gets a task from that local queue based on the priority function the user has defined in the JDF. A thread will attempt to steal a task from another thread when its queue is empty. A thread will try to steal from a thread on a close core first to provide efficient cache and memory locality [8]. PaRSEC achieves this work-stealing behavior by utilizing the hwloc framework [9] to gather information about the NUMA architecture of the machine.

2.4.4 Observations

PaRSEC is innovative in terms of separating parallelism from control flow and data distribution. However, writing a PaRSEC program is usually a three-phase process. A small modification to the JDF file would require recompiling both JDF and the main C code. Furthermore, writing a PaRSEC application requires the programmer to adopt two programming syntaxes. Because the JDF file syntax is uncommon,

many programmers would find it hard to write and maintain. Although the PaRSEC tool chain includes a PTG code generator, it is still immature and almost always requires the programmer to be involved and modify the generated code. For these reasons, PaRSEC would have a steep learning curve for many programmers as well as being complex to write, maintain, and debug.

Finally, the task scheduling in PaRSEC is fully distributed. Because this scheduling mechanism has many performance benefits, we studied the PaRSEC scheduler and adopted some of its scheduling concepts. Our TagHit scheduler is distributed as well. And similar to PaRSEC, it also allows a processing element to steal tasks when it exhausts all tasks in its own queue. However, because our TagHit implementation is a prototype, our scheduler can be optimized, especially in terms of the victim selection strategy to extract more performance. Section 5.6 discusses our TagHit scheduler in more detail.

2.5 XcalableMP (XMP)

XcalableMP enables parallelizing sequential code on a distributed environment using directives [21]. The XcalableMP design principle is performance-awareness, in which directives take all actions. The execution model allows combining MPI and OpenMP into XcalableMP for complicated and tuned parallel codes or for shared memory-thread programming (hybrid model) [32].

Data in an XcalableMP program can be global or local. Programmers can define how to distribute global data, and data in remote memory can be accessed

by explicit communication.

2.5.1 Tasks and Loops Execution

Each node, which might have multiple cores, starts executing the same main routine until it encounters an XcalableMP construct. When a node encounters a loop, array, or task construct and is specified by the directive's "on" clause, it executes the body of the construct. An executing node set is a set of nodes that executes a loop or block [32]. The loop construct executes a loop in parallel by mapping its iterations to nodes holding specified data. The array construct, an alternative to the loop, performs a parallel execution of an array assignment in which each node (determined by the on clause) executes element assignment and operation. The task construct, on the other hand, assigns an amount of work to a specified node set.

2.5.2 Scheduling

It is the user's responsibility to specify where to execute loop iterations and tasks. The loop and task constructs specify a node set by using the on clause.

2.5.3 Observations

Although the on clause adds performance benefits on highly heterogeneous systems, it could complicate the programming effort by forcing users to direct the parallel execution on specific node sets. This approach also restricts loop iterations and task scheduling in cases where idle resources are available but not specified by

the on clause. A scheduling system that uses the underlying resources efficiently would overcome this issue. Furthermore, XcalableMP does not support dependencies, which leaves the programmer with synchronization constructs only. Task dependencies provide many performance benefits by lowering the execution overhead when compared to conventional synchronization points. Finally, a node in XcalableMP is mapped to multiple hardware resources. In TagHit, however, we look for a mechanism to dynamically map a processing-element to any hardware resources (i.e., node, CPU, or single core). This will give users more flexibility to map processing elements to the appropriate hardware resources and extract more performance with different kinds of applications.

2.6 Argobots

Argobots supports two levels of parallelism: execution streams (ESs) and work units (WUs) [27]. WUs are execution units that can be either user-level threads (ULTs) or tasklets. However, ES is a sequential execution stream containing one or more WUs and bound to hardware resources. Thus, WUs can be associated with an ES, and WUs in different ESs can be executed concurrently.

2.6.1 ULTs and Tasklets

User-level threads (ULTs) and tasklets offer an abstraction for fine-grained parallelism. ULTs are excellent for expressing parallelism whose flow of control pauses and resumes. Unlike traditional OS threads, ULTs are not intended to be preempted

but yield control to the scheduler. Tasklets in Argobots, on the other hand, are work units with dependence only on their input data. Tasklets do not yield control but run to completion before returning control to the scheduler.

2.6.2 Work Units Scheduling

Argobots has schedulers that control the execution order of scheduling units (WUs), and each scheduler can be associated with an ES. Thus, each ES can use a different scheduler [27]. Argobots provides basic schedulers, but users can also define their own schedulers. Furthermore, users can change the scheduler for the ES during runtime. WUs are scheduled by context switching. Thus, when a work unit completes its execution, it context switches to the scheduler, which then chooses another scheduling unit to execute.

2.6.3 Observations

Argobots is similar to Qthreads [31] or QUARK [33]. However, Argobots is an advanced threading infrastructure with customizable scheduling capabilities. Argobots alone does not expand for distributed-memory environments but it can be integrated with other distributed models to support this. We have chosen to present Argobots in this section to explore its applicability for building TagHit.

To support massive on-node parallelism, Argobots can be integrated with TagHit to schedule tasks on a node basis and create ES, for example, on each processing element. However, we still need a mechanism to schedule tasks across multiple nodes.

Our prototype implementation of TagHit does not use Argobots in its underlying system but, for simplicity, we used conventional POSIX Threads to express parallelism on shared memory. Also, we implemented our own scheduler to distribute tasks across nodes. For future work, Argobots, or similar frameworks [31, 33], can be considered for an improved implementation of TagHit that supports massive on-node parallelism.

Chapter 3

Basic TagHit API

TagHit is a directive-based programming interface that combines the benefits of task-based programming with the simplicity of directive-based programming style. Tasks are created dynamically using minimal modification with simple OpenMP-like directives. The underlying system is then responsible for efficiently scheduling and executing tasks. In this chapter, we discuss the TagHit API. Specifically, we describe the basic TagHit task creation construct and the task synchronization construct. Our description of these constructs is for C/C++ as pragma directives. In future, the TagHit API will possibly add other constructs as well as clauses to the task creation construct to enable necessary features for large scale computing, but they are out of the scope of this thesis work.

3.1 TagHit Terminologies

A processing element in TagHit can be a complete compute node, CPU socket, or single core. Also, as this report later describes, processing elements can run one or more threads (execution threads). We use the term ***TagHit process*** to represent this concept throughout this report.

Furthermore, ***data environment*** is basically a set of variables referenced within the structured block of a task. They are defined before the task construct and may have been initialized with values. We also use the terms ***data item*** and ***data items*** to refer to one or more variables of this set. One or more data items could be a task input or output.

3.2 Task Creation Construct

To create a TagHit task, the ***task*** construct is used. The syntax of the construct is as follows:

```
#pragma taghit task   new-line
    structured-block
```

When a TagHit process encounters a ***task*** construct, it generates a task for the associated structured block. The task's data environment depends on the variables referenced within the structured block, and initially takes the values before using the construct. The task is then scheduled for execution by the encountering TagHit

process and may be executed by the same process or sent to another for remote task execution.

Using the task-synchronization construct discussed in section 3.3 can ensure completion of the task. After using the task-synchronization construct, user can safely access the result. If a ***task*** construct is encountered during the execution of another task, a child task, independent of its parent, is generated.

3.3 Task Synchronization Construct

To wait for the completion of TagHit tasks, the ***taskwait*** construct is used. The syntax is as follows:

```
#pragma taghit taskwait new-line
```

The task-synchronization construct waits for child tasks to finish. At the ***taskwait*** construct, the current task is suspended until all previously generated child tasks complete.

3.4 Compiler Assumptions

Because TagHit is a directive-based API, the compiler will be involved in translating TagHit constructs into their corresponding TagHit runtime functions. The compiler implementation of these constructs is out of the scope of this thesis. However, to

```
1 int main(int argc, char** argv) {
2
3   int x = 1, int y = 2, int z;
4   #pragma taghit task
5   {
6     z = x + y;
7   }
8 }
```

Listing 3.1: TagHit Program

ensure that future work complies with the runtime, we provide a simple example in which a TagHit ***task*** construct, Listing 3.1, is translated into a runtime version, as seen in Listing 3.2.

As in Listing 3.2, we assumed that the compiler implementation of TagHit constructs would be able to achieve the following for each ***task*** construct appearing in the program:

- Determine the number of data items of a TagHit ***task*** construct and their data types. Similarly, in OpenMP the compiler detects the visible variables in OpenMP constructs to determine data-sharing attributes. The number and the types of the variables perform the additional compiler work, as discussed in the next points.

- Determine the MPI data types that correspond to each data item. As Chapter 5 explains, to ensure portability and heterogeneous systems support, the MPI data types are passed to the MPI routines when TagHit processes need to send or receive data.

- If task data are not primitive data types, such as C structs, the compiler

```
1  task_meta taghit_tasks_lookup_table[1] = {
2    {
3      .tag = 0,
4      .task_function = &taghit_task_01,
5      .input_count = 3,
6      .input_data_types =
7          (MPI_Datatype[]) {MPI_INT, MPI_INT, MPI_INT},
8      .input_data_lengths = (int[]) {1, 1, 1}
9    }
10 };
11
12 void taghit_task_01(void** input) {
13   int x = *((int*) input[0]);
14   int y = *((int*) input[1]);
15   int z = *((int*) input[2]);
16
17   // structured block of the taghit task
18   z = x + y;
19
20   *((int*) input[0]) = x;
21   *((int*) input[1]) = y;
22   *((int*) input[2]) = z;
23 }
24
25 int main(int argc, char** argv) {
26
27   taghit_init(&argc, &argv, taghit_tasks_lookup_table);
28
29   int x = 1;
30   int y = 2;
31   int z;
32
33   void** inputs = malloc(sizeof(void *) * 3);
34   inputs[0] = &x;
35   inputs[1] = &y;
36   inputs[2] = &z;
37
38   taghit_create_task(0, inputs);
39 }
```

Listing 3.2: TagHit Program (Runtime Version)

implements a solution to parse any complex data types and generate either their corresponding derived MPI data types or provide the MPI data type of each and every element in the data structure.

- Encapsulate TagHit task structured-blocks in special TagHit functions that return nothing (i.e., void) and take an array of pointers in which task data are referenced. Listing 3.2 (line 12) shows the TagHit task function signature. The compiler must encapsulate a TagHit task structured-block in a TagHit function even if it consists of only a single line of code. The body of the function consists of three main parts. It begins by dereferencing task-data pointers into their actual values and assigning them to variables of the same type as in Listing 3.2 (line 13 to 15), and ends by assigning the task-data values again to the pointer array (line 20 to 22). The TagHit task structured-block is then placed as it is, as seen in Listing 3.2 (line 18).

- Construct a task data pointer array that consists of all addresses of variables referenced in a TagHit ***task*** construct, as seen in Listing 3.2 (line 33 to 36). This array is passed to the TagHit runtime call that creates a new TagHit task, as in line 35. Chapter 5 discusses task creation and execution in more details.

- Create a task-lookup table consisting of all task definitions using the information extracted for each task in the previous points. Lines 1 to 10 of Listing 3.2 show an example of the lookup table. The compiler also passes the lookup table to the TagHit runtime, as seen in line 27. Chapter 5 explains the purpose of the task-lookup table.

Chapter 4

TagHit Experimental Runtime Design: Task Execution and Scheduling

This chapter describes an experimental runtime design for TagHit. It begins by discussing the design goals of the TagHit-runtime system. It then presents the architecture design of the runtime and describes its various components.

4.1 TagHit Runtime Design Goals

The TagHit-runtime system design goals should be precisely defined because they are essential to validate the correctness of the TagHit-runtime design and implementation. This stage focuses on implementing the runtime to create tasks and schedule

them; however, the TagHit runtime must be well designed to accommodate future development efforts such as data distribution and tasks dependencies.

A task is the fundamental unit of work in TagHit. Tasks may have input data on which to work and from which they may produce results. They are issued in program order, exactly as they are written via directives; yet they are asynchronous and dynamic to execute until completion on any available TagHit process.

When a task is created, the TagHit runtime system decides where to execute it. The runtime system also migrates tasks between different TagHit processes if necessary. Furthermore, to minimize memory consumption, the runtime does not duplicate tasks, and possibly their data, unless it is a necessity, as in the case of an adoption of a fault-tolerance mechanism requiring duplication of tasks.

To decide where to execute a TagHit task, a proper scheduler distributes tasks across TagHit processes based on a certain scheduling mechanism. However, the TagHit scheduler distributes tasks fairly and load balances them across all available TagHit processes in a way that ensures correct execution of both parent and child tasks.

Finally, TagHit is a directive-based model. The TagHit-runtime design does not conflict in anyway with this goal such that tasks shall be created via directives, specifically ***#pragma*** in C, and the runtime system takes care of the rest by scheduling and executing them as described earlier.

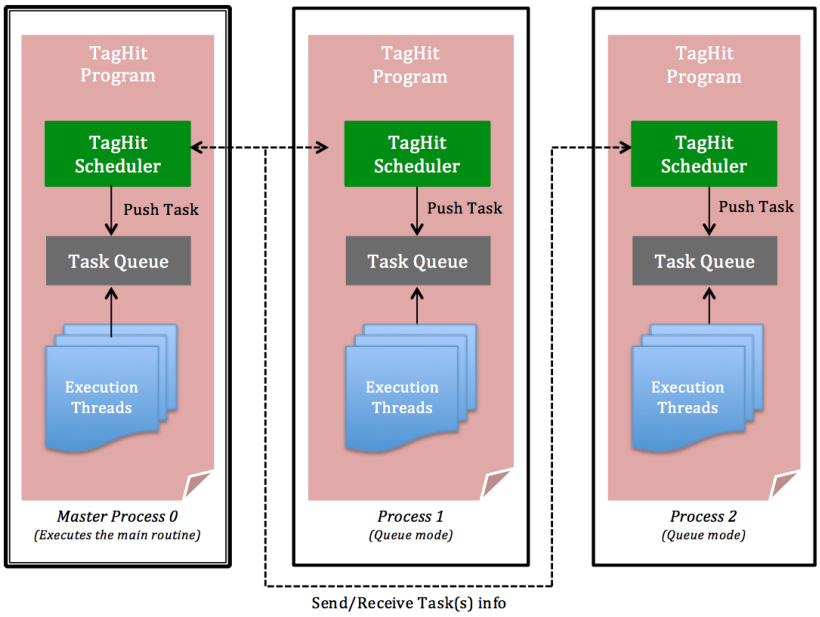

Figure 4.1: TagHit runtime architecture to create, schedule, and execute tasks in a distributed environment. It consists of four main components: TagHit porgram, TagHit scheduler, Task Queue, and Execution Threads. These components are distributed, and an instance of each component resides on every TagHit process.

4.2 Proposed Architecture Design

This section discusses our proposed architecture design of the TagHit runtime system. The design conforms to the design goals discussed in the previous section. We tried to make it simple yet fixable to accommodate future work and any possible enhancements to the API. Figure 4.1 demonstrates the general design and its various components.

The design of TagHit runtime is modular. Each design component defines a set of roles and functionalities in which the coupling between them is set to minimum.

This would make any further changes, improvements, or addition of new components easy as long as the new component defines a set of new roles and the way it interacts with the other components is clearly described. As Figure 4.1 shows, the TagHit runtime consists of four main components that define the basic semantics and mechanisms to create, schedule, and execute tasks in a distributed environment. These components are distributed, and an instance of each component resides on every TagHit process. This is essential for scalability, because centralized components usually create bottlenecks and increase contentions during the execution of a TagHit program. The roles of each component are described as follows.

4.2.1 TagHit Program

The TagHit program encapsulates the application code, which uses TagHit constructs, and all the runtime functionalities. The TagHit program is distributed across all TagHit processes. Only a master TagHit process, process 0 in Figure 3.1, executes the main routine, while other TagHit processes initially suspend and waits for tasks to be received for execution. The purpose of distributing TagHit programs across all processes is to make the runtime fully distributed and to facilitate direct access to tasks on every TagHit process. In fact, other mechanisms such as dynamic loading [4] would dynamically load tasks to processes, but the cost associated with dynamic loading at runtime makes our approach preferable. In addition, our approach simplifies the implementation of the TagHit runtime as well as the usage and implementation of TagHit applications by avoiding any restrictions imposed by other methods.

4.2.2 TagHit Scheduler

This component is mainly responsible for deciding where to execute a task. Scheduling in TagHit is fully distributed, and every TagHit process runs an instance of this component. When a TagHit process encounters a new task construct during the program execution, it creates a task and invokes the TagHit scheduler. The TagHit scheduler will then either schedule the task to run locally or send it to a remote TagHit process. If the task is sent to a remote process, the TagHit scheduler on that process will receive the task and schedule it for execution. Similarly, when a child task is launched during the execution of a task, the TagHit scheduler of the process that launches the child task will be invoked to decide where to execute it, eliminating the need to send it to a centralized scheduler.

The scheduling mechanism depends on the specific implementation of this component. The implementation of this component can range from a trivial round robin scheduling of tasks across TagHit processes to a more complex scheduling mechanism such as work stealing. Chapter 5 discusses this component's prototype implementation.

4.2.3 Task Queue

Task queues are FIFO queues that hold ready-to-run tasks. Each TagHit process operates on its own task queue, and the TagHit scheduler of that process pushes tasks into the task queue. A task queue in one TagHit process is not shared with any other process. This is essential to minimize both contentions and network

communications between TagHit processes while pushing tasks or consuming them from queues. In addition, this approach uses the memory of TagHit processes efficiently by allowing tasks to be distributed across different processes where they are scheduled to run, rather than residing only on one process.

Furthermore, task queues are not entirely locked by any kind of operation. This is necessary to allow concurrent access in the same TagHit process to different parts of the task queue and thus to extract more performance. For instance, a TagHit scheduler can push tasks directly to the task queue while other operations are accessing tasks in the same queue.

4.2.4 Execution Threads

Execution threads, which execute TagHit tasks, are key in our TagHit runtime design to exploit the potential benefits of shared memory parallelism to extract more performance. Each TagHit process runs one or more execution threads. An execution thread pulls tasks from the task queue of the process on which it is running and executes them.

4.3 Design Enhancement for Future Work

A global address space that spans all TagHit processes would further enhance the TagHit runtime. Exposing data globally would efficiently enable different (future) TagHit functions such as data distribution and task dependence. It would also allow tasks to be received asynchronously on a remote TagHit process without the need

to actively wait for them. Combining this scheme with task queues allows overlap of communication with computation to extract more performance. Follow-on studies, however, will lead to a broader understanding of how TagHit would achieve these features.

Chapter 5

TagHit Runtime Prototype Implementation: Task Execution and Scheduling

In this chapter, we discussed different TagHit runtime functions to create and schedule tasks. These functions implement (as a prototype) the experimental runtime design discussed in the previous chapter. The runtime prototype is implemented in C. The chapter first discusses the underlying libraries of the TagHit runtime and then describes the implementation of task creation, execution, and synchronization. Finally, the chapter discusses task scheduling and presents the approach implemented.

5.1 Underlying Libraries: MPI

Discussion of the TagHit experimental runtime design in Chapter 4 demonstrated that TagHit programs must be distributed across all TagHit processes to directly access task functions on every TagHit process. Furthermore, TagHit schedulers must send and receive TagHit tasks to achieve task execution on remote processes. These aspects of the design can be realized via MPI.

MPI [23] is the most common mechanism for expressing parallelism on distributed environments today. It is standardized, portable, and available on leading platforms. MPI also has several well-tested and efficient implementations, some of which are open source. Furthermore, the MPI standard has evolved to support derived data types, non-blocking collective communication, multithreading support, and remote memory access (one-sided) operations. These capabilities have made MPI a powerful system that enable a wide range of users, not only to write high-performance programs but also to build other platforms and runtime systems in C, C++, and Fortran.

The prototype uses MPI as the fundamental building block of the runtime system. We used MPI to launch TagHit processes and distribute the TagHit program. MPI is also used to facilitate communication and send and receive TagHit tasks between TagHit processes. In addition to these reasons for using MPI, the following subsections describe more powerful capabilities that make MPI preferable for implementing the TagHit runtime system.

5.1.1 Using MPI Data Types for Heterogeneous Systems

To achieve task execution on multiple TagHit processes, the processes must exchange task info with other process such as task input and output data of any type. MPI requires users to specify the type of data that need to be sent and received and uses special MPI data types for that purpose.

MPI data types are not just intended to show the type of the data at the send and receive times; it is also a powerful tool to ensure consistency of the data layout in different architectures. MPI data types allow the underlying MPI system to perform data representation conversion between source and destination machines [23]. Consequently, specifying MPI data types is crucial to prevent erroneous operations that might occur on heterogeneous systems. One concrete example of the advantage of using MPI data types is sending an integer from a machine that uses little-endian representation to another that uses big-endian. The bits can then be rearranged in the receiving machine to preserve the integer's original value.

5.1.1.1 The Challenge of Dynamically Determining MPI Data Types

Because TagHit is directive-based and the input and output data of tasks could be of any type, dynamically determining the correct MPI data type without user involvement is a challenge. The problem becomes even more complex when the data is a complex structure, such as C structs that may require building and using derived MPI data types.

A trivial solution for sending any data type via MPI is to serialize the data

into bytes and use ***MPI_BYTE*** as the MPI data type. However, ***MPI_BYTE*** prevents MPI from performing any type of data conversion [23], thus losing the benefit of using other MPI data types to support portability.

Another solution is to urge TagHit users (as an option) to specify the MPI data types of the task input and output data in the task creation constructs as they would do if they used the standard MPI routines. If the user does not specify the MPI data types, the runtime should serialize the data to bytes and use ***MPI_BYTE*** as a default option.

Alternatively, the compiler would assist in solving this issue. In fact, we decided to go with this solution for our runtime implementation. We assumed the compiler could extract the types of task data, match them with their corresponding MPI data types, and then provide them to the TagHit runtime system. Chapter 3 discusses the compiler assumptions.

5.1.2 MPI in Multithreaded Environments

One important feature of MPI is the support of multithreaded communication. MPI facilitates requesting a certain level of thread support (***single***, ***funneled***, ***serialized***, and ***multiple***). The ***multiple*** level allows an arbitrary number of threads to make MPI calls, simultaneously.

Multithreaded communication in MPI would potentially improve the performance of the TagHit runtime system. In TagHit, multiple execution threads may run to execute tasks. When a task completes its execution in a remote process,

the results must be sent to the original process that created it. Using multithreaded communications would facilitate this behavior by allowing multiple execution threads in one TagHit process to invoke MPI calls concurrently (via the TagHit scheduler), sending task results back to other TagHit processes. Furthermore, a TagHit scheduler could also independently receive messages from other TagHit processes by running on its own thread and without blocking execution threads from exposing additional work.

5.1.2.1 *MPI_Init_thread* to Support Multithreaded Communication

Making MPI calls directly in multithreaded applications without the required setup might cause a deadlock. The underlying system might generate a warning if the MPI library does not support such behavior. In fact, this deadlock might cause the runtime to degrade performance as well. To solve this problem, MPI has a provision to request a certain level of thread support by calling the ***MPI_Init_thread*** routine. Because the TagHit runtime uses threads to make MPI calls, as described above, MPI must be initialized with ***MPI_Init_thread*** instead of ***MPI_Init***. Unfortunately, however, some MPI implementations do not include multithreaded communication support by default. OpenMPI, for example, does not include this support if it has not been configured with the ***−enable-mpi-thread-multiple*** configure switch during installation. Another important point is that multithreaded communication support has only been lightly tested in OpenMPI at the time of writing this report. It likely does not work for thread-intensive applications. Other implementations of MPI such as MPICH and MVAPICH, however, have been supporting multithreaded communication for much longer and thus can be used to

ensure robustness.

5.2 Underlying Libraries: Pthreads

The discussion of the TagHit experimental runtime design in Chapter 4 demonstrated that TagHit processes may run one or more execution threads. Our prototype implementation utilized POSIX Threads [10], or Pthreads, to implement the execution threads. Based on the number of execution threads the user requests, each TagHit process launches a number of Pthreads at the runtime initialization stage. Basically, each Pthread is implemented as a loop that waits for tasks to be pushed into the task queue and is then triggered by the scheduler to pull one by one for execution. Once a Pthread pulls a task from the queue, it binds the task as its context by assigning the task instance to its thread-specific value. This will facilitate accessing different task information within the thread while executing, scheduling, or synchronizing the task. Furthermore, binding a context to the Pthread would easily enable task switching, which 5.5.1 discusses.

For future work, we recommend improving the TagHit runtime to extract even more performance by enhancing the implementation of the shared memory parallelism. For example, utilizing other frameworks such as Argobots, as Chapter 2 describes, would achieve massive on-node parallelism with dynamic and customizable on-node scheduling capabilities.

5.3 Tasks-Lookup Table

As we discussed in Chapter 3, the compiler encapsulates structured block of a TagHit task construct in a task functions. Thus, a pointer to the task's function must be known by the process that is going to execute it. In the case of executing a task on a TagHit process different from the one that has generated it, passing the task function pointer to that process is meaningless because TagHit processes do not share the same memory address space. In fact, as discussed in previous sections, each TagHit process has an instance of the TagHit program and thus the task function should be accessible to the remote process, but the pointer must be locatable. Moreover, other information about the data environment of the task, such as the number of data items and their types, must be available for the process to send or receive it.

We adopted a task lookup approach to facilitate finding a task definition, including both a pointer to the task functions and information about its data environment, in any TagHit process. As Chapter 3 explained, the compiler should construct a task lookup table when translating TagHit constructs into runtime code and pass it to the runtime system. Because each TagHit process has an instance of the TagHit program, which also encapsulates the runtime functionalities, the task-lookup table is available for all processes. As in Listing 3.2, the task-lookup table consists of all task definitions, each of which has fields as follows:

- **Task tag:** a unique identifier of the task definition and its index in the lookup table. Tags are exchanged and used by TagHit processes to look up

task definitions.

- **Task function:** a pointer to the task function. TagHit processes find the task function and execute it by using this field.

- **Input count:** the number of data items that make up the data environment of the task. This number is used to determine the number of data items to be sent or received.

- **Input data types:** an array of MPI data types, each corresponding to one data item of the task. They are passed to the MPI routines to ensure data portability on heterogeneous systems, as discussed in section 5.1.

- **Input data length:** an array of integers where each is the length of one data item of the task. They are also passed to the MPI routines to determine the length of each data item to be sent or received.

As the task-lookup table is available for all processes with the above fields, any TagHit process can retrieve a task definition by receiving its tag and, if necessary, receive the task's data environment, execute the task, and send it to another TagHit process.

5.4 Task Creation and Execution

TagHit tasks are created by the ***taghit_create_task*** runtime function, which is a result of translating the ***task*** constructs into runtime code. This function accepts two arguments as follows:

- **Task tag:** the unique identifier of the task definition and its index in the lookup table.

- **Input:** an array of pointers that consists of all addresses of data items (variables) referenced in the task structured block.

A call to this function creates a task with a set of information necessary to execute it, synchronize it with other tasks, and track its location and status. This information is organized in a data structure, named ***task_info***, consisting of the following:

- **Task id:** keeps track of the task when sent and received between TagHit processes.

- **Origin process:** the rank of the process that has created the task.

- **Executing process:** the rank of the process that has been assigned to execute the task.

- **Status:** the execution status of the task, which initially is set to ***WAITING*** and changed during the task's lifetime to either ***RUNNING*** or ***FINISHED***.

- **Task definition:** a pointer to the task definition item in the task lookup table of the process where the task resides.

- **Input data:** the pointer array that has been passed to the ***taghit_create_task*** runtime function and consists of all addresses of the task's data items.

- **Parent task:** a pointer to the parent of the task.
- **Child tasks:** an array of pointers to the current task's children.
- **Task lock:** a mutex to provide a locking mechanism to ensure mutual exclusion of the task.

When the task (i.e., ***task_info***) is first allocated, the task definition and input data members are assigned. Then, the runtime assigns an ID to the task and sets its status to ***WAITING*** and its origin process to the rank of the current process that has created it. If the task has a parent, a reference to the parent, which is the context of the execution thread that has generated it, is assigned to the task. Otherwise, the task will be assigned a default root task representing the main routine.

The TagHit scheduler is then invoked, passing the newly created task. The task is scheduled for execution locally or on a remote TagHit process according to the scheduling mechanism implemented in the scheduler. Either way, the scheduler will assign to the task the rank of the process that is going to execute it, then push it into the local queue to keep its reference. If the scheduler has chosen a remote TagHit process to execute the task, the scheduler will send the tag of the task to the remote process along with its ID and data environment. The scheduler running on the receiving process would then allocate a new task, assign the appropriate information to it, and then push the task into the task queue. However, it depends on the scheduling mechanism in place. Work stealing, for example, would initially schedule the task to execute on the process that has created it and allow other processes to steal it. Section 5.6 discusses the adopted scheduling mechanism.

When tasks are pushed into a task queue, the scheduler triggers the execution threads to pull it. An execution thread does not lock the entire task queue during the consumption or execution of any task. However, an execution thread only locks the task that is trying to pull to execute few simple instructions to check its status and then unlock it. This approach allows different execution threads to consume tasks without much overhead and without affecting other operations that may access other parts of the task queue or push tasks to it concurrently. Once a thread pulls a task from the queue, it changes its status to ***RUNNING***, binds the task as its specific value, and executes its associated task function.

If a child task is generated during the execution of a certain task, an independent child task that is not part of its parent is created and scheduled. However, a reference to the child task is added to the parent's array of child tasks so that the parent can wait on its child tasks or perform a task switch to them. Both task synchronization and task switching are discussed in the next section.

Finally, when a task completes execution, its status will be changed to ***FINISHED***. The execution thread that has finished executing a task will attempt to pull another task from the task queue.

5.5 Task Synchronization

When a task hits a synchronization point, the ***taghit_taskwait*** runtime function is called. This function will go though that task's array of child tasks and make sure all are executed. If the main routine hits the synchronization point, the child task

array of the root task, which represents the main routine, is used. The following subsection discusses aspects of task synchronization.

5.5.1 Synchronized Nested Tasks

At first glance, nested tasks appear to be easily scheduled and executed on any TagHit process. However, having a synchronization point adds internal dependencies between child and parent tasks in the task queues. Inefficient scheduling of parent and child tasks might lead to a deadlock.

The problem can be better understood as demonstrated in Figure 5.1. Suppose we have two tasks, Task 1 and Task 2, scheduled on the queue of process N. Also suppose that Task 1 generated Task 3, which happened to be scheduled on the same process as Task 1, as shown in Figure 5.1. Now, if Task 1 encounters a task synchronization point, it will suspend waiting for its child task (Task 3) to finish. However, because queues execute tasks in a FIFO fashion, Task 3 will not execute until Tasks 1 and 2 are finished, which consequently prevents the queue on process N from further progressing.

Similarly, suppose process X generated a child task T and decided to schedule it on a remote process Y. If process Y is also waiting on child tasks that are scheduled on other processes including process X, a cyclic dependency occurs on a large scale and between different TagHit processes.

The main trick here is that the TagHit synchronization point should not only suspend a task waiting for its child tasks. In fact, it should invoke a scheduling

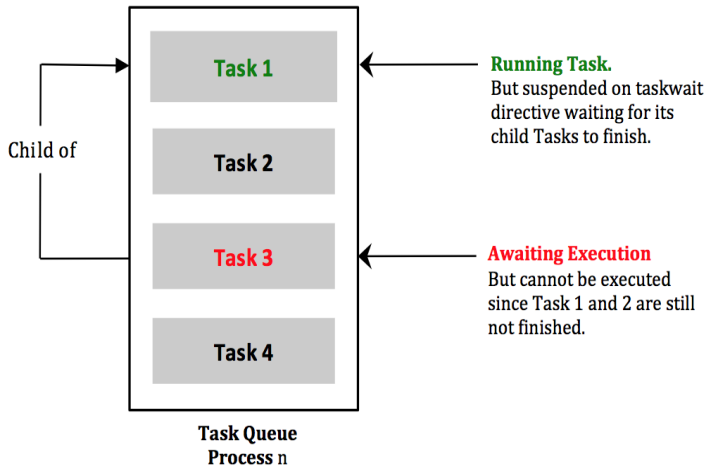

Figure 5.1: Example of a deadlock situation caused by nested tasks. Task 1, Task 2, and Task 3 are scheduled on the queue of process N. Task 3 is a child task of Task 1. If Task 1 encounters a synchronization point, it will suspend waiting for Task 3 to finish. However, because task queues are FIFO, Task 3 will not execute until Tasks 1 and 2 are finished.

point where it switches from executing the current task to executing other tasks found in the queue. This is, in fact, similar to what is known as task switching in OpenMP: the act of a thread switching from executing of one task to another, which normally happens at certain scheduling points, one of which is ***#pragma omp taswait***.

5.6 Task Scheduling

To implement a scheduling mechanism, we must consider the two main, yet challenging, design goals of TagHit runtime:

- Tasks should be fairly balanced between TagHit processes.
- Network communication between TagHit processes should be minimized.

Based on our experimental runtime design, scheduling in TagHit is also distributed where every TagHit process runs an instance of the scheduler. Thus, a centralized scheduling mechanism, such as work sharing or a centralized task queue, would violate this design. Furthermore, task scheduling in a distributed environment involves network activities, which are time-consuming operations and should be kept to a minimum. This section examines different scheduling mechanisms and specifies a preferred approach..

5.6.1 Trivial Task Scheduler

Round-robin or random task distributions between TagHit processes are the most trivial scheduling mechanisms to implement. In a round-robin scheduling mechanism, for example, when a TagHit process generates a new task, it invokes the TagHit scheduler running on that process to select a process in a round-robin fashion to execute the task. These scheduling mechanisms potentially cause unfair task distribution or unbalanced workload between processes.

5.6.2 Work-Stealing Scheduler

Work stealing [6, 1] is a distributed load-balancing mechanism whereby processors pull tasks from other processes. When a thief processor finds a victim with tasks, it steals a portion of the victim's task queue and then is able to continue executing tasks. That is, a process does not send generated tasks to other processes, but other (thief) processes pull them. A thief process does not attempt to steal tasks from

others until it exhausts all tasks in its own queue.

Work stealing can be done in a two-sided or one-sided manner [13, 30, 12]. Work stealing is efficient and scalable at utilizing one-sided communication, as shown in [30, 12]. However, considering the amount of modification to our runtime implementation and the complexity of managing data locks, the one-sided approach would require much more implementation effort. For this reason, we have implemented work stealing in two-sided manner. We reduce the performance overhead of this implementation by using MPI asynchronous operations and a multithreaded solution, as the next subsection describes.

5.6.2.1 Sending and Receiving Tasks

Communications in our TagHit runtime prototype are generated in the sender and receiver parts of the scheduler. Tasks are sent asynchronously via the scheduler utilizing MPI asynchronous communication. This would not block execution threads that invoke the scheduler, and thus they are able to expose more work. However, the scheduler receives tasks (and possibly other scheduling data) in a separate receiver thread, allowing tasks to be received and pushed to the queue without interrupting ongoing work in execution threads. In future work, if one-sided solution is adopted, it would eliminate the need for a receiver thread as it requires only one process to transfer data.

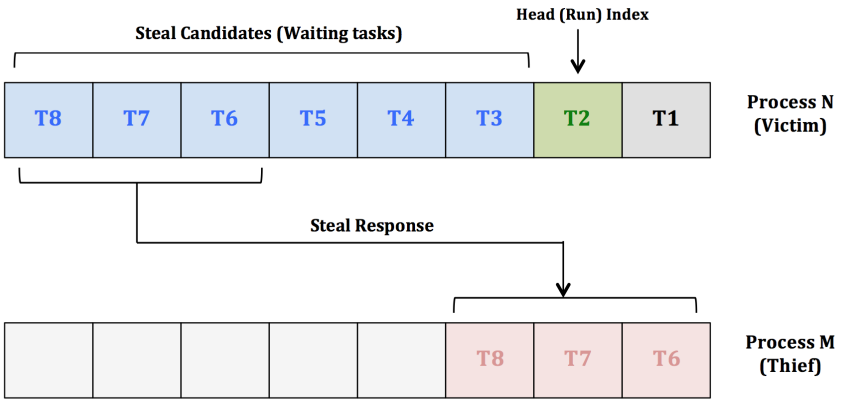

Figure 5.2: Work-stealing algorithm in TagHit. When a thief (process M) exhausts all tasks in its queue, it will steal tasks by randomly selecting a victim (process N). Process N will send half of the waiting tasks in its queue to process N.

5.6.2.2 The Work-Stealing Algorithm

The work-stealing algorithm is demonstrated in Figure 5.2 and described by the following sequence of steps (we divided the sequence into parts to distinguish the steps performed by the thief or the victim process).

During thief process:

1. A process will attempt to steal tasks when it exhausts all tasks in its queue. One of the execution threads will invoke the scheduler when it does not find more task to execute.

2. If (1) is true, the process selects a victim randomly or goes through other processes in a round-robin fashion. The second option is currently implemented for simplicity.

3. The process sends a **steal request** message to the victim process asynchronously. Then the execution thread that invoked the schedule suspends, waiting for tasks to be available.

   During victim process:

4. When a victim process receives a steal request, it checks its **steal candidates** counter if it is greater than 1. Steal candidates are tasks waiting for execution.

5. If (4) is true, the victim process calculates the number of tasks to steal by dividing the steal candidates counter by 2 (stealing half of the victim's work is an optimal strategy as demonstrated in [6]). Otherwise it sends a **steal abort** response message to the thief process.

6. Starting from the tail of the victim's task queue, the process checks if the task's current status is ***WAITING***.

7. If (6) is true, it locks the task and then changes its status from ***WAITING*** to ***STOLEN***. Then it unlocks it and stores its index. Otherwise, it moves to the next task from the tail of the queue.

8. The process repeats (6) to (7) until the number of tasks to steal is satisfied or the task's current status is ***RUNNING***, which means that it has reached the head of the queue.

9. The process decrements the steal candidate's counter by the number of stolen tasks.

10. The process then asynchronously sends stolen tasks to the thief process one by one.

During thief process:

11. The thief process receives either a **steal abort** or tasks as a response to its steal request.

12. If the thief process received a steal abort response, it repeats steps (2) to (12). Otherwise, it pushes the received tasks to the tasks queue so that execution threads can continue executing tasks.

Chapter 6

TagHit Experiments

In this chapter, we describe two benchmarks developed using our TagHit implementation. We have implemented a benchmark to compute PI and another to calculate the Fibonacci sequence of a given number as a recursive algorithm.

6.1 Environment Setup

Our experiment was performed using the Crill cluster at the University of Houston. This cluster has the following node and network specifications:

- NLE Systems nodes with the following configuration:
  - Four 2.2 GHz 12-core AMD Opteron processors (48 cores total)
  - 64 GB main memory
  - Two dual-port 4xDDR Infiniband HCAs

- Network Interconnect with the following configuration:
  - 144 port 4xInfiniband DDR Voltaire Grid Director ISR 2012 switch

6.2 Experimental Results

In the compute PI benchmark, we discussed the program scalability and the scheduler behavior. In the second, we demonstrated the functional nature of the runtime to handle synchronized nested tasks.

6.2.1 Compute PI

This benchmark calculates PI based on a mathematical approximation of

$$\int_0^1 \frac{4}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles as follows:

$$\sum_{i=0}^{n} F(x)\Delta x \approx \pi$$

The width of each rectangle is Δx and its height is $F(x)$, as shown in Figure 6.1. A pseudocode to compute PI serially is shown in Listing 6.1. Executing a serial program to compute PI in Crill takes about 26 seconds.

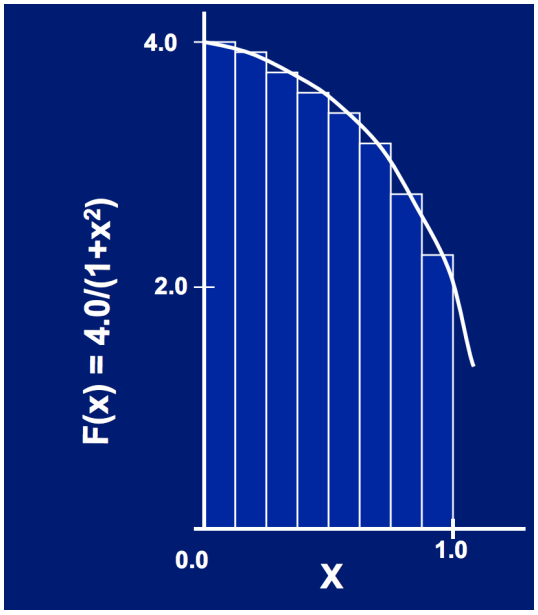

Figure 6.1: PI function. PI value is approximated as a sum of rectangles. The width of each rectangle is Δx and its height is $F(x) = 4/(1 + x^2)$

```
1
2  x, pi, sum = 0.0;
3  step = 1.0 / num_steps;
4  num_steps = 1000000000;
5
6  for (i=0; i < num_steps; i++)
7  {
8    x = (i + 0.5) * step;
9    sum = sum + 4.0 / ( 1.0 + x*x );
10 }
11
12 pi = step * sum;
```

Listing 6.1: Compute PI Pseudocode

We have created a TagHit version of this program to calculate PI in parallel by dividing the loop, as in the above pseudocode (line 6), into a set of 64 TagHit tasks. The average execution time of running the TagHit version using one, two, four, and eight processes is shown in Table 6.1.

TagHit Processes	Execution Time in Seconds
1	25.8
2	12.9
4	6.5
8	3.2

Table 6.1: TagHit compute PI execution times.

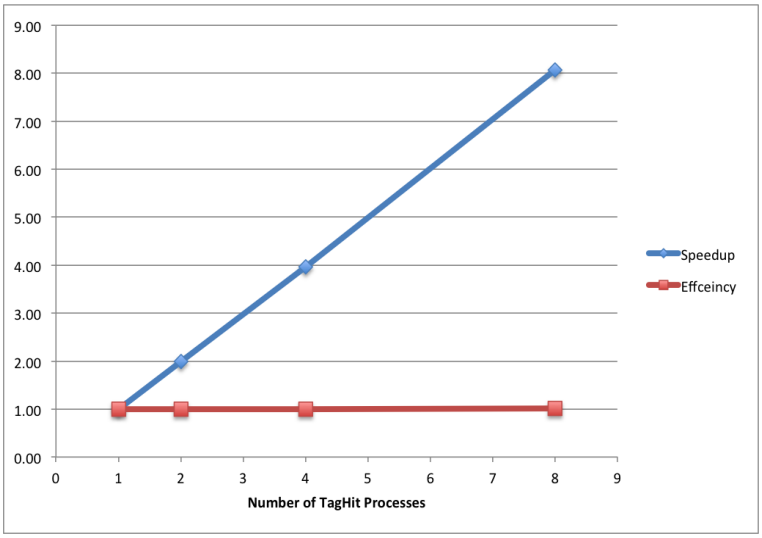

Figure 6.2: Speedup and parallel efficiency of the TagHit compute PI program. Using one, two, four, and eight processes, the program scales linearly and provides almost 100% efficiency with each run instance.

As in Figure 6.2, computing PI in parallel using TagHit provided the expected speedup ratio and efficiency. The program scales linearly as the speedup grows by a factor of 2 as the number of processes increases. In addition, how effectively TagHit processes are utilized defines the efficiency. Figure 6.2 shows almost 100% parallel efficiency with each run instance.

To study the behavior of the TagHit scheduler, Table 6.2 illustrates how tasks are distributed between TagHit processes. For all run instances using one, two,

Number of TagHit Processes	Rank 0	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5	Rank 6	Rank 7
1	64	-	-	-	-	-	-	-
2	32	32	-	-	-	-	-	-
4	16	16	16	16	-	-	-	-
8	8	8	8	8	8	8	8	8

Table 6.2: Number of tasks executed by each process of the TagHit compute PI program.

TagHit Process Rank	Aborted Steals	Successful Steals	Total Steal Request
0	1	1	2
1	2	3	5
2	9	3	12
3	27	3	30
4	7	5	12
5	13	5	18
6	16	6	22
7	1	1	2
Average	10	3	13

Table 6.3: Number of aborted and successful steals made by each process when running the TagHit compute PI program with eight processes.

four, and eight TagHit processes, the number of tasks executed by each process is shown. In the table, we used Rank 0, Rank 1, Rank 2, and so on to name each TagHit process.

In all run instances, Table 6.2 shows that our work stealing scheduler successfully balances the load between available processes. And because all tasks in our implementation do the same amount of work, tasks are evenly distributed between TagHit processes.

Furthermore, to study the work-stealing behavior of the scheduler, we counted

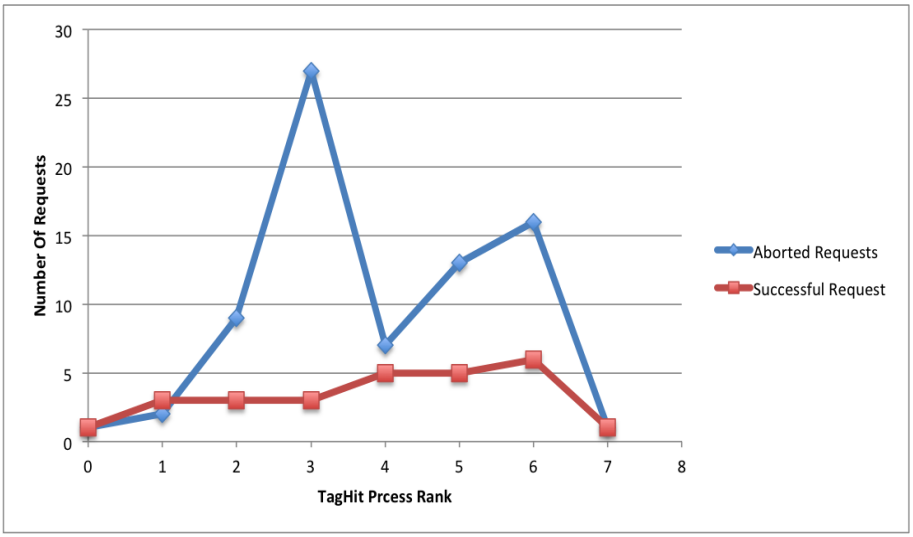

Figure 6.3: Successful and aborted steals request of the TagHit compute PI program. The figure shows a comparison between the number of successful and aborted steals made by each process when running the program with eight processes.

the total number of steal requests and the number of aborted steals (before receiving the last successful steal) each TagHit process made. Table 6.3 shows these numbers when running the program with eight TagHit processes. Figure 6.3 provides a comparison between the number of successful steals and aborted steals.

To analyze these numbers, we considered the average of aborted steals and the average of successful ones made by each process, shown in Table 6.3: 10 and 3 respectively. In other words, each process sent 10 failed requests and 3 successful ones on average. That is, each process approximately tried three times before each successful request. We beleive this is not optimal for such small scale and indicates that our victim-selection strategy, as well as the number of tasks to steal with each request, should be well studied and enhanced. In fact, several researches [30, 16, 25, 24] have discussed victim and work-size selection strategies for work-stealing schedulers. Locality-aware and hierarchical work-stealing are potential strategies

to consider for TagHit.

6.2.2 Fibonacci: A Recursive Algorithm

The Fibonacci series can be calculated by use of the formula

$$F(n) = F(n-1) + F(n-2)$$

This can be archived by a recursive algorithm as in Listing 6.2 .

```
1
2 int Fibonacci(int n)
3 {
4   if ( n < 2 )
5     return n;
6   else
7     return (Fibonacci(n-1) + Fibonacci(n-2));
8 }
```

Listing 6.2: Fibonacci Pseudocode

We created a TagHit program to calculate the Fibonacci series. Listing 6.3 shows the directive-based code. It creates two tasks to compute the recursive calls, ***fibonacci(n-2)*** and ***fibonacci(n-1)***, and waits for them before computing the sum of their return values.

The Fibonacci recursive algorithm would achieve little to no parallelism. We also note that this is not the fastest way to compute Fibonacci using parallel tasks. However, we created this program to demonstrate the functional nature of TagHit to handle synchronized nested tasks.

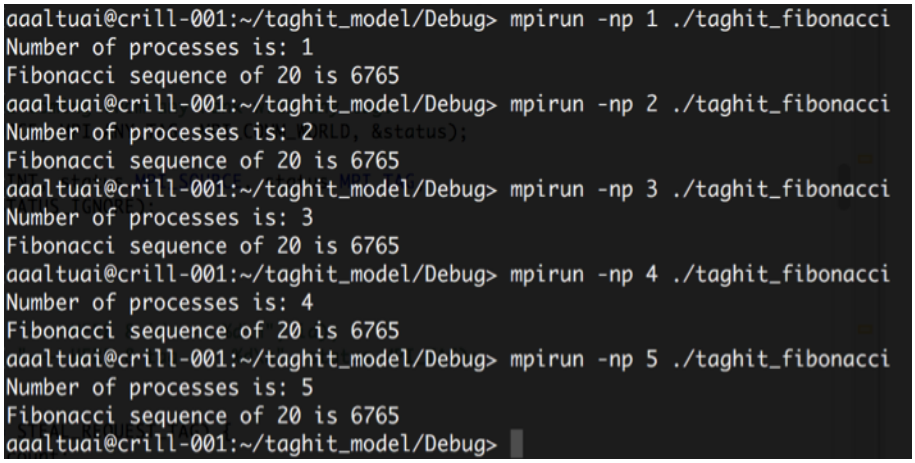

Figure 6.4: Screenshot of the TagHit Fibonacci program execution. The program calculates the 20th Fibonacci number, using one, two, three, four, and five processes.

```
1  long fibonacci(long n) {
2    long i, j;
3    if (n < 2)
4      return n;
5    else {
6      #pragma taghit tasking
7        i = fibonacci(n-1);
8      #pragma taghit tasking
9        j = fibonacci(n-2);
10     #pragma taghit taskwait
11     return i + j;
12   }
13 }
```

Listing 6.3: Fibonacci TagHit Code

Figure 6.4 shows screenshot of the program execution to calculate the 20th Fibonacci number, using one, two, three, four, and five TagHit processes. As shown, the TagHit runtime has been able to handle the parallel nested tasks in the Fibonacci algorithm and managed the synchronization exposed between them to produce correct results.

Chapter 7

Conclusion and Future Work

The first part of this thesis reviewed a set of tasking models. The second part discussed an experimental design and prototype implementation of the task creation and scheduling modules of the TagHit runtime. It described different aspects of the runtime and then detailed a prototype implementation of task creation and task synchronization directives.

Chapter 2 provided overview of six related task-based models, including their key features and our observations of them. Chapter 3 discussed the TagHit API, particularly the basic TagHit task creation construct and the task synchronization construct. The TagHit API may eventually include other constructs as well as clauses in the task creation construct, but they are out of the scope of this thesis. Chapter 4 defined the design goals of the TagHit runtime and then proposed an experimental design accordingly. Chapter 4 also includes a detailed discussion of

different aspects of the runtime design, including the architecture and its different components. The runtime design consists of four main components: TagHit program, TagHit scheduler, task queue, and execution threads. The runtime components are distributed such that each process runs an instance of the scheduler and has it own queue. In addition, The TagHit runtime exploits the potential benefits of shared memory systems by running a customizable number of threads per processing element.

Chapter 5 described how the prototype, which is implemented in C, creates and schedules TagHit tasks. Specifically, the chapter began by discussing the underlying libraries that we used to implement the runtime system, namely MPI and Pthreads. Then, the chapter described the technical implementation details to create, schedule, execute and synchronize tasks. Chapter 5 also presented the distributed work-stealing scheduling mechanism adopted to migrate and load balance tasks among processing resources.

Finally, Chapter 6 presented a study of two TagHit benchmarks. The chapter discussed the scalability and scheduler behavior of the first benchmark, which is a TagHit program to compute PI using 64 tasks. The result shows that the program provides good speedup and scalability using two, four, and eight TagHit processes. However, although the tasks were distributed nicely between TagHit processes, the behavior of the scheduler shows that the victim and work-size selection strategies are not optimal and should be studied further and enhanced. The second benchmark, however, is a TagHit recursive algorithm to calculate the Fibonacci series, which we used to demonstrate the functional nature of the runtime to handle synchronized

nested tasks.

The work presented in this thesis is an experimental runtime of the baseline functionalities of the TagHit API. Future development of TagHit, however, is required to build a complete solution that is targeted for exascale. TagHit should ultimately support other features such as tasks with different levels of granularity, data distribution mechanisms, and task dependencies. The TagHit API may be extended with other directives as well as clauses to the task creation directive to enable these features.

In addition, possible enhancements may be applied to our TagHit runtime design and implementation. An implementation of a global address space that spans all TagHit processes may further enhance the TagHit runtime and efficiently enable different TagHit functions. Work-stealing in TagHit can also be enhanced by implementing the one-sided approach and explore better strategies for victim selection.

Because TagHit is a directive-based API, the compiler will be involved in translating TagHit constructs into their corresponding TagHit runtime functions. Once a feature-complete TagHit runtime is in place, the compiler implementation of these constructs is required. We discussed the compiler assumptions in section 3.4 to contribute to the future development of the compiler and ensure it complies with our runtime.

Overall, although the experimental design and prototype discussed in this thesis have proven the general concept of TagHit as a potential API, we have assessed implementation costs and challenges to execute, synchronize, and schedule tasks. This work is not a complete solution for TagHit, but we expect it to guide the

definition of TagHit concept and semantics for further research and development.

Bibliography

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[3] M. E. Bauer. *Legion: Programming distributed heterogeneous architectures with logical regions*. PhD thesis, STANFORD UNIVERSITY, 2014.

[4] D. M. Beazley, B. D. Ward, and I. R. Cooke. The inside story on shared libraries and dynamic loading. *Computing in Science & Engineering*, 3(5):90–97, 2001.

[5] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

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

[7] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, J. Dongarra, and P. Luszczek. Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach. *Scalable Computing and Communications: Theory and Practice*, pages 699–733, 2012.

[8] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.

[9] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *PDP 2010-The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

[10] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[11] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra. Ptg: an abstraction for unhindered parallelism. In *Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on*, pages 21–30. IEEE, 2014.

[12] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 53. ACM, 2009.

[13] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[14] J. Dokulil, M. Sandrieser, and S. Benkner. Ocr-vx-an alternative implementation of the open community runtime. In *International Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures, in conjunction with SC15. Austin, Texas, November 2015*, 2015.

[15] P. Grubel, H. Kaiser, J. Cook, and A. Serio. The performance implication of task size for applications on the hpx runtime system. In *2015 IEEE International Conference on Cluster Computing*, pages 682–689. IEEE, 2015.

[16] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

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

[18] 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*, page 6. ACM, 2014.

[19] T. Kooburat and M. Hwang. Extending task-based programming model beyond shared memory systems, 2009. University of Wisconsin-Madison.

[20] J. Landwehr, J. Suetterlein, A. Márquez, J. Manzano, and G. R. Gao. Application characterization at scale: lessons learned from developing a distributed open community runtime system for high performance computing. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 164–171. ACM, 2016.

[21] J. Lee and M. Sato. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In *2010 39th International Conference on Parallel Processing Workshops*, pages 413–420. IEEE, 2010.

[22] T. Mattson, R. Cledat, Z. Budimlic, V. Cave, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar. Ocr: The open community runtime interface version 1.1. 0, 2015.

[23] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 3.1, 2015.

[24] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, 2011.

[25] S. Perarnau and M. Sato. Victim selection and distributed work stealing performance: A case study. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 659–668. IEEE, 2014.

[26] V. Sarkar, W. Harrod, and A. E. Snavely. Software challenges in extreme scale systems. In *Journal of Physics: Conference Series*, volume 180, page 012045. IOP Publishing, 2009.

[27] S. Seo, A. Amer, P. Balaji, P. Beckman, C. Bordage, G. Bosilca, A. Brooks, A. CastellAs, D. Genet, T. Herault, et al. Argobots: A lightweight low-level threading/tasking framework. https://collab.cels.anl.gov/display/ARGOBOTS/, 2015.

[28] 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*, page 81. ACM, 2015.

[29] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling. Preliminary design examination of the parallex system from a software and hardware perspective. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):81–87, 2011.

[30] A. Vishnu and K. Agarwal. Large scale frequent pattern mining using mpi one-sided model. In *2015 IEEE International Conference on Cluster Computing*, pages 138–147. IEEE, 2015.

[31] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An api for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[32] XcalableMP Specification Working Group. XcalableMP Specification Version 1.2.1, 2014.

[33] A. Yarkhan, J. Kurzak, and J. Dongarra. Quark users guide. *Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tennessee*, 2011.