Scalable Task Parallel Programming in the Partitioned Global Address Space

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

James Scott Dinan, M.S.
Graduate Program in Computer Science and Engineering

The Ohio State University
2010

Dissertation Committee:

P. Sadayappan, Advisor
Atanas Rountev
Paolo Sivilotti
© Copyright by
James Scott Dinan
2010
ABSTRACT

Applications that exhibit irregular, dynamic, and unbalanced parallelism are growing in number and importance in the computational science and engineering communities. These applications span many domains including computational chemistry, physics, biology, and data mining. In such applications, the units of computation are often irregular in size and the availability of work may be depend on the dynamic, often recursive, behavior of the program. Because of these properties, it is challenging for these programs to achieve high levels of performance and scalability on modern high performance clusters.

A new family of programming models, called the Partitioned Global Address Space (PGAS) family, provides the programmer with a global view of shared data and allows for asynchronous, one-sided access to data regardless of where it is physically stored. In this model, the global address space is distributed across the memories of multiple nodes and, for any given node, is partitioned into local patches that have high affinity and low access cost and remote patches that have a high access cost due to communication. The PGAS data model relaxes conventional two-sided communication semantics and allows the programmer to access remote data without the cooperation of the remote processor. Thus, this model is attractive for supporting irregular and dynamic applications on distributed memory clusters. However, in
spite of the flexible data model, PGAS execution models require the programmer to explicitly partition the computation into a process-centric execution.

In this work, we build a complete environment to support irregular and dynamic parallel computations on large scale clusters by extending the PGAS data model with a task parallel execution model. Task parallelism allows the programmer to express their computation as a dynamic collection of tasks. The execution of these tasks is managed by a scalable and efficient runtime system that performs dynamic load balancing, enhances locality, and provides opportunities for efficient recovery from faults. Experimental results indicate that this system is scalable to over 8192 processor cores, can achieve extremely high efficiency even in the presence of highly unbalanced and dynamic computations, and can also be leveraged to enable rapid recovery from failures.
To my parents who gave me breath

and to my wife, Kimberly, who took it away.
ACKNOWLEDGMENTS

When I started graduate school, I was like most; I didn’t know what I wanted to do, but I knew for certain that I would do it here. It was my good fortune to meet Saday and join his high performance computing research group. None of the work contained within this document would have been possible without his kind and patient mentorship. Saday introduced me to the fields of high performance and parallel computing and inspired me in the pursuit of these disciplines. Throughout my scholarly journey, he has encouraged me to pursue my interests as a researcher and he was always ready as an ardent supporter and skeptical reviewer when I needed each.

The many wonderful and talented members of the OSU computer science department have made a lasting impression on me and have given direction to my studies and my research. This dissertation would not have been possible without the strong support of my committee members, Paul and Nasko, who have always encouraged me to see things differently and provided a diversity of opinion that has kept me well-rounded and given me a broader outlook on the field. One of the greatest challenges during my graduate career was teaching. Bruce Weide and Kathryn Reeves were daring enough to trust me with an undergraduate core class and Bettina Bair, along with many others, taught me how to teach, for which I will always be grateful.
During my time in graduate school, I was fortunate to pursue three internships. My first was at Cray Inc where Brad Chamberlain, Steve Deitz, and Mary Beth Hribar gave me a home away from home working on Chapel, an exciting new parallel programming language. Later I spent a winter at Pacific Northwest National Laboratory, where I had the pleasure of joining Sriram Krishnamoorthy and Jarek Nieplocha who helped to greatly accelerate the pace of this work through their many insightful and valuable contributions. Finally, I visited Argonne National Laboratory where I joined in a very productive collaboration with Pavan Balaji, Rajeev Thakur, and Rusty Lusk, resulting in some interesting and exciting work on hybrid parallel programming.

I would not have made it to graduate school were it not for the encouragement, support, and hard work of my parents, Debra Cranwell and Scott Dinan; my two brothers Joe and Chris; and my sister, Rebecca. Nor could I have accomplished any of this without the loving support of my wife, Kimberly, who distracted me when I needed it and told me to stop procrastinating when I needed that too. Any accomplishment of mine will always be shared with these wonderful people who have supported me throughout all of my endeavors. When the going was not easy, they have made it easy to find kindness and in times of celebration, I am always glad that I have them to share it with.

Good friends have been my constant through core classes, quals, candidacy, paper deadlines, and defense. Too many to list have been a part of my life and I humbly thank those that I may here overlook. My labmates have been my daily companions on this journey. Especially Brian Larkins and Joshua Levine, whose sense of humor was never appropriate and always appreciated. Sriram Krishnamoorthy, Jerry Sabin,
Uday Bondhugula, Muthu Baskaran and the other senior members of the laboratory helped to show me the ropes and made me feel welcome when I joined the group. A new group of students, including Tom Henretty, Justin Holewinsky, Md Humayun Arafat, and Naznin Fauzia make me feel confident that I leave this place in good hands and excited for the future and the many wonderful things to come.

James Dinan
Columbus, Ohio
May 21, 2010
VITA

November 28, 1980 .......................... Born: Lee, MA, USA

May 2004 ................................. B.S.C.S.E. Comp. Systems Eng.,
University of Massachusetts,
Amherst, MA, USA

September 2009 ............................. M.S. Computer Science,
The Ohio State University,
Columbus, OH, USA

Spring 2005—Present ........................ Graduate Research Associate,
The Ohio State University

PUBLICATIONS

Research Publications


**FIELDS OF STUDY**

Major Field: Computer Science and Engineering

Studies in High Performance Computing: Prof. P. Sadayappan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvi</td>
</tr>
<tr>
<td><strong>Chapters:</strong></td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Background</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Extracting Parallelism</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Irregular and Dynamic Parallelism</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Parallel Programming Models</td>
<td>6</td>
</tr>
<tr>
<td>2.3.1 Partitioned Global Address Space Models</td>
<td>7</td>
</tr>
<tr>
<td>2.3.2 The Asynchronous Gap</td>
<td>9</td>
</tr>
<tr>
<td>2.3.3 Task Parallel Execution Model</td>
<td>10</td>
</tr>
</tbody>
</table>
3. The Scioto Task Parallel Programming Model ........................................ 11
   3.1 Scioto Programming Model .......................................................... 11
   3.2 Task Semantics ............................................................................ 12
   3.3 Task Input/Output Model ............................................................... 14
      3.3.1 Common Local Objects .......................................................... 15
   3.4 Task Execution Model .................................................................. 16
   3.5 Multilevel Parallel Task Collections ............................................. 18
      3.5.1 Multiprocessor Execution Groups ........................................... 19
   3.6 Scioto Programming Interface ...................................................... 20
      3.6.1 Data Types ........................................................................... 22
      3.6.2 Task Implementation .............................................................. 23
      3.6.3 Core Routines ....................................................................... 23
      3.6.4 Task Object Management ....................................................... 25
      3.6.5 Execution Group Interface ...................................................... 26
      3.6.6 Low-level Interface ................................................................. 27
      3.6.7 Example: Matrix-Matrix Multiplication .................................... 28
   3.7 Conclusion .................................................................................... 29

4. Scalable Runtime System ................................................................. 31
   4.1 Task Collection Design and Implementation .................................... 31
      4.1.1 One-sided Communication ...................................................... 32
      4.1.2 Distributed Task Queues ......................................................... 34
   4.2 Dynamic Load Balancing ............................................................... 34
      4.2.1 Work Stealing ....................................................................... 35
   4.3 Termination Detection ................................................................... 36
      4.3.1 Token Coloring Optimization ................................................... 38
   4.4 Improving Task Queue Performance and Efficiency ...................... 39
      4.4.1 Split Task Queues ................................................................. 39
      4.4.2 Lockless Release Operations ................................................... 41
      4.4.3 Efficient Task Creation .......................................................... 41
      4.4.4 ARMCI Runtime System, Latency and Overhead ..................... 42
   4.5 Improving the Scalability of Work Stealing .................................... 43
      4.5.1 Work Splitting ...................................................................... 44
      4.5.2 Contention Management ........................................................ 46
   4.6 Conclusion .................................................................................... 50

5. Experimental Evaluation ............................................................... 51
   5.1 Benchmarks .................................................................................. 52
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Benchmarking workload characteristics.</td>
</tr>
</tbody>
</table>
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Task parallel execution model.</td>
<td>16</td>
</tr>
<tr>
<td>4.1 Work stealing algorithm using one-sided communication.</td>
<td>37</td>
</tr>
<tr>
<td>4.2 Bakery locking algorithm $lock()$ operation using atomic increment</td>
<td>47</td>
</tr>
<tr>
<td>4.3 Spinlock $lock()$ operation using atomic swap</td>
<td>48</td>
</tr>
<tr>
<td>4.4 Aborting spinlock $lock()$ operation using atomic swap</td>
<td>49</td>
</tr>
<tr>
<td>5.1 Performing barrier synchronization using the termination detection</td>
<td>57</td>
</tr>
<tr>
<td>6.1 Fault tolerant task collection execution algorithm.</td>
<td>80</td>
</tr>
<tr>
<td>6.2 $TC.process$: Execute all tasks in a task collection</td>
<td>81</td>
</tr>
<tr>
<td>6.3 $TC.scan$: Detect data corruption due to failure and initiate recovery.</td>
<td>82</td>
</tr>
<tr>
<td>6.4 $TC.check_completion$ (naive): Find incomplete tasks and add to the</td>
<td>83</td>
</tr>
<tr>
<td>task collection.</td>
<td></td>
</tr>
<tr>
<td>6.5 $TC.check_completion$ (home-based): Find incomplete tasks and add to</td>
<td>84</td>
</tr>
<tr>
<td>the task collection.</td>
<td></td>
</tr>
<tr>
<td>6.6 SCF Pseudocode: Task collections can be used for one electron, two</td>
<td>88</td>
</tr>
<tr>
<td>electron, and density computations.</td>
<td></td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A partitioned global address space distributed across $n$ processes. A one-sided put operation is being performed using a low latency rDMA operation supported by many high performance interconnects.</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Logical view of a Scioto task pool on six processors. Colors indicate tasks with differing amounts of work.</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>A task descriptor is a contiguous object with a header that contains task meta-data and a body that contains task arguments.</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Example dynamic parallel task tree. Completed tasks are black, running tasks are shaded, and available non-started tasks are white.</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Multilevel parallel task collection snapshot on four groups. Each group executes a task as a team on multiple processors.</td>
<td>19</td>
</tr>
<tr>
<td>3.5</td>
<td>Snapshot of a task collection distributed across 4 processors. Tasks are prioritized based on their affinity (color).</td>
<td>21</td>
</tr>
<tr>
<td>3.6</td>
<td>Task descriptor implementation that incorporates a task body of user-defined size (<em>left</em>). An example user-defined task body for a blocked matrix-matrix multiplication task (<em>right</em>).</td>
<td>24</td>
</tr>
<tr>
<td>3.7</td>
<td>A simple task collection processing function implemented using the low level interface.</td>
<td>28</td>
</tr>
<tr>
<td>3.8</td>
<td>Task-parallel blocked matrix-matrix multiplication using Global Arrays.</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>The Scioto framework is built on top of ARMCI and interoperates with a variety of PGAS libraries including Global Arrays and Global Trees.</td>
<td>33</td>
</tr>
</tbody>
</table>
4.2 Distributed task queue implementation of the Scioto task collection shown in Figure 3.5. ......................................................... 35

4.3 Split Queue: Allows lock-free local access to the private portion of the queue and provides zero-copy transfers between public and private portions of the queue. ......................................................... 40

4.4 Illustration of work distribution over time using the steal half work splitting strategy on 16 processors. Shaded boxes represent processes with surplus work and empty boxes represent processes searching for work. ......................................................... 46

5.1 Example small UTS search trees, full UTS workloads can contain billions to trillions of nodes. Interior nodes are shaded and leaf nodes are white. ......................................................... 54

5.2 Bouncing producer consumer example task tree and corresponding task queue. The next producer node resides at the tail of the queue, available to be stolen. ......................................................... 55

5.3 Termination detection, ARMCI Barrier, and MPI Barrier timings on 64 cluster nodes. ......................................................... 57

5.4 Performance of split vs locked task queues for the BPC benchmark with \( n = 64 \) and \( d = 32768 \). ......................................................... 59

5.5 Task creation times for baseline, buffer recycling, and in-place task creation schemes. ......................................................... 60

5.6 Impact of steal-half over steal-1 (Cilk-style) and steal-2. ............... 61

5.7 Performance and percent efficiency for Madness, the BPC benchmark, and UTS on 8,192 cores. Efficiency is calculated as the ratio of time spent executing tasks to total time. ......................................................... 64

5.8 Total number of successful steals and average acquire latency for Madness, BPC, and UTS. ......................................................... 67

5.9 Total number of aborted and failed steals for Madness, UTS, and the BPC benchmark. ......................................................... 69
6.1 Penalty in tasks re-executed for a worst-case single process failure on SCF. 89

6.2 Execution time for no faults and worst case failures using naive and home-based recovery schemes. 90

6.3 SCF performance relative to the case with no faults. 91

8.1 Chained matrix-matrix multiplication: The dot product of block row 1 of $A$ with block column 1 of $B$ produces block $C_{1,1}$ enabling the corresponding tasks to begin in the second multiplication. 103

8.2 Matrix-matrix multiplication task dependence graph without and with clock synchronization for $C = A \cdot B$ followed by $E = C \cdot D$. 104
CHAPTER 1

Introduction

Clusters of commodity multicore processors have become the most powerful and
dominant systems in the field of high performance computing. The most popular
method of writing parallel programs for such systems is the industry standard Message
Passing Interface [57] (MPI). MPI provides the programmer with a Single Program
Multiple Data (SPMD) view of the computation where the programmer expresses
their algorithm in terms of a fixed number of processes that share data with each
other by exchanging messages. MPI has been a very successful model for applications
with regular communication patterns and applications which are amenable to static
or periodic synchronous partitioning of the workload. However, many important
scientific and engineering applications operate on irregular and sparse data which can
result in irregularity in the communication pattern and imbalance or dynamism in
the structure of the parallelism. Such characteristics are not well matched with MPI’s
programming model and solving these problems using today’s powerful distributed
memory clusters is challenging.

A new class of parallel programming models has emerged to support applications
that share data through irregular and asynchronous communication. These models
aggregate the memory of multiple nodes in a distributed memory cluster into a global
address space that is logically partitioned into patches that are associated with each process in the computation. Such Partitioned Global Address Space (PGAS) models provide the programmer with a global view of data that may physically distributed and allow for asynchronous, one-sided access to shared data. Examples of such systems include Unified Parallel C (UPC) [82], Co-Array Fortran (CAF) [65], Global Arrays (GA) [64], and Titanium [88].

These systems provide convenient mechanisms that can efficiently support irregular and dynamic accesses to shared data, however they still require the programmer to express their computation in the process-centric SPMD model used by MPI. Irregularity in the data access pattern also often results in irregularity in the computation, dynamically evolving parallelism, and load imbalance. Achieving good efficiency and high performance with such parallelism requires sophisticated techniques to manage the workload and such mechanisms are not provided by PGAS models. Thus, PGAS programmers have often resorted to simple but inefficient strategies to manage their parallelism.

In this work, we address this gap between the flexible PGAS data model and rigid SPMD computation model by introducing a task parallel approach to expressing the computation. Task parallel programming is a technique for expressing and managing parallelism by decomposing a computation into a dynamic collection of tasks that can be executed in parallel. Tasks in this collection can be automatically scheduled and executed by a runtime system to improve load balance, optimize for data affinity, and provide resilience in the presence of faults. In addition, the task model is able to support computations that exhibit dynamic parallelism where additional parallelism
is uncovered as the computation progresses and where the structure of the parallelism may be data-driven.

We begin in Chapter 2 with an overview of parallel programming and models for expressing parallel computations. In Chapter 3 we then introduce Scioto, our task parallel programming model which extends PGAS models with support for Scalable Collections of Task Objects. Next, in Chapter 4 we present an efficient and scalable dynamic load balancing runtime system for the Scioto programming model. This is followed in Chapter 5 by an experimental evaluation of the system that demonstrates scalability and high efficiency for several benchmarks on up to 8,192 processor cores. In Chapter 6 we present extensions to Scioto that leverage the task parallel programming model in order to perform selective restart from failures. This recovery technique reduces the cost for recovery from $O(N)$, where $N$ is the number of processors, to $O(N_{\text{failed}})$, where $N_{\text{failed}}$ is the number of failed nodes. In Chapter 7 we discuss related work in the areas of task parallelism, runtime systems, and fault tolerance. Next, in Chapter 8 we discuss future directions for research in these areas, including methods for expressing and managing dependence information and integrating with popular acceleration hardware. Finally, we conclude in Chapter 9.
CHAPTER 2

Background

In this chapter, we provide the context for this work within the field of parallel and high performance computing. We begin by describing the process of extracting parallelism from a computation, followed by a description of different classes of parallelism. We focus in particular on dynamic and irregular parallelism which motivate this work. Finally, we describe models for expressing the data and computation components of a parallel computation.

2.1 Extracting Parallelism

The first step in creating a parallel program is to identify concurrency present in the computation and decompose the computation into units of work, or tasks, that can be processed in parallel. Often, the programmer will partition the computation into the same number of tasks as computation resources. However, when tasks outnumber the processors, we say that the problem has been overdecomposed. Overdecomposition increases the parallel slack, or ratio of tasks to processors, and can be helpful when tasks vary in size and may need to be dynamically scheduled to reduce load imbalance.

Parallelism can be identified with respect to data that is being processed or with respect to the individual operations being performed. Data parallelism occurs when
the same operation is performed over multiple elements in a collection of data items and multiple of these operations can be performed concurrently. Such parallelism can occur at the loop-level (e.g. for arrays) or it can be nested (e.g. for trees and graphs) and additional parallelism may be encountered recursively. Task parallelism is more general and occurs whenever multiple operations can be performed concurrently. In general, data parallelism can also be converted into a task parallel representation (often as a task DAG with data elements as the vertices and data dependence expressed along the edges), but not vice versa.

2.2 Irregular and Dynamic Parallelism

When the amount of parallelism depends on dynamic information or when parallelism is dynamically exposed, the parallelism is referred to as dynamic. One example of such a computation is parallel state space exploration where, given a current state, the number of next states is unknown and depends on characteristics of the problem and the current state. Applications that exhibit dynamic parallelism may be unable to determine the total amount of parallelism that exists for a given problem and may also be unable to determine when work will be uncovered. For these reasons, dynamic parallelism evades static scheduling and requires special techniques to manage and distribute parallelism as it is dynamically uncovered.

In addition to variation in the availability of work, variation can also occur in the amount of time required to process each task. Parallelism with this characteristic is said to be irregular and must be scheduled carefully to avoid an unbalanced workload. The variation in execution time between tasks is referred to as the degree of irregularity. Irregular parallelism is difficult to manage because techniques are needed to
ensure that the work is evenly distributed across the processors. If the execution time of each task is known then work partitioning schemes can be effective at dividing the work among the available processing resources. However for many applications the variability is data dependent or may require significant of work to extract. When the degree of irregularity is unknown or there is not enough information available to perform effective partitioning, a dynamic load balancing solution must be used instead.

2.3 Parallel Programming Models

The method by which parallelism, dependence information, data sharing, and execution semantics are expressed is referred to as the parallel programming model. Many models are possible, from managed systems such as OpenMP [69] that provide a simple annotation language for identifying parallelism to explicit message passing [58] where individual processes communicate through messages. Different parallel programming models have advantages and disadvantages in terms of convenience, performance, portability, and scalability. In addition, different types of parallelism can be expressed and managed more readily in some models than in others. For example, loop level parallelism is convenient to express in OpenMP while recursive parallelism can be more conveniently expressed in Cilk[14].

At a minimum, a parallel programming model must define execution and a data sharing models. The execution model defines how units of the computation are executed and how they will be mapped to underlying resources. The most popular execution model is the Single Program Multiple Data (SPMD) model where multiple copies of the same program are run and each process gets an integer identifier, called
its rank, that it uses to identify itself and communicate with its peers. A popular alternative is the fork-join model used by OpenMP, Cilk, Chapel and many others. In the fork-join model, the program begins with a single logical thread of control and then enters regions where multiple threads are forked to execute work in parallel.

The data model defines how data will be shared by the individual execution contexts that arise due to the execution model. A popular example is the shared memory model where data is coherent and globally visible. Under this model, explicit synchronization is needed to ensure data is valid in the presence of concurrent accesses. An alternative and highly scalable model is message passing. Message passing is often used with the SPMD execution model and in this setting messages all data is private; all data sharing is accomplished through explicit messages.

2.3.1 Partitioned Global Address Space Models

In this work, we wish to develop a parallel programming model that provides efficient and scalable support for irregular and dynamic parallelism on large, distributed memory clusters. A family of parallel programming models that provides a promising possibility for supporting such parallelism is the Partitioned Global Address Space (PGAS) family.

PGAS parallel programming models offer a new technique for parallel programming through the use of a distributed shared global address space. Under a PGAS model each process has access to two address spaces: a local address space and a global address space. The global address space is logically partitioned into regions belonging to each process in the computation and each data element is associated with a distinct process. This partitioning is visible to the programmer and a pointer
Figure 2.1: A partitioned global address space distributed across \( n \) processes. A one-sided \texttt{put} operation is being performed using a low latency rDMA operation supported by many high performance interconnects.

to data in the global address space has the form \( \langle \text{proc}, \text{addr} \rangle \) where \texttt{proc} is the id of the process the data is located on and \texttt{addr} is the address of the data on process \texttt{proc}. Data in the global address space is accessed through \texttt{get} and \texttt{put} operations that copy data between the global and local address spaces. Efficient implementation of one-sided \texttt{get} and \texttt{put} operations on distributed memory clusters is achieved through high performance rDMA networks such as Infiniband or Myrinet.

PGAS models such as Global Arrays [64] provide a library based solution that extends existing languages such as C and Fortran with library-based interface to a global address space. Language extensions such as Unified Parallel C (UPC) [82], Co-Array Fortran (CAF) [65], and Titanium [88] offer convenient built-in support for interacting with shared data. These languages provide shared memory-like programming abstractions that automatically generate any needed communication when the programmer accesses non-local data.
PGAS models differ significantly from page-coherent Distributed Shared Memory (DSM) systems such as Treadmarks [43, 42] and object-coherent DSM systems such as Orca [6]. Under PGAS models, remote data is not operated on directly, instead it must be fetched into a local copy and written back into its location in the global address space. Thus, PGAS models offer no coherence and instead rely on the programmer to maintain data consistency through the memory model and explicit synchronization operations. This approach offers direct control over communication and can yield higher performance, sometimes at the cost of more work on the part of the programmer.

2.3.2 The Asynchronous Gap

The one-sided PGAS data model provides a high degree of flexibility in terms of data placement and accessing shared data. This makes it a good match for applications that exhibit irregularity or sparsity in the data. However, PGAS models use the Single Program Multiple Data (SPMD) view of the computation. In the SPMD execution model, the programmer executes the same program on multiple processes and each copy receives a unique integer rank to identify itself within the computation. Thus, the programmer must explicitly express the computation in terms of a fixed number of processes. However, irregularity and sparsity in the data can lead to irregularity and runtime imbalance in the computation. Mapping such unbalanced computations onto a fixed number of processes is very challenging and can require the development of sophisticated load balancing techniques. We refer to this gap between the flexible, asynchronous PGAS data model and the rigid SPMD computation model as the *asynchronous gap*. 
2.3.3 Task Parallel Execution Model

As described in Section 2.1, breaking a computation down into tasks is the first step in extracting parallelism. In SPMD execution models, the programmer generally gives a static assignment of tasks to processes and each process executes their batch of work. However, such work partitioning methods can be insufficient when expressing dynamic and irregular parallelism.

The task parallel execution model is a popular approach to addressing load imbalance. Under this model, the programmer overdecomposes the problem and creates a collection of tasks to be performed. This collection of tasks can be dynamic, allowing for more tasks to be entered as parallelism is dynamically discovered. Once a collection has been created, the program enters a task parallel execution phase where all tasks are executed and automatic resource management is performed to correct for imbalance in the workload that arises due to irregularity and dynamism in the computation.

As an extension to the SPMD model, the task parallel execution model introduces a layer of abstraction between the units of work and the underlying hardware resources. As we demonstrate in this work, such an abstraction can prove beneficial by exposing opportunities for a runtime system to perform load balancing, data affinity optimizations, and fault tolerance.
CHAPTER 3

The Scioto Task Parallel Programming Model

In this chapter we introduce Scioto: Scalable Collections of Task Objects. Scioto integrates with existing PGAS models and addresses the asynchronous gap by extending PGAS models with task parallel regions where tasks execute in parallel and operate on data in the global address space. Scioto defines a task parallel programming model that is designed for efficiency, performance, scalability, and compatibility with existing PGAS models and applications. We divide the discussion of Scioto into two parts; in this Chapter, we describe the task parallel programming model and in Chapter 4 we discuss its implementation.

3.1 Scioto Programming Model

As discussed in Section 2.1, tasks provide a concise, but powerful abstraction that allows the programmer, compiler, or automated tool to express the units of work in computation. A task parallel programming model defines the data sharing and execution semantics for tasks. In this work, we define a task parallel programming model that interoperates with existing PGAS data models. In particular, we focus on the PGAS model provided by the Global Arrays (GA) toolkit for distributed shared multidimensional arrays [64]. GA is a strong motivator because many computational
chemistry applications use GA, but suffer from poor performance due to the fundamental irregularities present in atomic and quantum mechanical systems. However, Scioto’s programming interface and task model are agnostic to the specific platform providing support for the global data view and have been designed to be compatible with a variety of one-sided and PGAS models.

In the Scioto task parallel programming model, the programmer exposes parallelism by creating task objects to represent work units; adding them to a shared task collection, shown in Figure 3.1; and then collectively processing the task collection in a task parallel region. During a task parallel region, the runtime system takes control of the execution and automatically schedules and executes tasks. The task parallel phase ends when all tasks, including those created dynamically, have completed execution. This model effectively create a layer of virtualization between the computation and the available resources. Thus, when processing a task collection the runtime system has many opportunities to apply automated techniques for improving performance, including load balancing and locality optimizations.

3.2 Task Semantics

In our model task objects (which we will refer to simply as tasks) are independent, transferable units of work that execute within the context of a PGAS environment. Each task is described by its task descriptor which contains task meta-information used by the runtime system as well as user-supplied task arguments. As shown in Figure 3.2, task descriptors are organized as contiguous objects that contain a header where task meta-information is stored and a task body where user-supplied task arguments are stored. The task body is defined to be a contiguous buffer which
the programmer uses to store a task’s arguments. The size of the body is defined by the type of task, or task class. Often the programmer may wish to use a struct to store the task’s arguments as is done with arguments to UNIX Pthreads. Among the arguments, the user can provide immediate data such as numerical constants, portable references to data stored in the global address space, or portable references to common local objects.

When a task is executed, it is provided with a reference to the task collection it is executing on as well as a pointer to its task descriptor. Tasks may gather inputs from their task descriptor, from common local objects, or from shared data in the global space. As a result of its execution, a task may create new subtasks (e.g. continuations) or store results in the shared global space or common local objects. Under the current model, we focus on providing support for independent tasks. Independent tasks are
Figure 3.2: A task descriptor is a contiguous object with a header that contains task meta-data and a body that contains task arguments.

tasks that do not rely on the values produced by, or operations performed by other tasks in order to execute. Once started, these tasks must be able to execute to completion when executed in any order and with any degree of concurrency. Tasks may synchronize through locking or other atomic operations when accessing shared data, however a consistent locking discipline must be adopted that ensures progress and avoids circular waiting that can lead to deadlock.

3.3 Task Input/Output Model

Under Scioto, tasks are permitted to read from and write to shared data. Many different mechanisms for asynchronous data sharing on distributed clusters are available. One-sided communication libraries such as MPI-2 [58], ARMCI [63], SHMEM [8], or GASNET [16] allow processes to expose regions of their address space for remote access. These tools provide primitives for allocating globally accessible regions of memory, performing remote get and put operations, and synchronizing accesses to
shared data. Additionally, some provide wait-free synchronization mechanisms such as atomic swap or atomic increment that allow for high degrees of concurrency. Higher level languages and tools such as UPC [82], CAF [65], or Global Arrays [64] allow the user to create high level shared data structures such as distributed multidimensional arrays and to interact with distributed global data objects through uniform or partitioned mechanisms.

Scioto’s task model is intended to support any of these data sharing mechanisms by allowing the user to place portable references to global data objects in the body of a task. For example, under many one-sided models for communication, a portable shared pointer is the tuple \((\text{process}, \text{address})\) where \(\text{process}\) refers to the process that has the data and \(\text{address}\) is address of the data in \(\text{process}\)’s address space. Some PGAS languages, such as UPC, provide language-level global pointers which can be easily placed within a task body. Under Global Arrays uses, integers are used to provide portable references to arrays and their indices as shown in the example in Figure 3.2.

### 3.3.1 Common Local Objects

The PGAS model provides a convenient abstraction for interacting with globally shared data. However it does not provide a means for handling persistent local data that tasks can interact with. For example, some applications may wish to replicate input data in local storage on each node or to store statistics or data that can be associatively reduced following execution. For this reason we have extended the PGAS memory model with *Common Local Objects* (CLO).
In addition to performing input and output with respect to the global space, tasks are also permitted to access common local objects. Common local objects are local data objects that are common across all processes. In other words, these are data objects where instances of the same type of object (with possibly differing values) are available in the local address space on all processes in the computation. These objects must be collectively registered with Scioto, which provides a portable reference so that no matter where a task executes, it is able to look up the instance of the object that is local. This functionality can be used to enhance performance by locally gathering intermediate results throughout a computation. It is also key for interoperability with MPI because it provides the only mechanism whereby tasks can produce results due to MPI’s lack of global address space.

### 3.4 Task Execution Model

In the task parallel programming model, the programmer first seeds the task collection with an initial set of tasks and then processes the task collection in parallel as shown in Algorithm 3.1. Tasks are processed from the task pool in LIFO order (Last In First Out) according to the work-first principle [14]. For tasks which dynamically create subtasks, this yields a depth-first traversal of the task tree and bounds the space requirements of the task collection as proportional to $O(T_{depth})$.

**Algorithm 3.1** Task parallel execution model.

```plaintext
{ Let $T_p$ be a task collection }
$T_p \leftarrow \{ initial\_tasks \ldots \}$

while $t \leftarrow next\_task(T_p)$ do
  execute\_task($t$)
end while
```
We restrict our task execution model by requiring that all tasks present in the task collection are independent and can be executed concurrently. Child-parent and descendant-ancestor data dependence can be expressed through dynamic task creation. However, once enqueued, tasks must be able to execute to completion without blocking. Tasks are allowed to communicate through data stored in the global address space, however tasks must not wait for results produced by any concurrently executing tasks. In addition, tasks are not permitted to make collective calls. These restrictions allow us to relax fairness constraints on task scheduling to improve performance and make it possible to avoid the need for migrating partially executed tasks when performing load balancing. This model captures the execution style of many significant applications [25, 26] and allows us to optimize for their efficient execution.

An example task tree is shown in Figure 3.3. This example shows a dynamic parallel computation where new sub-tasks have been spawned during the task parallel phase. Parent-child and ancestor-child data flow through the global address space is permitted. However arbitrary task-task dependence is not allowed.
3.5 Multilevel Parallel Task Collections

Each partitioning of a problem into multiple sub-problems that can be processed in parallel is referred to as a level of parallelism. A majority of parallel programs use a single level of parallelism, however many problems do not expose enough parallelism after a single partitioning and multiple levels of parallelism may be used to enable better performance. A common example of multilevel parallelism involves performing a coarse partitioning of the problem into chunks of work and then assigning each chunk to a group of processors or a group of cores on an SMP node. This chunk of work is then divided into smaller units of work that are processed in parallel by the group of processors. One distinct advantage of this strategy is that execution teams can leverage a high degree of locality and can execute algorithms that be challenging to scale to larger processor counts.

Many important applications from computational science and engineering utilize multilevel parallelism. In particular, support for multilevel parallelism in this work has been motivated by collaborative work on a Dynamical Nucleation Theory Monte Carlo (DNTMC) computational chemistry simulation [23]. In such programs the work can be partitioned into multiple tasks and each task is executed in parallel on a group of processors. In addition, many existing parallel solvers and libraries (e.g. PETSc [7], ScaLAPACK [12]) provide multiprocessor parallel routines that programmers wish to use within their tasks. In order for the programmer to get the full benefit of these parallel routines, tasks must be able to execute on a processor group rather than on an individual core.
3.5.1 Multiprocessor Execution Groups

In order to avoid adding complexity associated with dynamic resource management and scheduling, we have integrated support for fixed-size task execution groups into Scioto. In this model, shown in Figure 3.4, the programmer partitions processors into execution groups before entering a task parallel phase. During the task parallel phase, groups remain fixed and tasks are executed in parallel by these groups. We found that this model is representative of the needs of a majority of HPC applications and that it lends itself to an efficient and scalable implementation by avoiding complexity involved in dynamic group formation and scheduling of tasks with varying resource requirements.

During task parallel execution, Scioto designates the rank 0 process from the execution group as the group leader. The leader manages a task task queue for the
entire group and dynamic load balancing is done with respect to the queues present on leader processes. Once the group leader acquires a task for the group it broadcasts the task descriptor to the group and the group collectively executes the task. Collective operations are permitted within the execution group, however collectives between groups are not permitted (collectives are also not permitted when no groups are in use). If the computation reaches quiescence then the master instead broadcast a termination message.

3.6 Scioto Programming Interface

Scioto presents the programmer with a global view of a distributed collection of tasks referred to as a GAS task collection (GTC). This task collection may be physically distributed as shown in Figure 3.5 and each task is assigned an affinity with respect to a particular process. When selecting tasks for execution, tasks with the highest affinity to a given process or execution group are processed first and tasks with lower affinity are given lower priority. When load balancing is performed, low affinity tasks are given the highest priority to be transferred.

Scioto programs begin and end in the SPMD execution model used by Global Arrays and collectively enter into task-parallel regions when processing a task collection. A new task collection is first seeded either in parallel or sequentially with an initial workload. When seeding the task collection, the programmer is able to provide an initial assignment of tasks to processors by specifying task affinities. Initial task placements can be especially useful when integrating Scioto with existing task partitioning strategies.
Once the task collection has been seeded, all processors participating in the task collection collectively call `gtc.process()` to enter a task-parallel phase where tasks will be automatically scheduled for parallel execution across all available compute resources. During this stage, tasks may generate subtasks as they are executed. Alternatively, if the programmer would like to rely on their initial task placement scheme, dynamic load balancing can be disabled prior to entering the task parallel region, allowing for an implementation to reduce overheads when load balancing is not needed. Thus, a typical use case of Scioto is:

1. Create task collection (collective)
2. Calculate initial placements if desired
3. Seed task collection with initial workload
4. Process task collection (collective)
3.6.1 Data Types

Scioto defines a number of portable references that can be dereferenced on any node in the computation. These types can be copied via simple assignment and provided as arguments to a task. The preferred implementation for a Scioto portable reference is an integer.

The type corresponding to a Scioto GAS task collection is gtc_t. A gtc_t is a portable reference to a task collection and can be provided as an argument to a task. This can be useful when using multiple task collections to provide multiple task parallel phases; tasks can be placed into a second task collection while executing the first task collection.

A task collection contains a set of task objects. Every task object has a task class associated with it that defines the properties of that task, including the size of the task object’s body and how the task should be executed. A task class is required when creating a task, so the programmer must register task classes before creating tasks. When registering a task class, the programmer creates a task_class_t which is a portable reference to the task type and is used to when creating instances of that type of task. Because it is a portable reference, it can be include in task arguments. Such functionality is useful when tasks may create subtasks of a different type from the parent.

Scioto’s Common Local Objects (CLO) are also represented using portable references into the CLO Database. These references, called clod_t can be passed as arguments to tasks and used on any process to look up the local copy of the common local object.
3.6.2 Task Implementation

```c
void task_implementation(gtc_t gtc, void *descriptor);
```

A task is implemented by a function with the above signature. This function is registered with Scioto when a task class is created and will be called by the runtime system when a task is executed. When the task is executed, it receives a portable reference to the current task collection as well as a local pointer to the task descriptor that contains the task’s arguments. The task collection reference is passed to each task when it is executed and can be used for creating subtasks or interacting with the runtime system.

Every task is represented as a task descriptor object, shown in Figure 3.2. A task descriptor is an opaque object that contains a header with information used by Scioto and a body which contains user-supplied task arguments. The header of every task descriptor contains information such as affinity that can be used to schedule that task as well as information about the task’s class that is needed to execute the task. The task body contains arguments the task will use when it is executed.

The preferred implementation for task descriptors is as a contiguous object; this eliminates the need for marshalling when transferring the task. Thus, a task body is defined as a contiguous buffer that stores task arguments. The expected use of this space is to store a struct which contains the arguments to a task. An example of such a struct is shown in Figure 3.2.

3.6.3 Core Routines

```c
gtc_t gtc_create(int max_body_size, int max_tasks)
void  gtc_destroy(gtc_t gtc)
void  gtc_reset(gtc_t gtc)
void  gtc_add(gtc_t gtc, task_t *t, int patch)
void  gtc_process(gtc_t gtc)
```
typedef struct {
    /* Opaque header */
    task_class_t task_class;
    int created_by;

    /* Task Body */
    char body[0];
} task_t;

typedef struct {
    /* References to GAs */
    int A, B; // Input GAs
    int C; // Output GA

    /* Indices of C block to be produced. C = A*B */
    int block[3];
} mm_task;

Figure 3.6: Task descriptor implementation that incorporates a task body of user-defined size (left). An example user-defined task body for a blocked matrix-matrix multiplication task (right).

A task collection is first created by collectively calling `gtc_create()`. In order to achieve high efficiency, Scioto relies on pre-allocating space for the task collection. In order to facilitate this, the user provides as arguments the maximum size of each task (in bytes) and the maximum number of tasks that the collection must be able to hold. As discussed in Section 3.2, tasks contain an opaque user-defined body and Scioto allows for multiple different types of tasks to exist in the same queue. Thus, the user must inform the task collection of the largest task descriptors that they intend to use with each collection (through the `task_sz` parameter) in order to allocate appropriate storage. As an alternative, the user can specify a value of `AUTOMATIC` which will inform Scioto to automatically determine this value based on the task classes that have been previously registered. When a task collection is no longer needed, it can be destroyed by calling `gtc_destroy()`. If the user wishes to use a task collection multiple times, it must be reset between each reuse by calling `gtc_reset()`.

Tasks are added to the task collection by calling `gtc_add` and providing the process on which to add the task. When using execution groups, the rank of the group that
should receive the task is used instead of an individual process' rank. In addition, the user can specify AUTOMATIC to allow Scioto to automatically determine where the task should be placed. Tasks are added with copy in/out semantics. Thus, when the call to gtc.add() returns, the task buffer is available for reuse. Likewise, tasks are executed on a private copy of the task descriptor allowing this buffer to be reused by the executing task.

After adding the initial tasks to a collection, it is processed via a collective call to gtc.process() which enters a task parallel region where additional tasks may be spawned via calls to gtc.add(). Only one task collection may be processed at a time, but multiple task collections may be added to while one is being processed allowing for phase-based task parallelism. The call to gtc.process() returns collectively when global termination is detected.

3.6.4 Task Object Management

```c
#define task_class_t  gtc_task_class_register(int body_size,
                        void (*cb_execute)(gtc_t gtc, task_t *descriptor))
#define task_t gtc_task_create(task_class_t class)
#define void gtc_task_destroy(task_t *task)
#define void gtc_task_body(task_t *task)
#define void gtc_task_reuse(task_t *task)
```

A Scioto task is represented by its task descriptor object. Every task descriptor corresponds with a specific task class that specifies task properties such as how it is executed and how large (in bytes) the task body size is. Task classes must be collectively registered by calling gtc_task_class_register and providing this information. The result of this registration is a portable reference that can be used to create new task objects via a call to gtc.task.create(). Task classes need only be created once;
they are not bound to a particular instance of a task collection and are not lost when a task collection is destroyed.

A task descriptor contains a user-supplied task body which can be obtained by calling `gtc.task.body()` on a task object. When creating a task, the user must encode the task arguments within this space and when executing a task, the user must retrieve them. If a task object is no longer needed it can be destroyed and it can also be reused to add another task into the task collection if the new task is of the same type as the original task.

### 3.6.5 Execution Group Interface

```c
#include <Scioto/gtc.h>

// Function declarations

void gtc_create_grouped(int max_body_size, int max_tasks, int group);
void gtc_group_get_exec(gtc_t gtc);
void gtc_group_get_default(gtc_t gtc);
void gtc_group_ismaster(gtc_t gtc);
void gtc_group_groupid(gtc_t gtc);
void gtc_group_ngroups(gtc_t gtc);
```

Scioto allows the use of multiprocessor execution groups as described in Section 3.5.1. Logically, groups are always present and when they are not requested by the user, a "self" group containing only the local process is created to provide continuity in the behavior of the programming model. Thus, execution group routines are valid to use even when execution groups have not been explicitly specified by the user.

In order to create a task collection that uses groups, the programmer must call `gtc.create_grouped()` and provide the GA group id of the local process’ execution group. During task collection creation, Scioto will automatically determine group members and create one task collection patch per group. In addition, this call is
collective across the default GA group and this ”world” group is captured for later use by the runtime system.

Scioto provides an interface for querying the multilevel execution environment. In this model each process is identified by the rank in the global process group, rank in their execution group, and the execution group’s id. The rank in the global and execution groups is obtained through normal GA mechanisms. Every task is initially executed in the context of the default GA group. However, if the default processor group’s handle is needed, the programmer can call `gtc_group_default()` to obtain it. The execution group’s handle can be obtained by calling `gtc_group_exec()` and this group must be set to the default GA group before performing any groupwise collectives. The number of groups and the id of group that the current process belongs to can be obtained by calling `gtc_group_ngroups()` and `gtc_group_groupid()` respectively. Finally, the test `gtc_group_ismaster()` can be performed to determine if the current process is the execution group’s master.

3.6.6 Low-level Interface

```c
task_t *gtc_get(gtc_t gtc)
task_t *gtc_get_local(gtc_t gtc)
void gtc_task_execute(gtc_t gtc, task_t *task)
```

If the user wishes to customize the way in which tasks are processed, they can create their own task collection processing function using `gtc_get()` and `gtc_get_local()`. `gtc_get()` is a blocking call that either returns a task or NULL when global termination has been detected. `gtc_get_local()` is a non-blocking call that returns a task if one is available in the local patch of the task collection or NULL otherwise. Once a task has been obtained it can be executed by calling `gtc_task_execute*()`. A simple example task collection processing function is given in Figure 3.7.
void my_gtc_process (gtc_t gtc) {
    task_t * task;
    while (task = gtc_get (gtc)) {
        gtc_task_execute (task);
    }
}

Figure 3.7: A simple task collection processing function implemented using the low level interface.

3.6.7 Example: Matrix-Matrix Multiplication

Figure 3.8 provides a code listing for an example Global Arrays program that performs blocked matrix-matrix multiplication to solve for $C = A \cdot B$. Parallelization is performed at the level of result blocks of the output $C$ matrix and a block of the $C$ matrix is computed by the task function $mm\_task\_fcn()$.

In this example, all processes first collectively register the $mm\_cls$ task class for the $mm\_task\_fcn()$ task callback. This function takes as input a task with an $mm\_task$ body that contains portable references to the input and output arrays (integers under GA) as well as the indices of the blocks to multiply. Next, the task collection is collectively created. After this, an instance of the $mm\_cls$ task class is created on each process. The body of this task is filled in with references to the global arrays being multiplied.

After this, all processes seed the task collection with the multiplication tasks. Each processor creates only the tasks corresponding to patches of the output array that are local by calling the user-defined function $get\_owner()$ and comparing the result with its own process id. After the task has been added, the data in the $task$ buffer has been
copied into the task collection and the buffer is reused by calling `gtc_task_reuse()`.

Finally, the task collection is processed and all tasks are executed.

### 3.7 Conclusion

In this chapter, we have presented the Scioto programming model. Scioto defines a programming model based around collections of task objects that operate on data stored in a global address space. This model extends existing PGAS programming models such as Global Arrays and compliments their flexible, asynchronous data model with a flexible, dynamic view of the computation. Scioto is especially well matched with applications that exhibit irregular and dynamic parallelism. It provides convenient interfaces and mechanisms for expressing dynamic parallelism and provides opportunities for a runtime system to automatically perform load balancing and other optimizations to mitigate imbalance that can occur with these types of parallelism.
typedef struct {
    int A, B, C;
    int block[3];
} mm_task;

void mm_task_fcn(tc_t tc, task_t *task) {
    mm_task *mm = tc_task_body(task);

    // Perform multiplication on the given block
    Multiply_Block(mm->A, mm->B, mm->C, mm->block[0],
                   mm->block[1], mm->block[2]);
}

void main(int argc, char **argv) {
    gtc_t gtc; task_class_t *mm_cls;
    task_t *task; mm_task *mm;
    int A, B, C;

    GA_Initialize();
    // Initialize Global Arrays: A, B, and C
    mm_cls = gtc_task_class_register(sizeof(mm_task), mm_task_fcn);
    tc = gtc_create(sizeof(mm_task), MAX_TASKS);
    task = gtc_task_create(mm_cls);
    mm = gtc_task_body(task);

    mm.A = A;
    mm.B = B;
    mm.C = C;

    for (i=0; i < NUM_BLOCKS; i++)
        for (j=0; j < NUM_BLOCKS; j++)
            for (k=0; k < NUM_BLOCKS; k++)
                if (get_owner(i, j, k) == me) {
                    mm.block[0] = i;
                    mm.block[1] = j;
                    mm.block[2] = k;
                    gtc_add(gtc, me, task);
                    gtc_task_reuse(task);
                }

    gtc_process(gtc);
    gtc_destroy(gtc);
    GA_Terminate();
}

Figure 3.8: Task-parallel blocked matrix-matrix multiplication using Global Arrays.
CHAPTER 4

Scalable Runtime System

In this chapter, we describe an efficient and scalable implementation of Scioto task collections. Our implementation is built on top of the ARMCI low level one-sided communication substrate used by Global Arrays which is used to build a distributed runtime infrastructure that is geared toward high performance on distributed memory clusters. We discuss the overall design, followed by several performance and efficiency optimizations and enhancements to the load balancing scheme needed to achieve scalability on large scale systems.

4.1 Task Collection Design and Implementation

Many implementations of Scioto’s task parallel model and task collection structure are possible. One simple approach is to implement the task collection using a centralized task queue. Centralized approaches like this utilize a work server that provides a central location for storing the set of available tasks. This scheme is convenient and relatively easy to implement for small systems, however it has significant performance problems when scaling to larger machines.

In this work, we focus on building a task parallel system that scales to the large node counts common on current high performance systems. In order to achieve this
performance target, we have designed a distributed task collection runtime system
that stores patches of the task collection over a set of work queues that are distributed
across the processes in the system. The ratio of work queues to processors can be
varied, allowing for a variety schemes with different properties. In this work we focus
on a 1 : 1 scheme where each process or execution group maintains its own work
queue, which allows for efficient local access. We apply the work first principle [14]
which gives good bounds on the space required to store the tasks currently available
in the computation. Under this scheme, all processes perform work with respect to
their own task queue, pushing new tasks on the head of the queue and popping tasks
from the head to get the next task to execute in LIFO order. In addition, low level
performance tuning such as employing the help first [35] strategy or explicit task
placement are also supported.

4.1.1 One-sided Communication

Partitioned Global Address Space (PGAS) programming models provide new op-
portunities for the efficient and scalable implementation of task collections on dis-
tributed memory systems. These programming models provide a global view of phys-
ically distributed data along with mechanisms for performing efficient, one-sided ac-
cess. The ability to perform one-sided access to remote data is especially useful for
the implementation of distributed task collections where load balancing requests are
generated asynchronously in response to local work conditions. By storing distributed
task queues in the global address space, we gain the ability to perform one-sided work
stealing where, in spite of distributed memory, steal operations can proceed without
interrupting a working remote process.
Figure 4.1: The Scioto framework is built on top of ARMCI and interoperates with a variety of PGAS libraries including Global Arrays and Global Trees.

For our implementation, we focus on the PGAS programming model provided by ARMCI, the Aggregate Remote Memory Copy Interface [63]. ARMCI gives the benefit of interoperability with multiple parallel programming models including MPI [57], the industry standard message passing interface, and the Global Arrays toolkit [64] which provides a PGAS model for distributed shared multidimensional arrays. ARMCI maps directly onto remote Direct Memory Access (rDMA) one-sided communication operations provided by many high performance interconnects. In addition, more complex operations, such as atomic operations or non-contiguous transfers, are supported through a data server mechanism that guarantees implicit progress. Thus, no explicit progress is needed and such operations can still make progress even when the remote process is busy performing computation.

ARMCI is a portable, low level PGAS communication library that allows the programmer to allocate regions of memory that are available for remote access using one-sided get and put operations. Under this model, computations on shared data take on the get-compute-put model where data must be first fetched into a local buffer before processing and then copied back into a remote location. In addition, ARMCI
provides portable support for a number of one-sided atomic operations aiding in the
design of distributed data structures that support direct remote access.

4.1.2 Distributed Task Queues

Scioto task collections are implemented by pre-allocating contiguous arrays of task
descriptors in ARMCI shared space on each process. These arrays are treated as a
contiguous circular queues and head and tail indices are maintained to mark the
front and back of the queues. Each queue contains a process’ current pool of available
work and we refer to the collective aggregation of all queues as the task collection.

Processes can push and pop tasks to and from the head of their local queue. Pro-
cesses are also able to manipulate remote queues using ARMCI one-si
ded operations. Because queues are contiguous arrays, several tasks can be simultaneously pushed
onto or popped off of a remote queue using a single one-sided communication opera-
tion. During an operation on a remote queue, the queue must be locked to prevent
updates from colliding. This synchronization leads a reduction in concurrency that
can adversely affect the performance of the local process as it can end up waiting for
thousands of cycles while a remote process manipulates its queue. In Section 4.4 we
describe techniques to reduce overheads of interacting with the distributed queues.

4.2 Dynamic Load Balancing

As the computation progresses, processes may exhaust the work available in their
work queue. When this happens, dynamic load balancing must be performed in order
to obtain more work. Scioto uses a locality-awareness enhanced version of the work
stealing [14] dynamic load balancing algorithm. Under work stealing, processes that
have exhausted their local work must search among their peers for surplus work. This
4.2.1 Work Stealing

Under work stealing, each process maintains a double-ended work queue, or *deque*, of tasks. Processes execute tasks from the head of their deque and when no work is available they steal tasks from the tail of another process’ deque. The process that initiates the *steal* operation is referred to as the *thief* and the process that is...
targeted by the steal is referred to as the \textit{victim}. Because the thief is responsible for initiating load balancing requests, work stealing is a \textit{receiver-initiated} load balancing algorithm and the total volume of load balancing operations performed will therefore be proportional to the load imbalance. In addition, given an appropriate work division scheme, it has been proven that the load imbalance under work stealing is bounded making it a \textit{stable} load balancing algorithm [56, 11].

A one-sided version of the work stealing algorithm is given in Algorithm 4.1. When performing a steal operation, the thief must first select its victim. Many schemes are possible, however random victim selection has been proven to be optimal [14]. Once a victim has been selected, the thief must then fetch the metadata for the victim’s work queue to determine if they have work available. If they do, the thief locks the victim’s queue and checks the metadata again to ensure the work is still available. If so, it transfers one or more tasks from the tail of the victim’s queue to its own queue. If the victim has no work available, the thief selects a new victim at random and repeats this process until either work is found or global termination is detected.

\subsection{4.3 Termination Detection}

In order to determine when the computation has completed, processes must actively detect when all processes are idle and no more work is available. The process of detecting this stable global state is referred to as \textit{termination detection}. Many schemes are possible, ranging from centralized schemes using shared counters and termination detection servers to fully distributed schemes.
Algorithm 4.1 Work stealing algorithm using one-sided communication.

\[
\text{while } \neg \text{have\_work()} \land \neg \text{terminated} \text{ do}
\]
\[
v \leftarrow \text{select\_victim()}
\]
\[
m \leftarrow \text{get}(v.\text{metadata})
\]
\[
\text{if } \text{work\_available}(m) \text{ then}
\]
\[
\text{Check for work again under lock}
\]
\[
\text{lock}(v)
\]
\[
m \leftarrow \text{get}(v.\text{metadata})
\]

\[
\text{Perform steal operation}
\]
\[
\text{if } \text{work\_available}(m) \text{ then}
\]
\[
w \leftarrow \text{reserve\_work}(m)
\]
\[
m \leftarrow m - w
\]
\[
\text{put}(m, v.\text{metadata})
\]
\[
\text{queue} \leftarrow \text{get}(w, v.\text{queue})
\]
\[
\text{end if}
\]
\[
\text{unlock}(v)
\]
\[
\text{end if}
\]
\[
\text{if } \neg \text{have\_work()} \text{ then}
\]
\[
\text{lock}(\text{self})
\]

\[
\text{Try to vote for termination}
\]
\[
\text{terminated} \leftarrow \text{try\_vote()}
\]
\[
\text{unlock}(\text{self})
\]
\[
\text{end if}
\]
\[
\text{end while}
\]
In the context of Scioto, termination occurs when all processes become idle and when no load balancing operations are in progress. In order to detect this, we have implemented a wave-based algorithm similar to that proposed by Francez and Rodeh[30]. In this algorithm, a binary spanning tree is mapped onto the process space and a token wave is passed down and up the tree. The token is initially located at the root of the tree and is split at each node as it is passed down the tree. Once the token wave reaches the leaves, it reverses direction and as nodes become passive and have received their children’s tokens, they combine tokens and pass the result up the tree. Tokens are initially colored white, however if a node receives a black token from one of its children or has performed a load balancing operation since the last down-wave, it must color its token black to signal a re-vote. Token coloring is necessary to ensure that early termination is not detected when a passive process that has already passed a white token up to its parent transitions from passive to active.

Using this scheme, active processes do not forward termination messages until they run out of work, keeping overheads low. In the average case, termination is detected in $O(\log N)$ time steps, where $N$ is the number of processes.

### 4.3.1 Token Coloring Optimization

Coloring the token black results in additional termination detection waves and, due to one-sided stealing operations, also requires an extra communication operation between the thief and the victim process. This communication is necessary to mark the victim as dirty, informing them that they must color their token black to avoid early termination. However, marking the victim as dirty is not necessary under certain conditions, allowing for a reduction in the number of messages.
For the purpose of discussion, we establish a votes-before relation where \( p_i \rightarrow p_j \) indicates that \( p_i \) casts its vote before \( p_j \). In the case of a binary spanning tree, this means that \( p_i \) is a descendant of \( p_j \).

**Optimization:** The victim, \( p_v \), of a steal operation only needs to be marked as dirty if the thief, \( p_t \), has already voted and \( \neg (p_v \rightarrow p_t) \).

**Proof:** The purpose of \( p_t \) marking \( p_v \) dirty is for \( p_t \) to retract its vote. If \( p_v \rightarrow p_t \) then if \( p_t \) has voted, \( p_v \) must also have already voted and marking \( p_v \) dirty will have no effect.

### 4.4 Improving Task Queue Performance and Efficiency

In this section we present several optimizations to the distributed task collection implementation. We first present split task queues which remove locking from critical path local enqueue and dequeue operations. In addition, compared with maintaining two queues, splitting the queue allows for zero-copy movement of tasks between shared and private queues. We next discuss mechanisms to reduce overheads associated with task creation. Finally, we discuss adjustments to the ARMCI runtime system that can help reduce the latency of steal operations leading to better system utilization.

#### 4.4.1 Split Task Queues

It is essential that the task queue provide efficient local access, as enqueue and dequeue operations are executed on the critical path. A fully shared task queue would require expensive locking for every access to the task queue. This locking can be eliminated by using two queues per process: one for storing a portion of the work reserved by the local process and a second queue to store shared work. However,
Figure 4.3: Split Queue: Allows lock-free local access to the private portion of the queue and provides zero-copy transfers between public and private portions of the queue.

whenever work is moved between the local and shared queues a memory copy must be performed. This memory copy can be eliminated by using a single task queue that is split into private and public portions as described in [25].

The split task queue is shown in Figure 4.3. This queue splits a single shared queue into a local access only portion and a shared portion. The local portion is located between the head and split pointers and the shared portion is located between the tail and split pointers. The local portion of the queue can be accessed by the local process without locking and the shared portion can be accessed by any process and accesses are synchronized via a lock. The local process only needs to take the lock when adjusting the distribution of work between the public and private portions of the queue.

When using the split queue, the local process must periodically check to ensure that it is exposing enough surplus work in the public portion of its queue by moving the split toward the head of the queue. We refer to the operation of advancing the split toward the head of the queue as a release operation. Releases must be performed periodically to ensure that enough work is available for idle processes to steal. Likewise, when the local process has exhausted the work in the private portion
of its queue, it performs a *reacquire operation* to move the split toward the tail of the queue and reclaim public work into the private portion of the queue.

### 4.4.2 Lockless Release Operations

A further improvement to the split queue optimization can be achieved by carefully organizing the queue metadata. If the queue metadata is represented \{\text{tail}, \text{split}, \text{nlocal}\} then we can remove the need to lock when performing release operations. Using this new metadata, the size of the public portion of the queue and the head pointer are calculated on demand through simple arithmetic rather than stored.

With this metadata scheme, only updates to the \text{tail} pointer need to be synchronized, allowing data to be moved from the private portion of the queue to the public portion of the queue without locking. Because a release operation modifies the \text{split} and \text{nlocal} metadata and a steal operation modifies only the \text{tail} metadata, this scheme makes it possible to support a release operation concurrently with a steal operation. If the process performing the steal operation sees the old value for \text{split}, it simply appears that less work is available than is actually present in the public portion of the queue, which does not impact correctness. Thus, this metadata scheme enables *lockless release operations* to be performed.

### 4.4.3 Efficient Task Creation

The overhead of task creation is an important factor affecting task pool performance as many applications perform dynamic task creation on their critical path. As shown in Table 5.1, the UTS benchmark has a very short average task length but generates very large numbers of tasks. For applications that exhibit such characteristics, the efficiency of task creation can have a significant impact on performance.
The process of task creation involves three steps:

1. Create a task descriptor
2. Populate the descriptor with task arguments
3. Enqueue the task

In step 1, a temporary buffer is allocated to hold the task descriptor. In step 2, data is copied into the task descriptor, and in step 3 the data is copied from the user’s buffer into the task queue.

Since task descriptor buffers are generally short lived, a simple optimization is to recycle buffers to eliminate calls to the memory allocator in step 1. Furthermore, if the programmer knows that the head of the queue does not need to be accessed while the task is being created, further optimization is possible by allocating the new task descriptor directly from the head of the queue. We call this optimization *in-place task creation* and it eliminates both the memory allocation in step 1 and the copy operation in step 3.

## 4.4.4 ARMCI Runtime System, Latency and Overhead

On many platforms, ARMCI’s runtime system spawns one data server thread per SMP node to efficiently support primitives such as accumulate and strided/vector RDMA, and to guarantee progress for one-sided operations. Additionally, the data server has the job of ensuring portable support for the variety of atomic operations provided by ARMCI. The default behavior of the data server thread is to block on an incoming message and yield its processor to the application. However, we found that this mode of operation introduced long latencies for the atomic operations required to
implement work stealing. In this mode, when thieves perform a steal they must not
only wait for the operating system to schedule the data server thread, but also for the
data server to receive a signal from the networking layer indicating that a message
has arrived. These combined overheads often caused an operation that should take
tens of microseconds to take tens of milliseconds or more, depending on how heavily
loaded the remote node was. Due to these overheads and because the data server
is responsible for servicing requests for the entire SMP node, we found that with an
unpinned data server even small amounts of contention could result in very long wait
times.

In order to eliminate these overheads, ARMCI provides an alternative mode of
operation where the data server thread remains bound to a core and actively polls
the network for incoming requests. This eliminates latencies due to OS scheduler
noise and blocking on a signal from the networking layer. On our test system which
is configured with eight cores per node, we found that dedicating a core to the data
server is essential to lowering the latency of steal operations, tolerating contention,
and achieving scalability.

4.5 Improving the Scalability of Work Stealing

Work stealing is a fully distributed algorithm with no central bottlenecks which
makes it a very attractive algorithm for performing dynamic load balancing on large
processor counts. However, two significant challenges can impede this scaling: the
time to find work (number of hops) will increase with system size and the amount of
contention to access shared queues will increase with system size. In particular, it is
possible for the steal latency to grow linearly in system size if the work distribution
strategy is not carefully chosen. In addition, when a limited number of work sources are available pileups can occur leading to high degrees of contention on shared queues.

In this section we describe techniques to address these two concerns. We first describe a work splitting strategy that aims to rapidly disperse work and reduces the average steal latency to $O(\log N)$ where $N$ is the number of processors. Next, we present a new locking strategy that uses out-of-band resource information to detect contention and abort long waits on highly contended or exhausted resources.

4.5.1 Work Splitting

When performing a steal operation, a thief must determine how much work to steal. In the general work stealing algorithm, the thief attempts to balance the load evenly between itself and the victim [11, 49, 56]. Work stealing load balancers that focus on recursive parallelism, such as Cilk’s runtime system, take the approach of stealing the task from the tail of the victim’s queue. These systems reason that since the task stolen was at the tail of the queue it is likely to be of the largest granularity and roughly half of the victim’s work [14]. For Cilk-style applications this approach yields good performance, however for non-strict computations this strategy can yield an unstable load imbalance [11].

Many applications of interest do not follow a recursive parallel style or yield tasks where the amount of work present in each task varies widely. In our system, we wish to apply a strategy that will apply to a broad class of computations so we must not rely on a particular structure or task granularity. In addition we must satisfy the stability constraint given by Berenbrink et al in [11]. This constraint states that given a victim with $l$ work and a work splitting function $f$: 

44
\[ f = \omega(1) \]  

(4.1)

and

\[ 0 \leq f(l) \leq l/2 \]  

(4.2)

In our system we use the work splitting function, \( f \), that selects half of the tasks on the victim’s public queue for stealing. This approach satisfies the stability constraint for arbitrary \( l \) and also allows us to maximize the number of work sources in the computation. In the common case, this strategy leaves the victim with half of the work still available in its public queue and provides the thief with enough work so that it too can put surplus tasks in its public queue. By maximizing the number of work sources, we aim to enhance scalability by decreasing the average time required by any process to locate and steal work.

As shown in Figure 4.4, this effectively reduces the number of time steps to \( O(\log N) \) where \( N \) is the number of processors. At time step one it is likely that one process will randomly select the process with work. At time step two it is likely that two processes choosing their victim at random will select victim with work and perform a successful steal. Because each steal transfers half the work present at the victim (and assuming sufficient parallel slack) the number of work sources effectively doubles after each step. This doubling of the number of work sources after every steal operation allows the steal half strategy to effectively reduce search time to a logarithmic trend in the number of processors.
4.5.2 Contention Management

The performance and scalability of work stealing depends heavily on the cost of ensuring mutually exclusive access to the shared portion of each process’ queue and the performance of this mechanism under contention. In particular, the amount of contention to access work queues will increase with system size and when a limited number of work sources are available, pileups can occur that may greatly hamper performance. In this section we discuss two conventional locking techniques and how they are not able to effectively manage this contention. We then discuss a resource-aware extension to spinlocks that allows a thief to abort highly contended or exhausted resources. An experimental evaluation of these techniques is presented in Section 5.4.2.

ARMCI Locks

ARMCI provides a locking mechanism that uses a simplified version of the bakery (aka deli counter) algorithm for mutual exclusion [52]. It uses two shared global counters: next_ticket and current_ticket. The locking mechanism is analogous to the
ticket scheme used in many bakeries: new customers take a ticket when they arrive in the bakery and they are waited on when the number on their ticket comes up.

Under this scheme, when a process executes a lock operation, it first performs an atomic fetch-and-increment operation on \textit{next\_ticket} and stores the value fetched in \textit{t}. It then enters a loop where it waits until \textit{current\_ticket} = \textit{t}. Once \textit{current\_ticket} = \textit{t}, the process may enter its critical section. When the process has finished executing its critical section, it increments \textit{next\_ticket}, allowing the next process in line to enter its critical section. This algorithm is considered fair because it ensures FIFO access to the resource guarded by the lock.

\begin{algorithm}
\caption{Bakery locking algorithm \textit{lock()} operation using atomic increment}
\begin{algorithmic}
\STATE Let $l_v$ be a lock on victim \textit{v}, where $l_v.next$ is the next available ticket and $l_v.cur$ is the ticket that currently has access to the shared resource.
\STATE \vspace{1em}
\STATE \texttt{ticket} $\leftarrow$ \texttt{atomic\_increment}(\texttt{l}._v\texttt{.next})
\WHILE {$l_v.cur \neq \texttt{ticket}$}
\STATE \{ Perform linear backoff to avoid flooding the victim with lock traffic \}
\ENDWHILE
\RETURN \texttt{SUCCESS}
\end{algorithmic}
\end{algorithm}

**Spinlocks**

An alternative locking scheme is to create spinlocks using ARMCI's atomic swap operation, as shown in Algorithm 4.3. Spinlocks require a single communication operation to obtain a lock, whereas ARMCI's locks require an atomic increment followed by a get operation. Thus, if the lock is uncontended, spinlocks may provide a more efficient locking mechanism.
In comparison with ARMCI’s locks which ensure fair, FIFO access, spinlocks don’t guarantee access and can lead to starvation. In addition, spinlocks spin using an atomic operation that must be implemented using ARMCI’s data server, whereas ARMCI locks spin using a one-sided get() operation that is supported by RDMA interconnects. Hence spinlocks may also perform worse under contention.

Algorithm 4.3 Spinlock lock() operation using atomic swap

{ Let \( l_v \) be a lock on victim \( v \) }

\[
oldval \leftarrow \text{LOCKED}
\]

while \( oldval \neq \text{UNLOCKED} \) do

atomic\_swap(oldval, \( l_v \))

if \( oldval \neq \text{UNLOCKED} \) then

{ Perform linear backoff to avoid flooding the victim with lock traffic }

end if

end while

return SUCCESS

Aborting Spinlocks

ARMCI’s fair locks offer better performance under contention and are starvation free. However, in the case of work stealing they can cause a process to wait to steal from a queue that has been emptied by processes ahead of it in line. Because the process attempting the lock has already acquired a resource by taking a ticket, it must wait for its turn to release the resource even if it realizes that the queue it is waiting on has been emptied.

Spinlocks, on the other hand, allow the process attempting the lock to abort at any time. This provides the process the ability to abort a steal operation if it
determines that the victim no longer has any work available. While waiting on the
lock for a remote queue, the thief can periodically fetch the remote queue’s metadata
to determine if it still has work available. If no more work is available, the thief
can simply abort the lock operation and select a different victim. In comparison,
when using ARMCI locks, even if the thief determines that the victim has no more
work available, it must continue to wait until its ticket is up so it can increment
the current_ticket counter, signaling the next process in line that it may access the
victim’s queue. This requires waiting $O(T)$ steps, where $T$ is the number of thieves
in line, for each thief to release the lock.

Algorithm 4.4 Aborting spinlock lock() operation using atomic swap

{ Let $l_v$ be a lock on victim $v$ and
  resource_exists() be a query function to determine if the desired resource is still available.}

oldval ← LOCKED
while oldval $\neq$ UNLOCKED do
  atomic_swap(oldval, $l_v$)
  if $\neg$resource_exists() then
    return FAIL
  else
    if oldval $\neq$ UNLOCKED then
      { Perform linear backoff to avoid flooding the victim with lock traffic }
    end if
  end if
end while

return SUCCESS
4.6 Conclusion

In this chapter we presented a scalable runtime system that supports the Scioto task parallel programming model. This runtime system was constructed by associating work queues with each process or execution group. Queues are accessed using efficient one-sided communication operations provided by the ARMCI communication library that exposes the performance benefits of modern rDMA interconnects. Several significant enhancements to the design of the shared work queues and load balancing algorithm were presented to improve overheads and scalability.
CHAPTER 5

Experimental Evaluation

In this chapter, we evaluate the performance and scalability of the Scioto system through three benchmarks that have been selected because they exhibit dynamic parallelism with a high degree of imbalance. We first describe the benchmarks and the characteristics that make them a good match for the Scioto model. Next, we evaluate the performance impact of several performance enhancements presented in Section 4.4. Finally, we evaluate the impact of scalability optimizations presented in Section 4.5 and demonstrate strong scaling for each benchmark on 8,192 processor cores.

Experimental results presented in this section were conducted on a 2,310 node Hewlett Packard cluster located at Pacific Northwest National Laboratory. Each node is configured with two quad-core AMD Barcelona processors running at 2.2 GHz for a total of 8 cores per node and 18,480 cores in the full system. The high speed interconnection network is Infiniband which supports the low-latency rDMA communication needed for efficient execution of PGAS models. Nodes in the system run the GNU/Linux operating system and the message passing layer used for process management was HP-MPI.
5.1 Benchmarks

We have selected three benchmarks to evaluate the Scioto runtime system: MADNESS, UTS, and BPC. All of these benchmarks start from a single task and spawn new tasks dynamically. MADNESS is a real application kernel from the computational chemistry domain, UTS is a combinatorial optimization and state space exploration benchmark, and BPC is a new synthetic benchmark designed to stress test dynamic load balancing systems.

5.1.1 MADNESS Tree Creation Kernel

The MADNESS (Multiresolution ADaptive NumErical Scientific Simulation) project is a collaborative effort to develop a framework for scientific simulation using adaptive multiresolution analysis methods in multiwavelet bases [87]. The first step in every MADNESS execution is to project a user-supplied analytic function into its numerical representation through adaptive spatial decomposition. For our kernel we consider functions in 3-dimensions, resulting in an oct-tree based numerical representation of the analytic function. This sparse, multiresolution representation allows the computation to spend more effort in areas with a high degree of information and less time in computing on regions that have less content. In addition, operations such as differentiation and integration become less expensive in the multiwavelet representation.

Projection begins with a fixed 3d volume over which the analytic function is sampled to derive the numeric representation. The accuracy of the representation is then examined and if it is not high enough, the 3d volume of space is split into 8 subspaces and each subspace is recursively processed until the desired accuracy has been reached. We cast each call projection as a task. Thus, the computation
starts with a single task and each time greater accuracy is needed, eight new tasks are spawned. The size and shape of the resulting task tree is highly dependent on the user-supplied analytic function making a MADNESS workload very challenging to statically partition and a good candidate for dynamic load balancing.

5.1.2 The Unbalanced Tree Search Benchmark

The Unbalanced Tree Search benchmark (UTS) [27, 28, 67, 68] models the behavior of state space exploration and combinatorial search problems by measuring the performance achieved when performing an exhaustive parallel depth-first search on a parametrized, highly unbalanced tree. The UTS tree traversal starts with a single task and proceeds in nested parallel style to generate millions of tasks, one for each node in the tree. The tree is constructed using a splittable, deterministic random stream generated using the SHA-1 secure hash algorithm. Each node is represented by a 20-byte SHA-1 digest and its children are found by applying the SHA-1 algorithm to the parent node’s digest combined with the child id. There is a high degree of variation in the size of each subtree rooted at any given node in a UTS tree. Thus, if each node is taken as a task in a UTS execution there is a high degree of variation in the amount of work contained within each task. These properties make UTS a challenging problem that is effective for evaluating dynamic load balancing schemes.

UTS provides several different types of trees that can be used to model different classes of problems. Examples of two of these trees are shown in Figure 5.1. The geometric tree models combinatorial search problems where there is a depth limit corresponding to the number of variables. In this tree, the expected number of children for a given node is determined taking the branching factor $b$ provided by the user and
applying a geometric distribution in the depth of the node. An example binomial tree is also shown in this figure. Binomial trees are not depth limited and are representative of state space exploration problems that are unbounded. The expected number of children for any given node is determined using a branching factor $b$ given by the user. A node will have either $b$ nodes with probability $q$ given by the user or 0 nodes with probability $1 - q$. When $q \cdot m < 1$ this process generates a finite size tree.

Implementations of UTS are available in most popular parallel programming models, including MPI, UPC, OpenMP, PThreads, Shmem, Cray XMT, and others. In previous work [25] we have compared the performance of the Scioto and MPI implementations of UTS and found that both scale to 512 processors. However, Scioto outperforms the MPI implementation by a margin of 5-10% because Scioto’s implementation has lower overheads compared with the MPI implementation’s explicit polling-driven load balancing over two-sided messaging.
5.1.3 Bouncing Producer-Consumer Benchmark

The Bouncing Producer-Consumer (BPC) benchmark is a producer-consumer benchmark with a twist: the producer process may change as the result of dynamic load balancing. This benchmark is intended to create a scenario where locating work is challenging due to the migrating producer.

BPC is a task-parallel benchmark that dynamically produces two kinds of tasks: producer tasks and consumer tasks. In the example task tree shown in Figure 5.2 producer tasks are shown in black and consumer tasks are shown in white. Each producer task produces one producer task followed by $n$ consumer tasks. The corresponding dequeue has the producer task at its tail making it the first task that will be stolen. When applying the steal half work splitting strategy, this task could bounce across several victims before it is executes and the process that ultimately execute it will become the new producer. Each task executes for $t$ time and this process is repeated until a maximum depth $d$ is reached.
BPC is a dynamic parallel application that starts from a single producer task and generates new tasks to expose parallelism. This benchmark is intended to stress the ability of the dynamic load balancing system to locate and disperse work. In addition, given that the number of sources for dynamic parallelism is severely limited in BPC, it also tests the ability of the system to tolerate high amounts of contention to access the producers’ queues.

5.2 Termination Detection Latency

In Section 4.3 we described a scalable tree-based termination detection algorithm. An experimental comparison of the latency of this algorithm with the time required to perform MPI and ARMCI barriers is presented in Figure 5.3. This figure presents the average time per operation over 500 operations on up to 1024 processor cores. A bump in latency occurs after 256 cores due to the fat tree topology of the system.

From this data, we see that the time require to perform termination detection is comparable with the highly tuned MPI and ARMCI barrier implementations. However, a significant performance gap is present because our implementation is not topology aware and does not leverage the low latency intranode communication available on the 8 core nodes in the system. This gap is much more significant on low core counts where the ratio of intranode to internode links is higher and could be corrected if the layout of the spanning tree were made topology aware.

This similarity in performance is due to the similarities between termination detection (i.e. global consensus) and a barrier synchronization. At a high level, termination detection performs performs a notify (vote) followed by a wait (broadcast of the result). However, in comparison with barrier operations which are a single collective
synchronization, termination detection is a non-collective consensus mechanism that can require multiple iterations before global consensus is reached. In addition, the vote operation is by design non-blocking and multiple vote attempts are needed to make progress on the vote. As shown in Algorithm 5.1, repeated voting is needed to simulate a barrier using termination detection.

**Algorithm 5.1** Performing barrier synchronization using the termination detection algorithm.

```
repeat
  result = attempt_vote()
until result = PASS
```
5.3 Task Queue Optimizations

In Section 4.4 we presented several optimizations to the task queues used by the Scioto runtime system. We now examine the performance of these optimizations.

5.3.1 Split Queues

Figure 5.4 compares the performance of the split queue with lockless release against the performance of a shared queue that requires locking for all accesses to the queue. In this experiment we ran the bouncing producer consumer benchmark with producers that produce $n = 64$ tasks that each perform 1 msec of work and a total depth $d = 32,768$. At low processor counts, the parallel slack is relatively high and contention remains low. However, as the processor count is increased, producers have to contend with thieves to acquire the lock for their queue and enqueue new tasks. Performance begins to degrade past 128 processors as the time required to enqueue new tasks grows due to lock contention.

5.3.2 In-Place Task Creation

This optimization is intended to reduce the overhead involved in task creation. In this experiment we compare the performance of three schemes: baseline, buffer recycling, and in-place task creation. In the baseline scheme, a new task object is created, populated with task arguments, copied into the queue, and then freed. Under the recycling scheme a buffer is reused when adding tasks, removing allocation and free steps. In-place task creation removes these steps as well as the copy operation. Thus, only population is performed.
We compare the performance of these three schemes in Figure 5.5. Adding buffer recycling provides a significant benefit over the baseline approach for very small tasks. However, for larger tasks the cost of allocation is amortized by the cost of performing the copy operations. In-place task creation provides roughly a 15% performance improvement, which may seem smaller than expected, as it eliminates both the memory allocation and a copy. The reason that the gain from eliminating the second copy operation is not larger is that the data copy performed in step 3 is less expensive than the data copy performed in step 2, due to cache effects - the copy in step 2 brings the data into cache and the copy in step 3 is performed with the input data already in cache.
Figure 5.5: Task creation times for baseline, buffer recycling, and in-place task creation schemes.

5.4 Work Stealing Optimizations

In this section we evaluate the performance impact of the work stealing optimizations presented in Section 4.5. We first show the importance of the work splitting strategy and then demonstrate that with the addition of the aborting steals locking algorithm our system scales to 8,192 processor cores.

5.4.1 Work Splitting: Steal Half

We evaluate the performance of this scheme in Figure 5.6 in the context of the Unbalanced Tree Search benchmark. In this figure, we present the performance of UTS performing parallel exploration of a highly unbalanced tree for three schemes: steal-half, steal-1, and steal-2. UTS generates tasks recursively and thus has a workload where the steal-\(n\) strategy is stable. We see that all schemes scale well to 64
Figure 5.6: Impact of steal-half over steal-1 (Cilk-style) and steal-2.

processors, however as the processor count is increased, work becomes harder to find and the steal-1 scheme loses performance as most processors are idle searching for work. Stealing two tasks during each steal operation helps to better distribute the work, but it still does not scale past 128 processors. Performing steal-half gives the best performance, yielding over 95% efficiency on 256 processors. For this experiment, steal-half transferred an overall average of 3.8 tasks per load balancing operation.

5.4.2 Contention Management: Aborting Steals

We now combine all previous optimizations and evaluate the Scioto system at scale. In previous experiments, we demonstrated several techniques that are needed to achieve scaling on hundreds of processors. We now observe that additional contention management is needed at scale to disperse pileups and demonstrate that the aborting
steals technique is an effective mechanism to enable work stealing on thousands of processor cores.

**Benchmark Workloads**

In this section we conduct strong scaling experiments to evaluate the scalability of our system. For these experiments, the Madness kernel performed tree creation on a 128 element soft body system; the Unbalanced Tree Search benchmark performed exhaustive search on an unbalanced 270,751,679,750 node geometric tree with depth 18; and the Bouncing Producer Consumer (BPC) benchmark was run on a problem where each producer produced $n = 8192$ 10msec tasks with a total depth $d = 4096$. The total number of tasks and average task execution time these each workload is given in Table 5.1.

<table>
<thead>
<tr>
<th></th>
<th>Total Tasks Executed</th>
<th>Avg. Task Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Madness</td>
<td>1,801,353</td>
<td>102.28073 ms</td>
</tr>
<tr>
<td>BPC</td>
<td>33,558,529</td>
<td>9.96720 ms</td>
</tr>
<tr>
<td>UTS</td>
<td>270,751,679,750</td>
<td>0.00066 ms</td>
</tr>
</tbody>
</table>

Table 5.1: Benchmarking workload characteristics.

All three benchmarks start from a single task and dynamically generate subtasks. In the case of Madness, each task corresponds to a 3-dimensional region of space which may be subdivided to get finer precision for the numerical representation where needed. Each adaptive refinement results in 8 new subtasks. In BPC, each producer task produces 1 producer task and 4096 consumer tasks which are leafs and do not produce any subtasks. Finally, UTS was run to produce an unbalanced trees where
the expected number of children per node is geometric in the depth of the tree and proportional to the branching factor, $b_0 = 4$.

**Strong Scaling Performance**

In Figure 5.7 we show strong scaling experiments for each of the schemes discussed in Section 4.5.2. For each benchmark, we show the performance achieved from 512 to 8192 cores. Performance is reported as the total throughput in terms of thousands of nodes visited per second (kNodes/sec) to billions of nodes visited per second (GNodes/sec). We also report the efficiency of each execution as the percent of the total execution time spent executing tasks. This efficiency measurement separates the total execution time into active time, or time when a user task is executing, and idle time, or time spent searching for work. With this data we wish to characterize the load balance and have not subtracted any performance lost due to overheads, such as time lost when enqueuing tasks or balancing the queues. In general these overheads are low since we have created schemes to eliminate locking from the critical path and perform in-place task creation.

The benchmark data presented in Figure 5.7 is organized from left to right in decreasing task granularity with comparable runs of Madness processing tens of thousands of tasks per second; BPC, hundreds of thousands of tasks per second; and UTS, billions of tasks per second. As the number of cores is increased, Madness and BPC expose parallelism more slowly leading to limited parallel slack and creating the potential for long search times and high contention. UTS, in comparison, produces new work very rapidly and has a high degree of parallel slack. However, due to imbalance
Figure 5.7: Performance and percent efficiency for Madness, the BPC benchmark, and UTS on 8,192 cores. Efficiency is calculated as the ratio of time spent executing tasks to total time.
in the search space processes frequently run out of work as the subtree they are exploring terminates. For this reason, UTS requires lightweight load balancing, as the need to perform frequent load balancing can lead to high overheads.

The ARMCI and spin lock schemes scale to 4096 processors, however past this point limited parallel slack leads to contention to steal from the limited number of processes that have surplus work and long waits for thieves that frequently result in no work being stolen. By comparing the performance of ARMCI’s locks with spinlocks, we clearly see that for Madness and BPC ARMCI’s locks offer better performance. However, for UTS which exhibits a high degree of parallel slack, there is relatively little contention and spinlocks outperform ARMCI locks because they require only a single communication operation to take a lock, whereas ARMCI locks require two communication operations.

When contention is high and parallel slack is limited, steal operations performed using the ARMCI and spin lock schemes can often result in a long wait where the victim’s work dries up before all the thieves targeting it get work. In order to address this problem, we have implemented aborting steals which periodically check if the victim still has work available to avoid waiting on a stale queue. We see in Figure 5.7 that this scheme addresses the critical performance problem for Madness and BPC. For Madness, the aborting steals scheme achieves 88% efficiency on 8,192 cores in spite of severely limited parallel slack and for BPC it achieves 97% efficiency. For the UTS benchmark which has a high degree of parallel slack, the aborting steals scheme achieves the best performance with an efficiency of 99% on 8,192 cores. This is because it is built using spinlocks which are less expensive than ARMCI locks when the lock is uncontended.
Dynamic Load Balancing Activity

In Figure 5.8 we present the total number of successful steals performed by all processes and the average acquire latency across all processes for each benchmark. We define *average acquire latency* as the average amount of time spent from when a process exhausts its work until it acquired more work through a successful steal. This data is presented for each of the three locking mechanisms.

By examining the total number of steal operations, we see that MADNESS and UTS both scale linearly in the number of processes and perform on the order of tens of thousands of steal operations. In comparison, for BPC steal operations increase logarithmically in the number of processors and millions of steal operations are performed. This is because the workload run for BPC has a ratio of 8192:1 consumer tasks to producer tasks. Thus, by design, BPC incurs a very high degree of steal activity due to the limited number of producer tasks. The shape of BPC’s task tree results in cyclic bursts in the parallel slack where work is dispersed and contracts until the next producer is executed. Because of the steal-half work splitting strategy, a new producer task will be discovered in a logarithmic number of steals (dividing the work by two at each steal). This results in a logarithmic trend in the number of steals because the rate at which the producers are exposed becomes logarithmic due to the steal half strategy.

In the right column of Figure 5.8 we also plot the average acquire latency. This is the average time across all processors to acquire work after exhausting the work available in the local queue. We see that for BPC and UTS the average acquire latency is in the hundreds of microseconds. However, MADNESS produces only 1.8 million tasks and due to this limited parallel slack the average acquire latency is
Figure 5.8: Total number of successful steals and average acquire latency for Madness, BPC, and UTS.
on the order of tens to hundreds of milliseconds. We see also that contention is a contributing factor and that for ARMCI and Spinlocks the latency is much higher. Using aborting steals helps to drastically reduce performance loss due to long acquire latencies. In addition, we see that the data for ARMCI locks and spinlocks is noisy due to pileups that can occur when using these techniques. Aborting steals help to reduce the average acquire latency by aborting highly contended resources.

Failed and Aborted Steals

We present the total number of failed and aborted steals for each scheme in Figure 5.9. Failed steals are steal operations where the thief has acquired a lock on the victim, only to discover that the victim has no work available. Failed steals are an indication of the amount of contention present when accessing the shared portion of the queue. This contention can arise due to remote access by multiple processes or due to the local process performing reacquire operations concurrently with remote processes attempting to perform a steal.

Aborted steals are steal operations where the thief aborts the current victim without acquiring the lock, because it has determined that the victim has no work available. Aborted steals can occur under all schemes during victim selection when the thief initially fetches the victim’s metadata to determine if it has any work available before attempting to lock the shared portion of the victim’s queue. In the aborting steals scheme, this check is performed periodically even after a thief has decided to follow through with a steal on the current victim. If the thief determines that the victim no longer has work available, it can abort its current steal operation and move on to a different victim. Thus, the number of aborted steals reflects the parallel slack or the general availability of work present in the computation.
Figure 5.9: Total number of aborted and failed steals for Madness, UTS, and the BPC benchmark.
For Madness and BPC, we see that the performance loss past 4,096 processors is due to the large number of failed steals. The ARMCI and Spin locking schemes experienced between 2.5 to 3 billion failed steals compared with 0.25 billion for the Aborting steals scheme. If we look at the number of aborted steals for Madness, we see that the Aborting steals scheme performed twice as many aborts as the ARMCI locking scheme. This indicates that during execution of the Madness benchmark, idle processes experienced long searches for work due to limited parallel slack, which accounts for its 12% loss of efficiency at scale.

For BPC, we see that, surprisingly, the number of aborted steals is similar for all three schemes. This indicates that the primary cause for performance loss is due to a small number of pileups where multiple consumers were all waiting on a single producer. These pileups can be a significant performance problem for the ARMCI and Spin locks schemes because multiple thieves will have to wait for others to acquire the lock on the remote process, fail on the steal operation, and unlock before they themselves can move on to another victim. Thus, allowing consumers to abort when they detect that a producer’s work has been exhausted eliminates a possibly linear time wait to abort the steal on the current victim.

In addition to this, ARMCI and Spin locks both spin on the lock which can cause scalability issues as the networking layer for the process that owns the lock may become saturated by remote lock operations. In the case of ARMCI locks, the remote process issues repeated one-sided \texttt{get()} operations to fetch the \texttt{current\_ticket} counter. On our experimental system, these requests are truly one-sided and are performed using Infiniband RDMA operations. Spin locks perform repeated atomic swap operations which are serviced by ARMCI’s remote data server. From the performance
data, we can see that ARMCI’s locks scale better under contention as the data server becomes more quickly saturated with requests.

If we compare the number of failed steals for ARMCI and Spin locks in Figure 5.9, an interesting trend emerges: in the case of Madness and BPC, spin locks cause an increase in the number of failures, while in the case of UTS they cause a decrease in the number of failures. This phenomenon is due to how spin locks influence the behavior of the reacquire operation. As described in Section 4.4.1, a reacquire operation is performed when a process transfers work from the public portion of its queue to the private portion of its queue. When using ARMCI locks, this means that the local process must take a ticket and wait in line behind any other processes to acquire the lock on the public portion of its own queue. When using spin locks, the local process issues native atomic exchange instructions to acquire its lock. The local process can issue these instructions orders of magnitude more quickly than remote processes can issue atomic swap operations over the network. This effectively prioritizes the local process during a reacquire operation. This allows the local process to jump ahead of the queue and reclaim any shared work, causing many of the processes attempting to steal from it to fail. Thus, the total number of failures incurred when using spin locks is higher for both Madness and BPC.

In comparison with Madness and BPC, the duration of a UTS task is roughly three orders of magnitude shorter. Because of this, UTS also generates tasks orders of magnitude more rapidly that Madness and BPC. For this application, we observe that prioritizing the local process during reacquire operations gives a significant drop in the number of failed steals. This is because the local process is able to quickly
reacquire work from the shared portion of its queue and generate new tasks to satisfy the thieves waiting for the lock on its queue before their steal operation fails.

5.5 Conclusion

We have presented an experimental evaluation of the Scioto runtime system on three benchmarks: MADNESS, BPC, and UTS. We demonstrated that split queues and work splitting are needed to achieve scaling on hundreds of processors. Next, we showed that by combining these techniques with the aborting steals approach for contention management we are able to achieve strong scaling and high efficiency on 8,192 processor cores.
CHAPTER 6

Fault Tolerant Task Parallelism

Leadership class applications continue to pursue more sophisticated scientific and engineering simulations, demanding higher performance. This performance is often achieved through larger, more complex computer systems. Unfortunately, the Mean Time Between Failures (MTBF) for such systems decreases as systems grow in size and complexity. Thus, especially for long-running computations, job failures due to hardware faults are becoming a more frequent problem.

Conventional methods for handling faults involve writing periodic checkpoints to disk either at the application level or at the operating system process level. Upon failure, the computation can be resumed from the most recent checkpoint. Unfortunately, the overhead of taking checkpoints can be large due to the high volume of I/O generated and it grows with problem and system size.

As discussed in Chapter 2, task parallel programming is an effective parallel programming model that relaxes the process centric model for parallel computing by expressing the computation as a set of tasks that can be executed concurrently. In Chapter 5 we demonstrated that task parallelism can be used to enable automatic scheduling and load balancing of the workload and achieve significant performance
benefits. In addition, the layer of virtualization introduced by task parallelism between the units of work and the underlying hardware resources also offers unique opportunities for fault tolerance.

In this chapter we extend the Scioto model by leveraging task parallelism as a vehicle for fault tolerance. Our system adds dynamic, fine-grain tracking of task execution progress and uses this information to enable selective restart when recovering from failures that occur during task parallel regions of a program. This technique improves on conventional checkpoint/restart techniques by incurring a performance penalty for recovery that is proportional to the degree of failure rather than the full system size. In addition, the system can survive an arbitrary number of failures and the cost for maintaining recovery information is low and in-memory.

In Section 6.1 we provide an overview of the task parallel programming model, failure model and fault tolerance framework. In Section 6.2 we present a metadata scheme for fine-grain task completion tracking. In Section 6.3 we present the fault tolerant task execution algorithm and in Section 6.5 we evaluate our system using the Self Consistent Field kernel from the computational chemistry domain.

6.1 Overview

We extend the Scioto task parallel programming model [25] with fault tolerance. In this initial exploration, we place several restriction on the full task parallel programming model to better focus on core fault tolerance issues. This restricted programming model is sufficient to support a number of important computational chemistry applications and will be relaxed in future work.
6.1.1 Global Arrays

Scioto tasks operate on data stored in the global address space provided by Global Arrays (GA) [64]. GA is a Partitioned Global Address Space (PGAS) parallel programming model that provides support for distributed shared multidimensional arrays on distributed memory systems. In the GA model, array data is partitioned into patches that are distributed across all nodes in the computation and are remotely accessible through one-sided operations.

6.1.2 Task Parallel Programming Model

The task parallel programming model provides a globally accessible, but potentially distributed, collection of available tasks that the programmer can add new tasks into and remove tasks from for execution. In general, the job of selecting which task should be extracted by a particular processor is performed by the runtime system. This system is permitted to move tasks around in order to satisfy multiple criteria including load balance and locality. A popular and highly scalable load balancing strategy used by many task parallel systems is work stealing [26].

In this work, we restrict the full Scioto model by separating the operations of populating the task collection with work and processing the tasks in the collection into distinct phases. In particular, we do not permit tasks to add new sub-tasks into the task collection. This restriction allows us to focus on techniques for maintaining consistent data produced by tasks. In future work, we plan to investigate how these techniques for data consistency can also be applied to sub-tasks that are generated as task outputs.
We consider a task model where all input and output data is stored in global arrays. We require that tasks specify their inputs and outputs up front so that they can be managed by the fault tolerant runtime system. In addition, all input data must be read-only and all output data must be write-only. At present we focus on an output mode where tasks accumulate results into an output array using GA atomic accumulate operations. This model is sufficient for a broad class of iterative solvers and enables the system to track output completion of tasks in order to detect corruption and incompleteness due to failure.

### 6.1.3 Failure Model

In this work we focus on node failures and assume a pool of spare nodes that is available to the runtime system. These nodes can be used to replace failed nodes, keeping the total number of nodes in the computation fixed. In addition, we assume a fail-stop mode of failure where failed nodes crash and exit the computation. When a failure occurs, this results in the loss of a memory domain in global address space or, at the GA level, a patch of one or more global arrays.

### 6.1.4 Fault Tolerance Framework

Our fault tolerant task parallel system assumes a fault tolerant runtime stack. It requires fault tolerance from Global Arrays and GA’s communication layer, ARMCi, in order to support respawning failed nodes and re-establishing one-sided communication. In addition, a fault tolerant process management system is needed that can detect failures, respawn failed processes and inform the programming model that a
failure has occurred. These systems are currently under development but not yet complete enough to evaluate our system. Thus, in this evaluation we rely on simulated faults as described in Section 6.5.2.

Fault tolerant task collections are also targeted at supporting fine-grain restart for sections of code that can be expressed using task parallelism. This system works in collaboration with existing coarse-grain checkpoint/restart techniques that are used to protect non-task parallel code sections. We use such a checkpoint/restart system to take a checkpoint prior to entering a task parallel region. When a failure occurs within the task parallel region, only the failed process’ state is restored from checkpoint and it re-connects with the currently executing task collection.

6.2 Tracking Task Execution

When recovering from a failure, we wish to scan the set of tasks to determine which tasks have completed and which tasks are incomplete either because they did not execute fully or because some or all of the data they produced was lost. Tasks are permitted to produce outputs to regions of global arrays that span multiple memory domains. Thus, it is possible for a task that initially executed successfully to become partially or fully incomplete due to data loss from a failure.

In order to successfully detect this situation, we maintain a completion marker for every memory domain that a task writes to. This information makes it possible to detect partially complete tasks and reduce the number of cascading failures due to data corruption. Using the information on which patches of the global address space have been lost, tasks can be re-executed and re-write (i.e. atomic accumulate) outputs only to incomplete or recovered memory domains.
6.2.1 Distributed Task Execution Metadata

Given a set of tasks $T = \{t_1, ..., t_M\}$, a task $T_i$ produces outputs to a set of memory domains, $O_i$. A task can produce at most $N$ outputs, where $N$ is the number of memory domains. Thus, we can organize metadata to track the completion of these outputs by forming $M$, a $M \times N$ matrix where the tasks ids are the rows indices of the matrix and, for task $T_i$, $M_{i,j}$ indicates the state of task $i$ with respect to memory domain $j$.

Each $M_{i,j}$ may contain one of the following values that indicates the state of task $T_i$ with respect to memory domain $j$: $\bot$, started, completed. The initial value of all entries is $M_{i,j} = \bot$, which indicates that task $i$ has not produced an output to domain $j$. Started indicates that an update has been started and completed indicates that the previously started update has completed. During recovery, the system must ensure:

$$\forall T_i \in T, \forall j \in O_i \ M_{i,j} = \text{completed}$$

$$\forall T_i \in T, \forall j \notin O_i \ M_{i,j} = \bot$$

When a value of $M_{i,j} = \text{started}$ is seen during recovery, this indicates that a failure occurred while task $i$ was updating domain $j$ and that domain $j$ has become corrupted as a result of this failure. Thus, domain $j$ must be restored from the checkpoint taken at the start of the task parallel region and row $j$ of $M$ must be reset to $\bot$ to avoid erroneous results. When $M_{i,j} = \bot$ and $j \in O_i$, task $j$ has not completed with respect to output $j$ and should be scheduled to execution to produce this output.
6.2.2 Fault Resistant Metadata Storage

In order to avoid loss of important metadata when failure occurs, the metadata matrix $M$ is distributed across memory domains where domain $j$ stores the data in column $j$ of $M$. Thus, only the metadata corresponding to data stored on domain $j$ is stored on domain $j$. In the event that domain $j$ fails, all updates to $j$ are lost and only the metadata corresponding to $j$’s lost updates is lost. This co-location of metadata with the information that it tracks ensures that, at any given time, the state of all correct nodes is fully described by the available metadata.

6.3 Fault Tolerant Task Execution

In Algorithm 6.1 we present the fault tolerant task execution algorithm. This algorithm ensures all tasks in a task collection execute successfully and that they produce a consistent result. We divide this algorithm into three parts, covered in detail in the following sections: (6.3.1) task collection execution, (6.3.2) data corruption detection, and (6.3.3) completeness detection. At a high level, this algorithm proceeds by executing all unfinished tasks and checking to determine if a failure has occurred. If it has, it uses the metadata to detect data corruption and incomplete tasks. All incomplete tasks are then re-entered into the task collection and processing repeats until execution succeeds with no failures.

6.3.1 Task Collection Processing

Executing all of the tasks, or processing, a task collection requires performing a series of actions defined by each task object. In our task model, we require that all tasks in the task collection can be executed concurrently and prohibit tasks from
Algorithm 6.1 Fault tolerant task collection execution algorithm.

let: me be this process’ rank
let: nproc be the total number of processes
failed ← false
TC ← T
CP ← checkpoint()
repeat
    // Execute all tasks in the task collection
    TC.process()
    failed ← detect_failure()

    if failed = true then
        // Detect data corruption due to failures
        TC.scan()

        // Add all incomplete tasks back to task collection
        TC.check_completion()
    end if
until failed = false

producing sub-tasks. The algorithm for processing the task collection is given in Algorithm 6.2. This algorithm fetches the next available task from the task collection and executes it. When no more tasks are available, all processes line up at a barrier to wait for completion of all tasks before proceeding. If the underlying runtime system requires collective recovery, it can be triggered during this collective step.

We break down task execution into three steps: fetching task inputs, performing task execution, and writing task outputs. During execution, a task is not permitted to produce any outputs into the global address space. Instead, this is handled by the runtime system to allow for additional bookkeeping needed to detect data corruption if a failure occurs during the write operation. In our model, task outputs are specified up front by the user and metadata is used to determine which of these outputs need to be written (i.e. accumulated) to avoid producing an incorrect result. In order
to avoid extra communication, this information is generally encoded into the task
descriptor rather than gathering potentially distributed metadata in-line.

**Algorithm 6.2** TC.process: Execute all tasks in a task collection

\[
\textbf{while} \quad t_i \leftarrow \text{TC.next()} \textbf{ do} \\
\quad t.\text{fetch}_\text{inputs}() \\
\quad t.\text{execute}() \\
\quad \textbf{for all} \quad j \in O_i \textbf{ do} \\
\quad \quad \textbf{if} \quad M_{i,j} = \bot \textbf{ then} \\
\quad \quad \quad M_{i,j} \leftarrow \text{started} \\
\quad \quad \quad \text{contribute}(j) \\
\quad \quad \quad M_{i,j} \leftarrow \text{completed} \\
\quad \quad \textbf{end if} \\
\quad \textbf{end for} \\
\textbf{end while} \\
\text{barrier}() \\
\]

### 6.3.2 Corruption Detection

After the task collection has been processed we check to determine if a failure has occurred. If one has, we must check to determine if it occurred during a write operation resulting in an incomplete write and data corruption. This requires a scan of local metadata to detect any entries in the *started* state, indicating that they were started but not completed. If such an entry exists, this node must perform recovery by restoring its shared data from the checkpoint taken at the start of the task parallel region and resetting the corresponding metadata entries. This scan is necessary because we allow read-modify-write (i.e. accumulate) updates of shared data. If only non-overlapping writes were allowed then this step would not be necessary.
Algorithm 6.3 $TC\text{.scan}$: Detect data corruption due to failure and initiate recovery.

\begin{algorithm}
\For{$i = 1 \ldots M$} {
  \If{$M_{i,me} = \text{started}$} {
    \text{recover()} \quad \text{// Restore from CP and $M_{s,me} \leftarrow \bot$}
  }
}\end{algorithm}

6.3.3 Completeness Detection

When a failure has occurred, we perform analysis on the metadata to determine which tasks are incomplete due to the failure. Incomplete tasks may need to recompute some or all of their outputs and are added back into the task collection for re-execution during the next round. We next describe two algorithms to perform this step: a simple naive algorithm and a home-based algorithm that greatly improves on the communication efficiency of the naive approach.

Naive Algorithm

In Algorithm 6.4 we present a simple approach to detecting incomplete tasks. In this approach, for every task we check the metadata entries corresponding to the elements of its output set. If one of these entries is not completed the task is re-entered into the task collection. This can be done in parallel for each task in the set of tasks, preserving the property that a task can be added at most once to the task collection.

Home-Based Algorithm

The number of communication steps required for the naive approach is proportional to $O(|T| \cdot |O_{avg}|)$ where $T$ is the set of tasks and $O_{avg}$ is the average number of
Algorithm 6.4 $TC.check\_completion$ (naive): Find incomplete tasks and add to the task collection.

```plaintext
for all $T_i \in T$ do
  for $j \in O_i$ do
    if $M_{i,j} \neq \text{completed}$ then
      $TC \leftarrow TC \cup T_i$
      break
    end if
  end for
end for
```

outputs per task. The size of $T$ is generally much larger than the number of processors to allow for load balancing and continues to grow with problem size. In addition, due to data distribution of shared arrays, $O_{avg}$ also grows with system size for a given problem. This gives a worst case communication complexity that is $O(nproc^2)$ steps. Thus, a more scalable solution is needed.

In Algorithm 6.5 we present an algorithm that introduces the notion of recovery homes in order to reduce the communication complexity of scanning the metadata. In this scheme, the set of tasks is partitioned according to task ids (row numbers in the metadata matrix) and each block of tasks is assigned to a process who is the home for these tasks. The assignment of tasks to processes is known by all processes and a hashing function $H$ can be used to map task ids to their recovery homes. During recovery, each process scans its local metadata and compiles a message for each home indicating which tasks homed there have completed metadata entries on the scanning node. These messages are then sent to all the homes and used by the home to check for task completion. Thus, each process sends one message to every other process resulting in a total of $O(nproc)$ communication steps.
Algorithm 6.5  \textit{TC.check\_completion} (home-based): Find incomplete tasks and add to the task collection.

\begin{enumerate}
\item let: $T' \subseteq T$ be the set of tasks homed on me
\item let: $H : \mathbb{N} \rightarrow \text{rank}$
\item let: $MSG$ be an $M \times \text{MAX\_OUTPUTS}$ matrix
\item let: $M'$ be an $\text{nproc} \times \text{MAX\_OUTPUTS}$ matrix
\end{enumerate}

\begin{algorithmic}
\For{$i \in 1 \ldots M$}
\If{$M_{i,me} = \text{completed}$}
\State Append $i$ to $MSG[H(i), \ast]$
\EndIf
\EndFor

$M' \leftarrow$ All-to-All exchange of $MSG$

\For{$T_i \in T'$}
\For{$j \in O_i$}
\If{$j \cap M'[i] = \emptyset$}
\State $TC \leftarrow TC \cup T_i$
\State break
\EndIf
\EndFor
\EndFor
\end{algorithmic}
6.4 Discussion

The fault tolerant task parallel programming model yields several attractive properties which we describe in this section.

6.4.1 Tolerance of an arbitrary number of failures

The metadata tracking scheme described in Section 6.2 is robust in the presence of an arbitrary number of failures. Because the bookkeeping information lost due to failure corresponds only to data that is also lost, at any given time all correct nodes contain metadata that describes the complete state of the valid data in the computation.

6.4.2 Low performance overhead when no failures occur

The task execution algorithm presented in Section 6.3 adds only the overhead of maintaining the metadata to correct processes. This involves two sets of communication operations per task: one to set metadata bits to \textit{started} before writing outputs and one to set them to \textit{completed} when finished. These communication operations can be overlapped with task execution without any impact on robustness. Metadata bits can be set to \textit{started} using non-blocking communication at the start of task execution and setting them to \textit{completed} can be overlapped with execution of the next task. In this work we have focused on the cost of recovery and have not yet fully explored such techniques to mitigate the overhead of metadata maintenance. Thus, for the experiments where no faults occurred communication is still performed to maintain metadata. For SCF, this has only a small impact on performance because
it’s tasks are long running, however applications with different characteristics could require more aggressive overhead management techniques.

6.4.3 **Space overhead proportional to task collection size**

Conventional in-memory checkpointing techniques can also be used for selective restart, however these techniques store full copies of critical data and can double a program’s space requirements. In comparison, fault tolerant task collections provide finer-grain recovery while potentially using only a fraction of the space. The storage overhead for fault tolerant task collections is proportional to the size of the metadata matrix: the number of tasks multiplied by the number of outputs per task.

6.4.4 **Recovery cost proportional to degree of failure**

Conventional checkpoint/restart techniques respond to failure by restarting all nodes in the computation from the most recent checkpoint. The cost of this recovery mode is proportional to the total system size. In comparison, fine-grain recovery using the task parallel model involves recovering only the failed nodes and recomputing only the data lost from those nodes. In addition, the task parallel model provides load balancing, allowing all processes help with recomputation further accelerating this process.

6.4.5 **Bounded cascading failure**

Recovery is initiated on non-failed nodes to correct data corruption due to a failure that occurred while another node was writing to its memory domain. Because this recovery is isolated and does not trigger recovery on other nodes, the number of
cascading failures is bounded by the maximum number of memory domains a task is permitted to write to concurrently: the maximum number of task outputs.

6.5 Experimental Evaluation

In this section we present an experimental evaluation of our fault tolerant task parallel system using the Self Consistent Field (SCF) computation from the quantum chemistry domain. These experiments were performed on a 650 node IBM System 1350 cluster with Infiniband interconnect. Nodes in this system are configured with two quad-core 2.5 GHz AMD opteron processors and 24GB of RAM.

6.5.1 Self Consistent Field Computation

The Self Consistent Field (SCF) computation is a key, computationally intensive step in many computational chemistry applications that computes the ground-state wave function for a system of atoms. In particular, SCF forms the starting point for most \textit{ab initio} quantum chemistry simulations. In order to evaluate our system, we have extended an existing Global Arrays implementation of the closed-shell SCF method [79] with support for fault tolerant task collections.

This implementation computes the Fock matrix using the non-relativistic SCF method from the Born-Oppenheimer approximation, as described in detail in [77]. SCF exhibits irregular data access and computational granularity due to irregularity present in the underlying atomic system. Both the Fock and density matrices are distributed across all processors using Global Arrays. In the original implementation, load balancing is achieved by replicating a work queue across all processes and performing atomic increment on a shared counter to get the next available task. Pseudocode for SCF is shown in Algorithm 6.6. As shown, task collections can be
used to throughout the computation; in this work we focus on the performance of the

two electron computation which accounts for 90% of the execution time of SCF.

\begin{algorithm}
\textbf{Algorithm 6.6} SCF Pseudocode: Task collections can be used for one electron, two
electron, and density computations.
\begin{verbatim}
\textbf{while} $\text{deltad} < \text{thresh}$ \textbf{do}
  $\text{TC} \leftarrow \text{oneltasks}$
  $\text{TC}.\text{process}()$
  $\text{TC} \leftarrow \text{twoeltasks}$
  $\text{TC}.\text{process}()$
  $\text{diagolize}(\text{Fock})$
  $\text{TC} \leftarrow \text{denstasks}$
  $d \leftarrow \text{TC}.\text{process}()$
  $\text{deltad} = \text{global\_max}(d)$
\textbf{end while}
\end{verbatim}
\end{algorithm}

6.5.2 Fault simulation

A full fault-tolerant infrastructure for GA is currently under development. Thus,
in order to perform an early evaluation of our fault tolerant task parallel model we
have simulated failures in software. This is done at the user level by restoring a node’s
patch of the global array from a checkpoint and clearing the corresponding metadata.
Worst case failures are simulated in order to elicit the upper bound on the cost of
recovery. We define a worst case failure as a failure that occurs after the last task has
finished executing. Thus, the maximum amount of data is lost by the failing process.

In Figure 6.1 we show the percent of all tasks that must be re-executed for SCF on
a 48 beryllium atom data set experiencing a random worst-case single process failure.
From this data we observe that the number of re-executed tasks is proportional to the
number of processors via the data distribution. Due to the amount of global arrays
Figure 6.1: Penalty in tasks re-executed for a worst-case single process failure on SCF.

data per process, a low processor count incurs a higher penalty due to failure than higher processor counts where the same global arrays are spread across more memory domains.

6.5.3 Performance Study

We have extended the SCF benchmark presented in [25] to use fault tolerant task collections. We focused specifically on the kernel that calculates the two electron contributions to the Fock matrix since this accounts for over 90% of the execution time. However this technique can be applied to multiple task parallel sections of the code.

In Figures 6.2 and 6.3 we present the results from strong scaling experiments for the SCF computation on a system of 48 beryllium atoms on up to 256 processors. In
Figure 6.2: Execution time for no faults and worst case failures using naive and home-based recovery schemes.

Figure 6.2 we show the average execution time per iteration for the SCF kernel and in Figure 6.3 we show the relative performance as the ratio of the performance relative to the performance when no faults occur. Each graph shows data for three schemes: a baseline scheme where no failures occur, worst case failures with naive completeness detection, and worst case failures with the home-based completeness detection. For schemes that experience faults, we simulate one worst case failure per iteration of the SCF kernel.

By looking at the execution time for the naive completeness detection scheme, we see that the $O(nproc^2)$ cost for recovery can quickly dominate execution time on larger processor counts resulting in substantial slowdown. The home-based scheme offers the best performance in the presence of failures and closely follows the baseline
Figure 6.3: SCF performance relative to the case with no faults.

performance trend. From the slowdown data, we see that the average performance penalty to recover from a worst-case single process failure is less than 10\% for the home-based scheme while it grows very large for the naive scheme. In comparison, a conventional checkpoint/restart scheme incurs a penalty of 50\% slowdown for a worst case failure because it must repeat the computation a second time.

In addition, as shown in Figure 6.1, for small processor counts the penalty for a single failure is high because each process holds a larger amount of the global address space, inflating the number of tasks that must be re-executed. Higher processor counts result in a smaller data loss penalty but incur a higher communication overhead.
6.6 Conclusion

We have presented a new approach to fault tolerance that leverages the task parallel programming model to survive an arbitrary number of failures. This system improves on conventional techniques by offering a cost for recovery that is proportional to the degree of failure while maintaining low overheads. Fine-grain recoverability is achieved by tracking individual task progress with distributed metadata. This system was evaluated on the SCF computational chemistry kernel and demonstrated to achieve recovery penalties of less than 10% for worst case failures on 256 processors.
CHAPTER 7

Related Work

There is a wealth of literature in the area of task parallelism, however much of the prior work has focused on shared memory and distributed shared memory (DSM). This work is distinguished from the shared memory and DSM work in that we have focused on distributed memory systems and partitioned global address space (PGAS) memory model. We break down the related work into three primary areas: parallel programming models; dynamic load balancing and runtime systems; and fault tolerance. We further break down the programming models work into work that relates to the PGAS memory model which we have utilized and work that relates to the task parallel control model which we have developed. Overall the primary contributions of this work are:

**PGAS Task Parallelism:** We have created a parallel programming model called Scioto that addresses the asynchronous gap between the flexible PGAS memory model and process-centric SPMD control model. This programming model expresses the computation through tasks and exposes opportunities to enhance performance through automated techniques such as dynamic load balancing and scheduling to enhance affinity.
**Scalable Runtime System:** We presented a scalable runtime system to support the Scioto programming model. This system is centered around the use of work stealing to perform dynamic load balancing. We have presented techniques to enhance the scalability of work stealing and demonstrated high efficiency at 8192 processors on a distributed memory cluster. This is the first demonstration of work stealing at this scale.

**Fault Tolerance:** We have developed techniques for performing selective recovery from failures. These techniques leverage the unique properties of task parallelism to efficiently track fine-grain execution information and preserve valid partial results while re-computing invalid or lost data.

### 7.1 Partitioned Global Address Space

Distributed Shared Memory systems (DSM) such as Treadmarks [43] and Cluster OpenMP [38] were developed to provide the programmer with a global view of shared data on distributed memory computers. However, these systems have faced significant performance and scalability challenges due to communication overhead from maintaining data coherence and false sharing that can be incurred when using a page as the unit of coherence.

Partitioned Global Address Space (PGAS) programming models such Unified Parallel C (UPC) [82], Co-Array Fortran (CAF) [65], Global Arrays (GA) [64], and Titanium [88] have evolved to offer a compromise between full DSM and explicit messaging systems. These systems do not provide automatic coherence for shared data and instead rely on the programmer to maintain consistent data through explicit synchronization. In comparison with DSM models where data can migrate throughout the
computation, PGAS programming models fix the data in place but provide mechanisms to query exploit data locality. Thus, these systems offer a tradeoff between the ease of programming with a global address space and the performance possible with explicit data placement and communication management.

HPCS languages such as X10 [20], Chapel [19] and Fortress [75] also provide a partitioned global address space similar to existing PGAS systems. In addition, these new models relax the structure of the control allowing for asynchronous activities that support a task parallel programming model. These systems are still in their early stages and have not yet been fully realized on distributed memory machines. We hope that through this work we can give insight into their efficient implementation.

7.2 Task Parallel Programming Model

The Scioto projects aims to synthesize a new parallel tasking system that provides high-level support for lightweight task management, interoperability with PGAS programming models, and scalable performance on distributed memory machines.

A variety of systems have been developed that offer the programmer dynamically scheduled, task-based views of computation. Cilk[31] is a parallel extension to the C programming language that offers an elegant parallel tasking model, allowing the user to fork and join tasks. Satin is a adaption of Cilk for the Java programming language [84]. Cilk-NOW [15] extends support for Cilk to distributed memory systems and adds fault tolerance, however it focuses primarily on support for functional parallelism and does not support a global address space. KAAPI [33] is a recent effort to support data flow computations on large clusters and has achieved scaling for up to 3600 cores on a variety of benchmarks.
Concurrent object systems such as Charm++[39], PREMA[9] or the Illinois Concert System[40] target both shared and distributed memory systems and provide adaptive parallel programming models based on concurrently executing migratable objects. These systems provide an object-centric view of the parallel computation rather than a process-centric view and use message passing rather than a global shared address space for data. TStreams and Concurrent Collections (CnC) [17, 46] are new models where the programmer expresses their computation in terms of computational steps, items, and tags. Relations between these sets are used to provide dependence information (e.g. producer and consumer) as well as to prescribe (i.e. enable execution) of computational steps.

New high productivity parallel programming languages such as Chapel [19], X10 [20], and Fortress [75] offer language-level primitives to support asynchronously executing activities. The scalable implementation of the powerful task models of these new languages is still a challenging problem and we hope that the implementation and evaluation of the simpler Scioto model on a range of application programs can offer useful insights to their implementors.

Multithreaded programming in the context of distributed shared memory through systems like DSM-Threads [59] also allows the programmer to write dynamically scheduled computation in a global address setting. However, many of these systems rely on fair thread scheduling and thus must multiplex threads to ensure progress and perform full thread migration when balancing the load. In our work, we define a more restricted model which does not guarantee fairness. This allows for a lighter weight and more scalable system because it eliminates the need to perform time sharing and task migration.
7.3 Dynamic Load Balancing

Load balancing is a challenging problem and has been widely studied in the literature. Work in this area can be broken down into two categories: synchronous and asynchronous dynamic load balancing.

Synchronous dynamic load balancing is load balancing that is performed periodically, often with the cooperation of all processors. Such techniques may involve graph partitioning [18, 24, 41, 81] or space filling curves [2]. Task graph partitioning has also been explored [51] and focuses on optimizing the execution time given a set of tasks organized as a task graph. Locality-aware load balancing for a set of independent tasks using hypergraph partitioning has also been addressed in [10, 47]. Unlike the work presented in this proposal, these techniques require knowledge of all the tasks to be executed in order to perform load balancing.

In this work, we have focused on asynchronous dynamic load balancing. Asynchronous dynamic load balancing techniques perform load balancing continuously and without knowledge of all work to be performed. Work stealing is a popular asynchronous dynamic load balancing technique [14, 49, 74] and is the scheme we focus on in this work. Work stealing has been extensively investigated as a strategy for load balancing parallel depth-first search [50, 49]. It has also been shown to be space optimal for a broad class of problems [14], to provide stable load balance [11], and to exhibit good cache locality [1]. Cilk-style work first, or depth first, task execution has also been supplemented with a help first strategy [35] that increases parallel slack and is complementary to existing work stealing schemes. Our work in scaling work stealing to over 8000 processes has brought forth new issues, which we have
discussed in this proposal. We are not aware of any prior demonstrations of scalable load balancing through work stealing at such scale.

In Chapter 8 we introduced an approach to expressing tasks with dependencies. Asynchronous dynamic load balancing for such tasks is a challenging problem. Cilk [31] supports load balancing of fully strict computations based on work stealing and X10 work stealing [22] extends the Cilk algorithm to support terminally strict computations. Recent work [55] has investigated idempotent execution models in which each task is executed at least once rather than exactly once. NESL [13] allows for locality aware work stealing. All these efforts focus on work stealing in the context of shared memory (or distributed shared memory) systems.

7.4 Fault Tolerance

User-directed checkpoint/restart techniques such as Berkeley Lab’s Checkpoint/Restart [29] and GA checkpoint/restart [80] commit process checkpoints to disk periodically. Other process-level checkpoint/restart systems such as [32] and [60] perform checkpointing automatically throughout the computation but must log messages to ensure consistent global snapshots are captured. User-level checkpointing reduces the volume of data stored by storing only critical data and regenerating ancillary data during recovery. Some systems, notably NWChem [44], store a disk-resident database of intermediate results at each stage of the computation that can be used for restart. Fault tolerant task pools complement these coarse-grain recovery techniques with a fine-grain selective recovery mechanism that can be used to protect task parallel regions of a program against faults.
Efforts are also underway to develop a fault tolerant runtime stack, including a coordinated system-wide fault notification and handling backplane [36] as well as fault tolerant FT-MPI [21]. In addition to such reactive solutions, proactive fault tolerance [83] uses pre-failure feedback to prevent process failures via mechanisms like migration.

Cilk-NOW [15] provides fault tolerant task parallel computing on a network of workstations. However, Cilk-NOW is client-server based, focuses on functional parallelism, and does not support a shared global address space. In comparison, fault tolerant task pools has a fully distributed architecture and supports a shared global address space. BOINC [3] supports fault tolerant task parallel execution on a network of unreliable volunteer machines which can produce erroneous results due to failures. Thus, redundant computation must be performed in order to verify correct results for every unit of computation. BOINC is also client-server based and does not support a shared global address space.

Much work has been done to develop fault tolerance for the Linda programming model [5]. These efforts focus on maintaining a stable, content-addressable shared tuple space through replication and preventing tuple loss due to failure using atomic transactions. In contrast, this work uses the location-addressable global address space provided by GA and does not require transactions. Updates to shared data are performed via atomic accumulate, however failures can interrupt these operations resulting in memory corruption; metadata is used to identify and correct these inconsistencies. Linda also supports task parallelism through the shared tuple space. In contrast, FT task pools logically separates the tasks from the application’s shared
data and uses a checkpoint of the task list to detect lost or incomplete tasks, avoiding problems of duplication and task loss.
CHAPTER 8

Future Work

Scioto provides an exciting platform for research in the area of parallel programing models and scalable runtime systems. In this work, we have defined the baseline task parallel programming model, a scalable runtime system, and an application of the model to enable smart recovery from faults, however many directions for future work remain open. In this chapter, we discuss some of these opportunities and break them down into several core directions: programming model and scalable runtime support for expressing inter-task dependencies, utilizing the task parallel model as a means for managing computation on accelerators, supporting speculation within the task model, and continued work on fault tolerant task collections.

8.1 Task Dependencies

Many parallel computations exhibit data flow and control dependence between units of the computation. Such computations can only be supported in the current model through dynamic task spawning or phase-based parallelism. However, these approaches are not sufficient for exposing the rich parallelism available in a variety of computations, thus support for inter-task dependence is a key direction for future
research. In particular, scalable support for expressing and managing dependence information that is communication efficient remains an open problem.

Computations with inter-task dependencies can be expressed as a directed acyclic graphs (DAG) of tasks, \( G = (V, E) \). In this context, \( v \in V \) are tasks and for \( e \in E \), \( e = (v_1, v_2) \) is a directed edge from \( v_1 \) to \( v_2 \) which represents a dependence of \( v_2 \) on \( v_1 \). Scheduling tasks and managing dependencies for task DAGs can be complicated by the large number of edges present in general DAGs. It may be possible to reduce these overheads by reducing the number of edges through the insertion of additional clock vertices \(^1\) that represent discrete synchronization events. Clocks have multiple in edges and become enabled when all in edges are activated. A task becomes ready to execute when all of its clocks are enabled.

In Figure 8.1 we show a series of two blocked matrix-matrix multiplications, \( C = A \cdot B \) followed by \( E = C \cdot D \), where the result of the first multiplication is an input to the second. In the current model, this computation would require two task parallel phases, effectively inserting a barrier between multiplications, however this synchronization does not allow the programmer to fully expose the available parallelism. For each block of the \( C \) matrix that is completed, the tasks from the second multiplication that have the \( C \) block as an input dependence can begin. For example, once the \( C_{11} \) block is completed, the \( E_{11} \) \( \stackrel{\pm}{=} \) \( C_{11} \cdot D_{11} \) and \( E_{11} \) \( \stackrel{\pm}{=} \) \( C_{11} \cdot D_{11} \) can begin.

The corresponding dependence graph is shown in Figure 8.2(a). In this figure, we see that the \( E \) tasks all share the same dependencies. By inserting an clock vertex in the graph as shown in Figure 8.2(b) we are able to represent these dependencies.

\(^1\)Clock objects have also been introduced in the context of the new X10 programming language [20, 72], however X10 clocks are used as a generalization of split barriers and, in this discussion, clocks are used as a mechanism for expressing dependence between tasks.
Figure 8.1: Chained matrix-matrix multiplication: The dot product of block row 1 of $A$ with block column 1 of $B$ produces block $C_{1,1}$ enabling the corresponding tasks to begin in the second multiplication.

as a dependence on a single synchronization event. This representation may greatly reduce the complexity and cost of expressing and managing dependencies. In each case, the communication cost of sending completion information is proportional to the number of edges in the DAG. The introduction of clocks helps to reduce the total number of edges by reducing the out degree of a vertex to the number of events it contributes to rather than the number of tasks that have it as a dependence.

Many challenges are present in the efficient implementation of inter-task dependencies. Names must be assigned to clock events so that they can be referenced by tasks that enable them and tasks that subscribe to them. Clock vertices in the DAG must also be created at most once and may be dynamically created to support dependence in the context of dynamic parallelism. These challenges introduce information dissemination and ordering constraints on the problem of expressing and managing dependence information. In addition, clocks with high in and out degree can result in large amounts of communication, possibly with a single node who owns the
Figure 8.2: Matrix-matrix multiplication task dependence graph without and with clock synchronization for $C = A \cdot B$ followed by $E = C \cdot D$. 
clock. Such clocks may present performance challenges and may require a hierarchical scheme in order to reduce communication contention and scale to high processor counts. Finally, clock locality may play a role in the overhead of executing each task. When the clocks that the task advances are not stored locally, a task may need to perform several communication operations in order to update its dependent clocks.

8.2 Accelerator Tasks

Recently there has been widespread interest in the use of specialized hardware for application acceleration [45, 61, 66]. This trend has been largely centered around the use of graphics hardware for solving computational science and engineering problems and for many applications, this approach has yielded significant performance gains over conventional CPUs [4, 37, 53, 70, 71, 73]. Many applications that are unbalanced, operate on large data sets, or exhibit dynamic parallelism also stand to benefit from hardware acceleration. For such applications that the Scioto task model offers valuable assistance with managing the computation and the PGAS data model offers access to a large distributed global address space. Therefore, an important direction for future investigation is in extending the Scioto model to leverage the performance gains of hardware accelerators.

Many open questions exist in supporting the PGAS data model and task parallel control model on hardware accelerators. Many accelerators, such as GPGPUs, are simply connected to a conventional node and have a private memory on which they operate. Others integrate heterogeneous cores, providing one or more conventional scalar cores in addition to cores that are optimized for data parallelism. Such systems pose challenges in scheduling tasks and also present opportunities for runtime
optimizations by making conventional cores available to the runtime system while occupying accelerator cores with computation. Managing the data in the PGAS and potentially multiple copies of data on the local node also presents challenges to the programmer; recent work on asynchronous distributed shared memory for heterogeneous processing nodes may offer exciting possibilities for the integration with PGAS models [34].

Automatic overlap of computation with communication may be possible for PGAS tasks running on accelerators with private memory, such as GPGPUs and the Cell processor. For such accelerators, task execution can be broken down into the following sequence of steps:

1. Fetch inputs from global address space into local memory
2. Move data from local memory into device memory
3. Perform computation
4. Move data from device memory to local memory
5. Write data from local memory into global address space

When multiple tasks are available, it may be possible to pipeline these operations to automatically overlap data movement with computation. Furthermore, if device memory can be made addressable by the network interface for rDMA transfers then steps 2 and 4 can be eliminated.

Because such programs offload much of the computational work to an accelerator, this model opens up multiple opportunities to leverage the host CPU. The host CPU can be used to also execute tasks, especially for tasks that execute poorly on
the accelerator because they are control intensive or make non-streaming accesses to memory. In addition, some of the host CPU’s resources could be dedicated to optimizing performance during the task parallel region. For example, performance can also be enhanced by scheduling tasks that operate on the same input operands in sequence. This scheduling exploits inter-task locality in the accelerator’s memory and avoids repeated copies to device memory and communication to fetch operands from the PGAS. For example if we consider the blocked matrix-matrix multiplication, $C = A \cdot B$, each task that operates on a row block of $A$ requires the same column blocks of $B$. If these tasks can be scheduled in sequence then the communication cost of re-fetching blocks of the $B$ matrix can be eliminated.

8.3 Speculative Task Parallelism

Speculative parallelization refers to the use of speculation to generate multiple units of work that can be executed in parallel. Speculative parallelization has been successfully applied in a number of settings [48, 54, 76, 78, 86]. In much of the literature, speculation has been used to parallelize loops with dynamic data references where data dependence cannot be determined statically. Recently, this technique has generated a high degree of interest as a mechanism for dynamically parallelizing sequential codes on multicore processors [62, 85].

Previous work in this area speculates primarily on data and control dependence. However, some scientific computing applications that rely on the use of Markov Chains may benefit from speculation within the numerical method. In particular, one approach taken by some Computational Chemistry and Physics simulations for solving complex integrals is the use of Markov Chain Monte Carlo (MCMC) methods [23].
These methods essentially perform a random state space walk, performing a test at each state to determine if it is accepted and incorporated into the global solution or if it is rejected. When a rejection occurs, the walk continues from the last accepted state. In such a Markov Chain, the next state depends only on the properties of the current state. Thus, the process of walking a Markov Chain is inherently sequential.

In the context of the dynamical nucleation theory simulation presented in [23], the calculation involved in processing each state is an expensive energy computation that varies in intensity from $O(N^2)$ to $O(N^4)$ floating point operations. The workload can be spatially partitioned to use multiple walkers and increase parallelism, however the spatial irregularity present in chemical systems exacerbates the imbalance from the energy computation across walkers. In addition, there is an upper bound on the number of walkers that can be used before the accuracy of the result is affected. Due to these limitations in the degree of parallelism, applications such as this are severely challenged in achieving strong scaling.

For this and many other applications the acceptance rate can be 90% or higher. Given such a high acceptance rate, such applications can expose greater amounts of parallelism by speculating on future states of the Markov Chain. This parallelism can be used to accelerate the processing of Markov Chains, effectively shortening the critical path of the execution. Thus, the development of a task parallel approach to speculation can be used to expose the parallelism needed to achieve strong scaling on such simulations. In this context, task parallelism offers a succinct model for packaging speculative computations as well offering the ability to perform the dynamic load balancing needed to overcome imbalance present in the physical system under simulation.
In order to ensure correct execution of the program, speculative tasks must not be committed until they are accepted. One structure that can be used to support this mode of execution is an *accept buffer*. Task results are stored in a distributed accept buffer and once they are accepted, they will be committed to the global solution. Two processes must be performed as a part of such an execution: an execution process that executes tasks and an inspector/generator process that commits accepted tasks and manages the speculation. The inspector must cancel the execution of tasks when a rejection occurs and trigger the generation of additional speculative tasks when the volume of available work gets low.

States in a Markov chain are sequential. Therefore, tasks that speculate on future states of a Markov chain must be accepted sequentially. In order to achieve this, an inspector/generator model can be adopted. The inspector process must perform the task of scanning the accept buffer to determine task acceptance and incorporate results into the solution. It must also trigger the generation of new speculation as results are accepted to maintain a high level of parallelism. Finally, when the inspector detects a mis-speculation it must abort the execution of speculative tasks further down the chain to avoid performance losses due to mis-speculation. Performing these tasks in the context of a distributed execution where information on speculation is also distributed poses significant challenges. In addition, automated strategies to determine the degree of speculation given runtime feedback on the rate of acceptance will be critical in helping to control overheads.
8.4 Fault Tolerance

In Chapter 6 we developed extensions to the task parallel programming model that leverage the task abstraction in order to enable smart recovery from faults. This was a first step toward fault tolerant task collection and leaves open many interesting and exciting research problems. In particular, the fault tolerant task collections model described in Chapter 6 places significant restrictions on the model in terms of the input/output behavior of tasks. For example, this model does not provide infrastructure needed to support fault tolerant dynamic task creation. In this section we describe opportunities for future work in this area including much needed mechanisms for managing input and output data and dynamic parallelism while maintaining resilience in the presence of faults.

8.4.1 Data Management

The fault tolerant task collection model presented in Chapter 6 requires that the user specify input and output data in the global address space up front so that it can be managed by the runtime system. This is restrictive, but considered necessary in order for the runtime system to perform fine-grain tracking of the computation needed to perform selective restart. Given that input and output data is managed by the runtime system, multiple unexplored opportunities for performance and productivity enhancements have been exposed. For example, the runtime system can provide the programmer with a shared memory-like view of the data, potentially reducing the amount of data transferred by monitoring updates to output regions. In addition, opportunities exist to automatically overlap data transfers with computation.
In addition, further investigation is needed to decrease the space and communication overheads involved in maintaining task execution metadata. Recall that metadata can be in three states: \( \perp \), \textit{started}, or \textit{completed}. The initial value of all metadata entries is \( M_{i,j} = \perp \), which indicates that task \( i \) has not produced an output to domain \( j \). \textit{Started} indicates that an update has been started and \textit{completed} indicates that the previously \textit{started} update has completed.

When a task wishes to write its outputs, it first sets the corresponding metadata bit to \textit{started} and once complete it sets the metadata bit to \textit{completed}. If these operations are performed in sequence, significant overhead will be incurred since the network latency must be paid twice for each output. Instead, metadata can be eagerly set to \textit{started} and lazily set to \textit{completed}. This leaves the window for cascading failure open for a longer period of time. However, when failures are rare it can yield significant improvement in communication overheads.

### 8.4.2 Fault Tolerant Dynamic Task Creation

One of the strengths of the task parallel programming model is its ability to support dynamic parallelism. However, this capability is not supported by the techniques described in Chapter 6. One approach to supporting fault tolerant dynamic task creation is to view tasks as outputs produced by a parent task and track the production of these outputs using metadata similar to the tracking of data outputs described in Chapter 6. In this scheme, one would associate a \textit{home} with every task whether statically or dynamically created. This home must be identifiable by a globally known
hashing algorithm that takes the task name as input and produces its home id as output. One particular challenge in implementing this scheme is the need for a technique that assigns unique and deterministic names to each task.

When dynamic tasks are created, copies must be stored in a task history log with associated metadata. This would allow dynamic tasks to be treated similarly to data outputs. This history would allow the runtime system to track the progress of the computation so that when a failure occurs the computation does not need to start from the beginning and create the child tasks again. Instead it can do a selective replay of only the tasks corresponding to lost information.

When a failure occurs, a portion of the dynamic task history and its metadata will be lost. A significant research challenge in solving this problems lies in identifying the static or dynamic tasks that produced these lost dynamic tasks so that they can be recovered. Also, once recovered, such tasks must be checked to determine if they must be re-executed. In addition, ancestor, descendant data dependence poses significant challenges to this recovery process. If such a data dependence occurs then this induces an ordering on recovery where a tasks ancestors must all be completed before that task can be re-executed.
CHAPTER 9

Conclusion

In this work, we presented a task parallel programming model called Scioto that seeks to provide a complete environment for supporting irregular and dynamic parallelism by combining the PGAS data model with a task parallel execution model. Scioto’s runtime system has been tuned for performance and scalability and offers high efficiency on thousands of processor cores. Experimental results indicate that on a number of benchmarks Scioto is highly efficient, scales to over to 8192 cores, and incurs low overheads.

The task parallel programming model effectively decouples the computation from the underlying resources, exposing additional opportunities for fault tolerance. We have leveraged this property to define algorithms for selective restart after failures, effectively reducing the cost for recovery on $N$ processors to $O(N_{\text{failed}})$ compared with $O(N)$ for conventional checkpoint/restart schemes alone.

The Scioto system is a promising framework for future research and we have identified several directions for future work. These include scalable support for inter-task dependencies; tasks that execute on compute accelerators, such as GPGPUs; speculative tasks; and performance and functionality enhancements to fault tolerant task collections.
BIBLIOGRAPHY


114


[20] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebe...


