PARALLEL PROGRAM COMPOSITION
WITH PARAGRAPHS IN STAPL

A Dissertation

by

TIMMIE GENE SMITH

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2012

Major Subject: Computer Science

PARALLEL PROGRAM COMPOSITION

WITH PARAGRAPHS IN STAPL

A Dissertation

by

TIMMIE GENE SMITH

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Lawrence Rauchwerger
Committee Members,	Nancy M. Amato
	Gabriel Dos Reis
	Marvin L. Adams
Head of Department,	Duncan M. Walker

May 2012

Major Subject: Computer Science

ABSTRACT

Parallel Program Composition with Paragraphs in Stapl. (May 2012)

Timmie Gene Smith, B.S., Texas A&M University;

M.C.S., Texas A&M University

Chair of Advisory Committee: Dr. Lawrence Rauchwerger

Languages and tools currently available for the development of parallel applications are difficult to learn and use. The Standard Template Adaptive Parallel Library (STAPL) is being developed to make it easier for programmers to implement a parallel application.

STAPL is a parallel programming library for C++ that adopts the generic programming philosophy of the C++ Standard Template Library. STAPL provides collections of parallel algorithms (`pAlgorithms`) and containers (`pContainers`) that allow a developer to write their application without reimplementing the algorithms and data structures commonly used in parallel computing. `pViews` in STAPL are abstract data types that provide generic data access operations independently of the type of `pContainer` used to store the data.

Algorithms and applications have a formal, high level representation in STAPL. A computation in STAPL is represented as a parallel task graph, which we call a `PARAGRAPH`. A `PARAGRAPH` contains a representation of the algorithm's input data, the operations that are used to transform individual data elements, and the ordering between the application of operations that transform the same data element. Just as programs are the result of a composition of algorithms, STAPL programs are the result of a composition of `PARAGRAPHs`.

This dissertation develops the `PARAGRAPH` program representation and its com-

positional methods. PARAGRAPHs improve the developer's difficult situation by simplifying what she must specify when writing a parallel algorithm.

The performance of the PARAGRAPH is evaluated using parallel generic algorithms, benchmarks from the NAS suite, and a nuclear particle transport application that has been written using STAPL. Our experiments were performed on Cray XT4 and Cray XE6 massively parallel systems and an IBM Power5 cluster, and show that scalable performance beyond 16,000 processors is possible using the PARAGRAPH.

To Lidia

ACKNOWLEDGMENTS

I would first like to thank my advisor, Dr. Lawrence Rauchwerger, for his support and guidance throughout my studies. In our extensive discussions of research and life he impressed upon me the importance of keeping the long term goal in sharp focus while working through the details as they were encountered.

I would also like to thank Dr. Nancy M. Amato for her guidance in my research. Her insight in to what a user should be expected to provide as they developed a parallel application and what functionality they might find useful shaped the design of my work and made its application easier. Her constant involvement in the STAPL project throughout my graduate career has been a great resource.

The members of my committee deserve thanks for their unique contributions to my dissertation. Dr. Gabriel Dos Reis provided me with feedback on my proposal and dissertation that broadened the scope of related work I considered and strengthened the foundation of my work. Dr. Marvin L. Adams has spent years meeting with me, instructing me on the fundamentals of nuclear particle transport, and providing insightful critiques of performance issues in STAPL and PDT based on his experience developing parallel scientific applications.

I would like to thank Dr. Mauro Bianco, a postdoc in the STAPL group, for his careful review of my design documents and help generalizing the interfaces it provided.

Thanks to the members of Parasol support for helping me learn about system administration as we tried to keep the lab up and running. I would also like to thank Kay Jones for all of her help over the years with administrative and facility issues.

I have had the fortunate experience of having my research applied in several programs as it was being developed. I owe many thanks to the members of Dr. Adams'

and Dr. Amato's research groups for using STAPL as it was developed. I am grateful for the time they took explaining the problem being solved by their application and working to establish a common vocabulary that allowed us to communicate. I am also very thankful for their patience and perseverance working with the implementation as interfaces evolved and performance issues were addressed. The individuals I had the pleasure of working with the most are: Michael Adams, Anthony Barbu, Dr. Teresa Bailey, Dr. Jae Chang, Dr. Josh Jarrell, Daryl Hawkins, Sam Jacobs, Dr. Alex Maslowski, Roger Pearce, and Dr. Shawna Thomas.

Throughout graduate school I had the opportunity to interact with a large number of colleagues in the Parasol Lab, the members of the STAPL research group in particular. I would like to thank them all for contributing to my "other" education by teaching me about their countries, cultures, faiths, and philosophies as we worked on our research. I offer special thanks to Dr. Silvius Rus, Dr. Alin Jula, Dr. Gabriel Tanase, Dr. Dongmin Zhang, and Shishir Sharma for sharing an office with me at one point during my graduate career.

I thank Joel Barron, Mike Sconzo, and Dr. Nathan Thomas for being amazing friends. You have always pushed me to be better, supported me through challenges, and celebrated victories with me. I cannot thank you enough.

Hardy Carlyle was the first person outside of my family to encourage my interest in computers. As my computer teacher in elementary school he taught me that coding could be fun. In high school his instruction in computer science set me on the path that has led me to a Ph.D. He was also my golf coach, programming team school sponsor, and UIL academic meet school sponsor. During the time we spent at competitions and practice rounds he instilled confidence, integrity, and responsibility. I would like to thank him for investing so much time in me.

Finally, I would like to give very special thanks to my family. My wife, Lidia, has

been my constant support and source of encouragement. Thank you for your sacrifice as I finished my work and always being there with love and encouragement after a long day. I cannot imagine how I could have done this without you. I would like to thank my mother, Debbie, who stressed the importance of education and always encouraged my academic pursuits. I would like to thank my entire family for their patience, love, and encouragement.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

> "Informally, an **algorithm** is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**. An algorithm is thus a sequence of computational steps that transform the input into the output." Cormen et. al. [1]

The definition provided by Cormen et. al. introduces a sequential view of algorithms, where a single processor executes each step in the sequence one after the other. Parallel computer architectures that contain multiple processing elements have been available for decades, and the recent shift by computer processor manufacturers to multi-core processors has made them ubiquitous [2]. A different mindset is needed for the development of parallel algorithms that will run efficiently on parallel architectures. In addition to specifying the operations that will be performed on each data element a parallel algorithm needs to specify an ordering on the parallel operations to ensure correctness and have some awareness of data distribution and load balancing for efficient execution. Many approaches for developing parallel applications exist. They provide different programming models that address dependence specification, data distribution, and load balance differently.

A. Existing Approaches

Process-oriented approaches such as MPI [3] implement algorithms as concurrent sequences of instructions. While some collective operations are provided for syn-

---

The journal model is *IEEE Transactions on Automatic Control.*

chronization, the main mechanisms for ordering of operations or communication are point-to-point operations that must be handled explicitly in the algorithm by the developer. The developer must also address data distribution and load balancing in the implementation of their algorithm. Finished applications can execute very efficiently and are portable because the MPI library has ubiquitous on HPC systems. However, the developer is often focused on the communication required instead of the details of the algorithm itself. The performance is not always portable either due to differences in system architectures.

Divide-and-conquer systems such as NESL [4] and Cilk [5] provide a data-oriented expression of the parallel algorithm whose implementation is focused on split and join operations. The data locality of the resulting algorithm is good, and load balancing can be handled implicitly by the runtime system. The difficulty of developing algorithms in these systems is determining when to stop dividing and when to begin processing the data. It is also difficult to express applications that have complex dependence patterns between operations using these systems.

Task-oriented systems such as Pthreads [6] and Intel Threading Building Blocks (TBB) [7] allow the developer to focus on the operations of the algorithm instead of the processes executing the algorithm. Both systems utilize shared-memory for communication, making it implicit in the algorithm. Load balancing is handled for the developer by the operating system for Pthreads, or by the work-stealing scheduler of the TBB runtime system. Task synchronization (ordering tasks to ensure correct execution) is either limited or difficult to express. TBB's limited task synchronization restricts the dependence patterns that can exist in algorithms implemented using the library.

Pattern-based systems such as algorithmic skeleton libraries [8] [9] allow developers to easily develop new algorithms as a composition of parallel patterns, specifying

the operations that will be performed by the tasks of each pattern when it is instantiated in the code. The degree of extensibility (i.e., the ability to add a new skeleton to the library) varies, but is usually limited. The forms of composition of the patterns may be limited as well, which restricts the form algorithms implemented using the library may take.

Stream-base systems such as Intel Concurrent Collections [10], StreamIt [11], and the Pipeline skeleton in many algorithmic skeleton libraries, allow a parallel algorithm to be specified as a set of transformations that are applied in turn to data elements, which continuously enter the algorithm over time. Expressing the ordering between the transformations of the algorithm is easy in these systems. Load balancing and communication is also handled implicitly by the runtime system. The drawback of such systems is limited applicability, as it may be difficult to express the input to an algorithm as a stream.

B. Parallel Algorithm Definition

Considering the existing approaches, each with its advantages and difficulties, we developed the following goals for a developer when expressing a parallel computation:

- able to focus on expressing operations of the algorithm,
- able to specify any ordering between operations that is necessary,
- able to specify the pattern of a parallel algorithm instead of individual operations,
- able to add a new pattern for a parallel algorithm, and
- able to specify any ordering between or composition of the patterns used to express a parallel algorithm.

This set of capabilities leads to the following definition of a parallel algorithm that captures the necessary information and provides the proper mindset for developing parallel algorithms.

Definition 1 (Parallel Algorithm) *A parallel algorithm is a tuple* $\{O, D\}$ *where:*

- O *is the operations of the parallel algorithm.*
- D *is the partial ordering of operations that access the same data.*

The operations of O can be parallel algorithms themselves, which makes the definition above recursive, or they can be transformations of individual data elements that match the transformations on data in the definition by Cormen et. al. that began this chapter. The elements of O can be thought of as a *collection of tasks*. The elements of D provide the information needed to correctly schedule the execution of the tasks.

The representation of the tasks and their dependences used in this dissertation is a high level task graph of the computation called a `PARAGRAPH`. The specification of `PARAGRAPHs` at the level of individual tasks and dependencies is a difficult and error-prone process. This dissertation explores methods to simplify task graph specification in order to improve the productivity of the programmer developing a parallel application.

C. STAPL Programming Model

The Standard Template Adaptive Parallel Library (STAPL) [12–21] is a parallel programming library for C++ that is being developed to address the difficulties of parallel programming. STAPL provides functionality similar to STL, the Standard Template Library that is part of the ISO adopted C++ standard [22]. STL is a collection of

containers, iterators, and algorithms that may be used as the building blocks of a sequential application. Similarly, STAPL provides `pContainers` for parallel data storage, `pViews` to provide uniform data access operations by abstracting away the details of the particular `pContainer` being used to store data, and a collection of `pAlgorithms` that are parallel implementations of algorithms frequently used in applications.

The programming model in STAPL is a task graph representation of the application. The application is written as a task graph, called a `PARAGRAPH`, whose operations are task graphs themselves. The result is a hierarchical task graph that captures the entire computation. Hence, in STAPL, the `pAlgorithms` declare one or more `PARAGRAPHs`, whose execution performs the desired transformation of the input data. The implementation of the `PARAGRAPH` in STAPL is the validation of the concepts presented in this dissertation.

The `PARAGRAPH` has been used to implement the `pAlgorithms` and operations of the `pContainers` that are presented in [12, 16–21]. A paper that is focused on the work presented in this dissertation is under preparation.

D. Research Objective and Contributions

Our research objective is to define mechanisms for concise expression of task graphs and their composition to simplify the development of parallel algorithms and applications. The `PARAGRAPH` presented in this dissertation makes several novel contributions.

- The task dependence graph of a parallel application is a hierarchical construct built through the composition of task dependence graphs of smaller program units. It is explicit in the application and can be manipulated by the developer.
- Dynamic irregular task graphs can be specified in user-level code.

- A unified form for the specification of regular and irregular parallelism is provided.
- A unified form for the expression of data- and task- parallelism is provided.

E. Outline

The remainder of the dissertation is organized as follows. Chapter II summarizes the related work. Chapter III presents an overview of STAPL with a discussion of the primary components in the library. In Chapter IV we present the mechanisms used to specify a `PARAGRAPH`. The means used to compose `PARAGRAPHs` into larger computations is presented in Chapter V. An experimental evaluation of the `PARAGRAPH` implementation is presented in Chapter VI, and includes the performance of individual STAPL `pAlgorithms` and multiple benchmarks and an application written using STAPL. Finally, in Chapter VII we make our concluding remarks and identify possible directions for future research based on the work of this dissertation.

CHAPTER II

RELATED WORK

Several research projects have used task graphs to represent applications. The scope of a task graph in these projects ranges in scale from loop nests to large workflows for scientific problems in order to expose parallelism. The review of those efforts is organized by whether the task graph is constructed automatically, or expressed by the developer in some form. Research projects in the area of algorithmic skeletons are then surveyed, as this research area is closely related to the `task factory` concept developed in this dissertation.

A. Compiler Constructed Task Graphs

One of the fundamental concepts taught in compiler design courses is that the program being compiled can be represented by several different graphs. For example a data flow graph represents the instructions of a program as vertices of a graph with edges added between an instruction that produces a value and all of the instructions where the value is read. We are interested in projects that form a graph from the source code in order to identify instructions that can be executed in parallel.

1. HTG

The Hierarchical Task Graph (HTG) [23] was implemented in the Parafrase-2 compiler as an intermediate representation that captured the set of data and control dependencies of an application in order to enable task parallelism. At each level of the hierarchy a task is a sequence of instructions. Tasks at the same level in the graph may be executed in parallel if there is no control or data dependence between them. HTGs allow nested task parallelism as independent tasks at any level of the graph

can be executed in parallel. The task graph is extracted from the application source code, and as such there is no need for dynamic task graph support or allowing user specification of the task graph that is comparable to our work. The HTG works in addition to the detection of data-parallel operations (i.e., parallel loops) by additional passes in the Parafrase-2 compiler, and only supports task parallelism (i.e., MPMD programming) as a result.

2. Program Dependence Graph

The Program Dependence Graph (PDG) [24] is an intermediate representation that combines control and data dependences of a program in a uniform representation. The goal of the work was to aid in the development of optimizations for vectorizing and parallelizing compilers. The uniform representation allows some optimization passes, such as vectorization, to be simplified because the control dependences are no longer treated as special cases. The nesting of control structures in an application results in a PDG that is hierarchical as well. The hierarchical nature of the PDG was exploited to produce summaries of code that allowed for rapid querying of a region during optimization to determine if it needed to be processed, and to simplify reordering transformations. The PDG representation was intended to be used by all optimizations of a parallelizing/vectorizing compiler. It is able to expose both task and data parallelism. Like in the HTG, algorithm developer manipulation of the graph and dynamic task graph support is unnecessary in this work. The hierarchical nature of the PDG is similar to the hierarchical task graph representation that we explore in this dissertation.

3. Polytope Model

The polytope model [25] is an internal representation of the iterations of a set of nested for-loops. Each iteration is represented by an *iteration vector* whose components are the values of the loop indices for that iteration. A set of inequalities can be formed to describe the shape of an n-dimensional polyhedron that contains all the points of the iteration space of the loop nest. Transformations that have been established in the theoretical basis of the polyhedron can be applied then to change the shape of the polyhedron to a more regular shape. This allows better analysis of the loop nest by the compiler, possibly allowing it to be recognized as parallelizable when it would not have been in its original form. Like the HTG, the polytope is formed by the compiler during analysis of the application and is not exposed to the user, nor is dynamic modification of the polytope possible or desirable. The analysis is applied only to loop nests and thus only enables data parallelism.

B. User Expressed Task Graphs

The task of automatically extracting parallelism from a sequential code is difficult because the languages used to develop applications rarely allow the developer to include semantic information about the computation in its implementation. A sequential language also allows unrelated variables in a function to reuse the same memory and requires the developer to arbitrarily order independent instructions of a function to form a sequential execution, further complicating the task. In an attempt to address these concerns several research projects allow the application developer to express the computation as a task graph in some form, allowing parallelism to be easily detected and exploited.

1. Chimera

The Pegasus [26] Workflow Management System is a tool that coordinates the execution of large-scale scientific applications across multiple platforms in different locations. Pegasus performs the same operations as the `PARAGRAPH Executor` [27] and the Scheduler of the STAPL run-time system. Workflows can be expressed using multiple tools that differ in their level of abstraction. Pegasus is capable of handling explicitly specified workflows, Chimera [28] specifications, and CAT [29] specifications. These three levels of specification correspond loosely to the explicit specification, task factory specification, and composition of task graphs proposed in this dissertation. The Chimera system contains a collection of data sets that represent the results of scientific computations. A scientist is able to query the system and specify a new data set that they need. Chimera uses its knowledge of the results it already has and the computations that may be performed to transform those results to generate a directed acyclic graph (DAG) of the computations that must be performed on a specific subset of the results available to obtain the desired result. That DAG can then be given to Pegasus or another workflow management system to schedule the necessary computations.

2. Intel Concurrent Collections

Intel Concurrent Collections [10] [30] is a parallel programming model that allows a high level, dataflow-like description of an application. A domain developer that writes the application expresses the data to be processed in *item collections*, the computation to perform in *step collections*, and any control dependence between step collections in *tag collections*. Each of the preceding concepts is referred to as a collection to emphasize that it is a set of distinct units. This is important because

the step instances in a step collection are the minimal execution unit and schedulable entity, and item instances are the finest grain of communication that can occur in the application. The result of the domain developer's work is a model of the application that is fine-grained.

When the domain developer is finished with the model it is passed to what is referred to as a *tuning expert* to map the computation on to a system for execution. The tuning expert – which may be a software analysis tool or a human – is concerned with the coarsening of items into coarse-grained entities to minimize runtime overhead, distributing coarsened items across processes, and scheduling coarsened items for execution within a process [31].

The separation of concerns between the domain developer and tuning expert in expressing the computation and mapping the computation to a system, respectively, is similar to the separation of concerns between the `PARAGRAPH` developed in this dissertation and the `PARAGRAPH Executor` developed in [27] for algorithms written using STAPL. The application developer specifies the data to be processed, the operations to use, and the pattern of the computation as a `PARAGRAPH` instance. The `PARAGRAPH` instance is processed by the `PARAGRAPH Executor` that instantiates the tasks of the `PARAGRAPH` and maps them onto the system for execution and maintains the dependencies between the tasks to ensure correct execution.

3. Intel Threading Building Blocks graph class

Intel Threading Building Blocks (TBB) version 3.0 update 5 [7] has introduced a task graph class as a community preview feature. The library now provides a graph class that can be instantiated and explicitly populated with nodes and edges. The behavior of a node varies based on its type. There are nodes that execute user functions in addition to nodes that simplify setting up different communication patterns. After a

graph is constructed it may be repeatedly executed.

The specification of a TBB graph is explicit and requires the developers to add individual nodes to the graph and then add the appropriate edges. The usage examples in the TBB documentation indicate that the scale of the graphs is intended to be small, which makes its use practical. In Chapter IV we demonstrate that explicit construction of the tasks in a `PARAGRAPH` to process the elements of an input `pView` would be difficult due to the scale of the graph. Our work proposes using dependence patterns and task graph composition methods to simplify task graph specification and raise the level of abstraction used in a STAPL application.

C. Algorithmic Skeletons

Algorithmic skeletons [32] [33] are higher-order functions that implement the structure of a parallel algorithm and accept functions that implement the operations to be performed within the algorithm as arguments. An example of a skeleton is *reduce*, which can be thought of as a parallel implementation of the STL accumulate algorithm. The reduce algorithm accepts a functor that implements the operation to be applied on the elements to form the accumulation. How the code within the algorithm is parallelized and how the partial results are combined to form the final answer is hidden from the user. The DatTeL library [34] is a skeleton library for C++ that implements parallel versions of the STL algorithms as skeletons using this approach.

The `task factories` developed in this dissertation are the analog of skeletons. Indeed, the `task factories` provided in STAPL (map, reduce, prefix scan) were some of the earliest data parallel skeletons developed. `pAlgorithms` in STAPL use these `task factories` to instantiate the `PARAGRAPHs` they need in order to perform the desired

computation. While code written using skeletons is portable, the skeleton implementation usually is not. Skeletons are typically implemented using the native run-time system or a low level API such as MPI, and the implementation of each skeleton is independent of the others in a library. The STAPL `task factories`, on the other hand, all generate tasks of a `PARAGRAPH` that are independent of the system on which they will execute and utilize the same components of the STAPL run-time system for execution. By generating tasks of a `PARAGRAPH` that is executed by the `PARAGRAPH Executor` the implementations of the computation patterns are as portable and efficient as the `pAlgorithms` that use them. The `parallel_for`, `parallel_reduce`, and `pipeline` algorithms in Intel Threading Building Blocks [7] are also examples of skeletons whose implementations use a common, higher level of abstraction for their implementation.

CHAPTER III

STAPL OVERVIEW

STAPL [12] is an extensible parallel programming library for C++. It adopts the generic programming philosophy of the C++ standard template library (STL) [35]. When a developer begins using STAPL to develop an application the components she encounters closely parallel the components provided by STL. Figure 1 illustrates the similarities in the high level components of the two libraries. STL provides containers for storing application data, algorithms that implement frequently occurring computations, and iterators that abstract the containers and provide the algorithms with a uniform set of data access operations. Developers familiar with these components will be able to easily map them to their STAPL counterparts.

A. Component Overview

When a developer begins writing a program using STAPL there are three sets of components they begin using immediately. Parallel algorithms (`pAlgorithms`), distributed data structures (`pContainers` [19]), and `pViews` [21] are the high level building blocks that can be used to compose an application. The `pAlgorithms` and `pContainers` provided by STAPL have interfaces similar to the C++ STL containers and algorithms. `pViews` provide data access operations while abstracting away the details of the `pContainer` on which they are defined, analogous to the way STL iterators provide access to elements stored in containers independent of the container type. The `pAlgorithm` parameters that specify the data to process are `pViews`, and because the `pViews` provide generic data access operations a `pAlgorithm` is able to operate on multiple `pContainers`.

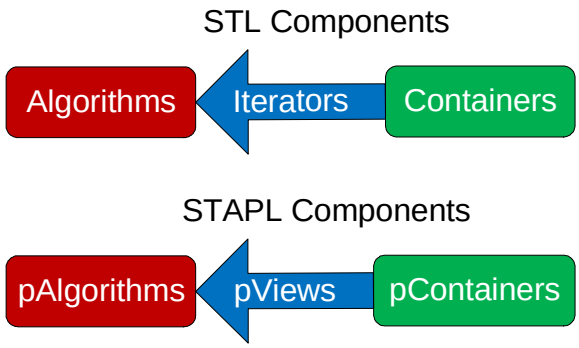


Fig. 1. Mapping between STL and STAPL components

`pAlgorithms` declare a set of `PARAGRAPHs` whose execution will perform the desired parallel operation. A `PARAGRAPH` is a task graph representing a parallel computation. The `PARAGRAPH`, the `PARAGRAPH Executor` which executes `PARAGRAPHs`, and related task concepts used in this dissertation are defined below.

Definition 2 (Task) *A Task T is a pair (A, D) where A is an algorithm and D is the data that represents the inputs and outputs of A.*

Definition 3 (Task Graph) *A task graph TG is a graph whose vertices are tasks, and whose edges represent data dependencies between the tasks.*

The algorithm performed by a task may itself be expressed as a `PARAGRAPH`, and it is implemented in STAPL by what we refer to as a `work function`.

The data to be processed by a task is represented by one or more `pViews`. The dependencies in the task graph ensure that a task completes execution before any of the tasks that depend on the results of its execution are allowed to begin execution. The shape of the task graph is determined by the `task factory` provided to the `PARAGRAPH`. Processing of a `PARAGRAPH` is handled by the components of the `PARAGRAPH Executor`. The `PARAGRAPH Executor` performs the creation and mapping of the tasks of a `PARAGRAPH` to the system and enforces the dependencies between the

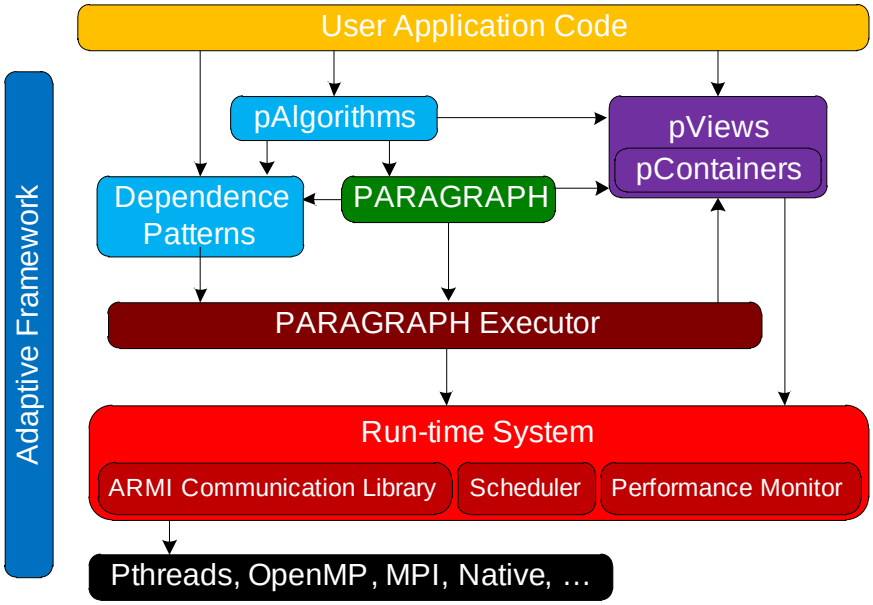


Fig. 2. STAPL Overview

tasks. Figure 2 illustrates the organization and interaction of the components of STAPL with an application written using the library.

The arrows indicate that the component at the base of the arrow uses the component at the end of the arrow (e.g., `pAlgorithms` use `PARAGRAPHs`). While there are no lines shown in the figure to the adaptive framework [15], each component of STAPL may use the adaptive framework to choose between multiple implementations of a component to improve performance when they are available.

The sets of `pAlgorithms` and `pContainers` provided by STAPL allow a developer to develop their application without having to re-implement the algorithms and data structures commonly used in parallel computing. However, for any application it is likely that the developer will need domain specific parallel algorithms and data structures that are not provided by STAPL. Composing `pContainers` (i.e., defining `pContainers` of `pContainers`) may provide the desired data structure very quickly. In more advanced cases, the STAPL Parallel Container Framework (PCF) [19] [36]

provides components that allow a developer to quickly define a new parallel data structure.

The STAPL components a developer may use to specify new parallel algorithms are the subject of this thesis. In order to develop a new `pAlgorithm` the developer must:

1. specify the pattern of the parallel computation to determine which dependence pattern to use,
2. specify the operations that will be performed on the data, and
3. specify the data that will be processed by the `pAlgorithm`.

Completion of the first step allows the developer to decide if one of the `task factories` provided in STAPL will allow them to construct the `PARAGRAPH` they desire, or if they will need to develop a new `task factory` as well. Completing the second step allows the developer to write the operations that will be the core of the `work functions` used in the tasks of the `PARAGRAPH Executor`. STAPL simplifies this step by allowing nested parallelism via calls to existing `pAlgorithms` in the operation. Completing the third step above will allow the developer to know how many `pViews` their new `pAlgorithm` will accept. The `pAlgorithm` the developer is writing may be made up of several `PARAGRAPHs` that need to interact with one another. STAPL provides a set of composition operators to allow `PARAGRAPHs` to be composed together and processed by the `PARAGRAPH Executor` concurrently.

The following chapter details how to construct an individual `PARAGRAPH`, and Chapter V defines the `PARAGRAPH` composition operators provided to allow more parallelism to be exposed as the `PARAGRAPH Executor` processes the `PARAGRAPHs`.

CHAPTER IV

PARAGRAPH SPECIFICATION

`PARAGRAPHs` are the building blocks of the parallel algorithms and applications developed using STAPL. The `PARAGRAPH` simplifies the development of a parallel algorithm by:

- allowing a developer to separate the implementation of the individual operations of their algorithm from the specification of the dependencies between the operations,
- allowing a developer to separate the specification of a `PARAGRAPH` from its mapping to and execution on processing locations by the `PARAGRAPH Executor`, and
- allowing seamless composition of task graphs.

The `PARAGRAPH` and `task factory` concept it uses are defined as follows.

Definition 4 (`PARAGRAPH`) *A* `PARAGRAPH` *is a task graph, and is defined as the tuple* $\{F, O, V\}$ *where*

- F *is a* `task factory` *that adds tasks and associated dependencies to the task graph when invoked by the* `PARAGRAPH Executor`,
- O *is a tuple of operations that will be applied by the tasks of the task graph, and*
- V *is a tuple of* `pViews` *representing the data to be processed by the task graph.*

Definition 5 (`task factory`) *A* `task factory` *is a tuple* $\{\mathrm{SP}, \mathrm{DP}(O)\}$*, where* SP *is a* structural pattern *and* DP *is a* dependence pattern *that is parameterized by the tuple of operations,* O*, provided to the* `PARAGRAPH`.

The *structural pattern*, SP, defines how many tasks will be generated for a `PARAGRAPH`, and which of these tasks can be predecessors of tasks in another `PARAGRAPH` through composition. For every point in SP the *dependence pattern*, DP, generates a specification of the task that should be created, including the operation from O to be applied, the set of preceding tasks, and the number of successors.

In the remaining sections of the chapter, `PARAGRAPH` construction and coarsening are presented first. The interfaces provided for specification of `PARAGRAPHs` using `task factories` are presented, followed by the interfaces for dynamic task specification. The chapter concludes with a case study illustrating how to implement a new `task factory` for a computation that cannot be expressed with the patterns provided in STAPL. The case we will examine is a sweep algorithm used in a discrete-ordinates particle transport code.

A. PARAGRAPH Construction and Coarsening

Specification of a `PARAGRAPH` instance requires constructing an object that contains all three elements of the tuple in the definition above. The tuple V of `pViews` is provided directly to the specification. These `pViews` provide access to the individual elements of the data to be processed. The `task factory` F is also provided directly. The operations of O, however, don't need to be provided directly to the `PARAGRAPH` specification.

The operations in O are only invoked during task execution when the `PARAGRAPH` is being processed by the `PARAGRAPH Executor`. Therefore the `PARAGRAPH` itself doesn't need to be aware of the number or type of the operations. The `task factory` F does need to know the number of operations and their exact type, because it is responsible for producing specifications of the tasks in the task graph when the

PARAGRAPH is processed by the PARAGRAPH Executor, and the specification includes an instantiation of the operation used by the task. Therefore, a convenient way to construct a PARAGRAPH instance is to pass the pViews and task factory to the constructor, and have the task factory instance store the work functions that are instances of the operations in O.

The pViews and work functions provided by the developer are fine-grained. An element of a pView is an individual datum that is processed by a single invocation of a work function. Passing a PARAGRAPH specified with these fine-grained pViews and operations to the PARAGRAPH Executor for processing would result in a task graph that is proportional to the size of the input data and whose tasks would have execution overhead near the cost of the operation execution. The runtime overheads in this scenario would negate the benefits of using the PARAGRAPH to express parallel algorithms. In order to minimize the overhead the PARAGRAPH introduces in application execution it is *coarsened* when it is constructed if the task factory used allows coarsening.

The coarsening of a PARAGRAPH requires transformation of both the operations and pViews. The transformation applied to the operations produces new operations that are able to handle multiple elements and return the coarsened result. A simple example is the coarsening of an operator that is used in the PARAGRAPH that computes an accumulation of a set of elements in a pView. The original operator could be std::plus<> from the C++ STL, and the coarsened operator would be similar to the listing shown in Figure 3. The developer may bypass the operator coarsening by providing a "coarsened" operator to the PARAGRAPH constructor. The operator contains an extra type definition to indicate that it is capable of handling multiple elements. Developer coarsened work functions are used in the implementations of NAS EP and PDT discussed in Chapter VI.

```
template <typename Op>
struct coarse_plus
{
private:
  Op m_op;
public:
  typedef typename Op::result_type result_type;

  coarse_plus(Op const& op) : m_op(op) {}

  template <typename View>
  result_type operator()(View& view)
  {
    typename View::value_type result = view[0];
    for (int i = 1; i != view.size(); ++i)
      result = m_op(result, view[i]);
    return result;
  }
};
```

Fig. 3. Coarsened accumulation operator.

The coarsening of the `pView` is a coarsening of the data that is passed to the operator of a task in the task graph. The result of data coarsening is a `pView` of `pViews`. Currently in STAPL, the coarsening performed when a `task factory` indicates that it can support coarsening (i.e., work functions that are the coarsened versions of all operations in O are available) maximizes the size of the coarsened element while maintaining data locality on each processing location. The developer may also precoarsen the data passed to a `PARAGRAPH` instance. The coarsening produces a well known `pView` type that we are able to detect when the `PARAGRAPH` is instantiated and bypass the coarsening for that `pView`.

Data coarsening is the opposite approach of the divide-and-conquer technique that has been adopted by existing approaches such as NESL [4], Cilk [5], and Intel TBB [7]. We chose to perform coarsening of the fine-grained computation instead of dividing a coarse-grained computation because the fine-grained approach exposes the maximum amount of parallelism to the system during execution. As the amount of parallelism available in systems continues to increase it will become more important

to find available parallelism in an application. The fine-grained specification allows a STAPL application to operate at the maximum degree of parallelism available if the overheads in the system are low enough to make it beneficial.

After coarsening is complete, the `task factory` of the `PARAGRAPH` generates tasks that apply an operation from the coarsened versions of the functions in O to the elements of the coarsened `pViews` that were in V. It is important to note that the behavior of the `PARAGRAPH` and `task factory` are the same whether coarsening was performed or not. The uniform treatment of fine-grained and coarsened `pViews` and operations prevents usage of the `PARAGRAPH` from being limited to one case or the other. This is demonstrated again in the presentation of a new `task factory` for a sweep algorithm in a discrete-ordinates particle transport application. In that application, the coarsening of the `pView` and operator is performed by the application developer before the `PARAGRAPH` for the sweep is instantiated.

1. STAPL Implementation

The declaration of the `PARAGRAPH` class and its constructor is shown in the listing of Figure 4. The notation for variadic templates, a feature of the latest C++ standard [22], is used to show that any number of `pViews` can be provided to the `PARAGRAPH`. The number of `pViews` cannot exceed the number of `pViews` expected by the `task factory`.

B. Specification Using Task Factories

Interfaces that would allow for the specification of individual tasks and their dependencies in a `PARAGRAPH` are too low level to allow for productive development of parallel algorithms. In order to raise the level of abstraction of populating a task

```
template <typename TaskFactory, typename Operations, typename ...Views>
class paragraph
  : public task_graph
{
  paragraph(TaskFactory const& factory, Operations const& operations, Views const&... views);

  paragraph(paragraph const& pr);
};
```

Fig. 4. Declaration of the `PARAGRAPH` class and its constructors.

graph, to enable reuse of the code that performs the population, and overlap task creation and execution, the idea of providing a generative functor to the task graph is explored. Each functor is parameterized with the operations to be used and is given the set of coarsened `pViews` – unless the `task factory` disallows coarsening – to process. The functor generates the tasks of a task graph with a particular pattern of dependencies between the tasks. The generators are instances of the factory method pattern [37], and we refer to them as `task factories`.

A `task factory` captures the computational pattern of a task graph. The parametrization of the functor with the operations to be performed by the tasks it generates allows the `task factory` to be reused to implement a wide range of algorithms that have the same structure in their task graph. We have found, along with researchers in the algorithmic skeletons community [8, 38], that there are a few computation patterns that occur with high frequency in parallel applications. Therefore, a small number of `task factories` written by a more advanced developer can cover a substantial number of parallel applications. Once a `task factory` is written it enables rapid development of efficient and correct parallel algorithms by all developers.

A `task factory` is the combination of a *structural pattern* and a *dependence pattern* that has been parameterized with the operations of the `PARAGRAPH`. The *struc-*

tural pattern specifies how many tasks will be generated given the size of the input `pViews`. For each task specified by the *structural pattern* the *dependence pattern* is responsible for forming the specification of the operation to be applied, the set of predecessor tasks, and the number of successors. While we believe there are only a few `task factories` that need to be implemented to capture the most common parallel computation patterns, we believe that domain specific patterns will be found as applications are developed using STAPL. The `task factory` definition provided above allows for the implementation of *structural patterns* and *dependence patterns* to be separated, simplifying the job of a developer writing a new `task factory` and allowing for greater reuse of both types of patterns.

An example that demonstrates the flexibility of the `task factory` definition is a comparison of the implementations of the `map factory` in STAPL presented in Section C and the `sweep factory` in PDT detailed at the end of this chapter in Section E.

The `map factory` generates a set of tasks that will apply the operation provided to the factory to every element in the input `pView`. There are no dependences between the tasks, so the *dependence pattern* returns a task specification for every point in the *structural pattern* that contains the operation, an empty set of predecessors, and indicates that there are no successors for the task. The *structural pattern* of the factory will specify a task for each element of the input `pView`. The *structural pattern* is similar to the *pardo* construct used in the pseudo-code of [39], so we use `pardo` as the name of the *structural pattern*. The *dependence pattern* is named `independent` to indicate that no dependence information will be specified. The `map factory` could be generated by the function shown in Figure 5.

The `sweep factory` in PDT is responsible for constructing the tasks of a `PARAGRAPH` that compute the movement of subatomic particles in a given direction across a spa-

```
template <typename Operation, typename View>
map_factory construct_map(Operation& operation, View& view)
{
  return pardo(independent(operation), view.size());
}
```

Fig. 5. Expression of map factory with structural and dependence patterns.

tial domain. The spatial domain is represented as a graph whose cells are discretized units of space and the edges in the graph represent a shared face between two discretized spatial units. The edges store a vector normal that indicates the orientation of the face from the face center to the center of the spatial unit. To determine the change in particle flow in a direction an operation must be applied to each vertex of the graph. This single application of the operation to a vertex indicates that the structural pattern needed is `pardo`. The *dependence pattern* of the `sweep factory` is such that an operation cannot be applied on a graph vertex until the operation has been applied to all vertices that have an edge to the current vertex with a vector normal whose dot product with the sweep direction is positive (i.e., the edge represents incoming particle flow). The `sweep factory` and the *dependence pattern* it uses can be written using the code shown in Figure 6.

1. STAPL Implementation

a. Task Graph Pattern Requirements

The `task factories` currently implemented in STAPL are classes that have their *structural pattern* and *dependence pattern* explicitly encoded in the implementation. Full generalization of the factories using the structural and dependence patterns as defined above is one of the directions for future research.

A `task factory` instance is a distributed object that is responsible for gener-

```
template <typename Operation, typename View, typename Direction>
sweep_factory construct_sweep(Operation& operation, View& view, Direction& direction)
{
  return pardo(categorize_edges(operation, direction), view.size());
}

template <typename Index, typename Vertex>
task_specification categorize_edges::operator()(Index& index, Vertex& vertex)
{
  vector<int> predecessors;
  int num_successors;
  for (Edge& e : vertex.edges())
    if (dotproduct(e.normal(), direction) > 0)
      predecessors.push_back(e.target());
    else if (dotproduct(e.normal(), direction) < 0)
      ++num_successors;
  return task_specification(vertex.id(), operation, predecessors, num_successors, vertex);
}
```

Fig. 6. Expression of sweep factory using structural and dependence patterns.

ating all tasks of a `PARAGRAPH` when the `PARAGRAPH` is processed by the `PARAGRAPH Executor`. The `task factory` is constructed outside of a `PARAGRAPH` and is passed to its constructor. The `PARAGRAPH` constructor inserts a task on each processing location in the task graph that will invoke the `task factory` on each location when it is executed. The `task factory` is implemented as a `dynamic task` [27] whose operator inserts the tasks of the computation into the task graph, possibly through multiple invocations on each processing location. It is the responsibility of the `task factory` implementer to decide how the generation of the tasks will be distributed across the processing locations and invocations of the `task factory` tasks on a location to ensure that the tasks of the `PARAGRAPH` are generated correctly. STAPL provides helper functions that assist the developer by providing a partitioning of the coarsened input `pView`elements across the processing locations. The `task factory` tasks on a particular processing location could then be responsible for generating the tasks that operate on the `pView`elements that are in the subset of the partition on that location.

Section c explains why the generation of the tasks of a computation is necessarily

done through the repeated execution of the `task factory` task on each processing location. The `PARAGRAPH` may call the `reset` method of the `task factory` to allow it to reset any internal state it may have so the `task factory` instance may be reused to execute the `PARAGRAPH` again (e.g. as part of an iterative algorithm where the `PARAGRAPH` is constructed outside of the computation for the iteration).

The `task factory` must provide a constructor that accepts function objects that are the operations that will be used as the task operations. For example, the `task factory` for the map-reduce pattern in STAPL provides a constructor that accepts two function objects; the first is the map operation to perform on the input elements and the second is the reduce operation that combines the outputs of the map operations to form the final result of the computation. The task factory for the map pattern provides a constructor that accepts a single function object that implements the operation to be applied to each input element. Table I lists the methods that must be provided by a `task factory` implementation.

Table I.: Methods required of a `task factory`.

Name	Purpose	Return Type	Arguments
Constructor	instantiate `task factory`	`task_factory`	Task operation functors
`finished`	Report if all tasks generated	bool	none
`reset`	Reinitialize internal state	none	none
`operator()`	Generate task specifications	none	coarsened input `pViews`

The `task_factory_core` structure in STAPL simplifies `task factory` creation by providing default implementations of the required `task factory` methods. The interface of the structure is listed in Figure 7. The class provides implementations of the `finished` and `reset` methods that are appropriate for use in a stateless `task factory` that generates all tasks of the task graph in a single invocation of its function operator. If the `task factory` performs incremental task generation then it must override the default behavior by calling the `task_factory_core` constructor

```
class task_factory_core
  : public p_object
{
protected:
  bool m_finished;

  bool initialized(void)  const;

  void reset_view_indices();

  void set_view_index_iterator(std::size_t idx, view_index_iterator_base* ptr);

  std::size_t view_indices_size() const;

  view_index_iterator_base*
  get_view_index_iterator(std::size_t n) const;

public:
  task_factory_core(bool b_incremental_generation=false);

  virtual bool finished() const;

  virtual void reset();

  virtual ~task_factory_core();
};
```

Fig. 7. Interface of task_factory_core class.

with `b_incremental_generation` set to true, and set the `m_finished` data member when all tasks on this location have been specified. If a `task factory` contains internal state it must redefine the `reset` method. The interface in Figure 7 contains several members that are related to incremental task generation, which is described in Section c.

The `task_factory_core` has an additional benefit for the `PARAGRAPH` implementation. The task graph is able to refer to the task factory by a pointer to the core class. This allows the `task_graph` class to be non-templated, which reduces the amount of code that is generated when a `PARAGRAPH` type is instantiated during program compilation. The number of invocations of the virtual methods is small compared to the number of tasks expected in the task graph, and in practice we haven't observed negative impact from the overhead of the indirections introduced by the virtual function

calls. The PARAGRAPH implementation requires that every task factory derive from this class.

The task_factory_core class provides the essential interfaces of a task factory and simplifies the development of basic task factories. The development of task factories for task graphs that contain dependencies is still a complicated process as the task factory contains code similar to what is shown in Algorithm 1, which computes the sum of an arbitrary number of elements using a task graph whose shape is shown in Figure 8. Algorithm 1 finds the set of the largest powers of two whose sum is the number of elements to process (e.g., $1345 = 1024 + 256 + 64 + 1$). A balanced binary tree task graph of each of these widths is generated, and tasks to combine trees are added to complete the task graph. This complexity in what is a very regular and straightforward computation pattern is the motivation for our future research direction that will allow the structural and dependence patterns of a task factory to be specified independently of one another as described in the previous section.

b. Task Specification

The code in Algorithm 1 used add_task functions to insert tasks into the PARAGRAPH being processed by the PARAGRAPH Executor. These functions are provided by the task_graph_access class.

The task_graph_access class, shown in Figure 9, is used as the base class of task factories and dynamic work functions to provide methods to insert tasks into the task graph for mapping and execution as the task graph is processed by the PARAGRAPH Executor. In the case of task factories, this class and the task_factory_core are used as the parent classes of task_factory_base. The task factory base class further simplifies the specification of tasks with its consume method that allows the

Algorithm 1 reduce(ReduceOp& op, View& view)

```
n ← view.size(), offset ← 0, id gets 0
while n ≠ 0 do
  logn ← ⌊log n⌋
  width ← 2^(logn-2)
  if n = 1 then
    width ← 1
  end if
  for i ← 1 to logn-1 do
    if i = 1 then
      for j ← 0 to width-1 do
        add_task(id++, op, view[offset+2*j], view[offset+2*j+1])
      end for
    else
      base ← id - width*2
      for j ← 0 to width-1 do
        add_task(id++, op, consume(base+2*j), consume(base+2*j+1))
      end for
    end if
    width ← width / 2
  end for
  n ← n − 2^(logn−1)
  offset ← offset + 2^(logn−1)
  root_ids.push_back(id-1)
end while
if root_ids.size() > 1 then
  count ← 0
  for i ← root_ids.size()-1 to 1 do
    if count = 0 then
      if view.size()  mod 2 = 1 then
        add_task(id++, op, consume(root_ids[i]), view[offset-1])
      else
        add_task(id++, op, consume(root_ids[i-1]), consume(root_ids[i]))
      end if
    else
      add_task(id++, op, root_ids[i-1], id-1)
    end if
  end for
end if
```

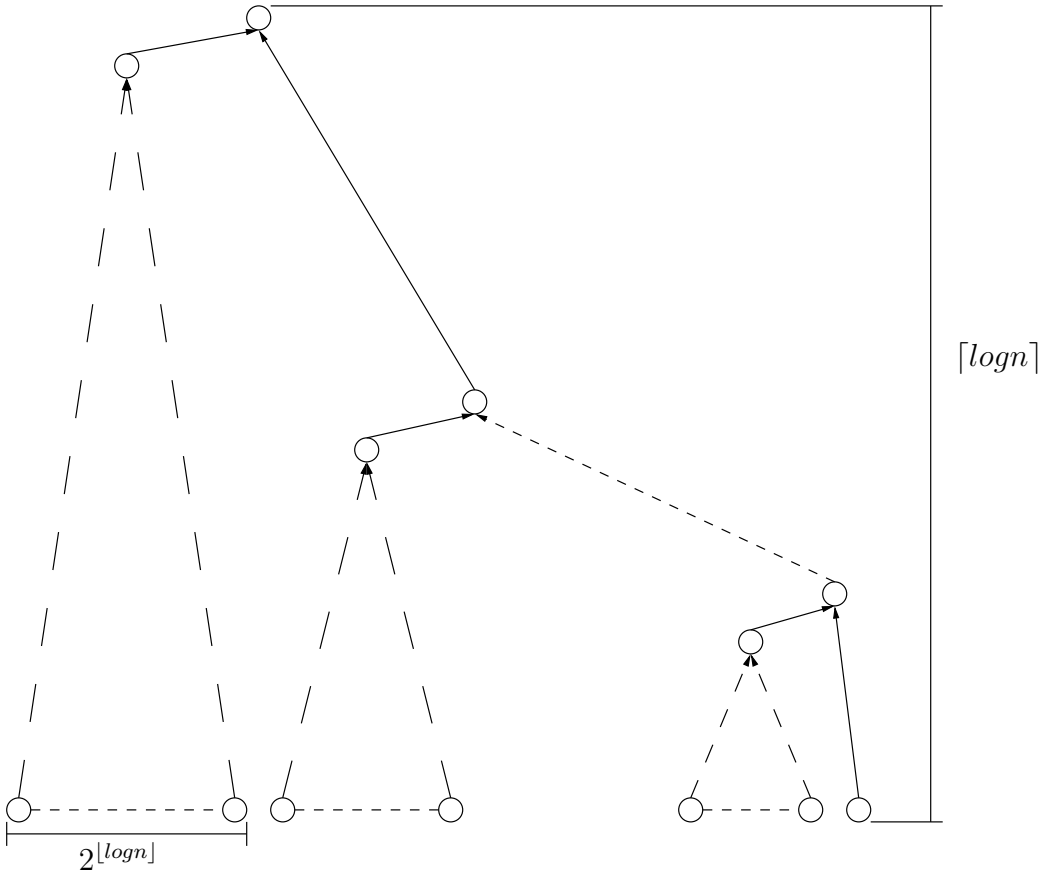


Fig. 8. A task graph to reduce an arbitrary number of elements to a single value.

view specification for the result of a task to be constructed for the `task factory`. The interface for the `task_factory_base` is shown in Figure 10.

It is clear that the `task_factory_base` significantly reduces the burden of writing a `task factory`, and the class should be used when new computation patterns are being expressed as `task factories` in STAPL.

c. Incremental Task Graph Specification

`Task factories` are generative functors that are parameterized by the input data

```
class task_graph_access
{
public:
  template<typename WF, typename ...Args>
  void add_task(std::size_t task_id, WF const& wf,
                std::size_t num_succs,
                Args const&... args) const


  template<typename WF, typename ...Args>
  void add_task(std::size_t task_id, WF const& wf,
                pred_list_t preds, std::size_t num_succs,
                Args const&... args) const


  template<typename WF, typename ...Args>
  void add_task(std::size_t task_id, WF const& wf,
                Args const&... args) const


  template<typename WF, typename ...Args>
  std::size_t add_task(WF const& wf,
                       Args const&... args) const
};
```

Fig. 9. A class for adding tasks to the current task graph.

that the task graph produced will process. The task graph produced by the functor will contain tasks whose number is proportional to the size of the coarsened input data. For example, a reduce task graph that processes 1024 coarsened elements will contain 2047 tasks (remember that the number of vertices in a balanced binary tree of width n is $2n - 1$). Completely generating a task graph before processing it would consume memory proportional to, and possibly greater than, the data being processed. This use of a limited resource in high-performance computing systems is wasteful and can lead to severe performance degradation. For this reason, the task graph has been designed to allow for construction and processing of tasks to be interleaved. This section details the interfaces provided and what a `task factory` developer must consider as they design their factory.

The `PARAGRAPH Executor` facilitates incremental construction of the tasks in the task graph it is processing by executing the task that invokes the `task factory` on

```
template<typename WFReturn>
struct task_factory_base
 : public task_factory_core,
   public task_graph_access
{
  using task_graph_access::add_task;

  //If user codes pattern for incremental generation,
  //set m_finished to false so that operator() will be called more than once.
  task_factory_base(bool b_incremental_generation = true);

  //Construct the view specification using the task graph's result view
  std::pair<result_view_t*, std::size_t>
  consume(std::size_t tid) const;

  void set_result_tid(std::size_t task_id);
};
```

Fig. 10. A base class for `task factories`.

a location only after processing the tasks that were generated by the previous invocation of the `task factory` or finding that the remaining tasks from that invocation have outstanding dependencies that prevent their immediate execution. After each invocation of the `task factory` it is queried to determine if it has finished generating tasks on this location, or if a task that will re-invoke the `task factory` needs to be inserted into the scheduler of the `PARAGRAPH Executor` again. This predictability allows the task factory developer to reason about the state of the task graph at each call and generate an appropriate number of tasks.

A `task factory` that incrementally generates its set of tasks must override the `finished` method of the `task_factory_core` shown in Figure 7. The task graph avoids unnecessary calls to the `task factory` on a processing location by querying the method and only calling the function operator of the task factory if the method returned false. The `finished` method is also used in the termination detection algorithm of the executor processing the task graph. A task specified on one location may be mapped to another processing location for construction and execution (see [27] for details of the task placement algorithm). Therefore the default termination condition

provided by the `PARAGRAPH Executor` for a task graph is that all tasks have been processed and the `task factory` on each processing location reports that it is finished generating tasks.

A `task factory` implementing incremental task specification will likely have internal state to track the input elements for which tasks have been specified. If a task graph is re-executed then it is necessary for the `task factory` to reinitialize its state to allow the entire task graph to be regenerated as if it had not been executed previously. The `task factory` must provide the `reset` method for this purpose. The method is only called at the point where the task graph begins re-execution, immediately before the function operator of the `task factory` is invoked for the first time in the processing of the task graph. All of the `task factories` provided in STAPL for regular computation patterns perform incremental task generation by specifying a fixed percentage (e.g., 10%) of the task graph on each invocation of the factory's function operator. In the future, this value will be adaptable based on the conditions of the processing location as reported by the STAPL run-time system.

C. Regular Task Graphs

There are computation patterns that arise frequently in the development of algorithms. The simplest of these patterns is the application of a function f to an element v to produce a new element w.

$$w = f(v)$$

Extending the pattern such that it accepts a set of input elements V and produces a set of output elements W of equal size results in the *map* pattern.

$$W = f(V) = \{f(v_i) \mid v_i \in V, 0 \leq i < n\}$$

```
template<typename Operation, typename ...Views>
DataFlowView<typename Operation::result_type>
map_func(Operation& op, Views...& views)
```

Fig. 11. Map_func function interface.

The task graph for this pattern is a degenerate graph that contains only vertices and no edges.

A set of commonly occurring task graph patterns has been explored and implemented in STAPL. The patterns provided are map, reduce, and scan. We have anecdotal evidence in our work implementing parallel equivalents of STL algorithms and the NAS benchmarks that with these three patterns a significant number of algorithms can be written. Exceptions are the PDT sweep presented at the end of the chapter and algorithms for graph traversal that use the dynamic task graph creation interfaces described in the next section.

1. Map

The map pattern implemented in STAPL has been generalized beyond what was presented above by extending the function to accept multiple input sets that are all of the same size.

$$W = f(X, Y, Z, \ldots) = \{f(x_i, y_i, z_i, \ldots) \mid x_i \in X, y_i \in Y, z_i \in Z, \ldots, 0 \leq i < n\}$$

The interface of this pattern is realized in STAPL as shown in Figure 11. The name of the pattern in STAPL is `map_func` instead of `map` because the C++ STL has used the name `map` for an associative container that associates a value with the specified key value.

A single view over the output values is produced regardless of the number of input

views. The view returned is a data flow view, which is a `pView` whose container is an `edge container` [27]. The `edge container` stores the results of tasks in the task graph as they finish executing. It forwards the result of each task to all processing locations that are waiting to execute tasks that require the result as an input.

The data flow view allows `map_func` to construct the `PARAGRAPH` for the computation, pass it to the `PARAGRAPH Executor` for processing, and return to the user code before execution of the `PARAGRAPH` begins. When a data element that is not yet available in the data flow view is accessed by the user code its computation will block until the task that produces the desired value is executed. If there is enough computation to do, it is possible that all elements of the data flow view will be available and the execution of the task graph produced with `map_func` will have been executed concurrently with the rest of the application.

There are no dependences between the tasks in the task graph. The task graph can assign arbitrary ids to the tasks, and there is no dependence information required in the specification. The `task factory` used to produce the map task graph can then use the fourth `add_task` method shown in Figure 9. The implementation of the map function and the function operator of the map factory are shown in the listing in Figure 12.

2. Reduce

The reduce `task factory` implemented in STAPL accepts a single coarsened `pView` of elements as input along with the operator that will be used to combine the fine-grain elements. The interface to construct a `PARAGRAPH` to perform the reduce operation is shown in Figure 13.

The task graph outputs a `pView` over a single element that is the result of the reduction of all elements. The reduce factory extends the algorithm shown in Algo-

```
template<typename MapOperator, typename ...Views>
DataFlowView<typename MapOperator::result_type>
map_func(MapOperator& op, Views...& views)
{
  // The function operator passes the PARAGRAPH to the PARAGRAPH Executor
  // for processing and returns the DataFlowView that will be populated
  // with the results of the execution.
  return paragraph<WorkFunction, Views...>(wf, views)();
}

template<typename MapOperator>
template<typename ...Views>
void
map_factory<MapOperator>::operator()(Views...& views)
{
  if (!m_initialized)
  {
    // define a iterator over the elements of each view, iter0...iter(n-1)

    // determine the maximum number of tasks generated on each call
    // and store_this in m_tasks_per_call.
  }

  //partition the id set of the views

  std::size_t task_cnt = 0;
  for (; task_cnt < m_tasks_per_call && !iter0->at_end(); /*increment all id sets*/)
  {
    add_task(m_wf, /* deref of all id sets */);
  }
  if (iter0->at_end())
    this->m_finished = true;
}
```

Fig. 12. Map function and factory function operator implementation.

rithm 1 with logic to perform incremental task generation. As with the map `task factory`, the reduce `task factory` computes the number of tasks that should be specified by a processing location on each invocation. The amount of state needed to implement this is significantly more than what is needed in the case of the map `task factory`.

In the map `task factory` the internal state was the set of iterators to the partitioned input `pViews` that indicated which elements did not yet have tasks specified for them. In the reduce `task factory` it is possible to reach the maximum number of tasks at any point in specifying one of the binary trees or the tasks to combine the

```
template<typename ReduceOp, typename View>
typename DataFlowView<ReduceOp::result_type>
reduce(ReduceOp& op, View& view)
```

Fig. 13. Reduce function interface.

```
template<typename SumOperator, typename typename View>
DataFlowView<typename View::value_type>
scan(BinaryOp& op, View& view)
```

Fig. 14. Scan function interface.

results of the trees. Since the trees are specified one level at a time the state must include the tree the factory is currently specifying, the level within the tree, and the offset within the level.

3. Prefix Scan

The prefix scan pattern, referred to as scan in the algorithmic skeletons literature [8] [40], accepts a coarsened `pView` of elements, $\mathcal{I}$, as input and returns a coarsened data flow view of the same size and type, $\mathcal{O}$, whose fine-grain elements are defined as:

$$\mathcal{O} = \text{scan}(\mathcal{I}) = \{o_0, o_1, ..., o_{n-1}\} = \{o_k \mid o_k = \Sigma_{j=0}^{j=k} \iota_j, 0 \leq k < n\}$$

The operator used to perform the summation for each output element is provided as a parameter to algorithm. The interface for the scan is shown in Figure 14.

While it is possible to compute every output element independently that would require on average $n/2$ applications of the `op` function for each output element, which is suboptimal in terms of the work to be done. Since the input data is distributed, this approach would require a large amount of communication to provide the necessary elements to each processing location, with the last processing location receiving a full

copy of the input data. This amount of memory usage on each processing location is also wasteful.

A work optimal parallel algorithm presented in [41] performs the combination of two elements and uses that result as the input to other combinations. The pattern of this algorithm has been implemented as a `task factory` in STAPL. The task graph for the scan of eight elements is shown in Figure 15. The input elements are represented by the array at the bottom of the figure, and the elements of the data flow view returned as output are represented by the array at the top. The lines represent the flow of values from the input view at the bottom to the inputs of the tasks, and from the outputs of the tasks up to the output view at the top of the figure.

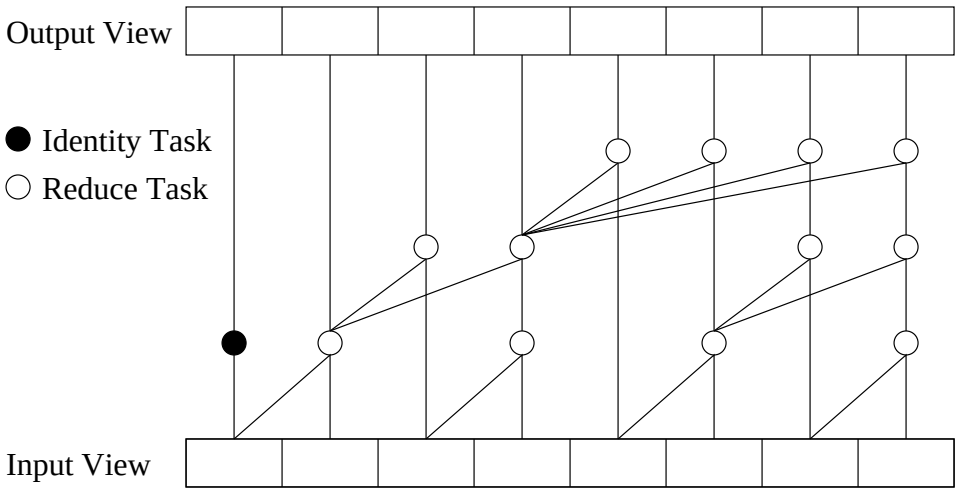


Fig. 15. Task graph for prefix scan of eight elements.

The task graph generated for a number of elements that is not a power of two follows the approach used by the reduce task graph. The largest powers of two that sum to the number of elements is found and a scan tree is generated for each. The result of the combine task that contains all elements of a tree is applied to all the smaller trees. This pattern can be seen in Figure 15 as well. The result of the

task that contains the sum of the first four elements is applied to each element to the right. For larger trees this broadcast of the value affects the scalability of the algorithm negatively, and in the implementation in STAPL it has been replaced by a set of identity tasks that form a broadcast tree to reduce the number of messages that a given processing location must send.

There are two types of tasks in the task graph. The reduce tasks apply the summation operator provided to two input values and the output is forwarded on to the tasks that depend on it. The identity task is needed to copy the first element of the input view to the first element of the output data flow view and to implement the internal broadcast trees.

In the cases of map and reduce, the tasks whose results formed the elements of the data flow view returned were easily defined. The tasks whose results form the data flow view are the identity task, the reduce tasks that make up the left half of the leftmost tree, and the final combine tasks that combine the result of the leftmost tree with the elements of the smaller trees.

D. Dynamic Task Graphs

The previous section presented computation patterns for which the specification of the task graph needs only to know the number of elements in the input views. The structures of the task graphs are regular and do not depend on the values of the data being processed.

There are algorithms whose behavior does depend on the values of the input elements. Consider the example of an algorithm to traverse a graph beginning from a specified set of vertices. The task graph of the algorithm would contain a vertex for each vertex of the input graph, and the dependencies would mirror the connectivity

```
struct dynamic_wf
  : public paragraph_impl::task_graph_access
{
  using task_graph_access::add_task;
};
```

Fig. 16. A structure to facilitate dynamic work function implementation.

of the input graph. It is clear that the construction of the task graph requires performing a computation that is similar in structure to the computation the task graph represents.

Algorithms whose task graph structure depends on the values of the input are referred to as irregular algorithms since the task graph isn't known a priori to contain any regular pattern. Expressing these algorithms as task graphs can be done by providing the ability to add tasks to the task graph dynamically from within the execution of a task of the task graph.

The previous section explained how implementing a `task factory` is simplified by deriving from the `task_graph_access` class shown in Figure 9. This class adds a task to the currently executing task graph. A task in a task graph can access the same interfaces if the function object implementing its operation inherits from the `task_graph_access` class. STAPL attempts to make this facility more visible by providing the `dynamic_wf` structure shown in the listing of Figure 16.

A developer writing a parallel dynamic algorithm can have the operators of their algorithm derive from `dynamic_wf` and have immediate access to the `add_task` functions of `task_graph_access` as members of their own function object. The developer would only need to write a simple `task factory` that would generate the initial tasks of the task graph.

E. Case Study: Specifying a `PARAGRAPH` for sweeps in PDT

We demonstrate the process of developing a `task factory` for a parallel algorithm whose computational pattern doesn't match any of the `task factories` provided by STAPL in this section. The application is a discrete-ordinates particle transport [42] code from the nuclear engineering field.

The problem solved by discrete-ordinates transport applications can be briefly described as follows:

Given:

1. a N-dimensional(2-D or 3-D) spatial domain made of known materials,
2. an initial flow of particles through the domain at a starting time,
3. an initial set of sources generating particles inside the domain, and
4. knowledge about the behavior at the domain boundary.

Compute: the flow of particles at a later point in time for every point in the spatial domain.

In addition to a position in the spatial domain, the particles also have a specific energy level and are traveling in a specific direction. A discrete-ordinates transport application must discretize the spatial domain, energy domain, direction domain, and time domain in order to be able to produce a system of algebraic equations modeling the production and loss of particles in the problem domain. The discretized units of the spatial, energy, and direction domains are referred to as cells, energy groups, and angles, respectively.

Once the problem has been discretized its solution can be approximated through computation. The problem can be represented as a large system of coupled equations

in the general form $\mathbf{A}\psi = q$, and can be solved using direct or iterative methods depending on its size.

The class of problems of interest to the researchers working on a parallel discrete-ordinates particle transport code that is being implemented using STAPL are too large to solve with direct methods. In fact, the size of the matrix is too large to explicitly form in memory. The implementation is a matrix-free implementation. The unknowns of the vector ψ are stored in the elements of the cells of the spatial discretization, one for every energy group. The number of elements within a cell depends on the spatial discretization method that was used. The effect of the different directions is an additive effect, so the same set of unknowns in a cell can be used for all directions. Matrix-vector multiplications in the iterative methods are performed by applying a transport operator to the vector that performs the action of the matrix. This is implemented in the code – which is named PDT – as an ordered traversal (sweep) over the cells in the spatial domain for each direction, and the sweeps are repeated for every energy group. Unless reflective boundary conditions are present, the directions are completely decoupled and the sweeps for all directions can be performed concurrently.

Figure 17 illustrates the discretized spatial, energy, and direction domains for a simple two-dimensional example. The cells of the spatial domain are numbered with a unique id that will correspond to the id of the task that processes the cell in the task graph that represents a sweep. The directions are assigned Latin letters that are used to refer to their task graphs. The energy discretization doesn't impact the task graphs formed because the full set of task graphs will be executed for each element in the energy discretization.

The discretized spatial domain is represented by a STAPL `pGraph` in PDT. The `pGraph` is weighted, undirected, and non-multi-edge. This means that between two vertices there are two edges, one in each direction. Each cell is a vertex in the graph,

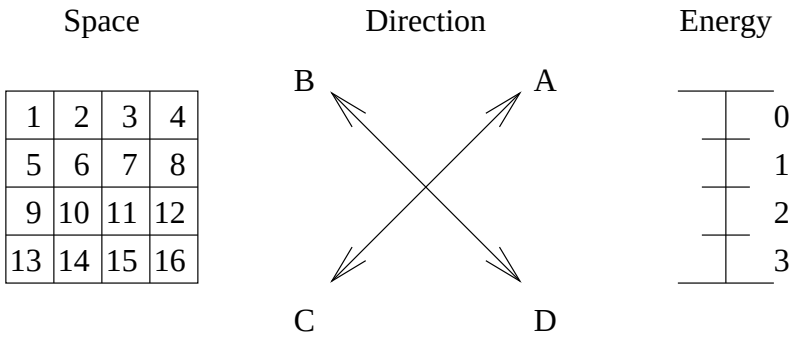


Fig. 17. Discretized domains of a 2D discrete-ordinates particle transport problem.

and the shared surface between two cells, known as the face, is represented by edges in the graph. The weights on the edges include a face normal to indicate the direction of particle flow across a face and the id of the other edge associated with the face (i.e., the "sister" edge). The `pGraph` uses an adjacency list implementation and edges are stored with the vertex that is their source. The id of the "sister" edge allows a computation on a vertex to call a method on an edge to which it doesn't have direct access.

Figure 18 is the `pGraph` that represents the discretized spatial domain in the 2D example of Figure 17. The normals stored on the edge are only shown for four edges. The normals for all edges with the same direction are the same. Note that the normal stored on an edge is the opposite of the direction of the edge (e.g., the edge from vertex 1 to vertex 2 has a normal with a negative x component). This is because the flow of particle information is in the opposite direction of the graph edge.

The coarsening of the views of the spatial, angular, and energy domains is currently dictated by the PDT user. Part of the input passed to PDT during execution is the grouping of angles into angle sets, energy groups into energy group sets, and cells into cellsets. The coarsening of cells into cellsets is done using the `hierarchical_view` of STAPL defined on the `pGraph`. The `hierarchical_view` de-

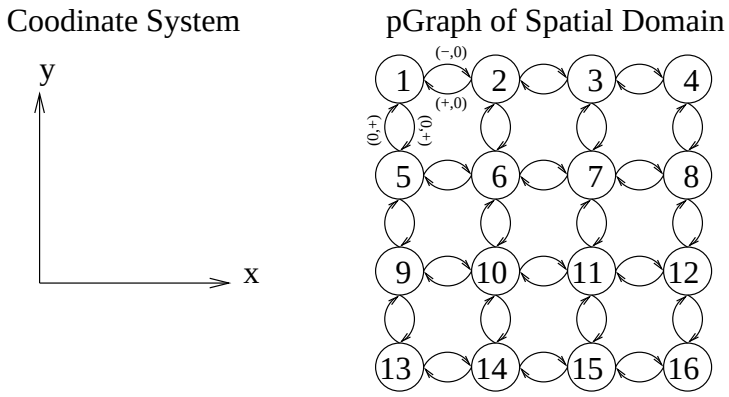


Fig. 18. Coordinate system and `pGraph` representation of the Spatial Domain.

fines a vertex for each grouping of cells into cellsets and associates an id with it as well. The `hierarchical_view` allows edges to be added between these coarse vertices that stores the average normal of the cell faces that is represented by the coarsened face. The result is that in the description of the problem above cells and cellsets can be used interchangeably. The `PARAGRAPH` is unaware of coarsening and represents the tasks of the computation and their dependencies uniformly whether the operations and the `pViews` they are applied to are coarsened or not.

The sweep operation over a `pGraph` can be described as:

1. Given an angle set from the coarsening of the discretized direction space find the set of cellset vertices that have no incoming edges whose normal is in the same direction (i.e., the dot product of the normal on the edge with the average angle of the angle set is positive).

2. Compute the particle flow information for each cell in each of these cellsets (applying this same algorithm on the cells within a cellset), returning the flow information across the faces of the cell.

3. Find the set of vertices whose incoming edges with a normal that produces a

positive dot product with the angle have all had their base vertex processed.

4. Repeat Steps 2 and 3 for the set of vertices that have just had their base vertices processed until there are no vertices left to process.

Implementation of this process as a `task factory` in STAPL is straightforward. Each processing location will construct the tasks for the cells that are stored locally. Each task created will be assigned a task id that matches the vertex id it is processing. The edge information contains the source and destination vertex ids, so for each vertex iterating through the set of edges that are rooted at the vertex, computing a dot product with the angle given, and building a list of the edge destination vertex ids for all edges with a positive dot product will produce the set of predecessor task ids for the task being created. The constructor and function operator of the `task factory` are listed in Figure 19. In order to keep the code focused on the creation of the task graph, the `task factory` listing does not perform incremental task graph construction. The code assumes that the `dotproduct` function has been implemented for the structure used to represent angles. Given the four angles in the direction space in our example and instantiating four task graphs, one for each sweep, produces the set of task graphs shown in Figure 20.

In the example above the cells and cellsets of the spatial domain are convex. When working with arbitrary discretizations of the spatial domain or with deformed discretizations the cells may not be convex. These cells are referred to as "reentrant" becase a sweep crossing through the nonconvex region would enter a cell multiple times. The task dependence graph in this case contains a cycle because the task processing the nonconvex cell would be a predecessor and successor of the cell that occupies the nonconvex region. These cycles must be broken in order to allow the `PARAGRAPH` to execute to completion. This is usually done by performing a cycle

detection algorithm [43] and removing the edge in the cycle that is closest to parallel with the sweep direction from consideration by the sweep `task factory`. An edge parallel with the sweep direction represents a face parallel to the sweep where few particles traveling in the sweep direction would cross. Removing the edge from the task dependence graph allows a task to process a cell using outdated information and allows the sweep computation to complete. In PDT, cycle detection is performed on the `pGraph` of the spatial domain before the sweep `PARAGRAPHs` are instantiated.

```
template<typename CellSetOperator, typename Angle>
pdt_sweep_factory::pdt_sweep_factory(CellSetOperator const& op, Angle const& a)
  : m_operator(op), m_sweep_angle(a)
{}

template<typename CellSetOperator, typename Angle>
template<typename HierarchicalGraphView>
void
pdt_sweep_factory<CellSetOperator, Angle>
::operator()(HierarchicalGraphView const& cellset_graph)
{
  //Scan the local set of vertices.
  typename GraphView::vertex_iterator vi = cellset_graph.vertices.begin();
  for(; vi != cellset_graph.vertices.end(); ++vi)
  {
    std::vector<std::size_t> predecessors;
    std::size_t num_successors = 0;

    //Scan the set of edges associated with this cell.
    typename GraphView::edge_iterator ei = (*vi).begin();
    for(; ei != (*vi).end(); ++ei)
    {
      //Check if the edge provides information to the task
      double dotprod = dotproduct((*ei).property().facenormal(), m_sweep_angle);
      if (dotprod > 0)
      {
        //This edge is input for the task
        predecessors.push_back((*ei).target());
      }
      else if (dotprod < 0)
      {
        //The task will write particle flow information to the sister edge.
        ++num_successors;
      }
    }

    //Add task with the vertex's id as task id.
    switch (predecessors.size())
    {
      case 0: // no predecessors
        add_task((*vi).descriptor(), m_operator, num_successors,
             std::make_pair(&cellset_graph, (*vi).descriptor()));
        break;
      case 1: // one predecessor
        add_task((*vi).descriptor(), m_operator, num_successors,
             std::make_pair(&cellset_graph, (*vi).descriptor()),
             consume(predecessors[0]));
        break;
      case 2: // two predecessors
        add_task((*vi).descriptor(), m_operator, num_successors,
             std::make_pair(&cellset_graph, (*vi).descriptor()),
             consume(predecessors[0]), consume(predecessors[1]));
        break;

      // The number of cases required is equal to the maximum number of faces
      // of a cell that can be exposed to a sweep direction.
      // E.g., for regular hexahedra three would be required.
    }
  }
}
```

Fig. 19. `task factory` constructor and function operator for PDT sweeps.

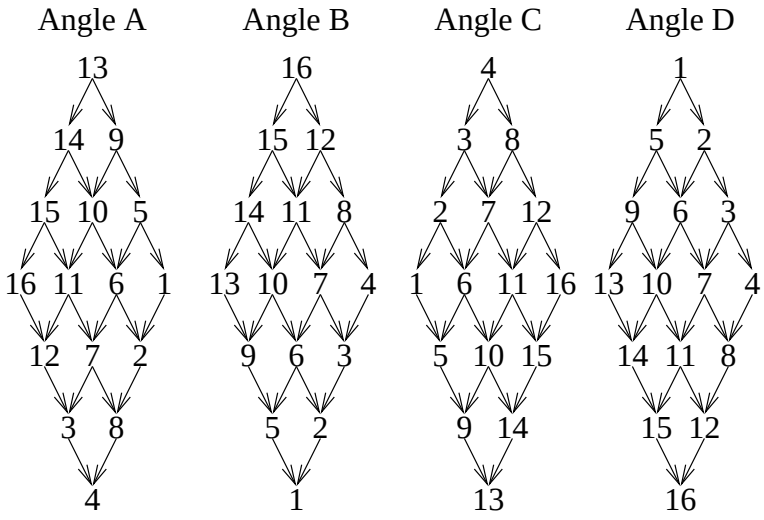


Fig. 20. Task graphs generated for the set of directions and the spatial domain.

CHAPTER V

PARAGRAPH COMPOSITION

The previous chapter demonstrated how a developer can express a simple parallel algorithm as a task graph using the PARAGRAPH. Parallel applications, and some parallel algorithms for tasks such as sorting, are made up of multiple calls to these basic algorithms. Each PARAGRAPH used in these more complex parallel algorithms and applications can be seen as a task that performs a computation. In order to provide a consistent development environment and expose the maximum amount of parallelism available in an application we have implemented composition operators for PARAGRAPH sequences and a loop construct that allow the developer to think of each PARAGRAPH as a task in a higher level task graph.

A. Sequence Composition

The composition of sequences of PARAGRAPHs differs from the composition of a sequential function in that independent PARAGRAPHs should be executed concurrently instead of in the arbitrary order that they were written in the code of the application. Only when PARAGRAPHs have a producer-consumer relationship between them should their execution be restricted. Full sequential ordering – requiring that all tasks of a PARAGRAPH producing a result be executed before any task of the PARAGRAPH consuming the result – is over-constraining the execution in most cases. Figure 21 shows the dependencies that are needed between the tasks of two PARAGRAPHs performing map computations that have a producer-consumer relationship.

In Figure 21 it is clear that a task in the PARAGRAPH that squares each element of the input view should only wait on the value to be produced by a single task in the PARAGRAPH generating random elements. This task-wise specification of the

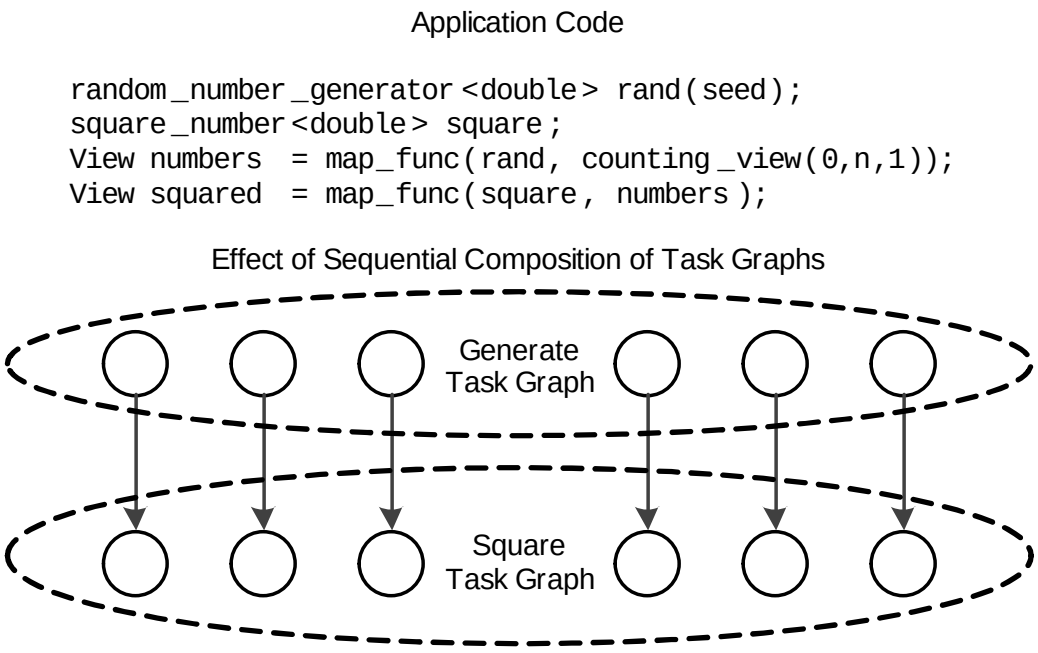


Fig. 21. Composition of two task graphs that have a producer-consumer relationship.

sequential composition allows for more flexibility in scheduling the tasks of the two task graphs and only constrains the execution order where it is absolutely necessary for correctness.

In cases where the PARAGRAPHs are independent computations the ordering provided by the code implementing the parallel algorithm is an artifact of the expression of the application in a sequential programming language, which requires a function to be written as a sequence of instructions. The composition framework only needs to make both PARAGRAPHs available for processing by the PARAGRAPH Executor.

The sequence composition operators we provide are implicit. A producer-consumer relationship between two PARAGRAPHs is specified by passing the data flow view returned by the producing PARAGRAPH as an argument to the consumer PARAGRAPH. If a PARAGRAPH is not passed the data flow view of another PARAGRAPH then the PARAGRAPHs are independent and can be executed concurrently.

A possible future extension of these sequence composition operators is the ability to specify that two PARAGRAPHs should be executed atomically with respect to one another. Atomic processing of PARAGRAPHs that share a producer-consumer relationship would result in the full sequential composition that was referred to at the beginning of the section as over-constraining. Atomic processing of independent PARAGRAPHs only prevents the interleaving of the tasks of the two PARAGRAPHs, it does not specify which of the PARAGRAPHs would be executed first.

1. STAPL Implementation

There are two key aspects of the PARAGRAPH implementation in STAPL that make the implicit composition of PARAGRAPHs possible. First, the PARAGRAPH function operator is non-blocking, and allows for multiple PARAGRAPHs to be made available to the PARAGRAPH Executor. Second, the function operator returns a data flow view, which is a pView whose domain may be known but whose elements are not available when the data flow view is returned from the PARAGRAPH.

The implementation of the PARAGRAPH's non-blocking function operator creates an initial task that will invoke the task factory with the pViews that were provided to the PARAGRAPH constructor. It then adds the PARAGRAPH to the set of available PARAGRAPHs for the PARAGRAPH Executor. The function operator then exits, returning a data flow view that will contain the results of the computation as they become available, and execution of the code continues in the function that called the PARAGRAPH function operator. The prototypes of the PARAGRAPH function operator are shown in Figure 22.

The optional integer parameter of the function operator is used to specify the priority of the PARAGRAPH relative to any other PARAGRAPHs in the parallel application, and is used when a priority scheduling policy has been specified for the PARAGRAPH

```
// Return is the type of element populating the df_view.
// The type is computed when the paragraph is instantiated.

df_view<Return>::reference
operator()(int priority = 0);

template<typename Scheduler>
df_view<Return>::reference
operator()(Scheduler scheduler, int priority = 0);
```

Fig. 22. Prototype of the PARAGRAPH function operator.

Executor, or for any executor processing nested PARAGRAPHs. The scheduler parameter accepted by the second function operator interface allows the developer to specify the scheduling policy to be applied between the tasks of the PARAGRAPH instance. The PARAGRAPH function operator constructs a single_executor instance that will be responsible for processing the tasks of the PARAGRAPH. This single_executor instance is in turn what is processed by the PARAGRAPH Executor using the priority parameter provided to the function operator.

The PARAGRAPH Executor contains an executor that is a persistent object that processes PARAGRAPHs it has been given according to a scheduling policy that was specified when the PARAGRAPH Executor was initialized. The policies available in the current system are a round-robin scheduler and a priority scheduler. When the executor is entered it invokes the function operator of a single_executor object for each available PARAGRAPH according to the scheduling policy in effect. The PARAGRAPH Executor is entered by an explicit call to its drain method, when a scheduling point in the run-time system is encountered (e.g., when waiting for the result of a remote method invocation), or when the threshold on the number of PARAGRAPHs that are being processed by the PARAGRAPH Executor is reached.

The data flow view returned by the PARAGRAPH function operator may be fully specified from the point of view that its container, operations, mapping function, and

domain are all known. What is unique in this situation is that the elements the `data flow view` represents may not exist when it is used to create a `PARAGRAPH`. If the domain of the `data flow view` is not known the coarsening of the `pViews` in the `PARAGRAPH` is deferred and the `task factory` task of the `PARAGRAPH` detects that the elements of the `pView` are not ready and therefore the `task factory` task cannot be executed yet. In this case the composition of the `PARAGRAPHs` is such that one `PARAGRAPH` will completely execute before any `PARAGRAPH` consuming its result. This over-constrained case is the exception, and is one that we have yet to encounter in practice. In most cases the domain of the `data flow view` produced by a `PARAGRAPH` can be deduced from the input `pViews` of the `PARAGRAPH` and the `task factory` used (e.g. a `PARAGRAPH` applying a map operation on `pViews` of size n returns a `data flow view` whose domain is $[0, n)$).

If the domain of the `data flow view` is known, the `task factory` task of a `PARAGRAPH` can execute and generate tasks that use the elements of a `data flow view` as parameters to the task's work function. The `data flow view` is queried to determine if the value of the element is available. If it is, then the task is created with only the dependence information from the `PARAGRAPH` of which it is part. If the `data flow view` element is not available, then the task has additional predecessor information added to it. When the value becomes available in the `data flow view` the task will be notified, at which time it can be made available for execution if all of its other predecessors have finished execution. The `data flow view` allows specification of the exact dependencies between two `PARAGRAPHs` by allowing individual tasks of a `PARAGRAPH` to depend only on the element of the `data flow view` that they need, which is generated by a single task in the `PARAGRAPH` that produces the `data flow view`.

2. Case Study: Specification of NAS CG

The utility of the implicit composition operators described above is demonstrated here by considering the body of the `conjugate_gradient` function of the NAS CG benchmark [44] with its loop removed. The sequence of operations is shown in Figure 23.

$$
\begin{aligned}
z &= 0 \\
r &= x \\
\rho &= r^T r \\
p &= r \\
q &= Ap \\
\alpha &= \rho/(p^T q) \\
z &= z + \alpha p \\
\rho_0 &= \rho \\
r &= r - \alpha q \\
\rho &= r^T r \\
\beta &= \rho/\rho_0 \\
p &= r + \beta p \\
||r|| &= ||x - Az||
\end{aligned}
$$

Fig. 23. Simplified sequence of operations from NAS CG.

Each mathematical operation in the sequence is represented by a `PARAGRAPH`. For example, $r^T r$ is implemented using the STAPL `pAlgorithm inner_product` and passing the `pView` of r to both arguments of the `pAlgorithm`. This creates and executes a `PARAGRAPH` that is a map-reduce that returns a `data flow view` over the value of the inner product. Figure 24 is the composed task graph of the operations.

Each node in the graph is labeled with the operation the task performs, and the edges between the nodes show the dependencies that will be enforced by the `data flow views` in the implementation of the sequence.

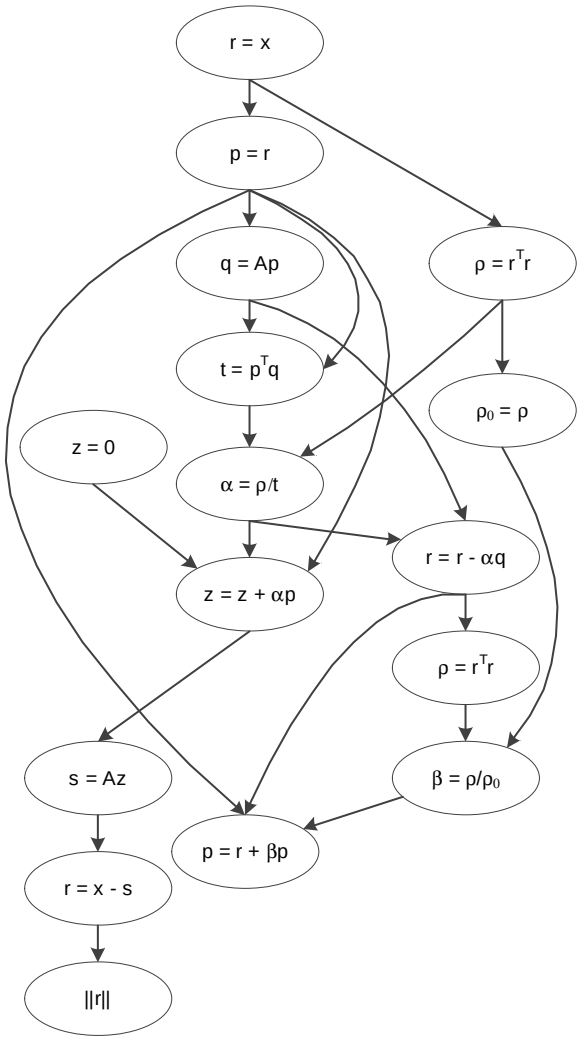


Fig. 24. Task graph of the simplified NAS CG sequence.

The graph in Figure 24 shows that after the copy of the elements of x in to `pView` r there is some parallelism between the tasks, but the computation is dominated by the flow of values from that copy down through the middle of the figure and ending

with the computation of the euclidean normal of the elements of r. The number of producer-consumer relations and the large variance in the distance along the critical path from producer to consumer (e.g., the consumers of the first task that produces p as a copy of r) reinforce the decision to use implicit composition operators to compose sequences of `PARAGRAPHs` into a higher level task graph.

B. Repetition Composition

The map `task factory` presented in the previous chapter provides the application of a function to each element in a `pView`, which can be seen as the repeated execution of the function over data. The function applied is implemented as a basic work function [27], and any `PARAGRAPHs` in the function are applied to the limited scope of the individual elements of the input `pViews`, resulting in nested parallelism. The repetition composition operator presented in this section differs in that it implements the repeated execution of `PARAGRAPHs` over time on the entirety of the input `pViews`. As such, the iterations of the repetition logically run across all processing locations instead of on a single `pView` element on a single processing location.

The repetition operator we provide handles `PARAGRAPH` composition when the bounds of the repetition are known. A set of statements that can include `PARAGRAPHs` is set to be executed a fixed number of times, and the execution of the set may depend on the value of the variable that represents the current iteration.

1. STAPL Implementation

The current implementation of the loop construct uses the sequence composition operators from the previous section. The interface is shown in Figure 25. The work function that represents the body of the for loop accepts the current iteration value

```
template <typename BodyWorkFunction, typename ...Views>
auto
do_loop(int iterations, BodyWorkFunction& body, Views&... views)
  -> decltype(body(iterations, views));
```

Fig. 25. Prototype of the for loop composition operator

and the set of input `pViews`. The work function returns a tuple of `data flow views` the represent the modified input `pViews`. The execution of the construct effectively unrolls the loop completely, making all `PARAGRAPHs` in the loop available for sequential composition. The threshold on the number of `PARAGRAPHs` that can be in the `PARAGRAPH Executor` limits the amount of memory consumed by the construct.

Our future work includes implementing the loop and other control constructs (e.g. switch and conditional repetition) as `PARAGRAPHs`.

C. Case Studies

When discussing the composition of `PARAGRAPHs` to form a parallel application a graphical representation is useful. National Instruments Labview [45] influenced our choice of composition operators, and Intel Concurrent Collections' [10] use of a "white-board representation" of the different collections they provide has emphasized its importance. We do not have a tool to translate from the graphical representations presented below like Labview and Concurrent Collections provide. Our graphical representation is merely to facilitate discussion at this point. Future work may include the development of a tool that allows graphical development of STAPL applications.

In our representation, a function is represented by a simple rectangle with the name of the function in it. These simple boxes can be used inside other boxes, and then defined as a larger box that itself uses other boxes as its implementation. Figure 26 demonstrates this with a function named foobar whose implementation is

the sequence of foo and bar.

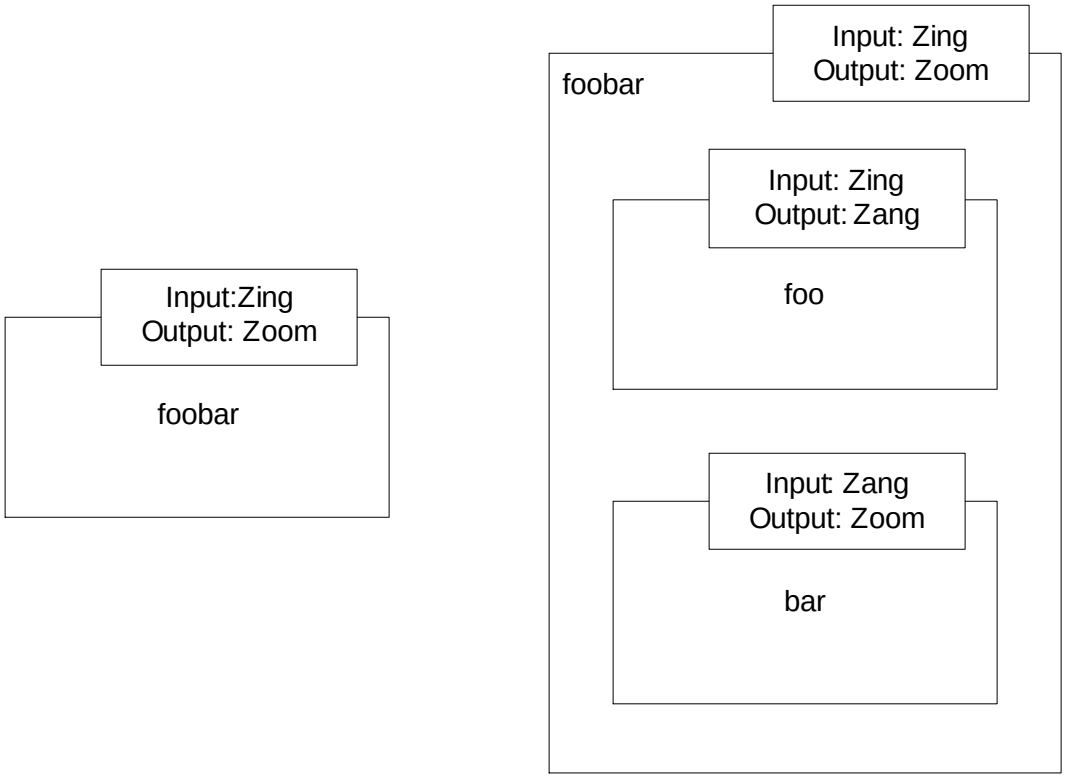


Fig. 26. Graphical representation of function and its definition.

The `pViews` accepted by the function as input and the `pView` returned as output are named in the small rectangle that decorates the top right corner of the function's box. In the definition of Figure 26 we can determine that the `foo` and `bar` functions will be composed using the sequence operator because `bar` accepts as input the `pView Zang` that is produced as the output of `foo`. The `pViews` are named in the figure for illustrative purposes only. In the implementation of a generic programming library such a STAPL a function or `PARAGRAPH` names its inputs as it deems appropriate and doesn't explicitly name its output. The producer-consumer relationship is established when an instance of a `data flow view` is passed from the output of a `PARAGRAPH` instance to the input parameters of another `PARAGRAPH` instance.

The graphical representations of the repetition composition operator is shown in Figure 27.

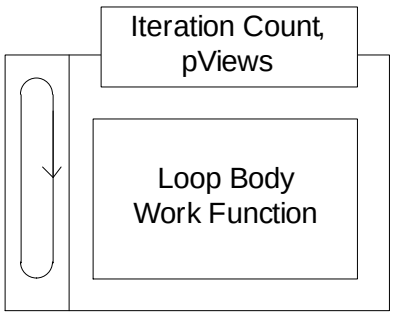


Fig. 27. Graphical representation of the repetition composition operator.

The repetition composition operator uses an ellipse with an arrow on it in the sidebar on the left to indicate that the operator is a repetition operator. The operator shows the work function representing the loop body as a process rectangle inside the main area of the operator's figure. The operator in Figure 27 shows its inputs of the iteration count and the `pViews` of the data to be processed in the rectangle in the top right corner of the figure.

Finally, `PARAGRAPHs` that use the task factories provided in STAPL are represented by the process blocks shown in Figure 28.

When discussing the composition of a parallel algorithm, a unique representation for each `PARAGRAPH` using a particular `task factory` is useful because the `task factory` being used defines the size of the output `data flow view`. You can see that Figures 28(a) and (c) accept either a set of views (noted by the plural use of `pViews`) or a single `pView`, respectively, and return a `data flow view` whose size is the same. Figures 28(b) and (d), on the other hand, accept a set of `pViews` or a single `pView`, respectively, and produce a `data flow view` of a single element as a result. The factories in use by each `PARAGRAPH` are `map` (a), `map-reduce` (b), `scan` (c), and

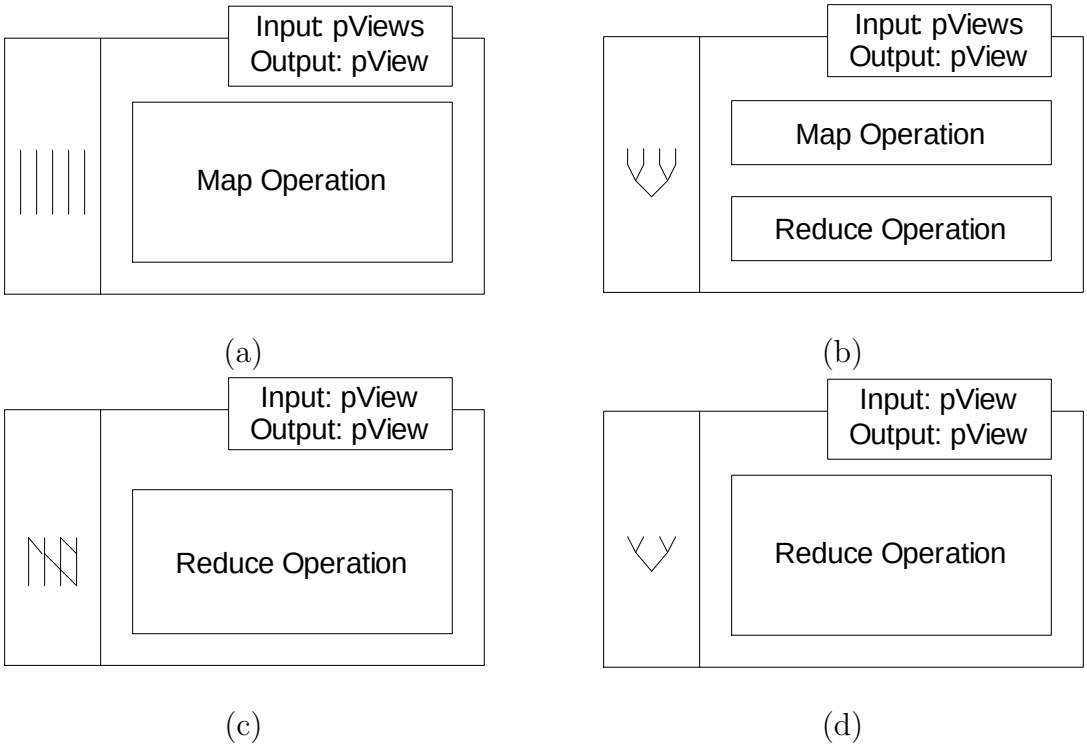


Fig. 28. Graphical representation of `PARAGRAPHs` using the task factories provided in STAPL.

`reduce` (d). The difference in the symbol used for `map-reduce` and `reduce` is the `map-reduce` symbol as long parallel lines at the top leading into the binary reduction tree. These lines represent the execution of the map operation, as the long parallel lines on the left of Figure 28(a) do for `map`.

Like the repetition composition operator in Figure 27 the operation performed by the `PARAGRAPH` process is shown in the center of the process block. In the case of `map-reduce` two operators are provided. Each of these operators is the fine-grained operation of the computation. Coarsening is performed on the input `pViews` if the `task factory` allows it. In these cases the operators provided by the code executing the `PARAGRAPH` are transformed appropriately to match the data coarsening. The code calling the `PARAGRAPH` is unaware of this transformation.

```
//A priority for each sweep direction is computed on each location.
//A sweep paragraph will have different priorities on different
//locations in the system.

//AngleSet is an aggregation of the discretized directions
//All angles in an angle set can use the same sweep.
for (AngleSet& angleset : anglesets)
{
  //sweep_factory_type is an instantiation of the factory described in
  //Section 4.5 with the spatial discretization (solver) method to use and
  //the type of the graph.
  //
  //cellset_view_type is a coarsened view of the graph that allows each task
  //in the sweep to process several cells.  This is explicit pView coarsening.
  paragraph<sweep_factory_type, cellset_view_type>* sweep =
    new paragraph<sweep_factory_type, cellset_view_type>
      (sweep_factory_type(energy_level, angleset.primary_direction,
                          problem_input, problem_kind),
       cellset_view)(priority);
}
```

Fig. 29. Concurrent execution of independent sweeps in PDT

1. Composition of Independent Sweeps in PDT

Chapter IV Section E introduced the PDT parallel application for particle transport that is written using STAPL. In that section we demonstrated how a `PARAGRAPH` for individual sweeps of the spatial domain can be expressed by developing a new `task factory` that specifies the dependencies between the tasks of a sweep `PARAGRAPH` based on the connectivity of the discretized spatial domain. It was noted there that unless the spatial domain has a reflective boundary on the spatial domain the individual sweeps are independent and can be processed concurrently.

The discretized direction information is replicated across all processing locations. Concurrent execution of the sweeps is achieved using the implicit parallel composition operator described in Section A of this chapter. The code is shown in the listing in Figure 29.

The sweep `PARAGRAPHs` in Figure 29 use the function operator that accepts the priority to be used when scheduling the different sweep `PARAGRAPHs` in the `PARAGRAPH`

`Executor`. The `PARAGRAPH Executor` assumes ownership of the `PARAGRAPHs` as part of the implementation of their non-blocking semantic. When a sweep `PARAGRAPH` has finished execution the `PARAGRAPH Executor` will destroy it.

2. NAS CG

Figure 23 provided a listing of the operations of the NAS CG benchmark with the loops removed to illustrate the sequence composition. The full set of operations including loops is shown in Algorithm 2.

Algorithm 2 NAS CG(λ, iterations)

{All vectors are of size n}
$x = 1$
for i = 1 to iterations **do**
  $z = 0$
  $r = x$
  $p = r$
  $\rho = r^T r$
  for j = 1 to 25 **do**
    $q = Ap$
    $\alpha = \rho / p^T q$
    $z = z + \alpha p$
    $r = r - \alpha q$
    $\rho_0 = \rho$
    $\rho = r^T r$
    $\beta = \rho / \rho_0$
    $p = r + \beta p$
  end for
  $||r|| = ||x - Az||$
  $\zeta = \lambda + 1/x^T z$
  $x = 1/||z|| * z$
end for

An implementation of the conjugate gradient method used in a production application would replace the inner for loop with a loop until convergence. The benchmark was designed to fix the number of operations required by the benchmark in order to make comparison of timing results across platforms easier. Using operator overloading

to define matrix-vector multiplication and vector scaling operations we can express the set of operations above very easily using the constructs presented in this chapter.

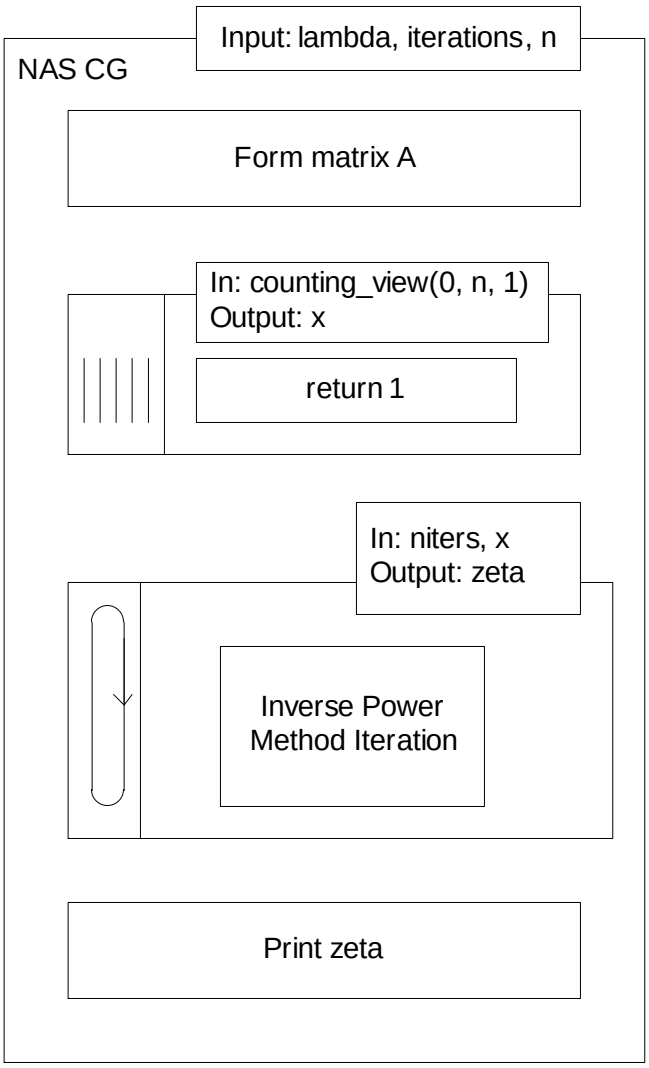


Fig. 30. The NAS CG benchmark main body.

Figure 30 shows the block representation of the main body of the CG benchmark and Figure 31 shows the work function of the loop it contains. Examining the inputs and outputs specified for the steps of the main body one can determine that the sequential composition operator will be used to completely order the execution of the

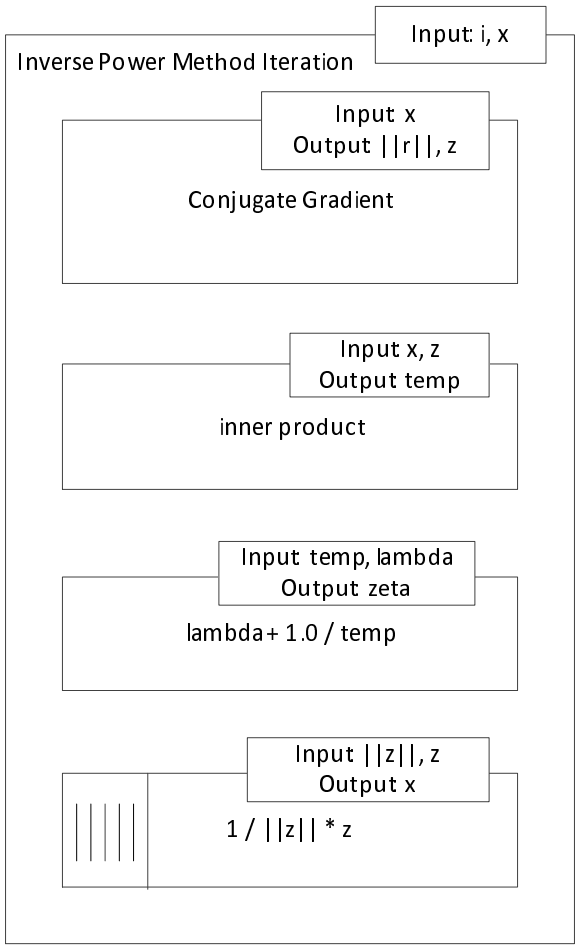


Fig. 31. The loop body work function of the loop in the NAS CG main body.

benchmark. In the work function of the iteration there are three computation steps that can be further expanded.

Figure 32, Figure 33 and Figure 34 show the representation of the conjugate gradient method, the work function used in its iteration, and the representation of the computation needed to find the euclidean norm of the residual vector, respectively.

Figure 35 provides the graphical representation of the matrix-vector multiplication and inner product operations that are needed by the benchmark. The matrix-vector operation is an example of nested parallelism in our representation. Each task

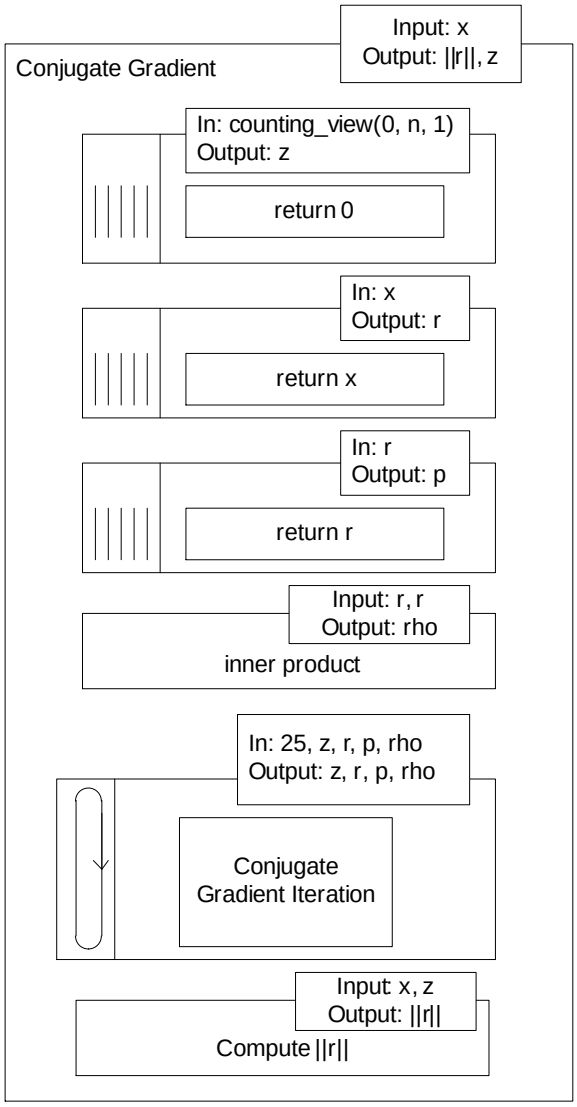


Fig. 32. The NAS CG conjugate gradient implementation.

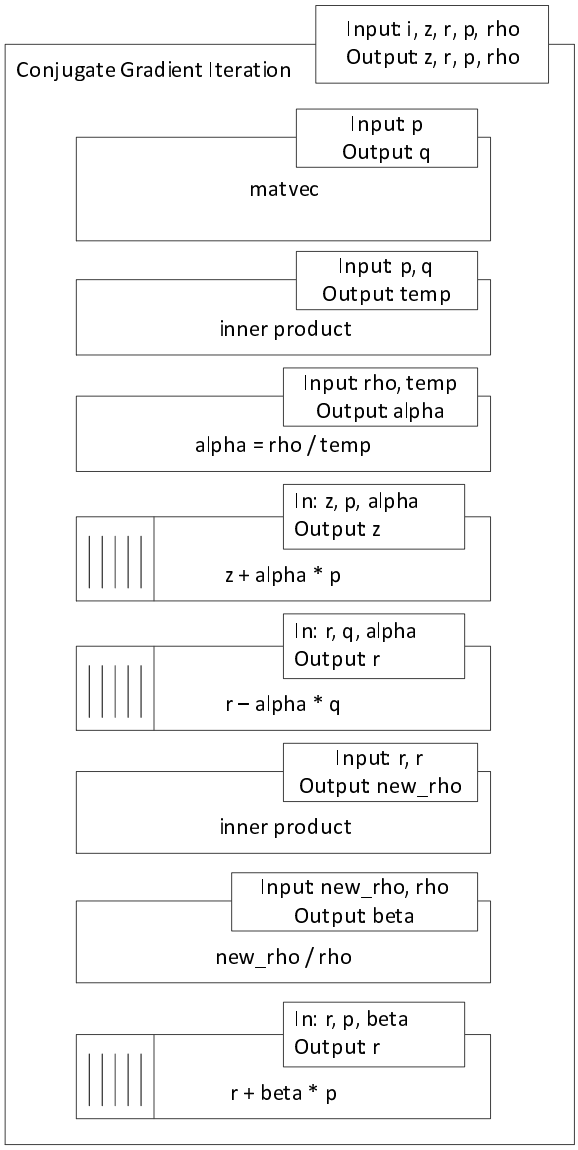


Fig. 33. The NAS CG conjugate gradient loop work function.

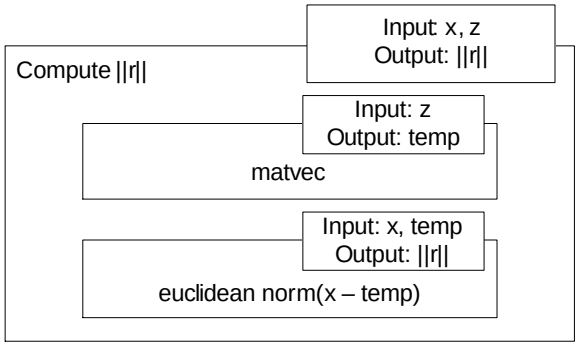


Fig. 34. The NAS CG method to compute $||r||$.

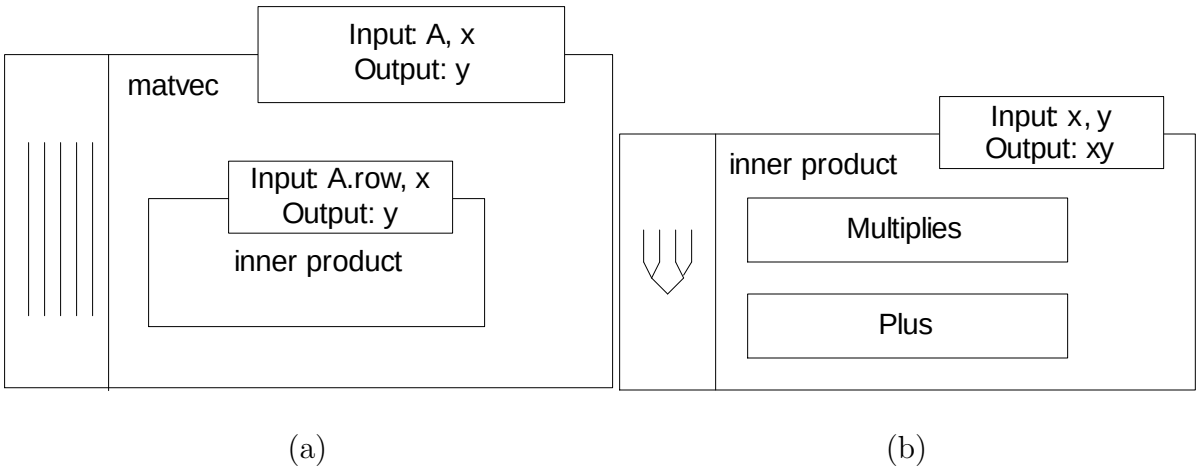


Fig. 35. Matrix-vector multiplication and inner product `PARAGRAPHs`.

of the map operation the matvec `PARAGRAPH` performs uses an instance of the inner product `PARAGRAPH`, which is a map reduce operation.

The listing of the main body of the implementation is given in Figure 36.

The implementation of the conjugate gradient method and its loop are shown in Figure 37.

The implicit composition of the sequences allow the bodies of each loop to execute with as much task parallelism as possible. The dependence graph in Figure 24 shows that there are three parallel tasks that can be executed concurrently. The ini-

```
template<typename MatrixView>
struct ip_loop_wf
{
  MatrixView&   A;
  double        lambda;
  int           n_iters;

  ip_loop_wf(MatrixView& a, double const& l, int iters)
    : A(a), lambda(l), n_iters(iters)
  { }

  template<typename IterRef, typename View1D, typename ZetaRef>
  auto operator()(IterRef it, View1D x, ZetaRef) const
    -> decltype(make_tuple(x, lambda + 1.0 / inner_product(x, x)))
  {
    auto cg_ret   = conjugate_gradient(A, x);
    auto norm_r   = get<0>(cg_ret);
    auto z        = get<1>(cg_ret);
    auto zeta     = lambda + 1.0 / inner_product(x, z);
    print_iteration(it + 1, n_iters, norm_r, zeta);
    if (it == n_iters - 1)
      return make_tuple(x, zeta);
    auto new_x    = 1.0 / euclidean_norm(z) * z;
    return make_tuple(new_x, zeta);
  }
}; // struct ip_loop_wf

void stapl_main(int argc, char** argv)
{
  // read input for benchmark traits
  // setup matrix A and vector x
  ip_loop_wf<view_t> wf(A, traits.lambda);
  auto zeta = get<1>(do_loop(traits.niter,1, wf, x, 0.0));
  runtime::anonymous_executor.drain();
  //print results and exit
}
```

Fig. 36. Implementation of the main body of the NAS CG benchmark

tialization of r and z can execute concurrently. Then the computation of ρ can overlap with the sequence of operations from the initialization of p down the computation of α where ρ is needed.

```
template<typename View2D, typename View1D>
struct cg_loop_wf
{
private:
  View2D&  A;
  int      n_iters;

public:
  cg_loop_wf(View2D& v1, int iters)
    : A(v1), n_iters(iters)
  { }

  template<typename IterRef, typename VecView, typename RhoRef>
  auto
  operator()(IterRef n, VecView z, VecView r, VecView p, RhoRef rho) const
    -> decltype(make_tuple(z, r, p, rho))
  {
    auto q       = A * p;
    auto alpha   = rho / inner_product(p, q);
    auto new_z   = z + alpha * p;
    if (n == n_iters - 1 )
      return make_tuple(new_z, r, p, rho);
    auto new_r   = r - alpha * q;
    auto new_rho = inner_product(new_r, new_r);
    auto beta    = new_rho / rho;
    auto new_p   = new_r + beta * p;
    return make_tuple(new_z, new_r, new_p, new_rho);
  }
}; // struct cg_loop_wf

template<typename View2D, typename View1D>
auto conjugate_gradient(View2D& A, View1D& x)
  -> decltype(make_tuple(euclidean_norm(x), x))
{
  // set initial values of loop variants
  auto z = vector_fill_n(0.0, x.size());
  auto r = x;
  auto p = r;
  auto rho = inner_product(r, r);
  cg_loop_wf<View2D, View1D> wf(A, x);
  // run loop and extract z output value
  auto final_z = get<0>(do_loop(25, wf, z, r, p, rho));
  // compute ||r|| = ||x - Az||
  return make_tuple(
     euclidean_norm(x - A * final_z),
     final_z
  );
}
```

Fig. 37. Implementation of the conjugate gradient method of NAS CG

CHAPTER VI

EXPERIMENTAL EVALUATION

We evaluate the performance of our implementation of the `PARAGRAPH` and underlying `PARAGRAPH Executor` [27] using several benchmarks of increasing complexity. The first set of experiments evaluate the performance of individual `PARAGRAPHs` used to implement parallel generic algorithms in STAPL, all of which are parallel equivalents of STL algorithms. The second set of experiments evaluates the performance of the STAPL implementations of EP and CG, two benchmarks of the NAS Parallel Benchmarks [44]. The CG implementation allows us to demonstrate the performance capabilities of the `PARAGRAPH` composition operators and the `data flow view` that makes it possible. Finally, we evaluate the implementation of the sweep operation of PDT in the case of non-reflective boundary conditions.

The results of the string matching experiment presented in this chapter were originally published in [21].

A. Experimental Setup

Our experiments are conducted on three parallel machines with different processor architectures and network interconnects. These machines include a 32,288 core Cray XT4(CRAY 4) [46] and a 153,216 core Cray XE6(CRAY 6) [47], both of which are available at NERSC. We also employed a 832 core Power5 Cluster(POWER 5) [48] available at Texas A&M University.

The CRAY 4 has 9,572 compute nodes, each with one quad-core AMD 'Budapest' processor running at 2.3 GHz. Each node has 8 GB of DDR3 800 MHz memory with 7.38 GB usable by the user application, allowing 1.85 GB of memory per core when the node is fully utilized. The compute nodes run the "Cray Linux

Environment" that supports a limited number of system calls and disallows the use of dynamically loaded libraries. The system interconnect forms a 3D torus that utilizes a SeaStar2 router with compute nodes connected through Hypertransport.

The CRAY 6 has 6,384 compute nodes each with two twelve-core AMD 'Magny-Cours' processors running at 2.3GHz and total of 32GB of memory. These compute nodes are also connected with a 3D torus via Cray's 'Gemini' interconnect. The system also runs the "Cray Linux Environment" on the compute nodes.

The POWER 5 has 52 compute nodes, each with 8 dual-core IBM Power5+ cores running at 1.9 GHz. Forty-nine of the nodes have 32 GB of DDR2 533MHz memory with 25 GB available for user applications, allowing 1.56 GB of memory per core when the node is fully utilized. The compute nodes run IBM's AIX operating system. Forty-eight of the nodes are connected together by a two-plane high-performance switch interconnect.

In all experiments, a location contains a single processor core, and the terms can be used interchangeably.

The experiments for the parallel generic algorithms and the PDT application are conducted using weak scaling. In this setup the number of elements to be processed by a core is kept constant as the number of cores is increased. This results in the size of the problem being solved increasing in proportion to the number of cores being utilized.

The specification of the NAS benchmarks provides several different sizes of input data for each benchmark in the suite. After selecting the input to use in an experiment the size of the input is fixed regardless of the number of cores utilized to solve the problem. This is referred to as *strong scaling*.

B. Parallel Generic Algorithms

A large percentage of the algorithms set forth in the standard for the C++ STL can be implemented using a single `PARAGRAPH` that makes use of one of the `task factories` provided in STAPL (see Chapter IV Section C). Table II gives the percentage of STL algorithms that can be expressed with either a `task factory` STAPL provides or as a combination of multiple `task factories`.

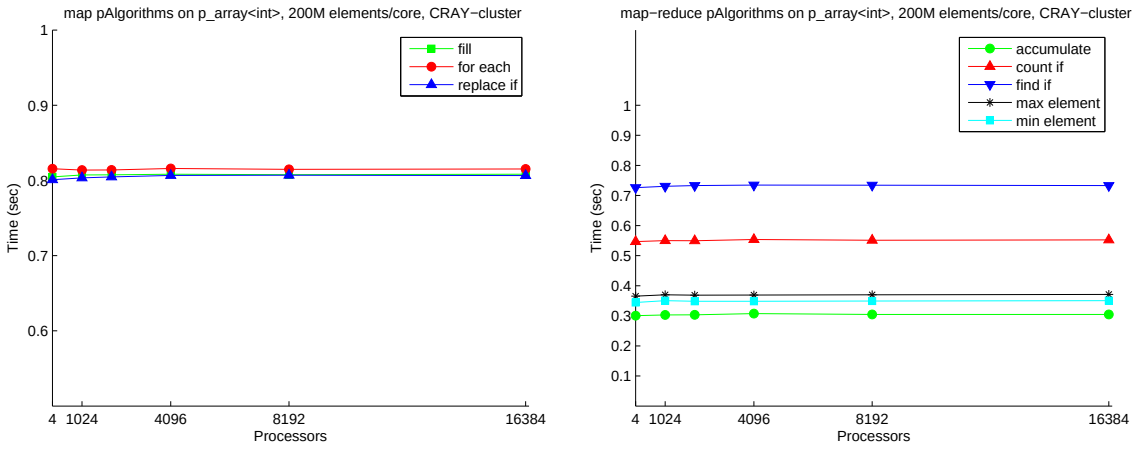
Table II.: Computation patterns used in STL algorithms

Pattern Name	Percentage of STL covered	Example algorithms
map	21%	for each, transform, replace
map reduce	36%	find, count, inner product
prefix scan	3%	partial sum
combination	40%	sort, partition, unique

The performance of `pAlgorithms` that use the `map task factory` to generate their task graph is shown in Figure 38(a). The experiments were run on the CRAY 4. There are 200 million integer elements on each processing location stored in a `pArray` in these weak scaling experiments. The difference in execution time between two `pAlgorithms` on a given processor count is due to the difference in the execution time of their operators. Each `pAlgorithm` results in a task graph with the same number of tasks and dependencies as it is processed by the `PARAGRAPH Executor`.

Figure 38(b) shows the results for the same scaling experiment conducted on the CRAY 4 where the `map reduce task factory` is used by the `pAlgorithms`.

Each data point in the lines of Figure 38(a) and Figure 38(b) are the mean of ten runs executed in a single batch job submission and the associated 95% confidence



(a) (b)

Fig. 38. Weak scaling of (a) map and (b) map-reduce `pAlgorithms` on CRAY 4.

interval.

The algorithms scale well as the number of cores is increased from 4 to 16,384. The difference in execution time of the 4 and 16,384 core experiments is less than 2% for the `map reduce` based `pAlgorithms`. The execution time of the `map` based `pAlgorithms` have a difference between the 4 core and 16,384 core execution time of approximately 1%. Part of the increase can be attributed to the termination detection algorithm that is run to ensure all tasks of the `PARAGRAPH Executor` have executed before control is returned to the calling code. Current work with the composition operators should remove the time needed by the termination detection from the critical path by allowing control to return to the caller as the results are available on a location instead of waiting on all processing locations to receive the result. The termination detection and destruction of the `PARAGRAPH` instance of the computation are already handled by the `PARAGRAPH Executor`, which could defer these activities on a processing location until there are no tasks of a `PARAGRAPH` available for processing.

A final example of a generic algorithm using the STAPL `PARAGRAPH` is an imple-

```
struct strmatch {
  const string& S;
  strmatch(const string& s): S(s) {}

  template<typename View>
  bool operator()(View v) const {
    return equal(S.begin(),S.end(),
                 v.begin());
  }
};

void stapl_main(int argc, char** argv)
{
  typedef stapl::p_array<char>
      p_string_type;
  typedef stapl::array_1D_view
      <p_string_type> pstringView;
  ...
  result=stapl::count_if(
           stapl::overlap_view(text,
           1,0,pattern.size()-1),
           strmatch(pattern));
  ...
}
```

Fig. 39. STAPL implementation of substring matching

mentation of string matching. Our implementation calls the STAPL implementation of `count_if` that accepts a `pView` and a caller-defined predicate. In order to obtain the behavior of substring matching the `pView` passed to the `pAlgorithm` has to be defined such that each element is a substring that the map operation will process. In this case, given a pattern of length M, we create an `overlapped pView` over the text whose elements will contain multiple characters. We declare the `pView` with a core of length 1, left overlap of size 0 and right overlap of size $M-1$. This will produce a `pView` whose elements are all the substrings of size M of the input text. The code illustrating the implementation of the `PARAGRAPH` operator and the construction of the `overlapped pView` is shown in Figure 39. In Figure 40, a MPI version of the program is shown. This case shows the additional complexity of the MPI code with respect to the STAPL version. The MPI programmer must explicitly handle the boundary regions where a string spans multiple processors by replicating the string. This is a

```
int main(int argc, char** argv) {
  ...
  MPI_Comm_size(MPI_COMM_WORLD, &P);
  N=N/P;
  std::vector<char> V(N);
  int M=S.length();

  for (int i=0; i <= N-M+1; ++i)
    if (equal(S.begin(), S.end(),
              V.begin()+i)) ++cnt;
  if (pid>0)
    MPI_Send((&V[0]), M-1, MPI_CHAR,
          pid-1, 1, MPI_COMM_WORLD);
  if (pid<P-1) {
    vector<char> BUFF(2*(M-1));
    copy(V.begin()+N-M+1, V.end(),
              BUFF.begin());
    MPI_Recv( &BUFF[M-1], M-1, MPI_CHAR,
              pid+1, 1, MPI_COMM_WORLD,
              &status );
    for (int i=0; i <= M-1; ++i)
      if (equal(S.begin(), S.end(),
            BUFF.begin()+i )) ++cnt;
  }
  int res;
  MPI_Reduce ( &cnt, &res, 1, MPI_INT,
               MPI_SUM, 0, MPI_COMM_WORLD );
  ...
}
```

Fig. 40. MPI implementation of substring matching

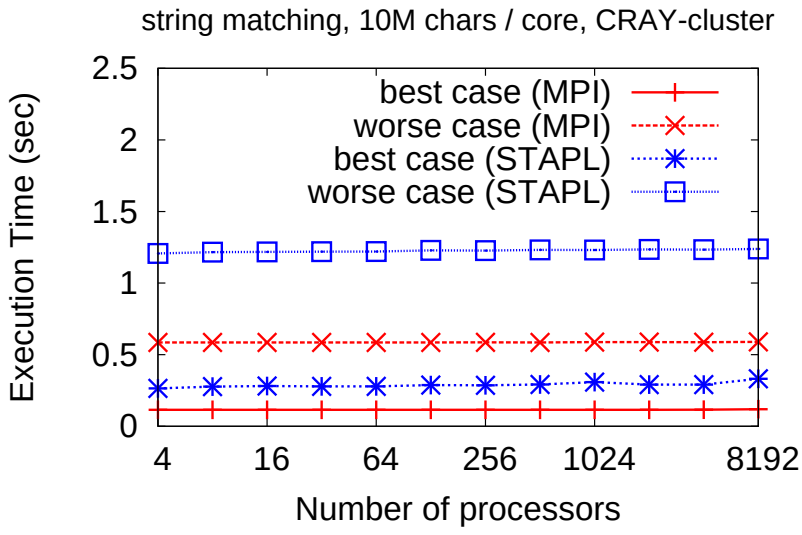


Fig. 41. Weak scaling of substring matching on the CRAY 4

special case of the use of ghost nodes, a well known technique in parallel processing [49, 50]).

Figure 41 shows that performance of the two versions is comparable. In the best case the substring to match is not part of the text, which allows the comparison operator to return immediately after comparing the first character of the string. The first character of the string is always stored on the location where the task is executed, so no communication results from the operation. In the worse case, both text and substring are composed of the same character, maximizing the number of occurrences and the amount of work and communication that the map operations of the task graph must perform.

C. NAS Parallel Benchmarks

The experiments evaluating the performance of individual `pAlgorithms` in Section B are meant to demonstrate the scalability of the components used to represent and process a single `PARAGRAPH` instance. In this section we evaluate the performance of STAPL implementations of the EP and CG benchmarks of the NAS Parallel Benchmarks [44]. The implementation of EP demonstrates the power of the `PARAGRAPH`'s ability to accept user-defined functions as its operations. The STAPL implementation is able to express the computation as a single `map reduce PARAGRAPH` while the FORTRAN reference implementation requires three successive calls to the `MPI_allreduce` function to compute the solution. The CG implementation makes extensive use of the composition operators presented in Chapter V. The result of the composition allows the `PARAGRAPHs` of the entire benchmark to be available for processing by the `PARAGRAPH Executor`.

1. Embarrassingly Parallel – EP

The Embarrassingly Parallel benchmark begins by generating n pairs of random numbers that represent points in the 2D plane. The algorithm used to generate the numbers is provided by the specification of the benchmark and is a deterministic uniform random number generator. Each pair is checked to see if it is a Gaussian pair. A pair (x, y) is a Gaussian pair if $x^2 + y^2 \leq 1$. If the pair is a Gaussian pair then each of its components are included in a pairwise global summation (i.e., the x components of each pair are summed together, and the y components of the pairs are summed together). The benchmark also accumulates data about which of ten annulus rings the pair lies within. The array of annulus counts is also globally accumulated. All of the steps above are part of the timed section of the benchmark. The specification provides various problem sizes that allow for a strong scaling study to be performed on platforms ranging from desktop development systems to massively parallel systems.

a. Implementations

The implementation provided by NAS (referred to as NPB in the figures below) is a Fortran-MPI code. There are two interesting features of the reference implementation. First, each processor blocks its work into sets of 2^{16} elements. The processor generates the elements for a block and stores them in an array. It then scans the array to find the Gaussian pairs and includes their values in the local portions of the data to be accumulated. The second interesting implementation detail is that three successive calls to `mpi_allreduce()` are required to perform the global accumulations needed to form the final results.

The STAPL implementation is a single call to the `map_reduce()` function. The operation provided for the map is a user coarsened work function that generates and

evaluates a point for each element in the `counting_view` it accepts as input. The map work function returns an instance of `deviate_info`, a structure that contains the contributions of the task to the component sums of the Gaussian pairs and the annulus counts. The reduce operator specified for the `PARAGRAPH` is `stapl::plus` instantiated with the `deviate_info`, which calls `operator+` that we have defined on the `deviate_info` structure. The primary difference between the NPB implementation and the STAPL implementation is that the STAPL implementation is able to perform all of the global accumulations in a single `PARAGRAPH`. By combining the work of the three distinct accumulations required in the benchmark the STAPL version achieves significant scalability improvements over the NPB implementation at higher processor counts.

b. Evaluation

Experiments were run on the POWER 5 and CRAY 4. The compiler used for the STAPL implementation was gcc 4.5.2 on both systems. The gfortran compiler was used to compile NPB. In all compilations -O3 was used as the optimization level. For these experiments, and the CG experiments in the next section, 30 runs of both implementations were performed at each processor count in order to compute the 95% confidence interval for the results.

Figure 42 shows the scalability of the STAPL and NPB implementations on the POWER 5 using the class B input ($n = 2^{30}$). Figure 43 shows the execution times of both implementations. The STAPL implementation has sequential overhead of approximately 21% compared to NPB. Our investigations of the usual causes of the "abstraction penalty" (e.g., lack of inlining) have not identified the cause. The amount of work performed by both implementations is the same and there is no fundamental reason the STAPL implementation should not be as fast as NPB. A timer placed around

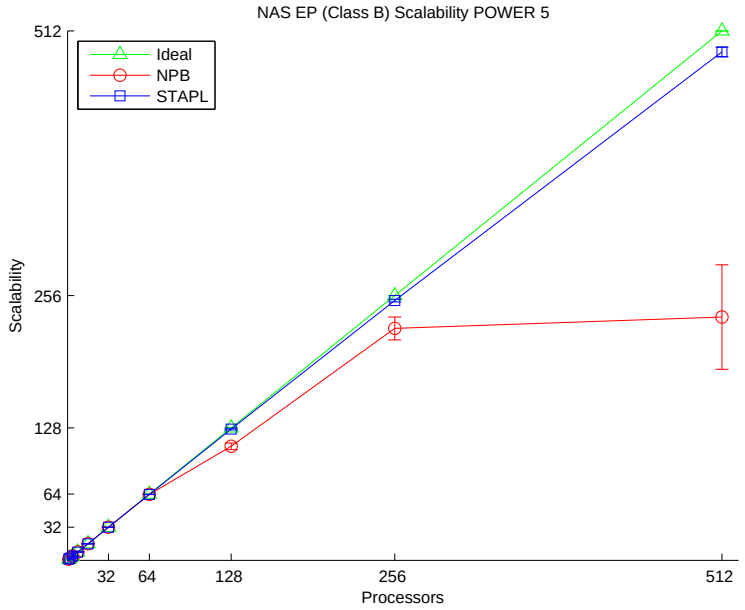


Fig. 42. NAS EP Class B Scalability on POWER 5

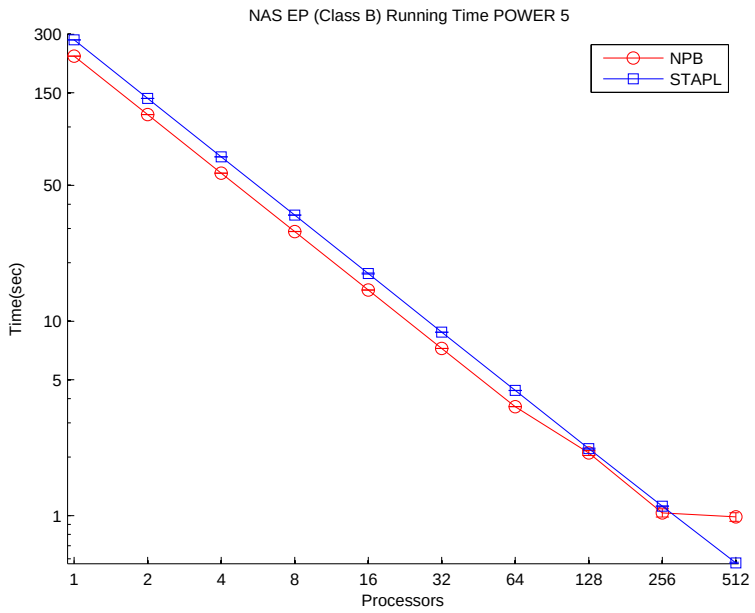


Fig. 43. NAS EP Class B Time on POWER 5 (logarithmic in both axes)

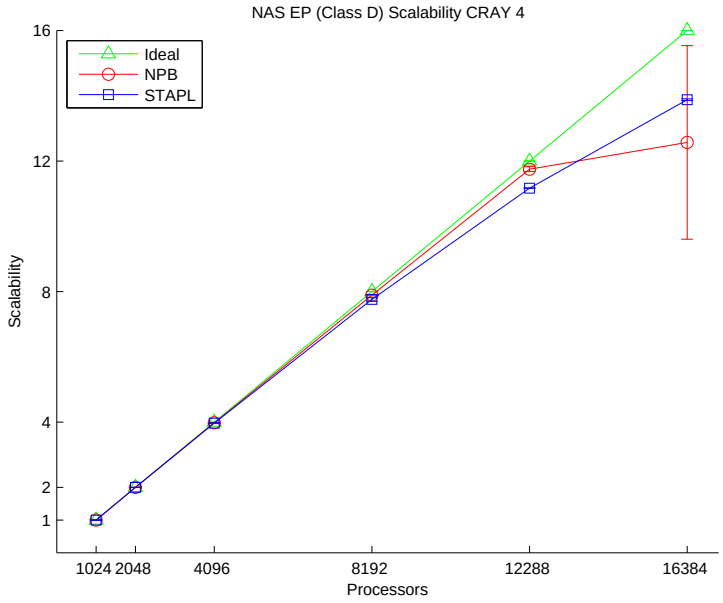


Fig. 44. NAS EP Class D Scalability on CRAY 4

the code identifying Gaussian pairs and computing the local sums reveals that the difference is in the core of the benchmark. We suspect that there is a difference in the language that enables additional optimization to be performed by the GNU FORTRAN compiler.

The scalability of the STAPL implementation is good, though improvement is still possible through customization or removal of the `PARAGRAPH` termination detection from the critical path.

The feature of interest in the graphs for the POWER 5 is the loss of scalability of NPB beyond 256 processors. Removing the three `mpi_allreduce()` calls pull the execution time of NPB back in line with the observations made up to 256 processors. We think the reduction in scalability caused by the successive calls to `mpi_allreduce()` may be due to process skew [51].

Figure 44 shows the scalability of the STAPL and NPB implementations of EP on

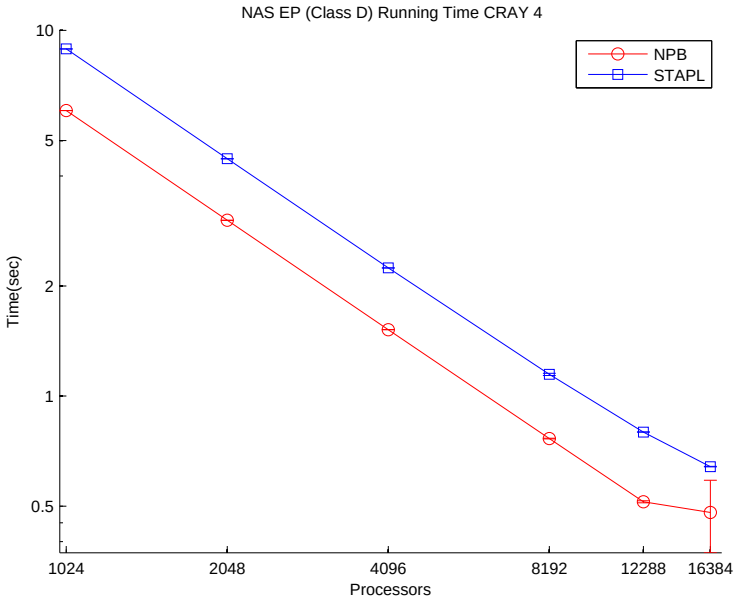


Fig. 45. NAS EP Class D Time on CRAY 4 (logarithmic in both axes)

the CRAY 4 using the class D input ($n = 2^{36}$). Figure 45 is the plot of the execution times for each implementation.

The sequential overhead of the STAPL implementation when compared to NPB has increased to 47% on this platform. We believe the causes are the same as those mentioned in the discussion of the results of the P5-cluster. The same rapid drop off in performance in NPB that we observed on the P5-cluster is seen in NPB as the number of cores is increased from 12,288 to 16,384. The scalability of the STAPL implementation decreases slightly in that range as well in a manner that we would expect from a `map reduce` computation with termination detection. While the scalability of the STAPL implementation could be improved, the run-to-run variability is very low with very tight confidence intervals at the higher processor counts.

2. Conjugate Gradient – CG

The Conjugate Gradient benchmark estimates the largest eigenvalue of a symmetric positive definite sparse matrix with a random patten of nonzeros using the inverse power method. One step of the method requires solving $Az = x$. This is done using the conjugate gradient iterative method. The intended purpose of the benchmark is to measure the performance of random communication between processors.

The steps of the benchmark were presented in Algorithm 2 in Chapter V Section 2. The specification of the benchmark sets the number of rows in the matrix using the parameter n and the number of iterations of the inverse power method using the parameter *niter*. The number of iterations of the conjugate gradient method performed in each iteration of the inverse power method is fixed at 25 by the benchmark specification.

a. Implementation

The implementation provided by NAS (referred to as NPB in the figures below) is a Fortran-MPI code. It uses a two-dimensional processor layout and block matrix distribution for the matrix A. The matrix-vector multiplication operation in the conjugate gradient method is implemented as a row-wise reduction of the results from each processor in the processor column, followed by a transposition of the vector to redistribute it so the vector-vector operations in the method work on data that is local to the processor. This complex use of MPI demonstrates that the NPB implementation is not a naive implementation, but has been optimized for scalability.

The STAPL implementation uses the same two-dimensional processor organization and block matrix distribution. The developer of the benchmark is isolated from the details of the communication pattern because the results of each task are forwarded

to the appropriate processing location due to the `PARAGRAPH` sequence composition operator. The result of STAPL's higher level of abstraction allows the benchmark to be implemented in 322 lines, while the NPB implementation is over $1,000$ lines. The STAPL implementation of the inverse power method was shown in Figure 36. The implementation of the conjugate gradient method was shown in Figure 37.

b. Evaluation

The benchmark was evaluated on the POWER 5 and CRAY 6 systems. The Class B problem ($n = 75000$, $niter = 75$) was used on processor counts ranging from 1 to 256. The Class D problem ($n = 1500000$, $niter = 100$) was used to continue the evaluation of the scalability of both implementations from from 256 to 16,384 cores on the CRAY 6 system. For these experiments 30 runs of both implementations were performed at each processor count in order to compute the 95% confidence interval shown in the figures below.

The scalability and execution time of the Class B problem from 1 to 256 cores on CRAY 6 are shown in Figure 46 and Figure 47, respectively.

The sequential overhead of the STAPL implementation compared to the NPB is 8% on 1 processor, and the difference in the execution times remains at 8% with the exception of the 64 core data point where it is 15%. The results show that the STAPL implementation with its high level of abstraction is able to perform comparably to an optimized FORTRAN-MPI implementation.

Figure 48 shows the continuation of the scalability study on the CRAY 6 from 256 cores and out to 16,384 cores using the class D input. Figure 49 is the plot of the execution times for both implementations.

In Figure 49 we see that the STAPL implementation is able to perform just as well as its FORTRAN counterpart on 1,024 and 4,096 cores, but doesn't see the same

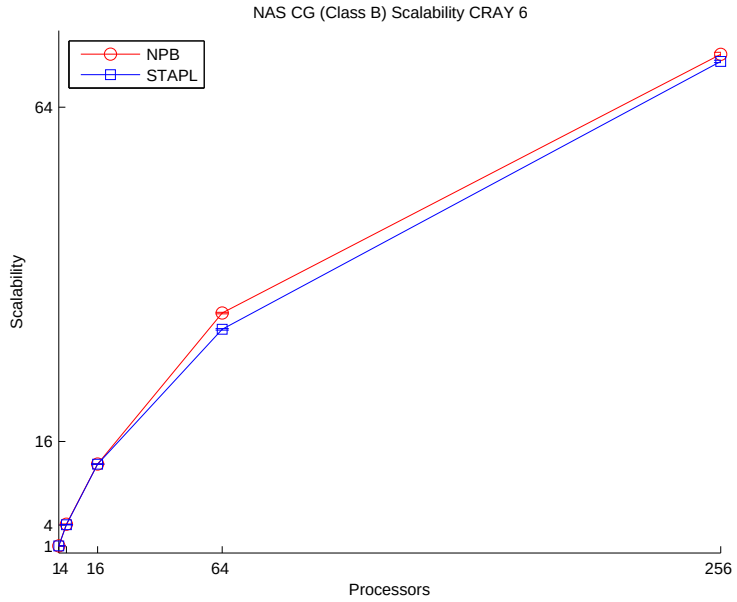


Fig. 46. NAS CG Class B Scalability on CRAY 6

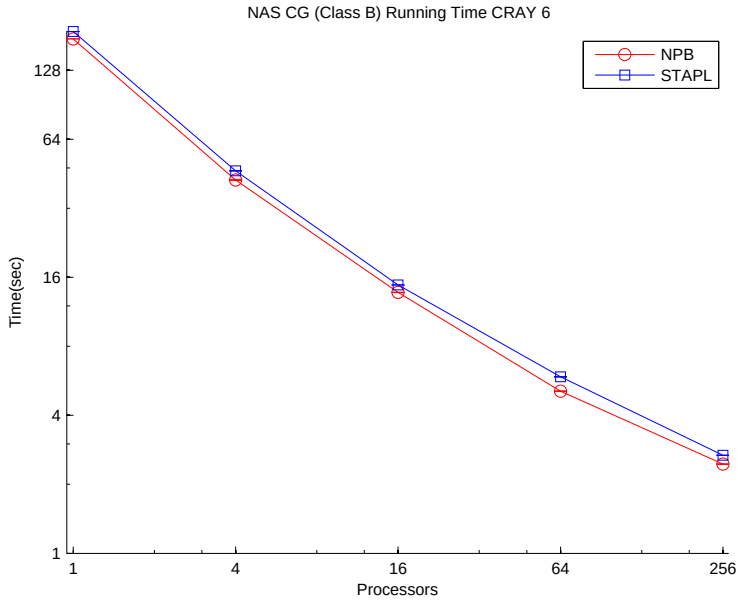


Fig. 47. NAS CG Class B Time on CRAY 6 (logarithmic in both axes)

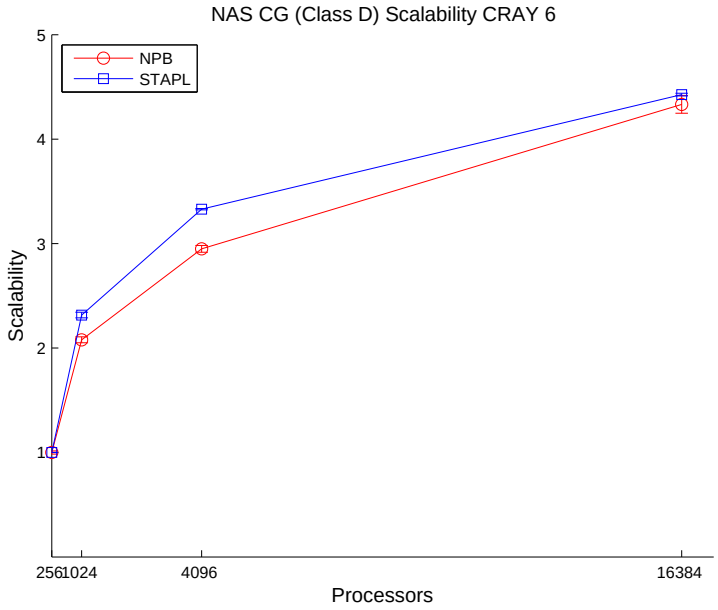


Fig. 48. NAS CG Class D Scalability on CRAY 6

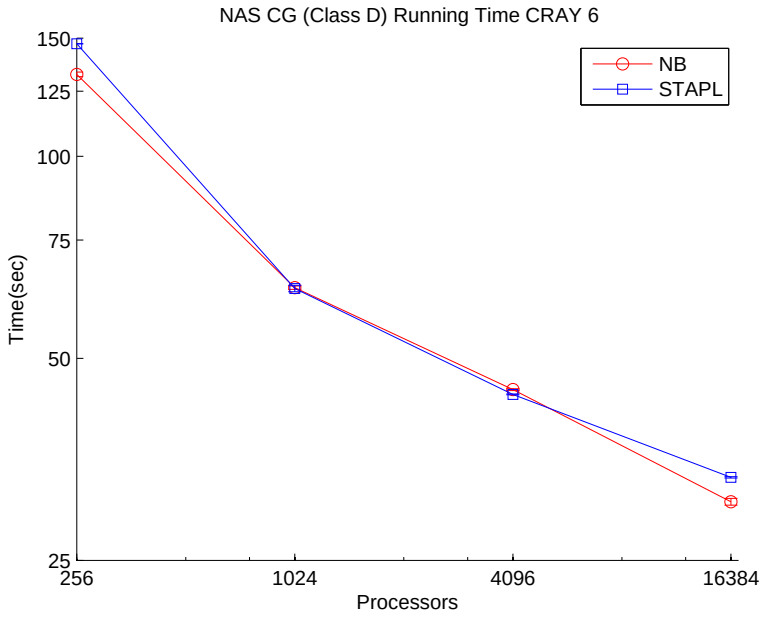


Fig. 49. NAS CG Class D Time on CRAY 6 (logarithmic in both axes)

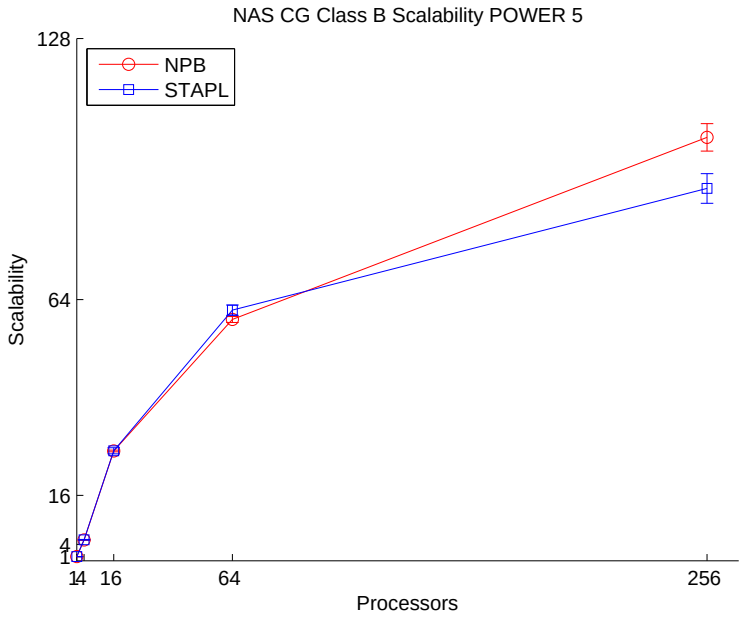


Fig. 50. NAS CG Class B Scalability on POWER 5

improvement in execution time as the NPB implementation when run on 16,384 cores.

Figure 50 shows the scalability of the STAPL and NPB implementations on POWER 5, and Figure 51 shows the execution time across the processor counts for the experiment. The sequential overhead on one processor is 6%. This is a reasonable number considering that the class B problem requires 11,624 `PARAGRAPHs` to be created and processed by the `PARAGRAPH Executor`. The difference between the execution time of the two implementations remains the same until 256 processors. At 256 processors the STAPL implementation improves less than the NPB implementation, which causes the scalability to suffer.

The behavior of the STAPL implementation for the Class B problem on POWER 5 and the Class D problem on CRAY 6 is similar. The STAPL implementation is able to match the performance of the NPB implementation on the lower core counts before the NPB implementation achieves better performance on the last data point.

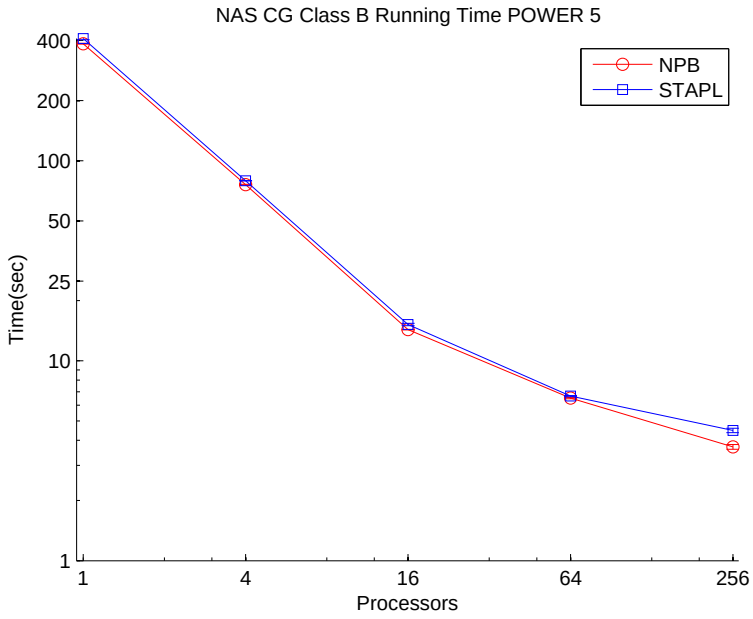


Fig. 51. NAS CG Class B Time on POWER 5 (logarithmic in both axes)

One possibility we are investigating is that the promiscuous MPI mode used by the STAPL communication library's implementation to check for incoming RMI requests negatively affects the execution time on larger core counts. Our preliminary experiments indicate that matching the MPI sends and receives, as the NPB implementation does, will reduce the difference in execution time between the NPB and STAPL implementations. The positive aspect of the STAPL implementation using `PARAGRAPHs` in this case is that the benchmark implementation and `PARAGRAPH` itself will be unaffected by any change in message processing behavior of STAPL run-time system. A similar change in the NPB implementation would require the developer to rewrite the benchmark itself.

D. PDT

Chapter IV Section E presented the high level overview of the problem solved by the PDT discrete-ordinates particle transport application and presented the `task factory` passed to the `PARAGRAPH` in order to construct the task graph to perform the sweep computation for a single set of directions. Chapter V Section 1 demonstrated how the individual sweep `PARAGRAPHs` can be composed using the composition operators presented in Chapter V in the case of non-reflecting surfaces on the spatial domain boundaries of the input being solved. In this section we evaluate the performance of the implementation of this composition using `PARAGRAPHs`.

1. Evaluation

We have designed an artificial input for PDT that allows us to perform a weak scaling study. We are interested in the performance of the sweep computations as that consumes a majority of the execution time in other applications solving the same problems, and this is where the independent composition of `PARAGRAPHs` using the custom `task factory` for the sweep is exercised.

Table III lists the values specified in the input file for each processor size in our experiment on the CRAY 4. "Cell Agg." in the table is the number of cells aggregated into a single cellset along a particular dimension. The input file format for the problem specifies everything related to the spatial domain in terms of the number of items in each dimension. For example, the input is a 3D spatial domain so the number of cells that make up the entire space is specified as the number of cells in the x-, y-, and z-dimensions. The processor layout is specified in the same way because the problem is parallelized by distributing the spatial domain.

The discretizations of the energy, direction, and time are kept constant across all

processor counts. The input performs 5 time steps, solving the steady state problem in each one. For each steady-state computation the direction domain is discretized into 80 angles that are aggregated into 8 angle sets, one originating from each octant of the direction domain. The energy domain is discretized into 10 energy groups that are aggregated into a single energy group set.

Table III shows that the processor arrangement is restricted to a two-dimensional layout. This results in a KBA partitioning [52] of the 3D spatial domain. The KBA partitioning produces a 2D regular grid of cell columns.

Table III.: Values used in PDT study on CRAY 4

Processor Count	1	2	4	8	16	32	64	128	256	512	1024	2048
Cells in X	8	16	16	16	32	32	32	64	64	64	128	128
Cells in Y	8	8	16	16	16	32	32	32	64	64	64	128
Cells in Z	32	32	32	64	64	64	128	128	128	256	256	256
Processors in X	1	2	2	4	4	8	8	16	16	32	32	64
Processors in Y	1	1	2	2	4	4	8	8	16	16	32	32
Processors in Z	1	1	1	1	1	1	1	1	1	1	1	1
Cell Agg. in X	8	8	8	4	8	4	4	4	4	2	4	2
Cell Agg. in Y	8	8	8	8	4	8	4	4	4	4	2	4
Cell Agg. in Z	2	2	2	2	2	2	2	2	2	2	2	2

The growth of the spatial domain occurs in all three dimensions. For each doubling in the processor count the spatial domain is doubled in one dimension in a cyclic order (x, y, and then z). The result is that for every 8-fold increase in the number of processors the number of cells in the z-dimension on each processing location is doubled.

The result of increasing the processors in two dimensions while increasing the number of cells in all dimensions is that the number of cells in a cellset – i.e., the number of cells processed by a task in the sweep task graph – is reduced as the number of processors increases. The number of tasks performed by each processor for a given sweep direction increases as the size of the tasks is reduced as well. The experimental design keeps the number of unknowns the computation is solving for on each processor fixed at 1.31×10^7 across all processor counts, but the number of messages sent increases along with the overhead that comes from invoking a larger number of tasks as the task graph is processed in the `PARAGRAPH Executor`. The end result is that the execution time of an experiment in this study must increase as the number of processors increases instead of remaining constant as expected in other weak scaling studies.

A simple model of the sweep execution has been developed. It uses the sweep time of a sequential execution to find the amount of time needed to solve for an unknown. The number of unknowns per core is constant, so the execution time per unknown (i.e., grind time) provides the base execution time. The number of double-precision values that are communicated between cores during the sweep is also known. The model takes this information and a specification of the system interconnect latency (i.e., the constant overhead of sending a message) and the time to send a single double as inputs. Timing information from the 1, 2, 4, and 8-core experiments is collected to determine the time required to construct a task in the task graph as it is processed by the `PARAGRAPH Executor`, and the overhead of invoking a task. The communication information combined with the task information and base execution time produces a simple estimate of the execution time needed for the sweeps in a computation. This estimate is a best-case scenario that assumes a message is processed by a location the instant it arrives.

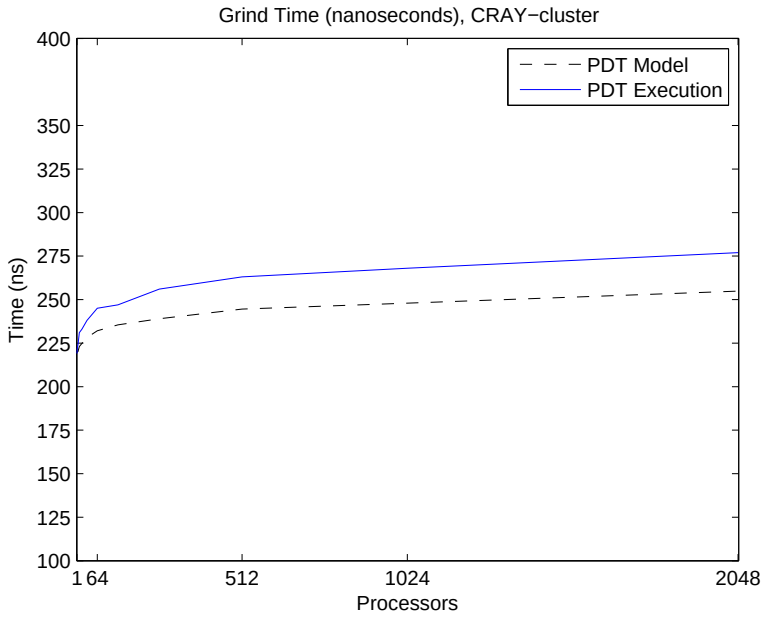


Fig. 52. Weak scaling of PDT on the CRAY 4

Figure 52 shows the execution time of the weak scaling PDT experiment on the CRAY 4 using a single core per node and the estimated execution time produced from the model. All cores on a node were not utilized because the application saturates the memory subsystem when a node is fully utilized. The base execution time is computed from the single processor execution result. The communication parameters we supplied to the model are 6000 nanoseconds latency and 1.14 nanoseconds to transmit a double. These values were obtained from the study of a CRAY XT4 system published in [53].

The execution times we observe deviate from the model, but the execution time differs from the predicted execution time by less than 10% at 2,048 cores. Table IV lists the percent by which the execution time differs from the predicted execution time at each core count. In the table we see four groupings – 1-4 cores, 8-32 cores, 64-256 cores, and 512-2048 cores – where the percentage the execution time varies

from the predicted time of the model is similar.

Table IV.: Percentage difference between actual and predicted execution times on CRAY 4

Processors	1	2	4	8	16	32	64	128	256	512	1024	2048
Difference (%)	0%	1%	2%	4%	4%	4%	5%	5%	7%	8%	8%	9%

Recall that every eight-fold increase in the problem size and processor count results in a doubling of the number of cells in the z-dimension on each core. This causes the number of tasks produced on each core for a sweep `PARAGRAPH` to double, cuts the number of cells processed by each task in half, and doubles the number of messages sent by a location during the sweep. Increasing the size of the task graphs produced for the sweep `PARAGRAPHs` in such a way increases the overhead of `PARAGRAPH` processing. The results in Figure 52 show that the overhead of processing the sweep `PARAGRAPHs` is low compared to the time of the computation being performed, and the amount it increases as the sizes of the `PARAGRAPHs` grow is limited. The results also indicate that the model needs further refinement to properly capture the overheads of increasing the problem size in this manner.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

Recent changes by computer processor manufacturers have made parallel architectures ubiquitous. Multicore processors are emerging in smartphones and tablets as core counts continue to increase in personal computers. At the same time the number of processing elements and their diversity in massively parallel systems continues to grow and redefine high performance computing. The implication for software developers is that more of them are expected to develop parallel applications. Developers working on HPC systems find that the difficulty of developing efficient applications using existing techniques is increasing as the problem complexity increases, and the performance of the algorithms they produce is not portable (i.e., the algorithms have to be tuned or rewritten for each new system).

STAPL, a library for parallel programming in C++, is being developed to address the difficulties of writing parallel algorithms that achieve portable performance. This dissertation presents the `PARAGRAPH`, a component of the library we have developed to allow natural development of a parallel application. The `PARAGRAPH` allows the developer to specify the necessary information in a manner that isolates the concerns of each component from the others and simplifies the development of all the components.

The components a developer specifies are:

- the input `pViews` that represent the data to be transformed by the algorithm,
- the `work functions` that implement the operations to be applied to the individual elements of the input `pViews`, and
- the `task factories` that capture the shape of the parallel computation and

generate the specifications of the tasks of the task graph as it is executed.

We began our discussion of the PARAGRAPH by demonstrating how basic parallel algorithms can be expressed simply. In the implementation section we presented the task factories provided by STAPL that capture computation patterns that occur frequently in parallel applications, and demonstrated how the developer can extend the library with new domain-specific task factories when necessary. We also showed how the PARAGRAPH supports the dynamic generation of irregular task graphs as they are processed to facilitate graph traversal algorithms and other dynamic programming problems. Finally, in the presentation of how the operators of a PARAGRAPH are developed we showed that the operators can use PARAGRAPHs in their implementation, enabling nested parallelism. It is clear from our work that the development of a new task factory is the most difficult activity when implementing a parallel algorithm using PARAGRAPHs. Future work in this area would simplify the development of the task factory by separating the expression of the dependence pattern from the structural pattern of the computation.

In the description of PARAGRAPH composition we demonstrated how multiple PARAGRAPHs can be processed concurrently by the PARAGRAPH Executor, and how producer-consumer relationships are established using data flow views to specify dependencies between the PARAGRAPHs when they occur. We also developed a loop construct that allows more PARAGRAPHs of an algorithm to be processed concurrently when parallel algorithm includes iteration. We demonstrated the capabilities of the composition operators by showing how they enable more parallelism in the STAPL implementations of the NAS CG benchmark and the sweep operation of a discrete-ordinates particle transport code.

We have performed an analysis of the PDT particle transport code and deter-

mined that more PARAGRAPHs could be considered for processing by the PARAGRAPH Executor and their executions optimized if the composition operators included equivalents of the control structures found in imperative sequential programming languages. Future work in this area will include the development of composition operators that allow selecting between PARAGRAPHs and their conditional repetition. Additionally, a composition operator that allows the atomic operation of PARAGRAPHs in a sequence may be of interest in some applications. Finally, simplifying the development of parallel applications written using STAPL by providing a graphical development environment for STAPL applications based on the depictions of the task factories and composition operators presented in this dissertation is another large area of potential future work.

We demonstrated the performance capabilities of the PARAGRAPH with experimental evaluation of applications and algorithms developed using the PARAGRAPH in STAPL. Our results show that the processing of PARAGRAPHs by the PARAGRAPH Executor is scalable to a large number of processing elements. We also show that the overhead of PARAGRAPH processing is small, allowing the STAPL implementations of the NAS benchmarks and the particle transport code to achieve execution times comparable to implementations written using lower level techniques.

The performance of the PARAGRAPH and its resource utilization can be improved further still by future work that explores increasing the amount of information captured in a PARAGRAPH instance and communicated to the PARAGRAPH Executor and the STAPL run-time system. We have identified how the generation of task specifications can be improved using information about the system load from the run-time system, and information about the relationship between PARAGRAPHs captured by the composition operators could allow the operations of the PARAGRAPHs to be merged. The termination detection required to determine when the PARAGRAPH Executor has

finished processing a PARAGRAPH can be modified or eliminated in cases where the composition patterns of the PARAGRAPHs are known.

As the need for improved parallel programming paradigms continues to increase with the number and size of parallel systems we believe that the techniques developed by the PARAGRAPH provide a promising path forward to reduce the difficulty of implementing parallel applications and achieving portable performance.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 2nd edition*. Cambridge, MA: MIT Press, 2001.

[2] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[3] M. P. I. Forum, *MPI: A Message-Passing Interface Standard, Version 2.2*. Stuttgart: High Performance Computing Center Stuttgart (HLRS), Sep. 2009.

[4] G. Blelloch, "NESL: A nested data-parallel language," Technical Report CMU-CS-93-129, Carnegie Mellon University, Pittsburgh, PA, 1993.

[5] M. Frigo, C. Leiserson, and K. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the 19th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Quebec, Canada, 1998, pp. 212–223.

[6] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. Sebastopal, CA: O'Reilly, 1998.

[7] Intel, *Reference Manual for Intel Threading Building Blocks, version 1.24*. Santa Clara, CA: Intel Corporation, 2009.

[8] F. A. Rabhi and S. Gorlatch, Eds., *Patterns and Skeletons for Parallel and Distributed Computing*. London: Springer-Verlag, 2003.

[9] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th USENIX Symposium on Opearting Systems Design & Implementation (OSDI)*, San Francisco, CA, Mar. 2004, pp. 10–10.

[10] K. Knobe, "Ease of use with concurrent collections," in *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar)*, Berkeley, CA, Mar. 2009, pp. 17–17.

[11] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction (CC)*, Grenoble, France, Apr. 2002, pp. 179–196.

[12] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: Standard template adaptive parallel library," in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR)*, Haifa, Israel, May 2010, pp. 1–10.

[13] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger, "STAPL: An adaptive, generic parallel programming library for C++," in *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, KY, Aug. 2001, pp. 193–208.

[14] L. Rauchwerger, F. Arzu, and K. Ouchi, "Standard templates adaptive parallel library," in *Proceedings of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR)*, Pittsburgh, PA, May 1998, pp. 402–409.

[15] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, "A framework for adaptive algorithm selection in stapl," in *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Chicago, IL, Jun. 2005, pp. 277–288.

[16] G. Tanase, M. Bianco, N. M. Amato, and L. Rauchwerger, "The stapl pArray," in *Proceedings of the 2007 Workshop on Memory Performance (MEDEA)*, Brasov, Romania, Sep. 2007, pp. 73–80.

[17] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger, "Associative parallel containers in stapl," in *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Urbana-Champaign, IL, Oct. 2007, pp. 156–171.

[18] G. Tanase, X. Xu, A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger, "The stapl pList," in *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Wilmington, DE, Oct. 2009, pp. 16–30.

[19] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulus, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The stapl parallel container framework," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, San Antonio, TX, Feb. 2011, pp. 235–246.

[20] A. A. Buss, T. Smith, G. Tanase, N. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger, "Design for interoperability in stapl: pMatrices and linear algebra algorithms," in *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Edmonton, Alberta, Canada, Jul. 2008, pp. 304–315.

[21] A. Buss, A. Fidel, Harshvardhan, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The stapl pView," in *Proceedings of the 23rd International Workshop on Languages and Compilers for*

Parallel Computing (LCPC), Houston, TX, Sep. 2010, pp. 261–275.

[22] ISO, *ISO/IEC 14882: Programming languages – C++, 2nd edition*. Geneva: International Organization for Standardization, 2011.

[23] M. Girkar and C. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 166–178, Mar. 1992.

[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[25] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model (Editors: G.-R. Perrin, A. Darte)* London: Springer, 1996, pp. 79–103.

[26] E. Deelman, G. Singh, M. Su, J. Blythe, A. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: a framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, no. 3, pp. 219–237, Jul. 2005.

[27] N. L. Thomas, "The paragraph: Design and implementation of the stapl parallel task graph," PhD Thesis, Texas A&M University, College Station, TX, 2012.

[28] I. T. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao, "Chimera: A virtual data system for representing, querying, and automating data derivation," in *Proceedings of the 14th International Conference on Scientific and Statistical Database Management (SSDBM)*, Edinburgh, Scotland, Jul. 2002, pp. 37–46.

[29] J. Kim, M. Spraragen, and Y. Gil, "An intelligent assistant for interactive workflow composition," in *Proceedings of the 9th International Conference on Intelligent User Interfaces*, Lisbon, Protugal, Feb. 2004, pp. 125–131.

[30] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, Atlanta, GA, Apr. 2010, pp. 1–12.

[31] H. A. Mandviwala, U. Ramachandran, and K. Knobe, "Capsules: Expressing composable computations in a parallel programming model," in *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Urbana-Champaign, IL, Oct. 2007, pp. 276–291.

[32] M. Cole, *Algorithmic skeletons: structured management of parallel computation*, Cambrige, MA: MIT Press, 1991.

[33] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[34] H. Bischof, S. Gorlatch, and R. Leshchinskiy, "Generic parallel programming using c++ templates and skeletons," in *Domain-Specific Program Generation (Editors: C. Lengauer, D. S. Batory, C. Consel, and M. Odersky)*, London: Springer, 2003, pp. 107–126.

[35] D. Musser, G. Derge, and A. Saini, *STL Tutorial and Reference Guide, Second Edition*, Reading, MA: Addison-Wesley, 2001.

[36] I. G. Tanase, "The stapl parallel container framework," PhD Thesis, Texas A&M University, College Station, TX, 2010.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Reading, MA: Addison-Wesley Professional, 1995.

[38] M. D. McCool, "Structured parallel programming with deterministic patterns," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism(HotPar)*, Berkeley, CA, 2010, pp. 5–5.

[39] J. JàJà, *An Introduction Parallel Algorithms*, Reading, MA: Addison-Wesley, 1992.

[40] J. Darlington, Y. Guo, H. W. To, J. Yang, H. Wing, and T. J. Yang, "Functional skeletons for parallel coordination," in *Proceedings of the 1st International Conference on Parallel Processing (EURO-PAR)*, Stockholm, Sweden, Aug. 1995, pp. 55–69.

[41] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.

[42] M. Adams and E. Larsen, "Fast iterative methods for discrete-ordinates particle transport calculations," *Progress in nuclear energy*, vol. 40, no. 1, pp. 3–159, 2002.

[43] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger, "Finding strongly connected components in distributed graphs," *Journal of Parallel and Distributed Computing*, vol. 65, no. 8, pp. 901–910, Aug. 2005.

[44] Numerical Aerospace Simulation (NAS) Facility, NASA Ames Research Center, *NAS Parallel Benchmarks*, Jun. 2011, http://www.nas.nasa.gov/Software/NPB/.

[45] G. W. Johnson, *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control, 4th edition*, New York, NY: McGraw-Hill, 2006.

[46] National Energy Research Scientific Computing Center, *Franklin: NERSC's CRAY XT4 System*, Jun. 2011, http://www.nersc.gov/users/computational-systems/franklin/.

[47] National Energy Research Scientific Computing Center, *Hopper: NERSC's CRAY XE6 System*, Jun. 2011, http://www.nersc.gov/users/computational-systems/hopper/.

[48] Texas A&M Supercomputing Facility, Texas A&M University, *hydra: an IBM p5-575 Cluster*, Jun. 2011, http://sc.tamu.edu/systems/#hydra.

[49] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in Parallel Graph Processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, Mar. 2007.

[50] J. Guo, G. Bikshandi, B. B. Fraguela, and D. Padua, "Writing Productive Stencil Codes with Overlapped Tiling," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 1, pp. 25–39, Jan. 2009.

[51] A. R. Mamidala, J. Liu, and D. K. Panda, "Efficient barrier and allreduce infiniband clusters using hardware multicast and adaptive algorithms," in *In Proceedings of the 2004 IEEE Conference on Cluster Computing (CLUSTER)*, San Diego, CA, Sep. 2004, pp. 135–144.

[52] K. R. Koch, R. S. Baker, and R. E. Alcouffe, "Solution of the first-order form of the 3D discrete ordinates equation on a massively parallel processor," *Transactions of the American Nuclear Society*, vol. 65, pp. 198–199, 1992.

[53] S. R. Alam, J. A. Kuehn, R. F. Barrett, J. M. Larkin, M. R. Fahey, R. Sankaran, and P. H. Worley, "Cray xt4: an early evaluation for petascale scientific simulation," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC)*, Reno, NV, Nov. 2007, pp. 39:1–39:12.

VITA

Timmie Gene Smith received his B.S. in computer science from Texas A&M University in December 1999. He received his M.C.S. in computer science from Texas A&M University in August 2002.

His research focus is parallel programming tools and their role in simplifying the development of applications deployed on massively parallel systems. He is also interested in parallel programming languages, high performance computing, and scientific application development. He received his Ph.D. in Computer Science from Texas A&M University in May 2012.

More information about Timmie Gene Smith's research and publications may be found at http://parasol.tamu.edu/people/timmie. He may be reached at: Parasol Lab, 301 Harvey R. Bright Bldg, 3112 TAMU, College Station, TX 77843-3112. His email address is timmie@cse.tamu.edu.

The typist for this dissertation was Timmie Gene Smith.