

GENERIC IMPLEMENTATION OF PARALLEL PREFIX SUMS AND THEIR
APPLICATIONS

A Thesis

by

TAO HUANG

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

MASTER OF SCIENCE

May 2007

Major Subject: Computer Science

GENERIC IMPLEMENTATION OF PARALLEL PREFIX SUMS AND THEIR
APPLICATIONS

A Thesis

by

TAO HUANG

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

Approved by:

| | |
|---------------------|----------------------|
| Chair of Committee, | Lawrence Rauchwerger |
| Committee Members, | Nancy M. Amato |
| | Jennifer L. Welch |
| | Marvin L. Adams |
| Head of Department, | Valerie E. Taylor |

May 2007

Major Subject: Computer Science

ABSTRACT

Generic Implementation of Parallel Prefix Sums and Their Applications.

(May 2007)

Tao Huang, B.E.; M.E., University of Electronic Science and Technology of China

Chair of Advisory Committee: Dr. Lawrence Rauchwerger

Parallel prefix sums algorithms are one of the simplest and most useful building blocks for constructing parallel algorithms. A generic implementation is valuable because of the wide range of applications for this method.

This thesis presents a generic C++ implementation of parallel prefix sums. The implementation applies two separate parallel prefix sums algorithms: a recursive doubling (RD) algorithm and a binary-tree based (BT) algorithm.

This implementation shows how common communication patterns can be separated from the concrete parallel prefix sums algorithms and thus simplify the work of parallel programming. For each algorithm, the implementation uses two different synchronization options: barrier synchronization and point-to-point synchronization. These synchronization options lead to different communication patterns in the algorithms, which are represented by dependency graphs between tasks.

The performance results show that point-to-point synchronization performs better than barrier synchronization as the number of processors increases.

As part of the applications for parallel prefix sums, parallel radix sort and four parallel tree applications are built on top of the implementation. These applications are also fundamental parallel algorithms and they represent typical usage of parallel prefix sums in numeric computation and graph applications. The building of such applications become straightforward given this generic implementation of parallel prefix sums.

ACKNOWLEDGMENTS

My respect and gratitude go to my adviser Dr. Lawrence Rauchwerger and co-adviser Dr. Nancy Amato for their guidance, support, and generous help since the very first day I met them.

I want to thank Dr. Marvin L. Adams, who helped me in the Generic Particle Transport Code project.

I also want to thank Dr. Jennifer L. Welch, a great professor who gave me the knowledge of algorithms needed for this work.

I want to acknowledge the work of Tim Smith and Gabriel Tanase, both of whom spent a lot of time working on the implementation platform.

TABLE OF CONTENTS

| CHAPTER | | Page |
|---------|--|------|
| I | INTRODUCTION | 1 |
| | A. Parallel Computation | 1 |
| | B. Prefix Sums Problem | 2 |
| | C. Contribution | 4 |
| | D. Thesis Structure | 5 |
| II | PRELIMINARIES | 6 |
| | A. Parallel Computation Models | 6 |
| | B. Parallel Algorithmic Techniques | 9 |
| | C. Parallel Algorithm Design Issues | 9 |
| | D. Synchronization in Parallel Algorithms | 11 |
| | 1. Barrier Synchronization | 12 |
| | 2. Point-to-Point Synchronization | 12 |
| | E. Parallel Library - STAPL | 15 |
| III | COMMUNICATION PATTERNS | 19 |
| | A. Communication Pattern I: Barrier Sequence | 19 |
| | B. Communication Pattern II | 21 |
| | 1. Recursive Doubling Pattern | 21 |
| | 2. STAPL FormulaDDG for Pattern II | 23 |
| | C. Communication Pattern III | 29 |
| | 1. Balanced Binary-Tree Based Pattern | 29 |
| | 2. STAPL FormulaDDG for Pattern III | 30 |
| IV | PREFIX SUMS ALGORITHMS IMPLEMENTATION | 40 |
| | A. 3-step Technique for $p < n$ | 40 |
| | B. RD Algorithm | 42 |
| | 1. RD Algorithm and Theoretical Complexity | 42 |
| | 2. Barrier Synchronization Implementation for RD Algorithm | 44 |
| | 3. Point-to-point Synchronization Implementation for RD Algorithm | 50 |
| | C. BT Algorithm | 54 |

| CHAPTER | | Page |
|---------|--|------|
| | 1. BT Algorithm and Theoretical Complexity | 54 |
| | 2. Barrier Synchronization Implementation for BT Algorithm | 57 |
| | 3. Point-to-point Synchronization Implementation for BT Algorithm | 59 |
| | D. Complexity of RD and BT Algorithms with Synchronizations | 61 |
| V | PERFORMANCE OF IMPLEMENTATION | 63 |
| | A. Optimization for Step 1 in the Implementation | 63 |
| | B. Experimental Results | 63 |
| VI | APPLICATIONS | 66 |
| | A. Parallel Radix Sort | 67 |
| | B. Tree Applications | 72 |
| | 1. Euler Tour Technique | 73 |
| | 2. Rooting a Tree | 76 |
| | 3. Three Other Tree Applications | 81 |
| | 4. Performance of Tree Applications | 82 |
| VII | CONCLUSION AND FUTURE WORK | 84 |
| | A. Conclusion | 84 |
| | B. Future Work | 85 |
| | REFERENCES | 86 |
| VITA | | 89 |

LIST OF TABLES

| TABLE | | Page |
|-------|---|------|
| 1 | Theoretical Complexity of RD Algorithm with Sync. Options | 62 |
| 2 | Theoretical Complexity of BT Algorithm with Sync. Options | 62 |

LIST OF FIGURES

| FIGURE | | Page |
|--------|--|------|
| 1 | Example communication structure using barrier synchronization . . . | 13 |
| 2 | Example communication structure using point-to-point synchronization | 14 |
| 3 | STAPL components | 16 |
| 4 | Communication pattern II example | 22 |
| 5 | Communication pattern III in BT algorithm | 30 |
| 6 | Example of applying three-step technique | 41 |
| 7 | Example of using the RD algorithm on 8 processors | 44 |
| 8 | Example using the BT algorithm on 8 processors | 56 |
| 9 | Processors' actions in the BT algorithm with barrier synchroniza- tion option | 60 |
| 10 | Example of the generic implementation with optimization in step 1 ($p=3$, $n=8$) | 64 |
| 11 | Speedup of parallel prefix sums (input size: 512M integers) | 65 |
| 12 | Speedup of parallel radix sort (input size is 128M integers) | 70 |
| 13 | Example of Euler circuit with successor function s [1] | 75 |
| 14 | Example of building an Euler tour in parallel | 77 |
| 15 | Example of "rooting a tree" | 80 |
| 16 | Performance of Euler tour applications | 83 |

CHAPTER I

INTRODUCTION

A. Parallel Computation

As computers become ever faster, user demands for solving very large problems are growing at an even faster rate. Such demands include not only high-end scientific computing needs motivated by numerical simulations of complex systems (e.g., weather, physics, and biological processes), but also emerging commercial applications that require computers to be able to process large amounts of data in sophisticated ways (e.g., video conferencing, computer-aided medical diagnosis, and virtual reality) [2].

At the same time, the speed of a uni-processor computer is limited by the physical size and speed of its transistors. To circumvent these limitations, computer hardware designers have been utilizing internal concurrency in chips, including pipelining, multiple function units, and multi-core processors.

Another important trend changing the face of computing is an enormous increase in the capabilities of the networks that connect computers, which enables applications to use physically distributed resources as if they were part of the same computer [2]. There is undeniable growing importance and interest in parallel computing.

However, most existing algorithms are designed specifically for single processor computers. In order to design and implement algorithms that can exploit multiple processors located inside a computer, and the additional processors available across

This thesis follows the style and format of *IEEE Transactions on Parallel and Distributed Computing*.

a network, the emphasis in algorithm designs has shifted from sequential algorithms to parallel algorithms, the algorithms in which multiple operations are performed simultaneously.

The basic idea behind parallel computing is to carry out multiple tasks simultaneously, thereby reducing execution time and handling larger scale problems. But the design and development of parallel algorithms are still far from well-understood. This is because there are some major issues in parallel algorithms that are not found in sequential algorithms. These include task decomposition, allocation, communication and synchronization, as well as performance issues such as scalability, speedup, and load balancing.

Fortunately, similar to sequential problems, many parallel problems can be solved based on a small set of essential parallel problems. Researchers have been working hard to design and develop the best parallel algorithms for these essential problems.

One of these essential problems is the parallel prefix sums problem. It has been shown that parallel prefix sums can be applied to a wide variety of parallel problems, as listed in Section B. This thesis deals with this problem by providing the generic implementation of two algorithms, handling the synchronization issues by building reusable communication patterns, and studying performance and applications.

B. Prefix Sums Problem

Parallel prefix sums might be the most heavily used routine in parallel algorithms [3]. They are one of the most important building blocks for parallel algorithms.

The definition of the prefix sums problem is [4]:

Given a binary associative operator \oplus , and an ordered set of n elements

$[a_0, a_1, \dots, a_{n-1}]$, return the ordered set $[a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus a_{n-1})]$.

For example, if the operator \oplus is addition, and the input set is $[3, 5, -2, 6, 2, 0, 4, 8]$, then the prefix sums of this ordered set are $[3, 8, 6, 12, 14, 14, 18, 26]$.

Even though the problem is called “prefix sums,” the operator \oplus could be any binary associative operation that can be applied to two input elements. That is, prefix sums can be used for more than just addition. But for simplicity, the remainder of this thesis uses $+$ to represent the operator, except in the pseudo-code where *bin_op* is used.

Different applications of prefix sums problems are possible by using different operators and different types of input elements. The applications include, but are not limited to, the following [4]:

- Lexically compare strings of characters;
- Solve recurrences;
- Implement radix sort;
- Implement quick sort;
- Implement tree applications.

Given the simplicity and wide application, a generic implementation of a parallel algorithm that supports different input element types, different operators, and different physical distribution of input elements would be valuable for solving many other parallel problems.

Due to the importance of prefix sums problems in parallel computing, there has been considerable research devoted to building effective parallel prefix sums algorithms, including [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15].

This thesis builds a generic implementation using two parallel prefix sums algorithms to build a generic implementation: a recursive doubling (RD) algorithm and a binary-tree based (BT) algorithm. The reason for selecting these two is that they utilize two important techniques widely used in parallel problems. Therefore, their implementation can be directly used to solve many other parallel problems.

C. Contribution

In order to make parallel programming efficient, portable, and reusable, a generic and efficient implementation of two important parallel algorithms for prefix sums in a parallel library is provided. These two algorithms, a prefix sums algorithm based on the recursive doubling technique (RD) and a balanced binary tree based prefix sums algorithm (BT), are implemented in C++ and analyzed using the Coarse Grained Multicomputer (CGM) model. This generic implementation supports a general linear sequence of any data structure as input, as long as the operation \oplus for prefix sums computation to be applied to the data structure is defined. There is no assumption about the distribution of input sequences across multiple processors.

Three pre-built reusable communication patterns are decomposed from these concrete parallel algorithms and they can be directly used by any parallel algorithm that follows the same communication pattern. These three pre-built communication patterns simplify the work of parallel algorithm implementation to a great extent. The usage of these communication patterns on the two prefix sums algorithms results in two different synchronization options used for each algorithm, thus producing four implementations for the two algorithms. All of them show reasonably good speedup in experimental results, while the point-to-point synchronization option has better performance than the barrier synchronization.

A few applications for parallel prefix sums are also implemented on top of the generic implementation, and they also perform well in experiments.

D. Thesis Structure

This chapter, Chapter I, introduces the prefix sums problem, motivates a generic implementation for parallel prefix sums algorithms, and summarizes this thesis' contributions.

Chapter II gives the preliminaries of parallel algorithm design, including some basic parallel computation models, parallel algorithmic techniques, and the major design phases and issues. For one of the main communication issue, synchronization, there are two typical techniques: barrier synchronization and point-to-point synchronization.

Chapter III introduces the three pre-built communication patterns.

Chapter IV presents the design and implementation of the two prefix sums algorithms using the three pre-built communication patterns. The implementations using the barrier synchronization use the first communication pattern with the global barrier operation supported by the library; the implementations using the point-to-point synchronization use communication patterns II and III to guarantee the minimal required synchronizations between tasks, represented by a directed graph.

Chapter V describes the performance study for the implementation of two algorithms with two different synchronization options.

Chapter VI provides a few other parallel problems' solutions through the application of parallel prefix sums. The performance results of these applications are also presented.

Chapter VII concludes this thesis and briefly discusses future work.

CHAPTER II

PRELIMINARIES

This chapter presents the basic ideas and issues in parallel algorithm design. The first two sections give a preliminary introduction to parallel computation models and important parallel algorithmic techniques. Section C briefly explains the major issues in parallel algorithm design. Section D introduces synchronization, the major issue in the design and implementation of parallel prefix sums this research deals with. The last section is a high level introduction of the implementation platform, a parallel / distributed standard template C++ library, STAPL.

A. Parallel Computation Models

A computation model is required to serve two major purposes. First, it is used to describe a computer. So, a computational model attempts to capture the essential features of a machine while ignoring the less important details of its implementation. Second, it is used as a tool for analyzing problems and expressing algorithms [16].

In the realm of sequential computation, the Random Access Machine (RAM) is a standard model that has succeeded in both of these purposes. It serves as an effective model for hardware designers, algorithm developers, and programmers alike.

However, when it comes to parallel computation, there is no such unifying standard model. This is due to the complex set of issues inherent in parallel computation.

Three important principles have emerged for successful parallel computation models: Work-Efficiency, Emulation, and Modeling Communication [17].

- **Work-Efficiency:** In sequential computation, an algorithm's work (or total number of operations) is equivalent to its execution time. On a parallel machine,

an algorithm taking t time on a p -processor machine performs work $W=p \cdot t$. An algorithm is *work-efficient* if it performs the same amount of work, within a constant factor, as the fastest known sequential algorithm.

- Emulation: A parallel computation model can be useful without mimicking any real or even realizable machine. Instead, it suffices that any algorithm that runs efficiently in the model can be translated into an algorithm that runs efficiently on real machines.
- Modeling Communication: To get the best performance out of a parallel machine, it is often helpful to explicitly model the communication capabilities of the machine, such as its latency.

One of the oldest parallel computation models is the PRAM model [18]. It assumes that the number of processors p is polynomial in the input size n , such that each processor only needs to work on partial input of a constant size. It is a simplistic model for parallel computation and is mainly of theoretical interest. Although it fails to capture the features of existing parallel computers by making the above unrealistic assumption, it does represent an upper bound for the performance of parallel algorithms. More importantly, any algorithm that runs efficiently in a p -processor PRAM model can be translated into an algorithm that runs efficiently on a p/L -processor machine with a latency L .

The BSP model [19] was proposed to serve as standard bridging model for hardware (machine architecture) and software (algorithm design and programming). As opposed to the PRAM model, parallel algorithms in the BSP model are organized in distinct computation and communication phases. Moreover, unlike the PRAM, the BSP model makes parallel computation coarse grained. In particular, it assumes that the number of processors and the input size are orders of magnitude apart. Due

to this assumption, the coarse grained model maps better on existing architectures where, in general, the number of processors is in the order of hundreds and the size of input data to be handled could be in the billions. This model uses four parameters: n , p , L and g . Parameter n is the input size; p is the number of processors; L is the minimal time between synchronous steps (measured in basic computation units); and g is the ratio of overall system computational capacity per unit time divided by the overall system communication capacity per unit time.

The introduction of the BSP model marks the beginning of increased research interest in coarse grained parallel computation. This model has been modified along different directions. For example, Culler et al. [20] suggest the LogP model as an extension in which asynchronous execution is modeled and a parameter is added to better account for communication overhead. In an effort to define a parallel computation model that retains the advantage of coarse grained models, while at the same time being simple to use (involving few parameters), Dehne et al. [21] suggest the CGM model.

The CGM model uses only two parameters: n and p . This model is a set of p processors, each with $O(n/p)$ local memory, interconnected by a router that can deliver messages in a point-to-point fashion. A CGM algorithm consists of an alternating sequence of “computation rounds” and “communication rounds” separated by barrier synchronizations (barrier synchronizations are introduced in Section 1).

This thesis uses the CGM model. The input for the parallel prefix sums problem is a linear sequence of n elements, and each of the p processors gets $O(n/p)$ input elements. How these elements are distributed to the processors is not assumed.

B. Parallel Algorithmic Techniques

There are some fundamental algorithmic techniques that are widely used in parallel algorithms. Some of these techniques are used by sequential algorithms as well, but they become much more important in parallel algorithms; others are unique to parallelism. Blelloch gives a brief description of three of the techniques in [17]:

- **Divide-and-Conquer:** By dividing a problem into two or more subproblems, the subproblems can be solved in parallel. Typically the subproblems are solved recursively and thus the next divide yields even more subproblems to be solved in parallel.
- **Randomization:** This is an algorithmic technique unique to parallel algorithms. It allows processors to make local decisions which, with high probability, add up to good global decisions.
- **Parallel Pointer Manipulations:** Many of the traditional sequential techniques for manipulating lists, trees, and graphs do not easily translate into parallel techniques. But such techniques can be replaced by efficient parallel techniques such as recursive doubling, the Euler-tour technique, ear decomposition, and graph contraction.

The algorithms implemented in this work use recursive doubling and Euler-tour techniques. Details of these techniques are introduced when the design and implementation are presented.

C. Parallel Algorithm Design Issues

Following the notation in [22], “task” represents a sequential program fragment that runs on a single processor together with its local storage. The basic point of parallel

computing is to let two or more tasks execute concurrently. And “communication channel” means the logical link between two tasks over which they can exchange information (either through shared memory or explicit message).

The parallel algorithm design process is divided into four steps [22]:

- Partitioning: Decompose a problem into fine-grained tasks, maximizing the number of tasks that can be executed concurrently.
- Communicating: Determine the communication pattern among fine-grained tasks, yielding a “task graph” with nodes and communication channels as edges.
- Agglomerating: Combine groups of fine-grained tasks to form fewer, but larger, coarse-grained tasks thereby reducing communication requirements. This makes the algorithm execute efficiently on the physical target parallel computer.
- Mapping: Assign coarse-grained tasks to processors, subject to tradeoff between communication costs and concurrency.

The work presented in this thesis focuses mainly on the second step. Although a bad partitioning / agglomerating / mapping strategy may produce an imbalanced load and decrease performance, the correctness of the algorithm is not affected. However, if communication between tasks is not correctly designed and implemented, the whole program will fail.

In terms of communication, different problems have different inherent patterns:

- Local versus Global Communication: In local communication, each task communicates with a small set of other tasks (or “neighbors”). In global communication, each task is required to communicate with many tasks (if not exactly all others).

- Synchronous versus Asynchronous: In synchronous communication, producers and consumers of communicated data execute in a pair-wise fashion to perform the data transfer operations; while in asynchronous communication, producers and consumers are not required to cooperate in this way.
- Structured versus Unstructured: In structured communication, a task and its neighbors form a regular structure, e.g., a tree or a grid. Unstructured communication may form an arbitrary graph.
- Static versus Dynamic: In static communication, each task's communication partners do not change over time. Dynamic communication may decide communication partners at runtime and change them frequently.

In this thesis, all the algorithms have structured static communication patterns. This means all the communication channels are independent from the runtime context and can be represented in a pre-determined regular structure. In the following sections it is shown that such regular structures in the algorithms can be a sequence of global communication operations or a mathematically formulated graph. Additionally, asynchronous communication is used, i.e., producer tasks do not need to be aware when consumer tasks require data; hence, consumer tasks must explicitly request data from producer tasks.

The next section discusses the synchronization issue in parallel algorithm design, focusing on two different synchronization options using global and local communications, barrier synchronization and point-to-point synchronization.

D. Synchronization in Parallel Algorithms

Synchronization is the coordination of simultaneous tasks to ensure correctness and avoid unexpected race conditions. There are two types of synchronization, barrier

and point-to-point.

1. Barrier Synchronization

A barrier is a point in parallel program code where a processor must wait until all the other processors participating the program have also reached the same point. After all of the processors reach the barrier, the processors continue issuing program code.

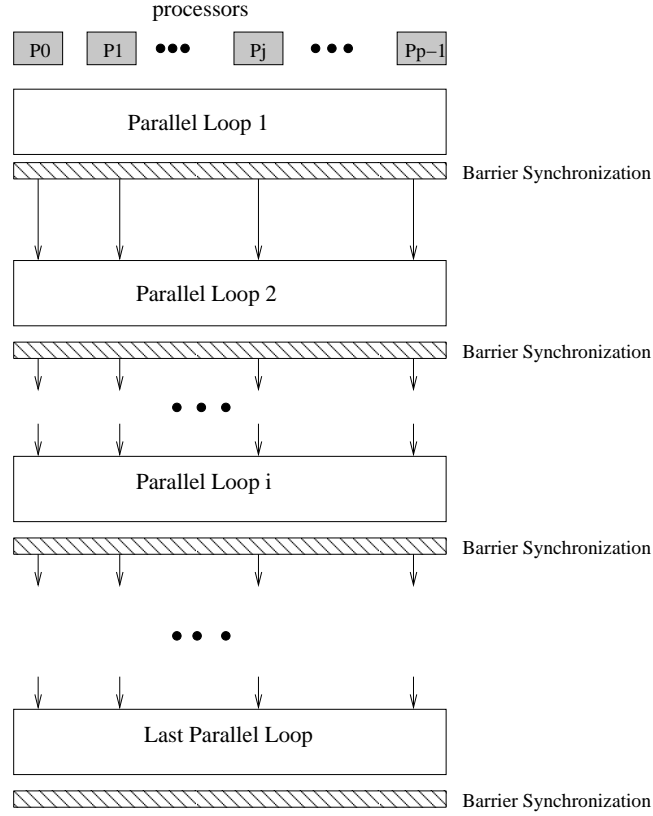
A typical place to use barriers is between parallel loop iterations which perform operations on the same set of data. These barriers guarantee that the processors associated with an iteration finish the operations in the iteration before any of them continue with other program instructions. This is necessary if one processor in a later iteration needs the results achieved on a process in a previous iteration.

Barrier synchronization is also useful for organizing the execution of a parallel program into a sequence of loosely coordinated phases. This matches the idea of the BSP and CGM computation models and there have been many parallel algorithms that utilize barrier synchronization for different domains of problems.

Basically, a parallel algorithm can be represented by a few parallel steps (e.g., loop iterations or loosely coordinated phases) with barrier synchronization as a regular sequence of alternating parallel steps and barrier points, as shown in Figure 1.

2. Point-to-Point Synchronization

Point-to-point synchronization happens only between two tasks, corresponding to only one communication channel in one direction. It typically means that only one single task needs data from another task in order to proceed. The whole communication pattern can be represented as a graph, with nodes as tasks and directed edges

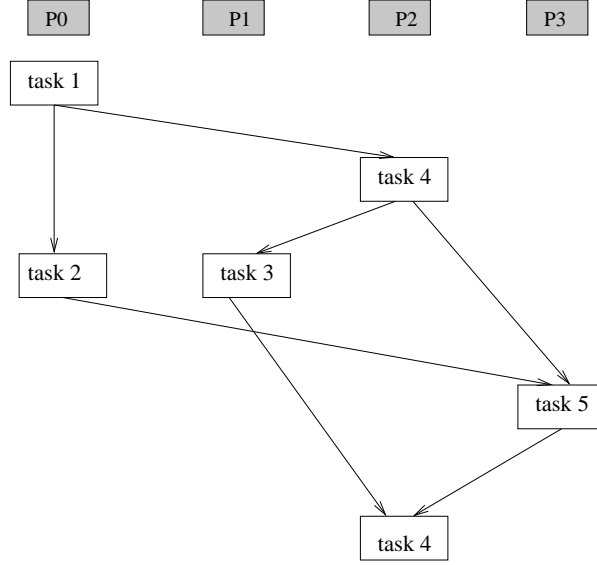


Example communication structure using the barrier synchronization technique of alternating parallel steps and barrier points.

Figure 1: Example communication structure using barrier synchronization

as the required point-to-point synchronizations between the tasks. Figure 2 shows an example of such a structure.

Point-to-point synchronization gives a fine-grained description that matches the minimal synchronization requirements. However, it is difficult to program this way because it requires a very careful design to guarantee all the minimal synchronization requirements are satisfied. The advantage is intuitively obvious: all the processors do not need to be synchronized just because a few tasks may need data from a single task on one processor. In such cases, barrier synchronization is too conservative, while



Example communication structures using point-to-point synchronization techniques;

Directed graph of communication are composed by tasks (nodes) and their point-to-point synchronizations (directed edges).

Figure 2: Example communication structure using point-to-point synchronization

the point-to-point technique can reduce the cost for communication operations.

In order to take advantage of the point-to-point synchronization technique, and to simplify parallel programming, a regular communication structure (the graph representing tasks and synchronizations) is decomposed from the parallel algorithms themselves. This regular structure defines a communication pattern, which can be pre-built separately and used by any parallel algorithm that follows the same communication pattern. For the two parallel prefix sums algorithms in this thesis, three different communication patterns using barrier synchronization and point-to-point synchronization are built. These communication patterns are not limited to the parallel prefix sums algorithms; instead, any parallel algorithms that follow similar communication patterns can directly use them to simplify design and implementa-

tion.

Before the details of these communication patterns are presented in Chapter III, a brief introduction of the implementation platform, a parallel library STAPL, is given.

E. Parallel Library - STAPL

The generic implementation of parallel prefix sums is built in STAPL (Standard Template Adaptive Parallel Library), which is a framework for parallel C++ code. Its core is a library of ISO Standard C++ components with interfaces similar to the sequential ISO C++ standard library (STL).

STAPL includes a run-time system, design rules for extending the provided library code, and optimization tools. Its goal is to allow a user to work at a high level of abstraction and hide many details specific to parallel programming. This is to allow a high degree of portability and performance adaptability across different computer systems. The components and their relationships are described in Figure 3.

Distributed data are stored in “pContainers”. STAPL hides the details of physical distribution of data across processors from the user and provides flexible logical “Views” to access the distributed data in pContainers. When implementing parallel algorithms, programmers can always access the data in the way the algorithms require by creating logical Views accordingly. One View can hide the physical distribution of data in a pContainer.

STAPL has a collection of “pAlgorithms,” which includes the parallel counterparts for most generic sequential algorithms in STL and some numeric and graph parallel algorithms. Parallel prefix sums is one of them, and is used by many other

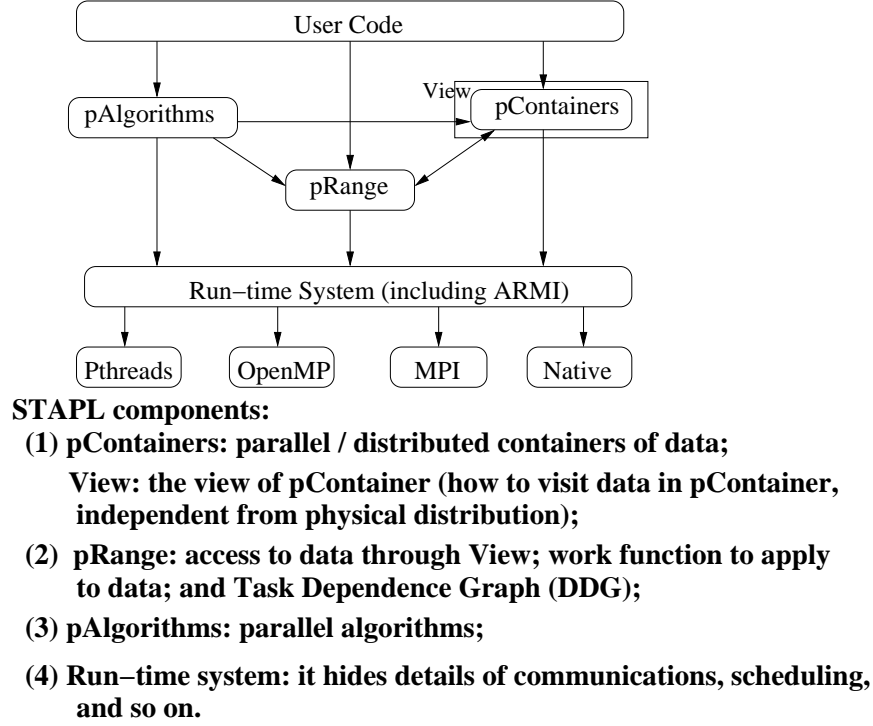


Figure 3: STAPL components

pAlgorithms.

A “pRange” connects pAlgorithms and pContainers together and enables pAlgorithms to be generic. Its core function call *p_for_all(pRange&pr)* is the main entry point to execute parallel programs. A pRange provides access to chunks of data (or “subviews”) in one or more Views, so each task will operate on one chunk (subview) of the View(s). A pRange also has a work-function object that describes the operation for each task, and a Data Dependence Graph (“DDG”) object which represents the tasks and their synchronizations (and could be an instance of a pre-built communication pattern). A pRange corresponds to a whole View of pContainer(s), and is a collection of subranges where each subrange has access to one chunk of the View. The function call *p_for_all(pRange &pr)* applies the work function of the

input pRange on each “subrange” of the pRange. Each such application is called as a “task”. The orderings of all tasks on the p processors follow the synchronizations in the DDG.

In the parallel prefix sums algorithms implementation this collection (pRange) corresponds to the View of a linear sequence of subranges. That is, subranges are organized as a sequence, with each subrange having an offset from the global beginning of the whole pRange and each subrange has a preceding neighbor subrange and a succeeding neighbor subrange (except the first and last subranges).

The run-time system, including the Adaptive Remote Method Invocation (ARMI) communication library, hides all the details of communication, scheduling, and so on.

To support barrier synchronization, ARMI provides an *rmi_fence()* function call that guarantees every processor stops there until all processors have reached the same place. More details and usage of *rmi_fence()* will be introduced in Chapter III.

To support point-to-point synchronization, the pRange follows the directed edges in its DDG to pick up tasks that have no predecessors or those whose all predecessors have been finished. A task X is a predecessor of task Y if task X must be finished before task Y can start, due to the synchronization requirement of X to Y , and task Y is a successor of task X . This ordering is represented by a directed edge from task X to task Y in the DDG.

The DDG can be represented by a parallel graph (pGraph), which is one type of the pContainers implemented in STAPL and physically contains vertices and edges. It can also be any class that can compute the predecessor / successor relationships between tasks given unique IDs. Since the tasks and synchronizations in the two parallel prefix sums algorithms (RD and BT) can both be determined statically (by mathematically expressing the preceding / succeeding relationships) based on the input size n , this determination function is provided as a class using formulas to

represent the DDG (or “FormulaDDG” in STAPL). The FormulaDDG can be pre-built out of concrete parallel algorithms and reused to represent inherent common communication patterns. The construction of the DDG is explained briefly in the following two sections when presenting the pre-built communication patterns.

CHAPTER III

COMMUNICATION PATTERNS

As one of the main issues unique to parallel algorithms, communication needs to be carefully designed in order to be correct and efficient. When tasks should communicate with other tasks, which tasks should be the destination, what the communication data size would be, and all such characteristics compose a specific pattern for the algorithm.

There have been observations [23] showing that some parallel algorithms share the same or similar communication patterns. Optimizing the design and implementation of such communication patterns gives insight into improving the performance of parallel algorithms. Pre-built tuned-up communication patterns can dramatically simplify the work for parallel programmers and guarantee good performance in the final parallel programs as well. With that motivation, this thesis attempts to decouple the usually embedded communication patterns from the whole parallel algorithms and build a library of reusable communication patterns; this library can then be used directly when programmers try to build parallel algorithms that follow similar communication patterns.

In this section, it is assumed that the input size n is equal to the number of processors p . This assumption only holds for this section and more general cases will be discussed later.

A. Communication Pattern I: Barrier Sequence

The simplest and most intuitive communication pattern supported is a sequence of alternating parallel steps and barrier synchronization operations, as introduced in Section 1. In STAPL, this is supported by the function calls *p_for_all(pRange)* and

rmi_fence(), exposed by ARMI.

Rmi_fence is a collective operation as the superset of a barrier. It does not release until all processors arrive (as the barrier requires) and complete all outstanding communication requests [24].

If a parallel algorithm follows this pattern, i.e., the algorithm can be divided into a few parallel steps with barrier synchronizations between each two adjacent steps, then the only thing specific to the algorithm is what operations are applied to each local set of data (or subrange in STAPL) in each parallel step. If these operations are put to work in function objects, the typical framework for this algorithm would be:

Framework of a parallel algorithm using
Communication Pattern I:

```
template <typename PRange,
          typename Workfunction1,
          typename Workfunction2>
palgorithm_foo(PRange& pr, WorkFunction1 wf1,
               Workfunction2 wf2)
{
    // parallel step 1
    pr.set_task(wf1);
    p_for_all(pr);

    rmi_fence();
```

```

// parallel step 2
pr.set_task(wf2);
p_for_all(pr);

rmi_fence();

...
}

```

As introduced in Section E, *p_for_all()* applies the work function object (set by PRange’s *set_task* function) to each local set of input data (subrange), which composes one parallel step. Then *rmi_fence* applies barrier synchronization.

B. Communication Pattern II

1. Recursive Doubling Pattern

The second communication pattern represents a commonly used parallel algorithmic technique, recursive doubling. This technique is used in parallel problems that first appear unavoidably sequential. “Recursive doubling” is also known as “pointer jumping” or “shortcutting”. It means replacing a pointer with the pointer it points to, in a linked structure. This technique is used for various algorithms on lists and trees.

An example of this communication pattern for a prefix sums problem with 8 input elements x_0 - x_7 is shown in Figure 4. Each processor keeps one input element $x[i]$ and its copy $y[i]$; ($y[i]$ is used to avoid unnecessary waiting for computation on $x[i]$ during communicating, as explained by a later example). In this figure, the lines with arrows represent the synchronizations. For example, task $x_4=y_2+x_4$ needs to

read the value of y_2 , so it has to wait for task $y_2=x_2$; it also needs to write to x_4 , so it has to wait until task $y_4=x_4$ correctly reads the old value of x_4 . Solid lines represent the synchronizations required by flow dependencies and output dependencies, while dotted lines represent the synchronizations required by anti-dependencies.

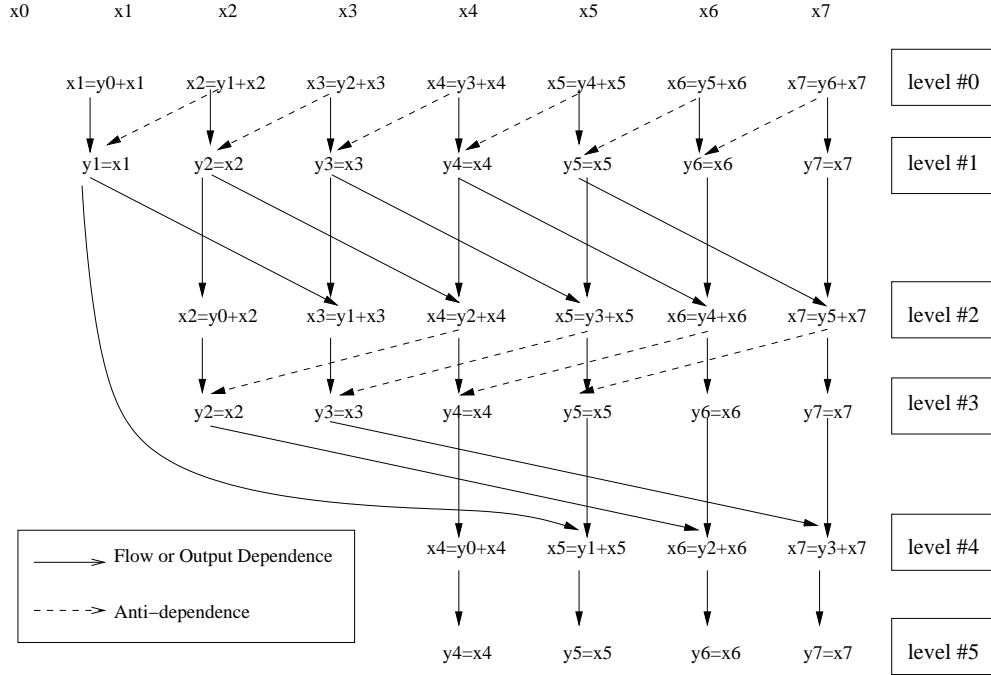


Figure 4: Communication pattern II example

Note that this pattern is not specific for the prefix sums problem. Any parallel problem using the recursive doubling technique has a similar synchronization structure.

Basically, this pattern is composed of a few levels of task pairs and synchronizations between tasks, as shown in Figure 4. The distances between source (or producer) tasks and destination (or consumer) tasks for synchronizations, called

“synchronization distances”, are different at different levels.

For example, in Figure 4, there are six levels of computations. Tasks in the j -th level, where j is even, apply addition operations $x[i] = y[i-d] + x[i]$, where d , the synchronization distance from the source task at the previous level, equals to $2^{j/2}$. Tasks in the j -th level, where j is odd, apply saving operations $y[i] = x[i]$, and these tasks have synchronization distances from the source task at the previous level as $2^{(j-1)/2}$. From a different perspective, the same d for input synchronization distance and output synchronization distance could be used for the tasks at even levels.

The value of synchronization distance d is decided by the underlying idea of the recursive doubling strategy: “replacing a pointer with the pointer it points to,” which leads to $\log_2(n)$ times of replacing. At each instance of replacing, the distance doubles; thus, the synchronization distance is doubled.

The reason to keep a copy $y[i]$ for $x[i]$ after each update step is to parallelize the reading and writing to the i -th element. For example, in the fourth step, $x[11]$ needs to read $y[7]$ (because $d_2 = 2^{4/2} = 4$), but $x[7]$ needs to be updated to $x[3] + x[7]$. Without a copy $y[7]$ for $x[7]$, the update to $x[7]$ has to wait until $x[11]$ has read the correct value of $x[7]$.

2. STAPL FormulaDDG for Pattern II

This pattern in STAPL is provided as a pre-built FormulaDDG *rd_dependence_oracle*. A FormulaDDG is constructed through three functions: *construct_tasks* to determine how many tasks need to be created for which subranges; *enables* and *depends_on* decide the successor tasks and predecessor tasks for each task, i.e., the synchronizations. The following are the three functions for Communication Pattern II:

Function "construct_tasks" for

`rd_dependence_oracle:`

(This function creates the vertices of DDG.)

Input: A subrange of the owner `pRange` for a
DDG instance of `rd_dependence_oracle`.

Output: A vector of pointers to the tasks (with
unique Ids) created for this subrange.

```
template < typename PRange >
vector<task_type*>
construct_tasks (PRange& subrange)
{
    int i = the subrange's offset;
        // since input pRange is a sequence of
        // subranges, each subrange has an offset
        // from the beginning.

    int tasknum;
        // #tasks for this subrange

    if ( i == 0) {
        // subrange 0 is never updated

        tasknum = 0;
    } else {

        tasknum = 2 * log2(i+1);
        // processor i has 2*log(i+1) tasks

    }

    vector<task_type*> new_tasks;
```



```

new_tasks.reserve(tasknum);

    // the vector to store the tasks to
    // be created

if (tasknum == 0)
    return new_tasks;

for ( int j = 0; j < tasknum; j = j+1) {
    // for each level
    int taskid = j * p + i;
    // a unique id for this task, meaning
    // that this task is for subrange i,
    // at j-th level in the whole DDG
    if (j is even) {
        // the task should do:
        //    x[i] = bin_op(y[i-d], x[i])
        d = 2 ^ (j/2);
    } else {
        // the task should do:
        // y[i] = x[i]; no need for d
        d = 0;
    }

    new_tasks.push_back(
        new task_type(taskid, subrange, d, bin_op));
    // create a new task with this "taskid"

```

```

        // that will work on this "subrange",
        // whose task-specific data is "d",
        // and operation is "bin_op".
    }

    return new_tasks;
}

```

The class *task_type* in the pseudo code for *construct_tasks* is a class that has a unique Id (a reference to the subrange it works on), a data member to save data specific to the task (like the synchronization distance d for this task), and a work function to specify the operation to be applied.

The following two functions, *depends_on* and *enables*, take a task Id as an input parameter and return a vector of task Ids as this task's predecessors and successors. The computation is based on the dependencies required by the RD algorithm. The “upper”, “lower”, “left”, and “right” neighbors in the pseudo-code refer to the geometric locations in Figure 4. The locations for these neighbors are always computable based on the input task Id and the type of dependences to/from the task. Mathematic details are not covered in this thesis.

Function "depends_on" to find predecessors
 for input task in rd_dependence_oracle:
 (This function creates input edges to a
 vertex in DDG.)

Input: The task Id for a task in a DDG
 instance of rd_dependence_oracle.

Output: A vector of task Ids for the predecessors of the task.

```

void
depends_on(int taskid,
          vector<task_id_type>& v)
{
    int i = taskid % n;
        // the index for the subrange
        // for this input task
    int level = taskid / n;
        // the level of the task in the DDG
    v.push_back(the upper neighbor task
                for same subrange);
        // flow dependence to this task
    if (level is even) {
        v.push_back(the left upper neighbor
                    that has non-vertical flow
                    dependence to the task);
    } else {
        v.push_back(the right upper neighbor
                    that has anti-dependence
                    to the task);
    }
}

```

Function "enables" to find successors
 for input task in rd_dependence_oracle:
 (This function creates output edges from
 a vertex in DDG.)

Input: The task Id for a task in a DDG
 instance of rd_dependence_oracle.

Output: A vector of task Ids for the
 successors of the task.

```
void
enables(int taskid,
        vector<task_id_type>& v)
{
    int i = taskid % n;
    //the index for the subrange
    int level = taskid / n;
    //the level of the task in DDG
    v.push_back(the vertically lower neighbor
                that has flow dependence from
                this task);
    if (level is even) {
        v.push_back(the left neighbor that has
                    anti-dependence from this task);
    } else {
        v.push_back(right lower neighbors that have
```

```

        flow dependences from this task);
    }
}

```

Given the pre-built FormulaDDG *rd_dependence_oracle*, any parallel problem using the recursive doubling technique only needs to build its own work function (i.e., the computation operation) for the tasks. It directly uses this DDG as its pRange's DDG and thus the correct synchronizations between tasks are guaranteed.

C. Communication Pattern III

1. Balanced Binary-Tree Based Pattern

This pattern first follows a balanced binary tree to compute partial results, and then follows another tree structure to pass these partial results back in order to get the final results. An example is shown in Figure 5. Similar to Figure 4, the solid lines represent the synchronizations required by the flow dependencies and output dependencies, while the dotted lines represent the synchronizations required by anti-dependencies.

This pattern includes two sweeps, the up sweep and the down sweep. As labeled in Figure 4, there are in total $2^{\log_2(p)} - 1$ levels of tasks, and the value of synchronization distance d at level j is: 2^j if the level is in the up sweep; or $2^{2^{\log_2(p)} - 2 - j}$ if the level is in the down sweep.

This is a general communication pattern which has both the up sweep and the down sweep. A specific parallel algorithm could specify whether and which processors have tasks in the down sweep. If all the processors do not have tasks in the down sweep, then the communication pattern is specialized as a simple binary tree structure.

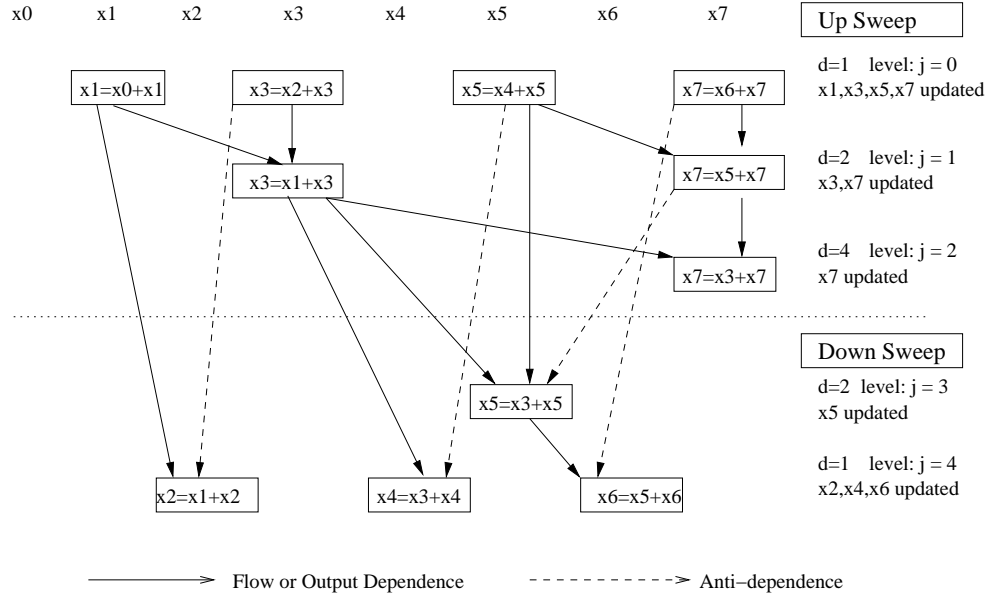


Figure 5: Communication pattern III in BT algorithm

2. STAPL FormulaDDG for Pattern III

A helper function *compute_distance* is used to compute the value for synchronization distance d for given level j :

Helper function "compute_distance":

Input: the level j of some task.

Output: The synchronization distance d for
this level (such that the task should
update its element by adding neighbor
at d places away: $x[i] = y[i-d]+x[i]$).

task_id_type

```

compute_distance(int j)
{
    if ( j < log2(p) ) {
        // this is an up sweep task
        return 2^j;
    } else {
        // this is a down sweep task
        return 2^(2*log2(p) - 2 - j);
    }
}

```

Similar to Communication Pattern II, this pattern is represented by a pre-built FormulaDDG *bt_dependence_oracle*. The condition to decide whether this subrange (with offset *i*) has task(s) in the down sweep, $i \geq 3 \cdot (2^t) - 1$, comes from the BT algorithm since this communication pattern is currently used only for the implementation of parallel prefix sums. However, this can be modified to match the needs of other similar parallel algorithms.

Function "construct_tasks" for

bt_dependence_oracle:

(This function creates the vertices of DDG.)

Input: A subrange of the owner pRange for a DDG
instance of *bt_dependence_oracle*.

Output: A vector of pointers to the tasks (with
unique Ids) created for this subrange.

```

vector<task_type*>
construct_tasks (PRange& subrange)
{
    bool has_downsweep_task = false;
    int i = subrange's offset;
    int nt = #times that (i+1) can be
        divided exactly by 2;
    // #tasks for this subrange in the up sweep
    // Refer to mathematic issue (a) below.
    int tasknum = nt;
    if (i >= 3 * (2nt) - 1) {
        // condition for the subrange to own
        // down sweep task, coming from BT algorithm;
        has_downsweep_task = true;
    }

    if (has_downsweep_task)
        // if the subrange does have down sweep
        // task, then it only has one such task.
        // Refer to mathematic issue (b) below.
        tasknum++;

    vector<task_type*> new_tasks;
    new_tasks.reserve(tasknum);
    // the vector to store the tasks to be
    // created

```



```

for (int j = 0; j < nt; ++j) {
    // for each level in the up sweep
    int taskid = j * p + i;
        // a unique id for this task, meaning
        // this task is for subrange i, at
        // level j in the whole DDG
    int d = 2j;
        // the distance for this level in up sweep
    new_tasks.push_back(
        new task_type(taskid, subrange, d, bin_op));
        // create a new task with this "taskid"
        // that will work on this "subrange",
        // whose task-specific data is "d",
        // and operation is "bin_op"
}

if (has_downsweep_task) {
    //create the down sweep task
    int j = 2 * log2(p) - 2 - nt;
        // the down sweep task is at level
        // 2*log2(p)-2-nt.
        // Refer to mathematic issue (c) below.
    int taskid = j * p + i;
    int d = 2( 2*log2(p) - 2 - j );
    new_tasks.push_back(

```

```

        new task_type(taskid, subrange, d, bin_op));
    // create a new task with this "taskid"
    // that will work on this "subrange",
    // whose task-specific data is "d",
    // and operation is "bin_op"
}

return new_tasks;
}

```

There are three mathematical issues in *construct_tasks* that require detailed explanations.

- (a) For a subrange with offset i , its number of tasks in the up sweep is the number of times that $(i+1)$ can be divided exactly by 2. In other words, it is the number of occurrences of 2 as $(i+1)$'s factor.
- (b) A subrange has either 0 or 1 tasks in the down sweep of the BT algorithm.
- (c) If a subrange does have a down sweep task, then the task's level can be computed based on the offset i .

The following are the detailed explanations for these claims.

(1) The number of tasks in the up sweep can be observed based on the balanced binary tree structure of the up sweep. At any level in the up sweep, only when $(i+1)/d_j$ is even can the subrange (with offset i) have a task at the level.

So, for a subrange to have tasks from level 0 to level j but not at level $j+1$, it must have: $i+1 = (2^{\alpha_1}) * d_j = \alpha_1 * 2^{j+1}$, but $i+1 \neq (2^{\alpha_2}) * d_{j+1} = \alpha_2 * 2^{j+2}$, where α_1 and α_2 are both natural numbers. In other words, $(i+1)$ must be multiple

of 2^{j+1} , but not a multiple of 2^{j+2} . This means the subrange's number of tasks is equal to the number of times that $i+1$ can be divided exactly by 2.

(2) Parts of the subranges have their results computed in the up sweep, so they do not need down sweep tasks at all. If a subrange has a down sweep task, it cannot have more than one such task (proved below). In (2) and (3), the symbol m is used to represent the value of $2 * \log_2(p) - 2$, for clearer expression.

Based on the BT algorithm (introduced in Chapter IV), the condition for the subrange with offset i to own a down sweep task at down sweep level $j1$ is: $i+1 = 3*d_{j1}$, or $5*d_{j1}$, or $7*d_{j1}$, i.e., $i+1 = (3+2*\beta1)*d_{j1} = (3+2*\beta1)*2^{m-j1}$, where $\beta1$ is a constant non-negative integer.

Suppose there exists another task at any other down sweep level $j2$, then $i+1 = (3+2*\beta2)*d_{j2} = (3+2*\beta2)*2^{m-j2}$, where $\beta2$ is also a constant non-negative integer.

Since $(3+2*\beta1)$ and $(3+2*\beta2)$ are both odd, the times that $(i+1)$ can be divided by 2 cannot be both $m-j1$ and $m-j2$ since $j1 \neq j2$. Therefore, this subrange can have only one down sweep task.

(3) The level for the down sweep task of a subrange with offset i , if it exists, is equal to the number of its up sweep tasks (nt in the pseudo-code). Because in (1) it is known that the last level (level j) of an up sweep task for subrange with offset i satisfies: $i+1 = \alpha * 2^{j+1}$, where α is an odd integer; and based on (2) it is known that this subrange's down sweep task must be at level $j1$ which satisfies: $i+1 = (3+2*\beta)*2^{m-j1}$. So $\alpha * 2^{j+1} = (3+2*\beta)*2^{m-j1}$. Since both α and $(3+2*\beta)$ are odd, we conclude that $j+1 = m-j1$, that is, $j1 = m - (j+1) = m$ - the number of the subrange's up sweep tasks.

The *depends_on* and *enables* functions for *bt_dependence_oracle* are briefly described by the following pseudo-code. The left, right, upper, and lower neighbors refer to the geometric locations in Figure 8.

In these two functions, to decide whether a task is an up sweep task or a down sweep task, simply compare the task's level with the number of up sweep tasks for the subrange i (as explained in *construct_tasks* function).

Function "depends_on" to find predecessors

for input task in bt_dependence_oracle:

(This function creates input edges to
a vertex in DDG.)

Input: A taskid for a task in a DDG instance
of bt_dependence_oracle.

Output: A vector of taskids for the predecessors
of the task.

```
void
depends_on(int taskid,
          vector<task_id_type>& v)
{
    int i = taskid % n;
        //the index for the subrange
    int j = taskid / n;
        //the level of the task in DDG
    v.push_back(the vertically upper neighbor
                that has flow dependence to
                this task);
    if (this task is an up sweep task) {
```

```

        v.push_back(the left upper neighbor
                    that has flow dependence to
                    this task);
    } else {
        //this task is a down sweep task
        v.push_back(the left upper neighbor
                    that has flow dependence to
                    this task);

        v.push_back(the right first-pass
                    neighbor that has anti-
                    dependence to this task);
    }
}

```

Function "enables" to find successors
for input task in bt_dependence_oracle:
(This function creates output edges
from a vertex in DDG.)

Input: A taskid for a task in a DDG instance
of bt_dependence_oracle;
Output: A vector of taskids for the successors
of the task.

```

void
enables (int taskid,

```

```

        vector<task_id_type>& v)
{
    int i = taskid % n;

        //the index for the subrange
    int j = taskid / n;

        //the level of the task in DDG
    if (this task is an up sweep task) {
        v.push_back(the vertically lower neighbor
                    that has flow dependence from
                    this task);
        v.push_back(the right lower neighbor that
                    has flow dependence from
                    this task);
        if (this task is the lowest up sweep task
            for the subrange)
        {
            v.push_back(some right neighbors' down
                        sweep tasks that have flow
                        dependences from this task);
        }
        v.push_back(the left neighbor's down sweep
                    task that has anti-dependence
                    from this task);
    } else {

        //this task is a down sweep task
        v.push_back(some right neighbors' down sweep

```

```

        tasks that have flow dependences
        from this task);
    }
}

```

Given the above pre-built FormulaDDG, the BT prefix sums algorithm only needs to specify its work function for the tasks and it uses this FormulaDDG as its own DDG, so all the synchronizations between tasks are guaranteed.

Other parallel algorithms using a similar communication pattern can use this FormulaDDG with only slight modifications.

CHAPTER IV

PREFIX SUMS ALGORITHMS IMPLEMENTATION

A. 3-step Technique for $p < n$

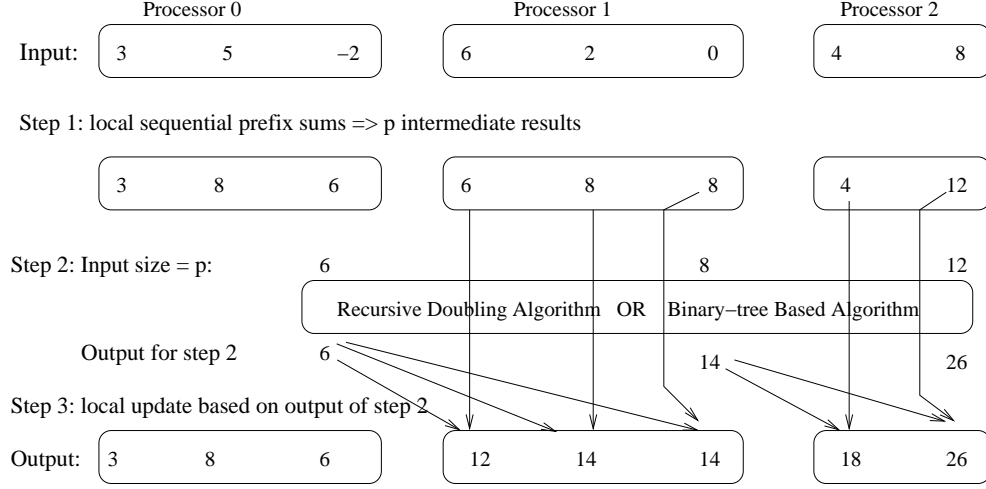
When the pre-built communication patterns were introduced in the previous chapter, it was assumed that each processor owns and works on only one input element. This means the number of processors p is equal to the number of input elements n , which is an over-simplified assumption.

However, there is a standard 3-step technique for solving the more general situation where $p < n$:

- Step 1: Each processor applies the fastest sequential algorithm on its own local chunk of data, of size $O(n/p)$, and gets some intermediate results.
- Step 2: Processors communicate with each other and do global computation on these p intermediate results using PRAM parallel algorithms.
- Step 3: Each processor does left-over local computations to get the final results based on the results of the previous global computations.

The underlying logic of this three-step technique is that after the first step, there are only p intermediate results, one for each processor; so a parallel problem using these intermediate results as input has input size equal to p . Then the algorithms introduced in the following sections of this chapter can be applied to the input set of size p and use the pre-built communication patterns to guarantee correct synchronizations.

In the STAPL generic implementation of parallel prefix sums, the three-step technique is used as follows:



Step 1 applies sequential prefix sums on each processor;
Step 2 applies RD or BT algorithm on p intermediate results;
Step 3 finalizes local computation based on the output of step 2.

Figure 6: Example of applying three-step technique

- Step 1: Each processor applies a sequential prefix sums algorithm on its own $O(n/p)$ input elements, and gets its own local total sum as the processor's intermediate result.
- Step 2: The RD or BT algorithms are applied to compute the prefix sums of the p intermediate results, along with corresponding communication pattern to guarantee correct synchronizations.
- Step 3: Each processor uses the globally computed prefix sums of intermediate results to update its own $O(n/p)$ elements.

Figure 6 follows the three steps for computing prefix sums of an input set of 8 integers, $[3, 5, -2, 6, 2, 0, 4, 8]$, on 3 processors, rather than the 8 processors needed in the PRAM model.

The input size is n for both Step 1 and Step 3. Each processor simply scans $O(n/p)$ elements one by one. So the complexity in these two steps is Time = $\Theta(n/p)$ and Work = $\Theta(n)$, assuming the input data are distributed evenly across the p processors.

The input size for Step 2 is p . Different algorithms (RD or BT), together with different communication patterns used for implementation, have different effects on the complexity of this step.

B. RD Algorithm

1. RD Algorithm and Theoretical Complexity

The recursive doubling technique was first used by Kruskal, Rudolph, and Snir [25] to solve parallel prefix sums problem. The input sequence x can be regarded as a linked list of elements with links representing their preceding / succeeding relationships.

This algorithm can be described by the following pseudo-code. d is used to represent the distance for the update operation $x[i] = y[i-d] + x[i]$. *Bin_op* is the binary associative operator, which could also be as simple as addition, but could be very complicated. *Parallel_for* means every processor executes this for-loop in parallel, with each working on one iteration.

Recursive doubling (RD) Algorithm:

Input: A set of elements "x" a copy "y";

A binary associative operator "bin_op".

Output: Prefix sums of original input elements
 stored in "x".

begin

```

for (d = 1; d < n; d = d * 2)
{
    parallel_for (i=d; i<n; i=i+1) {
        // parallel_for: each processor
        // works on one "i" of the for-loop
        // at the same time, i.e., i-th
        // processor does:
        // x[i] = bin_op( y[i - d], x[i] )
        // bin_op: the binary associative op
    }

    parallel_for (i=0; i<n; i=i+1) {
        // parallel_for: each processor works
        // on one "i" of the for-loop at the
        // same time, i.e., i-th processor does:
        // y[i] = x[i];
    }
}
end

```

Figure 7 exemplifies how to use the RD algorithm to compute the prefix sums of the input set $[3, 5, -2, 6, 2, 0, 4, 8]$ on 8 processors in parallel. The binary associative operator is simple addition in this example.

The distance d for each step changes from 1 to 2, then to 4, to 8, and so on until $p/2$. So it takes $\Theta(\log_2(p))$ iterations to finish the work. In each iteration, processor i updates $x[i]$ or writes $y[i]$ once; both are constant time operations. So, the total

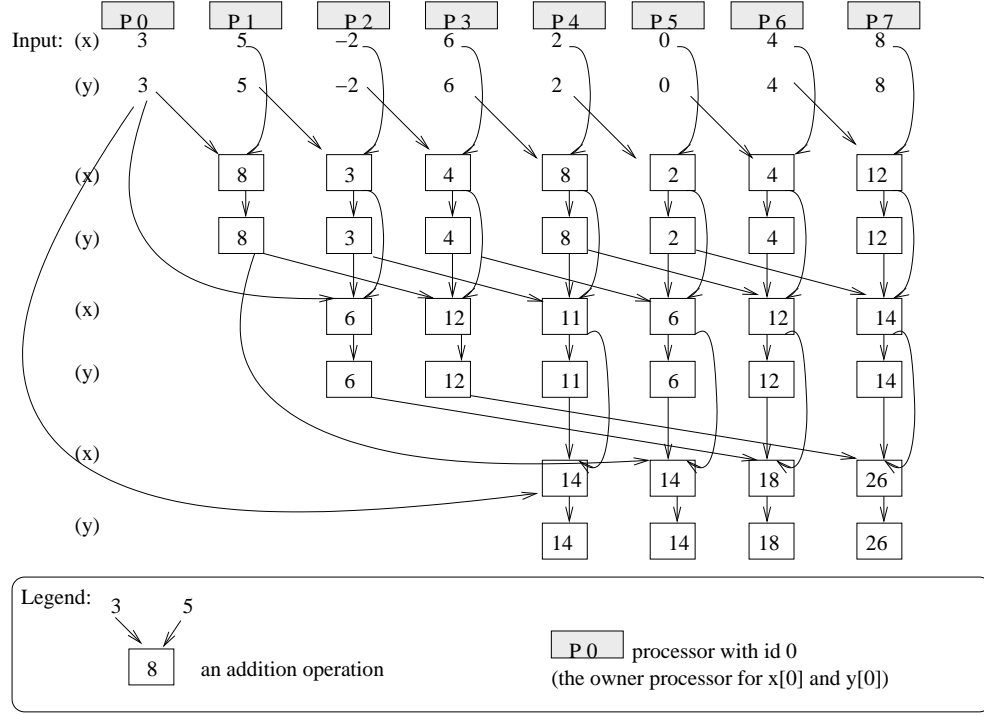


Figure 7: Example of using the RD algorithm on 8 processors

complexity for the RD algorithm in Step 2 is $\text{Time} = \Theta(\log_2(p))$ and $\text{Work} = \Theta(p^* \log_2(p))$.

2. Barrier Synchronization Implementation for RD Algorithm

As explained in Section A, in order to implement the RD algorithm with barrier synchronization, global barriers are put at the beginning and end of each parallel iteration using Communication Pattern I. The only thing that needs to be specified is the operation for each task, i.e., the work function object of the pRange. After that, the synchronizations between tasks are correctly guaranteed.

The RD algorithm's framework looks like the following pseudo-code.

Framework of RD algorithm with

Barrier Synchronization option:

(Using Communication Pattern I)

Assumption: $p = n$ (for step 2 only)

Input set "x" and its copy "y" are stored
in 2 pContainers.

Input: A pRange "pr" which has access to two views,
one for "x" and the other for "y";

A binary functor "bin_op" for prefix sums.

Output: Prefix sums of elements in "x" stored in
the view0 of "pr", which is the View for
input pContainer "x".

```
template <typename PRange, typename BinaryFunction>
p_prefixsums_RD_Barrier (PRange& pr,
                        BinaryFunction bin_op)
{
    for (d = 1; d < n; d = d*2 )
    {
        plus_workfunction<PRange,
                        BinaryFunction> plus_wf(d);
        // work function: x[i]=bin_op(y[i-d],x[i]);
        // the "distance" d for this level is
        // passed as an argument to this work
        // function because all the tasks at this
        // level have same "d"s.
```

```

pr.set_task(plus_wf);
p_for_all (pr);
    // applies plus_wf on each element of
    // 'x' and 'y'
rmi_fence();
    // barrier to guarantee every processor
    // has updated x[i]

save_workfunction<PRange> save_wf;
    // work function for y[i] = x[i];
pr.set_task(save_wf);
p_for_all(pr);
    // applies save_wf on each element of
    // 'x' and 'y'
rmi_fence();
    // barrier to guarantee every process
    // has saved updated x[i] to y[i]
}
}

```

Since the RD algorithm needs to operate on the elements of the copy y , the input pRange object pr for the algorithm has to support access to two Views, one for x 's View and the other for y 's View. This is supported by STAPL's combined-view pRange.

Similar to the original algorithm's pseudo-code, the STAPL generic implementation applies a work function on a pRange in each parallel step by calling *p_for_all*.

A DDG does not need to be specified explicitly for the pRange object here, because Communication Pattern I by default uses the DDG including one task per subrange and no orderings between tasks.

The first work function type *plus_workfunction* has its *operator()* applying the input binary operator on $y[i-d]$ and $x[i]$ on one subrange in each task. This only deals with the second step; with $p=n$, each local chunk on one processor has only one single element. That is why only the first element in view0 and view1 in the subrange are worked on. The following is the pseudo-code for this work function.

Work function "plus_workfunction" in RD

Algorithm's Barrier implementation:

(Operator() applies bin_op on each subrange.)

```
template <typename PRange, typename BinaryFunction>
class plus_workfunction :
    public work_function_base<PRange>
{
private:
    BinaryFunction bin_op;
    // the input binary associative operator
    int d;
    // the distance for this parallel step
public:
    workfunction(BinaryFunction _bin_op, int _d) :
        bin_op(_bin_op), d(_d) {}
```

```

void operator() (typename
                  PRange::subview_set_type& subrange_data)
{
    int offset = at<0>(subrange_data)->offset();
        // the offset for this subrange, i.e., "i"
    if ( i < d) {
        ; // no-op
    } else {
        x_iterator xit =
            at<0>(subrange_data)->begin();
            // the element x[i]
        y_iterator yit =
            at<1>(subrange_data)->begin() - d;
            // the element y[i-d]
        *xit = bin_op(*yit, *xit);
            // x[i] = bin_op(y[i-step], x[i])
    }
}
};

```

In order for each task to know whether it should apply the binary operation (like addition) or just do *no-op*, the task has to make a decision based on the distance d for the current step and the subrange's index i . Basically, if $i < d$, this task should do *no-op*. Since each call for *p_for_all* is working for only one single step, this step's distance d is simply passed as an argument for this work function.

The *save_workfunction* is very similar, but simpler.

Work function "save_workfunction" in RD

Algorithm's Barrier implementation:

(Operator() saves $x[i]$ to $y[i]$.)

```
template < typename PRange >
class save_workfunction :
    public work_function_base<PRange>
{
private:
    // no more need for bin_op and d
public:
    void operator() (typename
                    PRange::subview_set_type& subrange_data)
    {
        x_iterator xit =
            at<0>(subrange_data)->begin();
        // the element x[i]
        y_iterator yit =
            at<1>(subrange_data)->begin();
        // the element y[i]
        *yit = *xit;
        // y[i] = x[i]
    }
};
```

Since there are $\Theta(\log_2(p))$ parallel steps in the RD algorithm, the number of

barriers is also $\Theta(\log_2(p))$. Each processor is required to participate in each step, so the total number of synchronizations is $\Theta(p * \log_2(p))$.

3. Point-to-point Synchronization Implementation for RD Algorithm

As explained in the previous section, barrier synchronization causes some processors to do unnecessary *no-ops* at some parallel steps based on the algorithm itself.

Alternately, point-to-point synchronization eliminates the requirement for all processors to participate in all parallel steps, i.e., it avoids the unnecessary *no-ops*.

Basically, when implementing a parallel algorithm using point-to-point synchronization, it is necessary to only respect the point-to-point synchronizations between tasks. For the RD algorithm, this is guaranteed by simply using Communication Pattern II in Section B.

The following is the pseudo-code for the implementation of the RD algorithm with point-to-point synchronization.

Framework of RD algorithm with
Point-to-point Synchronization option:
(Using Communication Pattern II)

Assumption: $p = n$ (for step 2 only);

Input "x" and its copy "y" are stored in 2
pContainers.

Input: A pRange "pr" which has access to two views,
whose view0 is for "x", view1 is for "y";
A binary functor "bin_op" for prefix sums
operation.

Output: Prefix sums of elements in "x" are stored
into view0 of "pr".

```
template <typename PRange, typename BinaryFunction>
p_prefixsums_RD_Point2point (PRange& pr,
                             BinaryFunction bin_op)
{
    Build a new pRange "ipr" with same views
    as "pr", using an instance of "rd_dependence_
    oracle" as its DDG;
    workfunction wf(bin_op);
    // if the task-specific data "d" of the task
    // is 0, then the task does: y[i] = x[i];
    // else it does: x[i] = bin_op(y[i-d], x[i]).
    ipr.set_task(wf);
    p_for_all(ipr);
}
```

Point-to-point synchronization does not care about single parallel steps; instead, it builds and applies the DDG for the whole algorithm. Tasks in different parallel steps could run at the same time and tasks in later parallel steps could also run earlier as long as all synchronizations (caused by data dependences) are respected. This is why the value of d as task-specific data needs to be stored; it is statically computed based on the level for a task in *construct_tasks*.

Therefore, a unified work function object wf will work for any task. It does $x[i]$ = $bin_op(y[i-d], x[i])$ if the task is at an even level, otherwise it does $y[i] = x[i]$.

The following is the definition of the work function class. As mentioned before, since the $p=n$ situation is assumed, each subrange has only one single element for x and one single element for y .

Work function "workfunction" in RD

Algorithm's Point-to-point implementation:

(Operator() either updates $x[i]$ or saves it to $y[i]$.)

```
template < typename PRange, typename BinaryFunction >
class workfunction :
    public work_function_base<PRange>
{
private:
    BinaryFunction bin_op;
public:
    workfunction(BinaryFunction _bin_op) :
        bin_op(_bin_op) {}

    bool use_ddg_vertex_data() { return true; }

    // this method tells the run-time system
    // to pass task-specific data to the work
    // function's operator()

    void operator() (
        typename PRange::subview_set_type& subrange_data,
        typename PRange::DDGType::VERTEX& task_data)
```

```

{
    int d = task_data.d;

    // retrieve the "d" statically computed
    // and saved in the task during
    // "construct_tasks"
    if ( d > 0 )
    {
        // this is an even-level task
        {
            x_iterator xit =
                at<0>(subrange_data)->begin();
            // the element for "x" in this subrange
            y_iterator yit =
                at<1>(subrange_data)->begin() - d;
            // the element for "y" in subrange of
            // "d" distance away
            *xit = bin_op (*yit, *xit);
            // x[i] = bin_op(y[i-d], x[i])
        } else {
            // this is an odd-level task
            x_iterator xit =
                at<0>(subrange_data)->begin();
            // the element for "x" in this subrange
            y_iterator yit =
                at<1>(subrange_data)->begin();
            // the element for "y" in this subrange
            // at same position

```

```

        *yit = *xit;

        // y[i] = x[i]

    }

}

};

```

When *p_for_all* is executed by the run-time system of STAPL, those tasks that do not depend on any others are ready to be picked to run. When all of a task's predecessors are finished, it becomes ready to run. When all the tasks of a DDG are finished, the execution is complete.

C. BT Algorithm

1. BT Algorithm and Theoretical Complexity

[1] introduces another parallel prefix sums algorithm based on a balanced binary-tree structure which consists of two passes (the up sweep and the down sweep).

As introduced in Section C, in the up sweep, a binary-tree based pair-wise summation is applied: $x[1] = x[0] + x[1]$, $x[3] = x[2] + x[3]$, ..., $x[2^*i+1] = x[2^*i] + x[2^*i+1]$, ..., $x[n-1] = x[n-2] + x[n-1]$. This up sweep composes a balanced binary-tree. In the down sweep, sums are passed back from parents to children to compute the final results. The following pseudo-code describes how the BT algorithm works.

Binary-tree based (BT) Algorithm:

Input: A set of elements "x";

A binary associative operator "bin_op".

Output: Prefix sums of original input elements
stored in "x".

```

begin
    for (d = 1; d <= n/2; d = d*2) {
        // up sweep
        parallel_for (i = 2*d-1; i < n; i = i+2*d)
        {
            x[i] = bin_op( x[i-d], x[i] );
        }
        // each processor is in charge of summation
        // of one pair of elements in each step
    }

    for (d = n/4; d >= 1; d = d/2) {
        // down sweep
        parallel_for (i = 3*d-1; i < n; i = i+2*d)
        {
            x[i] = bin_op( x[i-d], x[i] );
        }
    }
end

```

Figure 8 shows an example of the BT algorithm used to compute prefix sums of the same input set $[3, 5, -2, 6, 2, 0, 4, 8]$ on 8 processors in parallel. The binary associative operator is still simple addition. This figure is essentially the same as Figure 5 except for the layout of tasks which focuses on the two-pass algorithms rather than the synchronization structure.

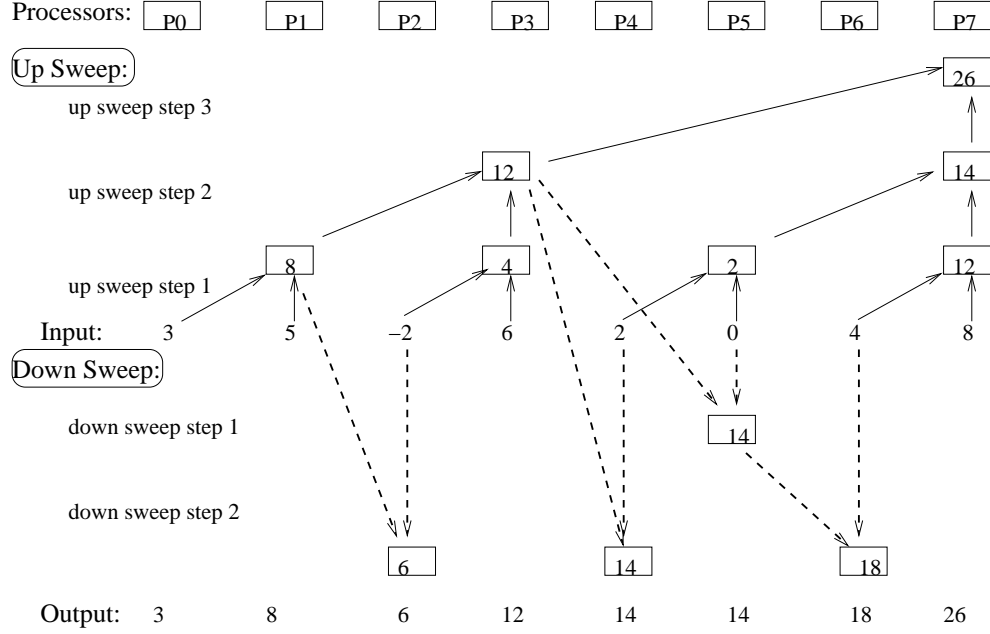


Figure 8: Example using the BT algorithm on 8 processors

In the up sweep of the BT algorithm, the synchronization distance d changes from 1 to 2, 2 to 4, 4 to 8, and so on until $p/2$ (same as the RD algorithm). It also takes at least $\Theta(\log_2(p))$ iterations to finish. However, based on the example, many processors only need to do operations in a few iterations. In the first step of the up sweep, $p/2$ operations are executed; in the second step, $p/4$ operations are executed, and so on. Following a balanced binary tree, only $p/2 + p/4 + p/8 + \dots + 1 = \Theta(p)$ operations are executed. At the end of this up sweep, $p/2$ elements have already been updated to their final results. In the down sweep, only the other $p/2$ elements need to compute their final results by executing one operation. This leads to constant time on each processor and $p/2$ work in total only. So, all together, Time = $\Theta(\log_2(p))$ and Work = $\Theta(p)$.

2. Barrier Synchronization Implementation for BT Algorithm

In Section C, it was assumed that the input size n is a power of 2. For an arbitrary natural number n , the second for-loop needs to be modified by rounding up its loop limit from n to the next power of 2.

This is the same for the implementation of the RD algorithm with barrier synchronization. Implementing the BT algorithm with Barrier synchronization simply means building the work function for each task and applying Communication Pattern I (*p_for_all* and barrier operation *rmi_fence*) to compose a sequence of parallel steps.

Similar to the algorithm's original pseudo-code description, the implementation is composed of two for-loops, one for each sweep. In each for-loop, each parallel step (*parallel_for*) is implemented as a *p_for_all* on a pRange, whose work function always does $x[i] = \text{bin_op}(x[i-d], x[i])$. The pRange object does not need to specify a DDG because it by default uses a DDG which has only one task on each subrange and no orderings between tasks based on Communication Pattern I. *Rmi_fence()* guarantees that processors finish each parallel step at the same time.

Framework of BT algorithm with
Barrier Synchronization option:
(Using Communication Pattern I)

Assumption: $p = n$ (for step 2 only)

Input set "x" is stored in a pContainer.

Input: A pRange "pr" which has access to a View
for "x";

A binary functor "bin_op" for prefix sums.

Output: Prefix sums of original elements stored
in view0 of "pr", which is the View for "x".

```
template < template PRange, typename BinaryFunction >
p_prefixsums_BT_Barrier (PRange& pr,
                        BinaryFunction bin_op)
{
    for (d = 1; d <= n/2; d = d*2) {
        // up sweep
        Workfunction<PRange, BinaryFunction>
            wf (bin_op, d);
        // work function for
        // x[i] = bin_op(x[i-d], x[i]);
        pr.set_task(wf);
        p_for_all(pr);
        // one up-sweep step
        rmi_fence();
        // barrier to guarantee this step's
        // computations are all finished
    }

    for (d = n/4; d >= 1; d = d/2) {
        // down sweep
        Workfunction<PRange, BinaryFunction>
            wf(bin_op, d);
        // same work function as in up sweep:
```

```

        // x[i] = bin_op(x[i-d],x[i]);
pr.set_task(wf);
p_for_all(pr);
        // each function does down-sweep
        // computation
rmi_fence();
        // barrier to guarantee each level's
        // down-sweep computations are all finished
    }
}

```

Again, in order for each task to decide whether it should apply the binary operation or do a *no-op*, the synchronization distance d for each step is passed as an argument to the work function.

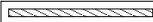
The processors' actions in the BT algorithm with barrier synchronization are shown in Figure 9. Compared with Figure 8, this figure has many *no-ops* where processors do not need to do any operations required by the algorithm; but they have to participate due to the requirement of barrier synchronization.

In the barrier implementation of the BT algorithm, there are also $\Theta(\log_2(p))$ steps, and $\Theta(\log_2(p))$ barriers. So the total number of synchronizations is also $\Theta(p * \log_2(p))$ since every processor has to participate in each parallel step.

3. Point-to-point Synchronization Implementation for BT Algorithm

The implementation of the BT algorithm using point-to-point synchronization uses Communication Pattern III as its pRange's DDG to guarantee the synchronizations between tasks. Its framework is represented by the following pseudo-code:

| | P0 | P1 | P2 | P3 | P4 | P5 | P6 | P7 |
|---------------|-------|----------|----------|----------|----------|----------|----------|----------|
| Input: | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 |
| up sweep | no-op | x1=x0+x1 | no-op | x3=x2+x3 | no-op | x5=x4+x5 | no-op | x7=x6+x7 |
| step 1 | | | | | | | | |
| up sweep | no-op | no-op | no-op | x3=x1+x3 | no-op | no-op | no-op | x7=x5+x7 |
| step 2 | | | | | | | | |
| up sweep | no-op | no-op | no-op | no-op | no-op | no-op | no-op | x7=x3+x7 |
| step 3 | | | | | | | | |
| down sweep | | | | | | | | |
| step 1 | no-op | no-op | no-op | no-op | no-op | x5=x3+x5 | no-op | no-op |
| down sweep | | | | | | | | |
| step 2 | no-op | no-op | x2=x1+x2 | no-op | x4=x3+x4 | no-op | x6=x5+x6 | no-op |

 Barrier
 no-op: does nothing real work but has to participate the step

Every processor has to participate in each parallel step.

Figure 9: Processors' actions in the BT algorithm with barrier synchronization option

Framework of BT algorithm with

Point-to-Point Synchronization option:

(Using Communication Pattern III)

Assumption: $p = n$ (for step 2 only)

Input set "x" is stored in a pContainer.

Input: A pRange "pr" which has access to a View
for "x";

A binary functor "bin_op" for prefix sums.

Output: Prefix sums of original elements stored in
view0 of "pr", which is the View for "x".

```
template < typename PRange, typename BinaryFunction >
p_prefixsums_BT_Point2point (PRange& pr,
```

```

                                BinaryFunction bin_op)
{
    Build a new pRange "ipr" with the same view as "pr",
    but whose DDG is an instance of "bt_dependence_oracle";
    workfunction wf(bin_op, step);

    // this workfunction does:
        // x[i]=bin_op(x[i-d], x[i])

    ipr.set_task(wf);
    p_for_all(ipr);
}

```

The usage of pre-built communication patterns significantly simplifies the implementation of the parallel algorithm. The only issue for a parallel programmer is to implement the work function object, which is extremely simple.

In the BT algorithm, as shown in Figure 5, each task has roughly two outgoing dependence edges. The number of synchronizations is roughly twice of the number of tasks. In the j -th parallel step ($j=0,1,2,\dots, \log_2(p)-1$), there are $p/2^j$ tasks, so the total number of synchronizations is: $2 * \sum_{j=0}^{\log_2(p)-1} (p/2^j) = \Theta(p * \log_2(p) - p)$.

Although asymptotically speaking, $\Theta(p * \log_2(p) - p) = \Theta(p * \log_2(p))$, it will be shown in Section B that this makes some performance difference,

D. Complexity of RD and BT Algorithms with Synchronizations

As shown in Section A, Step 1 (local sequential prefix sums on each processor) and Step 3 (local updates for elements on each processor) have complexity Time = $\Theta(n/p)$ and Work = $\Theta(n)$. This is independent from synchronization options.

The choice of synchronization options affects the theoretical complexity of Step

Table 1: Theoretical Complexity of RD Algorithm with Sync. Options

| Complexity | Without sync. | Barrier sync. | Point-to-point sync. |
|------------|-----------------------------|-----------------------------|-----------------------------|
| Time | $\Omega(n/p + \log_2(p))$ | $\Theta(n/p + \log_2(p))$ | $\Theta(n/p + \log_2(p))$ |
| Work | $\Omega(n + p * \log_2(p))$ | $\Theta(n + p * \log_2(p))$ | $\Theta(n + p * \log_2(p))$ |

Table 2: Theoretical Complexity of BT Algorithm with Sync. Options

| Complexity | Without sync. | Barrier sync. | Point-to-point sync. |
|------------|---------------------------|-----------------------------|---------------------------|
| Time | $\Omega(n/p + \log_2(p))$ | $\Theta(n/p + \log_2(p))$ | $\Theta(n/p + \log_2(p))$ |
| Work | $\Omega(n + p)$ | $\Theta(n + p * \log_2(p))$ | $\Theta(n + p)$ |

2 in the implementation. The theoretical complexity considering synchronization options is described in Table 1 and Table 2, listed together with the complexity without synchronization cost.

Based on the two tables, it is obvious that for the BT algorithm, the point-to-point synchronization option is more efficient than the barrier synchronization option. It is expected that better performance will occur with point-to-point synchronization.

CHAPTER V

PERFORMANCE OF IMPLEMENTATION

A. Optimization for Step 1 in the Implementation

As analyzed in Section A each parallel prefix sums implementation scans each input element twice, once in Step 1 and Step 3. Theoretically speaking, if the time spent on synchronization is small enough, the best speedup compared to sequential prefix sums would be $p/2$.

However, a simple optimization can be applied to the implementation to get a practical speedup better than $p/2$. Rather than applying the sequential prefix sums algorithm on each processor's local chunk of data in Step 1, let each processor read every local element and sum them up. Since Step 3 always needs to read and write each local element, the update operations in Step 1 can be avoided. This simply modifies the example in Figure 6 to Figure 10.

B. Experimental Results

The experimental performance results of the generic implementation of parallel prefix sums algorithms are compared with the sequential prefix sums algorithm in STL. The parallel version's input was a parallel container in STAPL, pArray, while the sequential version used an STL vector. Both containers had 512M integers, which needed 2GB storage on the experimental platform.

Experiments were done on a machine at the National Energy Research Scientific Computing Center (NERSC), a 712-CPU Opteron cluster running Linux. It has 356 dual-processor nodes interconnected with an InfiniBand network. The memory size for each node is 6GB, and the cache size for each processor is 1MB. The input set

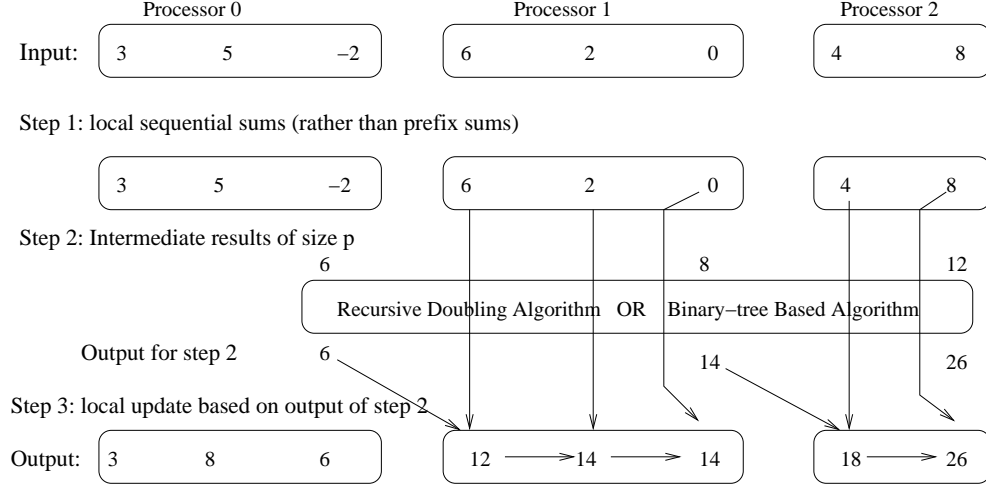


Figure 10: Example of the generic implementation with optimization in step 1 ($p=3$, $n=8$)

chosen for the experiments does not fit in the cache, but always fits in the memory. Experiments are repeated at least three times on 1, 2, 8, 16 and 64 processors respectively. In order to avoid the effects of memory contention, all processors used in the experiments reside on separate nodes.

Figure 11 shows the speedup for four combinations of algorithms and synchronization options in the generic implementation: the RD algorithm + barrier synchronization, the BT algorithm + barrier synchronization, the RD algorithm + point-to-point synchronization, and the BT algorithm + point-to-point synchronization.

Based on this figure, the speedup for any of the four combinations is as good as can be expected and higher than $p/2$. When comparing the four combinations in the implementation, it is found that when p is smaller, there is not much difference in performance because Step 1 and Step 3 dominate the total running time. However, when using more processors, the figure shows the following order in performance.

- The BT algorithm + point-to-point synchronization is the best, and much

Speedup of parallel prefix sums over sequential version (with sequential container) in STAPL

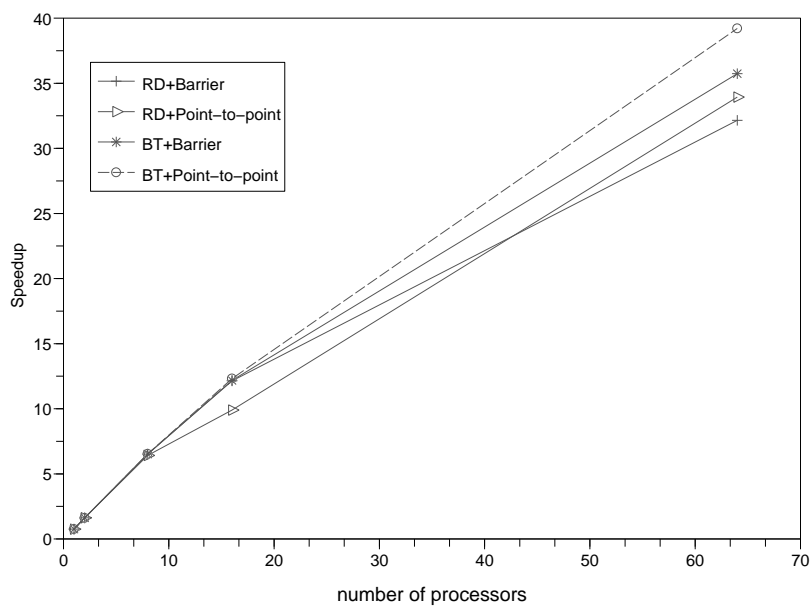


Figure 11: Speedup of parallel prefix sums (input size: 512M integers)

better than the other three (because it has less synchronization to perform);

- The RD algorithm + point-to-point synchronization is faster than the RD algorithm + barrier synchronization when p is big enough; as shown in Section 3 its number of synchronizations is actually $\Theta(p * \log_2(p) - p)$, rather than exactly $\Theta(p * \log_2(p))$.

It is concluded that for both the BT and RD algorithms, the point-to-point synchronization leads to better performance than barrier synchronization. For the RD algorithm, the point-to-point option does not show an advantage until p grows large enough; for the BT algorithm, the point-to-point option has obvious advantages because it enables the implementation to approach the lower bound for the number of synchronizations required by the algorithm.

CHAPTER VI

APPLICATIONS

A. Parallel Radix Sort

As listed in Section B, parallel prefix sums can be used to implement a parallel radix sort. This section briefly describes the implementation of a parallel radix sort as a numeric application of the parallel prefix sums implementation presented above.

Since radix sort is a sequential loop over a stable sort (typically counting sort) on each r bits of the input integers starting from the least significant bits, the parallel implementation in STAPL is simply a loop around a parallel counting sort.

The basic idea of counting sort is, if the range for the input values to sort is $[0, 2^r]$, count the occurrences of each value and remember the numbers of occurrences in an array *count* of size 2^r . Then apply the prefix sums algorithm on the array *count*, such that the value of *count*[j] is the total occurrences of values less than j ; i.e., it is the correct place to put the first element of value j in the final sorted sequence. Finally, for each value j , every input element with this value is written to the position *count*[j] of the final sequence and *count*[j] is incremented by 1 each time.

This sequential algorithm can be modified to the following parallel algorithm:

- Step 1. Each processor applies the sequential counting sort on its local chunk of data with size $O(n/p)$. The counting results are saved in a sequential vector *local_count* of size 2^r . All of the p *local_count* vectors compose a pContainer (pArray<vector>) *pcount*, with each *local_count* as one element of *pcount*.
- Step 2. Processors use the generic implementation of the RD or the BT parallel prefix sums algorithms on the pContainer *pcount* by specifying a binary associative operator on two *local_count* vectors.

- Step 3. Each processor i goes through each of its local elements, writes it to the global position $pcount[i][j]$ if its value is j , and increments $pcount[i][j]$ by 1.

At the end of Step 1, $pcount[i][j]$ remembers the number of occurrences of value j on processor i . Parallel prefix sums on $pcount$ in Step 2 change $pcount[i][j]$ to the number of occurrences of value j on all processors 0 to $i-1$. These can be converted to the total number of occurrences of all values less than j on processor 0 to $i-1$ by a sequential prefix sums computation.

The binary associative operation used in Step 2 simply adds up the corresponding elements (at the same position) from two vectors and returns another vector.

Binary associative function object

for parallel radix sort:

Functionality: Given two "local_count" vectors,
 the operator() returns a vector whose each
 element is the sum of corresponding elements
 in the two vectors.

```
class p_radix_sort_vectorplus : public binary_function<
    vector<int>, vector<int>, vector<int> >
{
public:
    vector<int> operator()(const vector<int>& v1,
                           const vector<int>& v2)
    {
        int len = std::min(v1.size(), v2.size() );
        vector<int> v;
```

```

        for (int i=0; i<len; ++i) {
            v.push_back(v1[i] + v2[i]);
        }
    }
};

```

Speedups of the parallel algorithms on pArray compared to sequential radix sort on an equivalent STL vector are shown in Figure 12. The experiments were done on the same machine used in Section B. The input dataset has 128M integers, which still does not fit in the cache, but always fits in the memory.

Since the computation of Step 1 and Step 3 in the parallel radix sort dominates the total running time, there is only a slight difference in the performance of using any of the four combinations (in Section B) in Step 2. Since both the sequential version and the parallel version need to scan all input elements twice, a speedup close to p would be expected. However, the experimental results did not achieve this theoretical expectation due to the overhead of reading and writing the pContainer *pcount* repeatedly. This is much heavier than reading and writing a STL sequential container.

An issue that has to be clarified for this result is that the speedup is valid only for a special kind of input data, where the distribution of chunks on processors makes every element not need to be moved across processors when being sorted. That is, the input chunks of elements are already sorted between processors. The reason to choose this kind of input data for the experiments is to compare the real speedup of a parallel implementation over a sequential version, excluding the cost of moving elements across processors, which is heavily dependent on the properties of the input data and the underlying interconnection of processors.

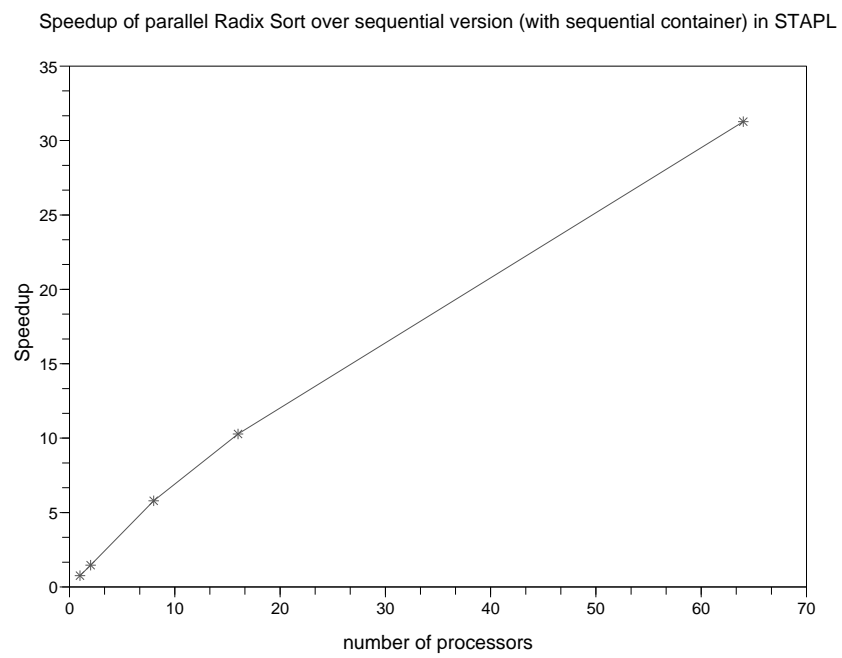


Figure 12: Speedup of parallel radix sort (input size is 128M integers)

Another numeric application that can be built on top of the implementation for parallel prefix sums algorithms is *p-partition*, which is implemented similarly to the parallel radix sort. *P-partition* is the parallel equivalent of STL's *partition* algorithm. This algorithm reorders the elements in the input sequence based on an input function object *pred*, such that the elements that satisfy *pred* precede the elements that fail to satisfy it.

Similar to parallel radix sort, *p-partition* has three steps:

- Step 1. Each processor applies the sequential partition algorithm on its own chunk of data, remembering the sizes of each left and right parts in a pair of integers $\langle \text{left-size}, \text{right-size} \rangle$. All the p pairs compose a pContainer (pArray<pair>) *pcount*.
- Step 2. Processors use the generic parallel prefix sums implementation on all the left-sizes in *pcount*, and again on all the right-sizes in *pcount*, to decide the total number of elements that should occur previous to each part.
- Step 3. Each processor copies its own left and right parts to the correct positions based on the result of *pcount* in Step 2.

This is very similar to parallel radix sort. Essentially, *p-partition* can be converted to a simplified parallel radix sort with an input value range of only [0,1]. When an element satisfies the *pred* condition, it is regarded as 0, otherwise it is 1.

The simplicity in implementing parallel radix sort and parallel partition algorithm comes from the generic nature of the implementation of parallel prefix sums. In the next section, another category of parallel applications is introduced that can also be built using this generic implementation together with another important parallel technique, the parallel Euler tour.

B. Tree Applications

This section presents four parallel tree applications. All of them are similar to each other, and all use two important techniques in parallel computation, the Euler tour technique and parallel prefix sums.

These four applications and their algorithms are introduced in [1]:

- Rooting a Tree: Given a tree and a vertex r as its root, compute the parent for each vertex in the tree.
- Post-order Numbering: Compute the post-order number of each vertex in a binary rooted tree.
- Computing the Vertex Level: Compute the level of each vertex, the distance (number of edges) between the vertex to the root r .
- Computing the Number of Descendants: For each vertex, compute the number of vertices in the sub-tree rooted at the vertex.

All of these are based on the Euler tour technique; the algorithm and implementation in STAPL are introduced in Section 1. Since the last three applications all use the information computed in rooting a tree and follow the same strategy, the second subsection presents the implementation of rooting a tree and the third subsection briefly introduce the other three applications.

Since the goal of this section is to further show the wide application of the parallel prefix sums implementation, the implementation details of the tree applications are not covered. For the same reason, the complexity analysis and performance results are only briefly presented because they are dependent on other factors besides the prefix sums step; e.g., the properties of input tree, the selection of root r , the distribution of the tree, and so on.

1. Euler Tour Technique

The following is the definition for Euler tour from [1]:

Let $T=(V,E)$ be a given tree and $T'=(V,E')$ be the directed graph obtained from T when each edge $(u,v) \in E$ is replaced by two arcs $\langle u,v \rangle$ and $\langle v,u \rangle$. Since the indegree of each vertex of T' is equal to its out-degree, T' is an *Eulerian* graph; that is, it has a directed circuit that *traverses each arc exactly once*. It turns out that an *Euler circuit* of T' can be used for the optimal parallel computation of many functions on T .

An Euler circuit of $T'=(V, E')$ can be defined by specifying the *successor function* s mapping each arc $e \in E'$ into the arc $s(e) \in E'$ that follows e on the circuit.

A suitable successor function is: For each vertex $v \in V$, we fix a certain ordering on the set of vertices adjacent to v - say, $\text{adj}(v) = \langle u_0, u_1, \dots, u_{d-1} \rangle$, where d is the degree of v . We define the successor of each arc $e = \langle u_i, v \rangle$ as: $s(\langle u_i, v \rangle) = \langle v, u_{(i+1) \bmod d} \rangle$, for $0 \leq i \leq d-1$.

This definition for successor function is implemented as [1]:

A successor function to build Euler tour:

Input: An euler_arc $\langle u,v \rangle$ (from u to v)

Output: Another euler_arc $\langle v,w \rangle$ which will occur
next to $\langle u,v \rangle$ in an Euler tour

procedure successor($\langle u,v \rangle$)

```

{
    if u is not the last one on v's adjacency list
        w = the next one following u on v's adjacency list
    else
        w = the 1st one on v's adjacency list
    return <v,w>
}

```

An example of an Euler tour is shown in Figure 13. The thick lines represent the edges of the input tree, while lines with arrow compose an Euler circuit. The table in the figure shows the successor function s for each arc.

The implementation of finding the Euler tour in the input parallel tree (a pContainer, pTree) in STAPL takes two steps.

- Step 1: Each processor applies the successor function on its local vertices and edges, trying to find sequences of edges as long as possible.
- Step 2: Processors communicate with each other to connect their sequences together to get the whole Euler tour.

The total Euler tour is stored in a parallel list (pList<euler_arc>), in which each element is an object of class *euler_arc* with three data members.

- *first*: the starting vertex ID.
- *second*: the ending vertex ID.
- *weight*: an integer to store the weight of the arc in applications, originally -1.

An execution example is given in Figure 14. The input tree has 10 vertices distributed across 3 processors.

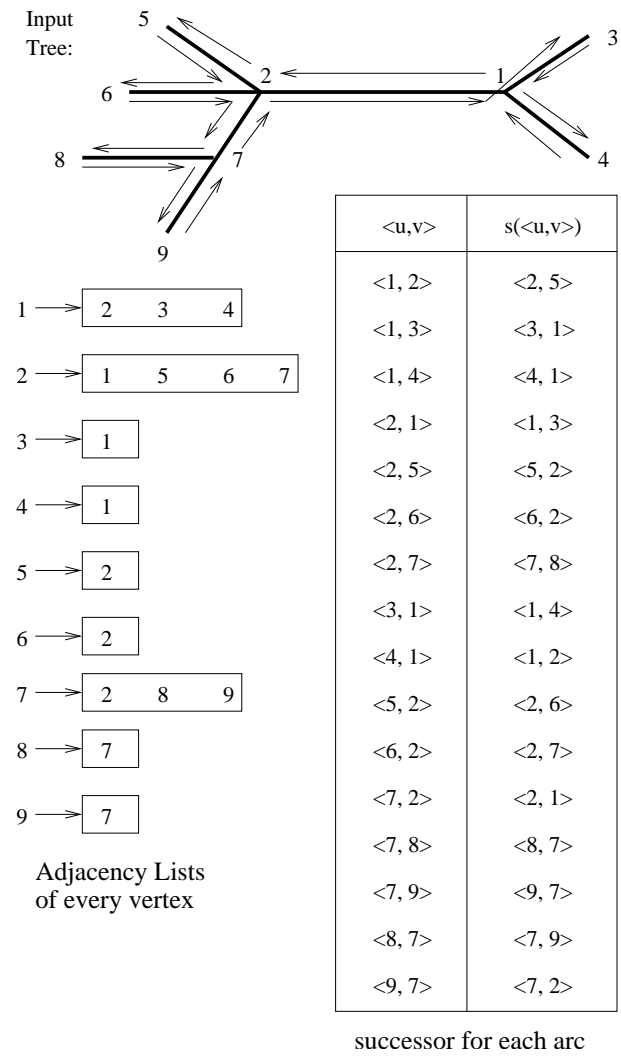


Figure 13: Example of Euler circuit with successor function s [1]

In the first step, each processor applies successor function s to build local sequences of *euler_arcs*. When a processor encounters an out-going cross-processor arc (called *cut-arc* in STAPL), it stops the construction for this sequence and starts working on other arcs. These local sequences are connected together in Step 2 by communicating among processors.

Since the number of communications in this step is equal to the number of local sequences, these sequences should be as long as possible. This is achieved by starting and finishing local sequences at *cut-arcs*.

In Figure 14, the vertical solid lines connect *euler_arcs* to local sequences, while the dotted lines connect these sequences together to become an Euler circuit. When the starting arc is specified ($\langle 7,0 \rangle$ in the example), the circuit is broken to become an Euler tour.

Since the number of edges in a tree is $\Theta(n)$, where n is the number of vertices of the tree, the total work to build a list of all edges is also $\Theta(n)$. Suppose the tree is distributed evenly across processors, then the theoretical complexity for parallel implementation to build an Euler tour would be $\Theta(n/p)$.

2. Rooting a Tree

The following definition and algorithm for rooting a tree comes from [1].

Consider a tree $T=(V, E)$ rooted at vertex r , where V is the set of vertices in T , E is the set of edges in T , r is a vertex in V that is named as the tree's root. The problem of "rooting a tree" is to find the parent vertex $p(v)$ of each vertex v except r , which is the vertex that stands immediately previous to v on the path from r to v .

Input: A tree T defined by the adjacency lists of

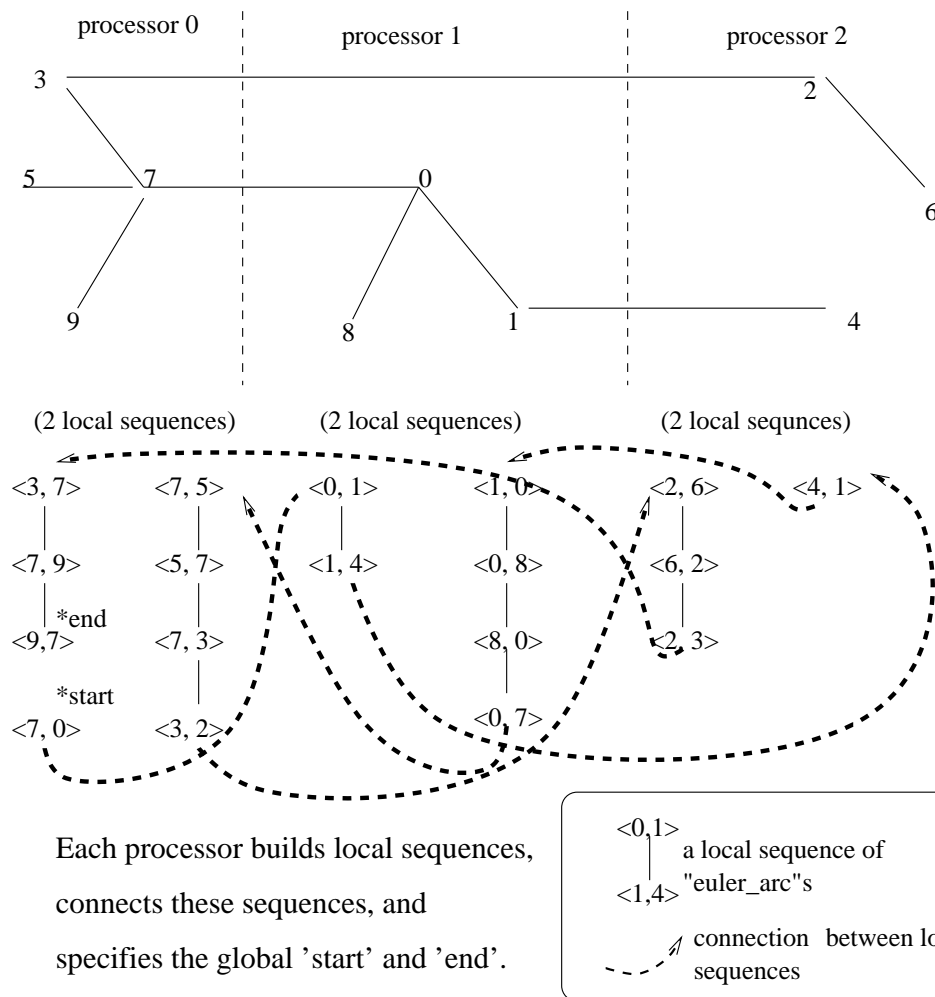


Figure 14: Example of building an Euler tour in parallel

its vertices;

A special vertex r

Output: For each vertex v except r , the parent $p(v)$.

begin

1. Find an Euler tour in T defined by function s ;
2. Identify the last vertex u appearing on the adjacency list of r , and set $s(\langle u, r \rangle) = 0$;
3. Assign a weight of 1 to each arc $\langle x, y \rangle$ on the Euler tour, and apply parallel prefix sums on the list of arcs defined by s ;
4. For each arc $\langle x, y \rangle$, set $x = p(y)$ whenever the prefix sum of $\langle x, y \rangle$ is smaller than the prefix sum of $\langle y, x \rangle$.

end

Step 1 is explained in previous section. Step 2 can simply be specified when connecting the local sequences for the whole Euler tour.

For Step 3, apply the generic implementation of parallel prefix sums algorithms on the Euler tour as a $pList\langle euler_arc \rangle$. The binary associative operation is defined as follows to add up the two arcs' weights and return an *euler_arc* with the sum as its weight. For example, given a sequence (with each *euler_arc*'s weight as 1):

$\langle 7, 0 \rangle[1] - \langle 0, 1 \rangle[1] - \langle 1, 4 \rangle[1] - \langle 4, 1 \rangle[1]$

applying prefix sums on this binary operation will return:

$\langle 7, 0 \rangle[1] - \langle 0, 1 \rangle[2] - \langle 1, 4 \rangle[3] - \langle 4, 1 \rangle[4]$.

Binary associative function to add

up two "euler_arcs".

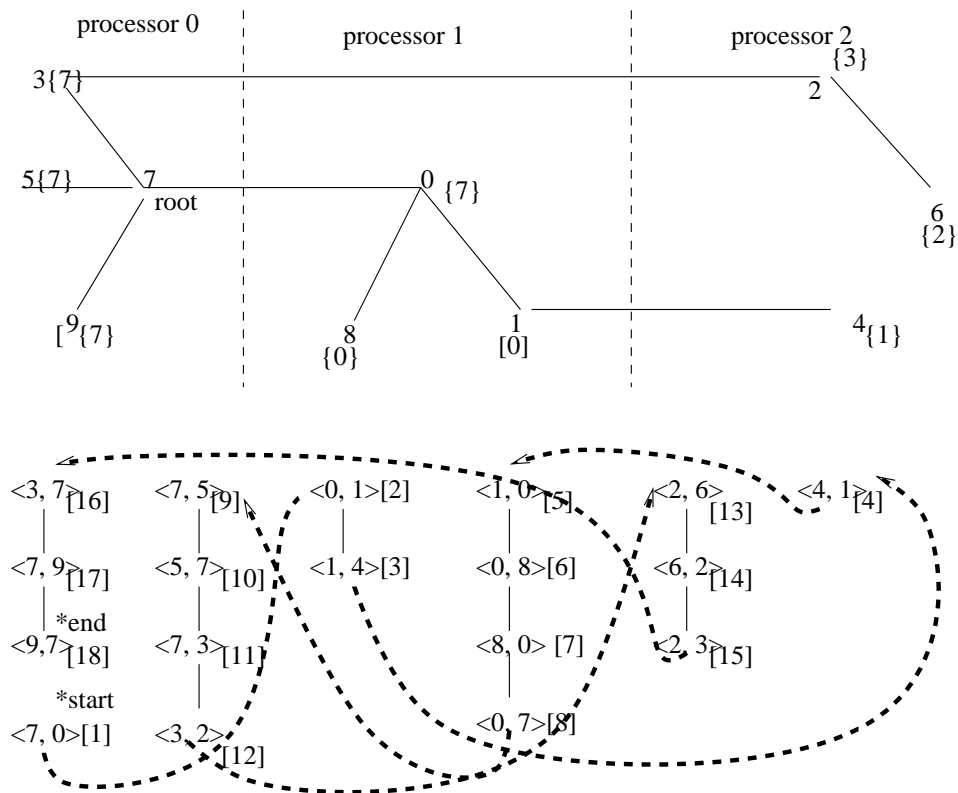
Functionality: Given two "euler_arcs":

```
<u,v>[weight1] and <v,w>[weight2],
return an euler_arc <v,w>[weight1+weight2].
```

```
class weight_plus : public binary_function<
    euler_arc, euler_arc, euler_arc>
{
public:
    weight_plus() {}
    weight_plus(const weight_plus& rp) {}
    euler_arc operator()(const euler_arc& e1,
                        const euler_arc& e2) const {
        return euler_arc(e2.first, e2.second,
                        e1.weight+e2.weight);
    }
};
```

For Step 4, the prefix sums of pairs of arcs $\langle x, y \rangle$ and $\langle y, x \rangle$ are compared to decide whether x is the parent of y or the reverse. If both x and y are on the same processor, this comparison is trivial. If they are located on two different processors, then these two processors do some communication and have only one processor compare them.

Figure 15 exemplifies Step 3 and Step 4 for rooting a tree for the same input tree as in Figure 14. The weights of *euler_arcs* are represented by the numbers in brackets beside the arcs; the parent of each vertex is in { } beside the vertex.



Rooting a Tree: set initial weight 1 to every arc in Euler tour, but 0 to the last one;
 apply generic implementation of prefix sums algorithm on the arcs.
 compare $\text{prefix-sum}\langle x, y \rangle$ and $\text{prefix-sum}\langle y, x \rangle$ to decide parents.

$3\{7\}$: vertex 3's parent is 7

$\langle 0, 1 \rangle [2]$: the prefix sums of the weight for arc $\langle 0, 1 \rangle$ is 2

Figure 15: Example of "rooting a tree"

3. Three Other Tree Applications

The other three tree applications all follow a similar strategy as following [1]:

Input: A rooted tree (a rooted binary tree if computing post-order numbers), an Euler tour defined by the successor function, root vertex r , and parent $p(v)$ for each vertex v .

- Postorder Numbering:
 - 1. For each vertex $v \neq r$, assign the weights of $\langle v, p(v) \rangle$ as 1 and weights of $\langle p(v), v \rangle$ as 0;
 - 2. Perform prefix sums on the list of arcs;
 - 3. For each vertex $v \neq r$, set $post(v) = prefix-sum(\langle v, p(v) \rangle)$, for $v = r$, set $post(r) = n$;
- Computing the Vertex Level:
 - 1. For each vertex $v \neq r$, assign the weights of $\langle v, p(v) \rangle$ as -1 and weights of $\langle p(v), v \rangle$ as 1;
 - 2. Perform prefix sums on the list of arcs;
 - 3. For each vertex $v \neq r$, set $level(v) = prefix-sum(\langle p(v), v \rangle)$, for $v = r$, set $level(r) = 0$;
- Computing the number of Descendants:
 - 1. For each vertex $v \neq r$, assign the weights of $\langle v, p(v) \rangle$ as 1 and weights of $\langle p(v), v \rangle$ as 0;
 - 2. Perform prefix sums on the list of arcs;

- 3. For each vertex $v \neq r$, set $size(v) = prefix-sum(\langle v, p(v) \rangle) - prefix-sum(\langle p(v), v \rangle)$, for $v = r$, set $size(r) = n$.

All the three applications, together with rooting a tree, are actually implemented in one generic framework in STAPL which includes the following steps:

- Step 1. Initialization: Setting the initial weights in each arc's weight.
- Step 2. Prefix Sums: Applying the generic implementation of parallel prefix sums on the weighted Euler tour.
- Step 3. Finalization: Computing final results using the prefix sums of arcs, based on the requirement of each application.

4. Performance of Tree Applications

Experimental results are shown in Figure 16. When n (number of vertices in the input tree) becomes as big as 4,000 or 8,000, the speedup is super-linear, probably due to the fact that the distributed graphs now fit in the cache.

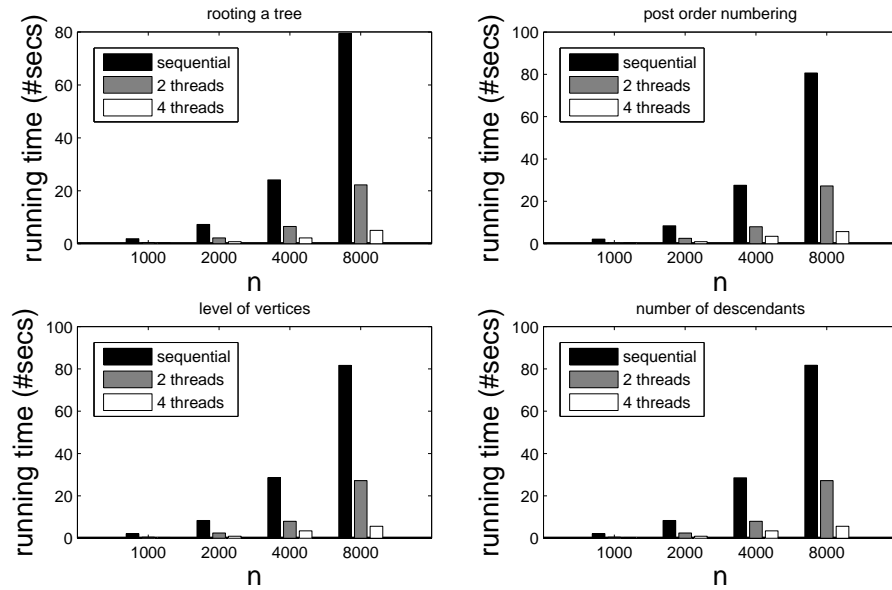


Figure 16: Performance of Euler tour applications

CHAPTER VII

CONCLUSION AND FUTURE WORK

A. Conclusion

In this thesis, the design and implementation of a generic implementation of parallel prefix sums in STAPL is presented.

The implementation has no extra assumptions about the input element type, the binary associative operator to be applied to these elements, the physical distribution of the input data on multiple processors, or the relationship between input size n and number of available processors p .

This implementation supports two different commonly-used prefix sums algorithms, the recursive doubling (RD) algorithm and the binary-tree based (BT) algorithm. It provides good examples for other parallel problems following similar techniques.

The implementation uses two different synchronization options: barrier synchronization and point-to-point synchronization. While the former is easier to implement, the latter has fewer synchronizations and thus better performance. All four combinations of algorithms and synchronization options give a reasonably good speedup in experiments.

Isolating the fixed communication patterns in concrete parallel algorithms, three pre-built communication patterns are designed to simplify the work of parallel programmers to a great extent. At the same time, they help to guarantee correct synchronizations in parallel algorithms.

The implementation is easily extended to two different categories of parallel applications. For numeric applications, a parallel radix sort algorithm is implemented.

For tree applications, four applications that also use Euler tour technique are implemented.

B. Future Work

This generic implementation is the first and currently the only pAlgorithm that uses point-to-point synchronization in STAPL. This actually motivates the implementation of FormulaDDG (rather than using a physical pGraph as DDG). All others use only barrier synchronization.

However, it is believed that there will be other kinds of algorithms that require group-based processor synchronization. In such cases, group-based barrier synchronization would benefit the algorithms the most. But a group-based barrier synchronization is not yet supported in STAPL.

When a run-time system's task scheduler picks ready tasks from a DDG, currently it just randomly picks any ready task to run. The scheduler may be made smarter by considering the impact of each task on its successors in the DDG. Therefore it may be able to make better decisions that will expedite the execution of the overall tasks.

REFERENCES

- [1] S. G. Akl, *Parallel Computation: Models and Methods*. Prentice Hall, Upper Saddle River, NJ, 1997.
- [2] Blelloch, “Prefix sums and their applications,” in *Synthesis of Parallel Algorithms*, J. H. Reif, Ed. Morgan Kaufmann, San Francisco, CA 1993.
- [3] G. E. Blelloch and B. M. Maggs, “A brief overview of parallel algorithms,” <http://www.cs.cmu.edu/~scandal/html-papers/short/short.html>, 1994.
- [4] V. Bokka, K. Nakano, S. Olariu, J. L. Schwing, and L. Wilson, “Optimal algorithms for the multiple query problem on reconfigurable meshes, with applications.” *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 9, pp. 875–887, 2001.
- [5] R. Cole and U. Vishkin, “Faster optimal parallel prefix sums and list ranking,” *Information and Computation*, vol. 81, pp. 334–352, 1989.
- [6] D. Culler, R. Karp, D. Patterson, K. E. S. A. Sahay, E. Santos, R. Subramanian, and T. von Eicken, “Logp: Towards a realistic model of parallel computation,” Tech. Rep. UCB/CSD-92-713, EECS Department, University of California, Berkeley, 1992.
- [7] P. de la Torre and C. P. Kruskal, “Towards a single model of efficient computation in real parallel machines,” *Future Gener. Comput. Syst.*, vol. 8, no. 4, pp. 395–408, 1992.
- [8] F. Dehne, A. Fabri, and A. Rau-Chaplin, “Scalable parallel computational geometry for coarse grained multicomputers,” *International Journal on Computational Geometry and Applications*, vol. 6, no. 3, pp. 379–400, 1996.

- [9] Ö. Egecioğlu and Ç. K. Koç, “Parallel prefix computation with few processors,” *CMA*, vol. 24, no. 4, pp. 77–84, 1992.
- [10] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Reading, MA, 1995.
- [11] I. Foster, “Parallelism and computing,” <http://www-unix.mcs.anl.gov/dbpp/text/node7.html>, 1995.
- [12] T. Hagerup, “The parallel complexity of integer prefix summation,” *Information Processing Letters*, vol. 56, pp. 59–64, 1995.
- [13] W.-J. Hsu and C. V. Page, “Parallel tree contraction and prefix computations on a large family of interconnection topologies,” *Acta Inf.*, vol. 32, no. 2, pp. 145–153, 1995.
- [14] J. JaJa, *An Introduction to Parallel Algorithms*. Addison Wesley, Reading, MA, 1992.
- [15] R. Karp and V. Ramachandran, “Parallel algorithms for shared-memory machines,” in *Handbook of Theoretical Computer Science*, vol. A, J. van Leeuwen, Ed. MIT Press, Cambridge, MA, pp. 869–941, 1990.
- [16] J. Kim and D. J. Lilja, “Characterization of communication patterns in message-passing parallel scientific application programs,” in *CANPC '98: Proceedings of the Second International Workshop on Network-Based Parallel Computing*. Springer-Verlag, London, UK, pp. 202–216, 1998.
- [17] C. P. Kruskal, L. Rudolph, and M. Snir, “The power of parallel prefix,” *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 965–968, October 1985.

- [18] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, 1994.
- [19] R. E. Ladner and M. J. Fischer, “Parallel prefix computation,” *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 822–830, October 1980.
- [20] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Francisco, CA, 1992.
- [21] A. Ping, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, “Stapl: An adaptive, generic parallel c++ library.” *Int. Workshop on Languages and Compilers for Parallel Computing*, 2001.
- [22] C. P. K. L. Rudolph and M. Snir, “The power of parallel prefix,” *IEEE Transactions on Computers*, vol. 34(10), pp. 965–968, 1985.
- [23] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.
- [24] U. Vishkin, “U. vishkin. thinking in parallel: Some basic dataparallel algorithms and techniques.” Monograph, in preparation. <http://citeseer.ist.psu.edu/vishkin02thinking.html>, 2002.
- [25] S. G. Ziavras and A. Mukherjee, “Data broadcasting and reduction, prefix computation, and sorting on reduced hypercube parallel computers,” *Parallel Comput.*, vol. 22, no. 4, pp. 595–606, 1996.

VITA

Tao Huang received her B.E. and M.E. in Computer Science and Engineering at the University of Electronic Science and Technology of China (UESTC), Chengdu, in 2000 and 2003, respectively. She completed her M.S. in computer science at the Department of Computer Science, Texas A&M University in May 2007. Tao Huang may be reached at 301 Harvey R. Bright Building, College Station, TX, 77843-3112. Her email address is thuang@cs.tamu.edu.