A HYBRID SPMD – COARSE GRAIN DATAFLOW PARALLEL PROGRAMMING MODEL

Adrian M. Soviani

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE
Adviser: Prof. Jaswinder Pal Singh

June 2014

To my beloved

Lizi and Valona

Abstract

The design of parallel programming models that achieve a good trade-off between productivity and efficiency, while maintaining performance portability and cost transparency, remains a challenging task. Similarly, parallel runtime cost modeling is essential for application and architecture design, as well as performance optimization; however, cost accuracy remains limited when modeling the effect of bandwidth bottlenecks for globally unbalanced communication.

This dissertation proposes a hybrid dataflow model (CGD) that leverages the simplicity and elegance of dataflows and the good performance scalability of Single Program Multiple Data (SPMD) computations. Benchmark analysis shows that the CGD model increases the productivity while maintaining or exceeding the performance of the MPI and pthreads models. The thesis also presents a hierarchical bandwidth machine model (αDBSP) that can estimate the execution time of CGD collective communication by naturally extending and improving the Decomposable Bulk Synchronous Parallel (DBSP) model.

The CGD model is a dataflow graph with SPMD computation nodes and datastructure decomposition data nodes, which exploits dataflow semantics to express data and task parallelism at a high-level, and relies on imperative languages to express efficient sequential computations. Data and computation partition and assignment are explicit, while communication, synchronization, and machine specific optimizations are handled automatically.

This dissertation introduces a coordination language with dataflow semantics that implements the CGD model, and presents several applications and their optimizations implemented in this language. The CGD runtime supports MPI, SHMEM, and pthreads running on both shared memory and cluster machines. The results from an 128 processor SGI Altix 4700 system show that the optimized CGD FT outperforms NPB2.3 MPI by 27%, the optimized CGD stencil is 41% faster vs. handwritten MPI, and the CGD Barnes-Hut particle simulation improves SPLASH2 by 14%.

The αDBSP model extends DBSP by associating a bandwidth growth factor α to message patterns, improves DBSP in terms of execution time, and helps machine bandwidth budgeting by estimating application hierarchical bandwidth. Consequently, for some globally unbalanced problems the αDBSP analysis is more accurate, and sometimes simpler. E.g., the single-element nearest-neighbor message exchange running on a pruned butterfly requires $O(log^3(p))$ on αDBSP vs. $O(\sqrt{p})$ on DBSP, while optimally modeling the one-to-all broadcast requires a single communication step on αDBSP vs. $O(log(p))$ steps on DBSP. We present three scientific computing kernels that illustrate the differences between αDBSP and DBSP analysis.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor Jaswinder Pal Singh, for his continuous guidance, wisdom, encouragement, and friendship throughout all these years. His liberal advice and insight into emerging software and hardware HPC systems shaped my understanding of parallel computing, and proved essential to completing my dissertation research.

I would like to thank my committee members, Prof. Kai Li, Prof. David August, Prof. Andrea LaPaugh, and Prof. Brian Kernighan for their enthusiasm, thoughtful comments, hard questions, and suggestions. Our technical discussions and their unique perspective helped me improve tremendously the quality of my research and presentation. My sincere thanks go to Melissa Lawson, our graduate department coordinator, for her constant help and support during my Ph.D program.

We are grateful to the Flexible Modeling System group at the Geophysical Fluid Dynamics Laboratory affiliated with Princeton University for presenting to us real world large-scale scientific computing problems, and for kindly allowing us to use their supercomputing facilities.

Several distinguished professors and researchers have played an important role in my professional development. I would like to express my gratitude to Prof. Charles Leiserson for instilling in me the love for parallel computers and algorithms while working on the Cilk project, and to Prof. Thomson Leighton for teaching me to think pragmatically and bridge the gap between theory and real world problem solving during my

work at Akamai Technologies. Last, I would like to thank my late Romanian literature professor Lucian Cristea for his boundless efforts to make us who we are.

I would also like to thank my fellow MIT and Princeton University colleagues for the stimulating discussions and all the fun we have had in the last few years: Alexandru Salcianu, Cristian Soviani, Mihai Badoiu, Erich Schmidt, Christian Bienia, Berk Kapicioglu. Finally, I would like to thank my family for their love, encouragement, and unconditional support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Writing efficient parallel applications that are portable across architectures remains a daunting task due to implementation complexity, optimization effort, architecture heterogeneity, and lack of cost transparency. A programming model greatly impacts the addressing of these issues, and inevitably, some of its most desired features include good productivity, efficiency, portability, and ease of design-space exploration. The development of high productivity programming models has recently received greater attention, representing a central aim of the HPC Challenge initiative and the newly developed Partition Global Address Space (PGAS) language family [CCDI09].

In the world of programming models—especially parallel models—achieving a good trade-off between productivity, efficiency and portability is not trivial. While low-level communication libraries and memory access primitives can potentially provide the best performance, they require a substantial development effort and may exhibit limited portability[1]. On the other hand, highly abstract languages require developers to specify fewer implementation details, thus increasing productivity; however, machine and problem coverage is often limited, and compilers need to solve complex optimiza-

[1]Communication libraries include pthreads [But97], MPI [GS99], SHMEM [Fei95], and GASNET [Bon02], and memory access primitives rely on cache-coherent shared address space (CC-SAS) [CSG98] mechanisms implemented in hardware.

tion problems, in many instances underperforming handwritten code, or requiring programming workarounds that defeat the stated high productivity goals[2].

However, on considering both low- and high-level programming models, an interesting question emerges: how much low-level detail is required by compilers to maintain the performance of low-level models and produce efficient implementations without sacrificing productivity?

The answer to this question probably lies along the continuum between low- and high-level programming models, mixing elements of both. A significant efficiency loss is not expected if programmers describe what they know best—including data layout, computation assignment, and algorithmic optimizations—and if compilers handle repetitive work such as communication, synchronization, data handling, and well-known scheduling and machine specific optimizations. Instead of aiming to automatically solve all difficult problems, such a model could provide developers with the best tools needed to solve them.

Hybrid Model This dissertation introduces a new programming model called CGD (for Coarse Grain Dataflow), which exploits the simplicity and elegance of dataflows and the good performance scalability of Single Program Multiple Data (SPMD) computations [SS09, CSG98]. The CGD model is a dataflow graph with SPMD computation nodes and datastructure decomposition data nodes, which leverages dataflow semantics to express data and task parallelism at a high-level, and relies on imperative languages to express efficient sequential computations. Section 2.2 presents how CGD reconciles the dataflow and SPMD elements.

In the CGD model, the developer specifies the SPMD computations, datastructure decompositions, distribution rules, and dependencies between SPMD computations and

[2]Parallel programming languages popular within the HPC community include OpenMP [DM98], CoArray Fortran [NR98], Cilk [cil04], and more recently PGAS languages such as UPC [CHea03] and Chapel [CCZ07].

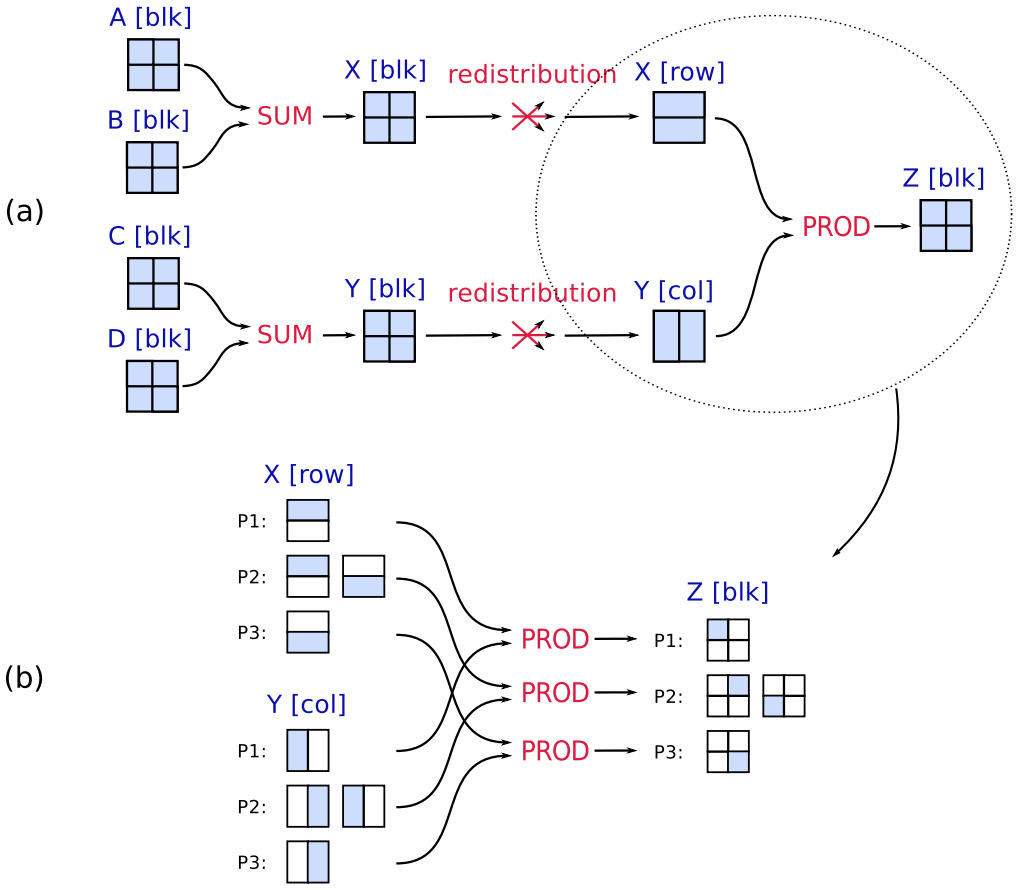


Figure 1.1: (a) CGD graph computing $Z \leftarrow (A + B) * (C + D)$, where A, B, C, and D are matrices; the graph specifies the dependencies between matrix decompositions and SPMD computations; and (b) SPMD computation for $Z \leftarrow (X * Y)$ mapped to three processes

datastructure decompositions (Fig. 1.1). Dependencies define a graph between datastructure decompositions nodes and SPMD computation nodes; distribution rules define how data decompositions can be automatically transformed. SPMD computations are functions written in an imperative language such as C++, and operate on the local domains of global datastructures [KR88].

On the other hand, the compiler orders the computations to obey data dependencies, and automatically inserts communication, data handling, and synchronization. Fur-

thermore, the system exploits the large optimization scope of CGD semantics to include architecture-specific communication, synchronization, datastructure access, and scheduling optimizations. E.g., similar to the Unified Parallel C (UPC) [CHea03] NPB FT "slab" optimization reported by [BBNY06], the CGD NPB FT implementation exploits the communication overlap allowed by fine-grain tasks as well as architecture-specific communication fine-tuning to significantly improve MPI performance on both large-scale distributed memory and small symmetric multiprocessing machines (Section 4.2.2).

The CGD model aims at achieving a good trade-off between programming productivity and implementation efficiency by having developers leverage their unique understanding of application particularities to specify data and computation decomposition and assignment, while allowing dataflow compilers to solve easier scheduling and orchestration problems.

Comparison Our programming model combines and develops several concepts defined by dataflow and SPMD-centric models. Similar to dataflows, CGD uses the single assignment rule, and data dependencies determine the scheduling. However, unlike dataflows, our model relies on user specified datastructure and computation decompositions and assignments. We feel that these requirements are essential for providing performance transparency and efficiency.

Similar to SPMD applied to message passing models and certain shared address space models such as SHMEM, CGD relies on explicit data and computation decomposition and assignment. Similar to newly developed PGAS languages such as UPC, Chapel [CCZ07], and X10 [CGS+05], CGD uses datastructure distributions and allows remote data element access. However, unlike these models, CGD computations are described as dataflow graphs, and thus, a CGD compiler can take advantage of dataflow semantics and explicit distribution rules to schedule computations, and automatically insert, aggregate, overlap, and optimize block communication. Moreover, message passing

implementations require explicit messaging and a greater programming effort, while PGAS languages don't allow multiple datastructure distributions, and suffer from fine-grain message overhead. These issues are presented in more detail in Section 1.3.1.

1.1 Programming Model Desiderata

This section discusses some of the most desired features of parallel programming models, and showcases a few successful approaches that address these features. The programing model Holy Grail of good performance and good programmability has been widely discussed, and several schools of thought have stated similar yet not identical aims; to better understand how to address and trade-off model features, we analyze some of the success stories as well as practical issues encountered by existing models.

Ease of Programming Most application developers are not familiar with concurrent reasoning, particularly with building a mental model of concurrent processes, inter-process communication, and synchronization. Reasoning regarding the correctness of concurrent applications is a challenge, and debugging such applications is equally demanding. A programming model that avoids explicit concurrency improves both the developer's productivity and application robustness [BVZ+07, SS10].

Many applications can exploit data, task, pipeline, and recursive parallelism [SSOG93, CSG98]. A model that supports more than just one type of parallelism improves programmability.

Most high performance computing (HPC) users have gradually developed a large domain-specific code base. Developers are familiar with a set of languages taught in school or extensively used by the community. A model that can incorporate components written in other languages, and that uses a familiar syntax has a greater chance of early adoption and success.

The separation of algorithm and datastructure implementation is not well supported by most programming languages, requiring algorithms to be modified when internal datastructure representations change. For example, converting a 2D grid into a 2D array of smaller 2D grids—a change often employed by parallel partial differential equation (PDE) solvers—requires recomputing and updating all array references. Similar algorithmic changes are needed when converting dense arrays into sparse arrays, row-major matrices into column-major matrices, etc [CCZ07]. A programming language that decouples the algorithm and datastructure implementation reduces the work required by changes in the datastructure representation.

Good Performance Historically, SPMD-centric applications have had a good performance track record for a wide range of architectures, partly owing to the developer's ability to explicitly decompose data domains and assign computations to processes. Thus, SPMD-centric applications balance the workload and minimize communication by taking advantage of application particularities. A model supporting explicit data and computation decomposition and assignment allows users to balance the workload and optimize the communication, leading to a good performance.

In many cases, message passing [SOW+95] and pthreads [But97] applications are optimized for performance by adding sections of architecture-specific code. Such examples include speeding up data copy by rewriting loops and reorganizing buffers, and reducing synchronization and messaging overhead by using faster primitives supported by the target machine. A programming model that defines distributed datastructure abstractions can automatically implement some of these architecture-specific optimizations, reducing the programming effort and improving performance portability [JSS97, SSOB00].

Cost Transparency Real-world HPC applications have frequently grown in size and complexity, incorporating the work of scientific teams over periods of several years.

Finding scalability bottlenecks in such systems is a non-trivial task without relying even on a heuristic measure of cost [SRG94]. To cope with this issue, developers have tried to create a mental map between operations and their hardware implementation, and thereby their cost; message passing models and the C programming language have proved successful in this respect. A cost-transparent model allows programmers to understand the communication and computation complexity during the design stage, and helps them make an educated choice between implementation options.

Design-Space Exploration Flexibility The development cycle of most HPC applications starts with the code development, and then continues with the optimization for performance [CSG98] on target machines. The initial implementation frequently requires major redesigns to address the typical load balancing and communication costs [JS99].

Programming models closer to the hardware layer such as message passing and pthreads provide good flexibility, allowing both lower-level architecture-specific optimizations and higher-level algorithmic changes, e.g., using shared memory for fine-grain element access, and hiding latency by replicating the computation and aggregating the communication in the case of PDE solvers. Unfortunately, many applications interleave the two optimization types, containing code that is complex and prone to concurrency errors, and potentially limits performance portability; such applications require a significant time investment to reorganize datastructure layouts, messaging, and buffer handling. In this context, the performance optimization process is greatly simplified using a programming model that decouples datastructure and algorithm implementation [CCZ07], and easily allows the exploration of datastructure and computation decomposition and assignment.

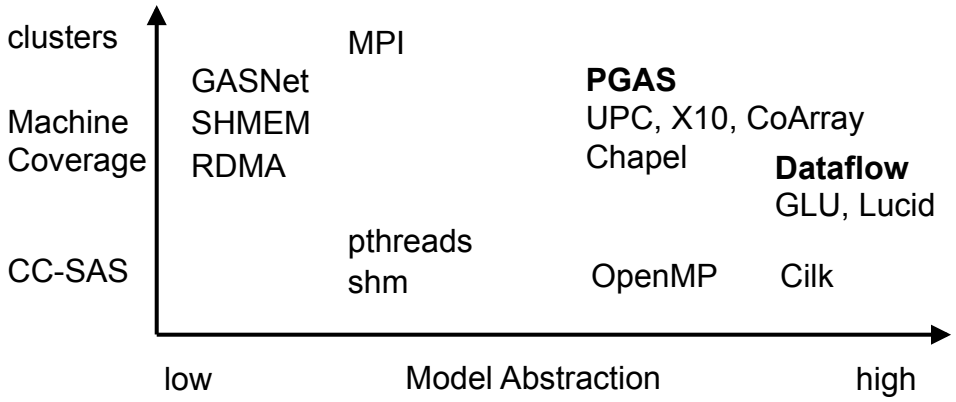


Figure 1.2: Programming model classification in terms of machine coverage and abstraction level [CSG98, CCZ07, Jag95, JHRM04]

1.2 Programming Model Landscape

This section presents an overview of the most influential parallel programming models, including Shared Address Space (SAS), message passing, Partitioned Global Address Space (PGAS), and dataflow models [CSG98, BCBY04]. These models can be classified in terms of machine coverage and programming abstraction (Fig. 1.2).

Some parallel programming languages and libraries require Cache-Coherent Shared Address Space (CC-SAS) [CSG98] machines, PGAS languages require support for at least Remote Direct Memory Access (RDMA) [Bon02] or one-sided messaging, some dataflow languages rely on CC-SAS, and the Message Passing Interface (MPI) [SOW+95] is supported by virtually any architecture. If message passing libraries and pthreads [But97] expose primitives closer to the hardware layer, parallel languages rely on constructs that partially hide the communication and synchronization details.

1.2.1 Shared Address Space

Symmetric Multiprocessing (SMP) systems, such as workstations and servers, and some large-scale distributed memory systems implement the CC-SAS memory model. On these machines, all processors access a unified address space (Fig. 1.3a), and specialized hardware implements a coherence protocol that propagates writes to a memory location to all readers, and ensures that all cache coherence conditions are met [CSG98].

Shared address space architectures do not necessarily provide cache coherence. Larger machines such as Cray T3E allow each processor to access any memory location; however, only local memory elements are handled by the cache, and developers explicitly access remote memory relying on SHMEM operations [Fei95]. More recently, clusters of SMPs connected via Infiniband or Quadrics switches provide a similar RDMA functionality.

The implementation cost of cache-coherent mechanisms on large-scale Cache-Coherent Non-Uniform Memory Architecture (ccNUMA) machines has traditionally been considered prohibitive; however, techniques such as directory-based protocols have reduced the cost of building such machines [SJHG93, LLG+90].

Shared Virtual Memory (SVM) is a software solution that provides a coherent shared address space on commodity clusters lacking CC-SAS hardware support. SVM implements coherence at page granularity via virtual memory management [KL89, MAB94]. Valid page entries correspond to up-to-date locally cached shared pages, while invalid entries correspond to locally unavailable pages. When an invalid page is accessed, a page fault occurs, and the SVM protocol brings an up-to-date page copy locally. CC-SAS applications running on SVM might exhibit performance issues caused by fragmentation and false-sharing, which are typically addressed by changing page granularity and memory access patterns to avoid frequent updates [PL, JSS97]. A family of SVM models has been developed to address such issues first by adding coherence protocol

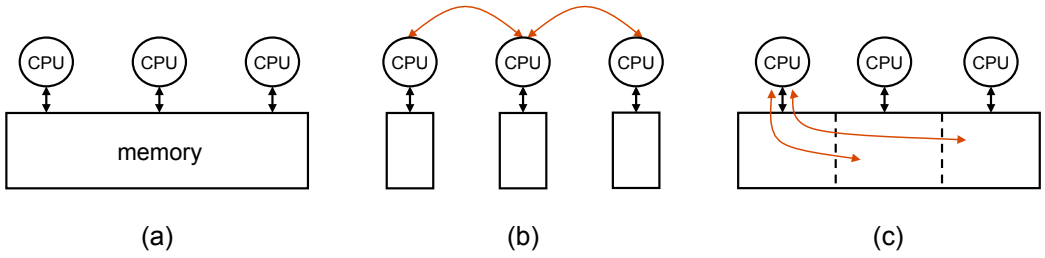


Figure 1.3: Programming model memory organization: (a) SAS threads access the global memory using a single address space [CSG98]; (b) message passing processes access local memories and exchange messages; and (c) PGAS processes access local and remote domains of distributed datastructures [CCZ07].

laziness (release, eager release, and lazy release consistency), and then by limiting the scope of page updates (entry and scope consistency) to reduce update frequency and size, thereby improving performance [Ift98].

All threads or processes of SAS applications access a single shared address space, making development easier in comparison to message passing models, where each process accesses only its local address space and handles messages explicitly [SS99]. Moreover, the CC-SAS model provides better problem coverage and requires significantly less effort vs. message passing when implementing irregular or adaptive algorithms [SSOB00].

While the CC-SAS memory model simplifies the application development, achieving good scalability on large systems comes at a cost. The CC-SAS model provides a uniform view of the entire memory address space; however, memory access latency is not uniform, remote access latency being larger than local access latency. This discrepancy increases as the memory hierarchy deepens, becoming most significant on large-scale ccNUMA machines. Subsequently, programming for performance requires understanding data placement, and trying to avoid remote accesses.

In most cases, good performance is achievable even on large systems if developers design good data decompositions that maintain access locality, and the problem size is

large enough for the target machine, i.e., the working set is sufficiently large to maintain a good communication-computation ratio [RSG93]. Other helpful optimizations include avoiding false-sharing between distinct data elements mapped to the same cache line, latency hiding techniques such as prefetching, write buffering, and higher-level algorithmic changes that reduce inherent communication or improve load balance [HSH96, JS99, JSS97].

Pthreads POSIX threads (pthreads) is a library that provides thread management and synchronization primitives on CC-SAS machines [But97]. Application programming interfaces (APIs) are available for most languages, from C and Fortran to Python and Ada. Pthreads is one of the most popular choices for small-scale applications developed for SMP systems such as x86 based servers.

All pthreads access a single shared address space, and shared datastructures are therefore easily implemented by adding synchronization. The unified memory view allows a gradual transition from single- to multi-threaded implementations, providing a practical approach to parallelizing existing applications. However, good performance is obtained after optimizing at least the synchronization, partition, and placement of shared datastructures.

The pthreads library provides users with a significant degree of freedom, and thus good problem coverage. However, expressing concurrency requires highly specialized developers, and similarly, achieving good scalability requires time consuming optimizations; the burden of these tasks is somewhat alleviated by the expertise accumulated by the community. Reported performance ranges from average scalability on SMPs to good performance on large-scale distributed shared memory systems [SSOB03, WOT+95, WMT08, CLZ+11].

An interesting alternative to calling low-level pthreads primitives is parallelizing compilers that automatically extract threads from sequential codes. Automatic paralleliza-

tion is a promising strategy for multi-core systems, allowing existing sequential codes to exploit implicitly available parallelism. However, common sequential programming practices can limit the ability of compilers to produce efficient code. Datastructure reuse occasionally leads to false datastructure element contention when loop iterations use the same memory location to store distinct values; this issue is addressed through the replication or privatization of elements, and [JKP+12] reports good results even for dynamic datastructures. Similarly, the sequential program structure sometimes unnecessarily constraints execution order, and thus, the ability to concurrently execute essentially independent tasks; here, implicit parallel programming models on top of sequential models extend the language to better describe data dependencies and help compilers perform more aggressive optimizations, thereby improving scalability [PGZ+11]. Automatic parallelization is a promising approach to parallelization on tightly-coupled systems, and its adoption depends on the continuous evolution of compiler technology, as well as the ease of achieving good parallel performance.

SHMEM SHMEM was first introduced by Cray to exploit its hardware support for non-cache-coherent remote memory access within a shared address space [Fei95]. Later, SHMEM gained a larger support on SGI and IBM machines, while at the same time having an open source implementation. SHMEM provides a large set of primitives for remote memory data *read (get)* and *write (put),* where the *put* and *get* operations access shared memory pages; these primitives are highly optimized for low latency fine-grain data access. Synchronization is provided as barriers, waits on data arrival, and semaphores.

In contrast to MPI, all SHMEM processes can access a shared address space by mapping memory pages to this space. Subsequently, each process holds a local copy of the memory page, and can access remote page copies via SHMEM calls; the same address can be used at a given time by several processes, each operating on its own copy. There is no hardware mechanism that keeps these copies in sync, which is a deliberate choice

taken for performance considerations. Since SHMEM is not cache-coherent, data consistency becomes the programmer's responsibility.

The SHMEM shared address space abstraction makes data access more elegant compared to MPI on large distributed memory machines; however, the lack of cache coherence makes SHMEM more vulnerable than CC-SAS programming models to subtle data race conditions and synchronization errors.

SHMEM allows processes to execute arbitrary communication via remote memory access, and similar to message passing, it provides good performance on large systems at the cost of significant programming effort. The usage of SHMEM is currently restricted to a few proprietary systems, whereas RDMA systems that supply similar functionality are becoming more common and affordable.

OpenMP Open Multiprocessing (OpenMP) is a programming language that augments the sequential C and Fortran syntax with parallel compiler directives [DM98]. With the aid of these directives, the compiler understands data, and recently, task parallelism, and automatically generates multi-threaded code.

The idea is promising: the OpenMP code is algorithmically sequential, datastructures use a single address space, and all communication and synchronization issues are transparent to the user. However, there are important drawbacks: i) applications are restricted to cache-coherent machines; ii) users rely on private variables to produce efficient codes, manually assigning arrays elements and loop iterations to threads, and hence breaking the single-view datastructure abstraction; and iii) automatically extracting algorithmic parallelism from sequential code does not always produce the best results given the present day compiler technology. Historically, OpenMP has not performed well on large systems, its use being mostly limited to SMPs; hybrid OpenMP–MPI solutions for SMP clusters have reported results improving MPI only performance [FG06, BBNY06].

Cilk Cilk annotates sequential C code with keywords that allow recursively spawning and synchronizing logical threads using modified function calls [RLR98, cil04]. The runtime starts a predefined number of processes that access shared memory pages, and load is balanced among processes using a work-stealing scheduling algorithm. When a function call is spawned, its context is saved on a globally accessible stack; when a process runs out of work it tries to steal a task from processes that have non-empty stacks. Unfortunately, dynamic work-stealing means that tasks are assigned to processes at runtime, and therefore data access locality becomes hard to maintain.

Cilk allows task and data parallelism by decomposing a problem into sub-problems—frequently employing the "divide and conquer" principle—that are executed as recursively spawned function calls. All logical threads access datastructures globally, however, without explicitly assigning data or computation to processes.

Unfortunately, optimizing the work-stealing algorithm to maintain data access locality remains an open problem for Cilk. While Cilk shows a good performance on smaller SMPs, scalability suffers on larger ccNUMA systems where access locality is critical, and additional remote accesses have a larger performance impact [Kus06, OP]. However, its particular ability to exploit task parallelism makes the possibility to overcome this limitation more promising.

1.2.2 Message Passing

Message passing architectures—such as SMP clusters connected via Infiniband or Quadrics switches, and SGI Altix and IBM Blue Gene supercomputers connected via high-performance custom backplanes—implement a message passing programming model. Here, all processes access only their local memory address space, and communicate with each other by explicitly sending and receiving messages that copy data blocks between local memories (Fig. 1.3b).

Message passing machines are typically implemented as individual physical nodes

linked by an interconnect, similar in this regard to large-scale ccNUMA machines, with the exception of communication, which is integrated into the I/O system rather than into the memory system [CSG98].

MPI The Message Passing Interface (MPI) provides solid performance across a wide range of parallel architectures, representing a mainstream programming model within the HPC community [For94, SOW+95, GS99]. The MPI library offers explicit communication primitives such as send, receive, broadcast, reduction, and synchronization. The MPI standard is continuously being updated by the MPI Forum, and currently defines bindings for the C, Fortran, and C++ (MPI-2) programming languages; unofficial ports to other languages such as Java and Python are also available. If several MPI implementations such as OpenMPI and MPICH are freely available, vendor implementations—typically based on MPICH—provide improved performance on their target machines.

The MPI-1 standard defines two-sided communication, where each *send* called by a sending process matches a *receive* called by the receiving process (Fig. 1.3b). The MPI-2 standard takes some clues from other high performance communication libraries such as SHMEM and Silicon Graphics (SGI)'s MPI extension, and allows both two-sided and single-sided communication. A process using single-sided communication executes *get* or *put* operations on remote memory addresses without requiring matching operations be executed by remote processes.

The semantics of two-sided communication can be summarized as follows:

```
process A:  send(process_B, source_addr_A, size)
process B:  recv(process_A, destination_addr_B, size)
```

where *source_addr_A* and *destination_addr_B* represent local memory buffers, and processes are not aware of remote addresses.

On the other hand, one-sided communication is initiated by a single process:

```
process A:  put(process_B, destination_addr_B, source_addr_A, size)
process A:  get(destination_addr_A, process_B, source_addr_B, size)
```

where *process_A* needs to know remote buffer address *destination_addr_B*, an address that is otherwise meaningless in the local address space.

One-sided communication is more prone to synchronization errors than two-sided communication. In the latter, processes progress more or less in lockstep, being implicitly synchronized by the *send-receive* call pairs. In the former, processes are free running while communication occurs in the background, changing remotely accessed local data without notice. However, for small messages, one-sided communication has lower protocol overheads, the one-sided latency approaching performance levels once reserved for SHMEM and RDMA requests.

MPI passes to the programmer the burden of handling communication details such as carefully interleaving various communication primitives with computation, marshaling and unmarshaling data into buffers, and managing distributed datastructures. These issues are orthogonal with algorithm design, breaking the algorithm-datastructure decoupling principle (Section 1.1). E.g., reorganizing distributed datastructure layouts requires changing the algorithm to send and receive new data slices at different points in time, and to update synchronization accordingly. Nevertheless, MPI is popular within the HPC community, being a well-established and well-specified free standard that has efficient implementations on most architectures.

1.2.3 Partitioned Global Address Space

PGAS languages are based on a global memory model that is logically partitioned between processors (Fig. 1.3c). Roughly speaking, these languages can be viewed as parallel extensions of existing imperative languages such as C, Fortran, or Java, where computation decomposition and data partition are practically explicit, while accessing data from remote memory partitions implicitly generates communication [BCBY04].

Code parallelism is typically expressed as data-parallel loops similar to *forall*, task creating function calls similar to *spawn*, and explicit SPMD constructs. Datastructures are partitioned among processes by assigning domain ranges to each process; some languages such as UPC allow only fixed 1D partitions, while more recent languages such as Chapel allow arbitrarily defined datastructures and partitions. The union of all domain ranges covers the entire datastructure domain; however, all domains have to be disjoint, sometimes limiting replication.

PGAS languages are usually implemented using one-sided communication libraries such as ARMCI and GASNet; these libraries allow processes fine-grain access to both local and remote datastructure elements. While this approach seems very attractive, it has certain fallacies: local access is fast compared to remote access, which could be orders of magnitude slower, and yet PGAS languages support both access types using an essentially identical syntax.

GASNet and ARMCI These libraries attempt to provide a standardized API for low-overhead fine-grain remote memory access, and remote code execution for a wide array of architectures ranging from hardware CC-SAS and RDMA machines to generic MPI clusters [Bon02, NC99]. These libraries are not designed to be used directly by application developers, but rather, to be used as a unifying bottom layer by parallel language compilers.

Subsequently, both libraries are one-sided communication libraries that extend the RDMA and Active Messages concepts; Active Messages [VECGS92] is a light-weight layer that allows code execution upon message receipt. Both GASNet (Berkley) [Bon02] and Aggregate Remote Memory Copy Interface (ARMCI, Pacific Northwest National Labs) [NC99] are built directly on top of a multitude of native network communication interfaces and resources.

An important particularity of these libraries is their careful tuning for performance.

Not only are they heavily optimized for the supported target architectures, but the same functionality may also have different implementations on the same target given practical details such as message size. Their performance is very good, and the library overhead remains low. However, their low-level abstractions make them unpractical as programming models, which is not a surprise given their declared goal of serving as a base layer for PGAS languages; their success depends on the success of the latter.

UPC Unified Parallel C (UPC) is a C parallel extension that introduced first what would later become the PGAS memory model [CHea03]. UPC defines two pointers types: shared pointers that address both local and global memory, and private pointers that address local data only. This gives compilers a significant optimization potential since private data accesses are guaranteed to be executed locally.

Shared arrays are usually distributed among processes in a cyclic or block-cyclic manner. 2D block decompositions are not supported directly, and thus require developers to create arrays of arrays to achieve a similar functionality. Parallel primitives such as *upc_forall* loops assign iterations to processes using an affinity expression, or otherwise trying to maximize access locality.

UPC can be considered a textbook representative of PGAS: shared pointers access global memory, and the physical location of data remains transparent to users. When addresses are local, simple *read*/*write* memory accesses are executed; otherwise, accesses are translated into library calls resulting in remote communication.

While the transparency of distributed datastructure access provides a powerful abstraction, it can quickly become a weakness by reducing a developer's locality awareness, and leading to poor data partition and unnecessary communication. Similarly, unstructured datastructure access patterns can lead to fine-grain rather than coarse-grain remote memory access, and therefore a significant latency overhead [MTT+09, PG08]. Moreover, accessing the local domains of distributed arrays can be slow if the

compiler cannot determine at compile time whether the address is local, and inserts a branch before each local memory access [EGC02]. Fortunately, language instrumentation using the *shared*, *private*, and *affinity* constructs is provided to help the programmer keep such issues under control.

Co-Array Fortran Co-Array Fortran (CAF) is a popular parallel extension of Fortran, which was eventually added to the Fortran 2008 standard. This extension adds parallelism using data-parallel loop constructs, and elegantly hides communication through distributed array element access [NR98].

CAF introduces a new array dimension called the co-array; a variable with such a dimension has its own copy on each process. Any array copy can be accessed by any process by indexing the variable with the desired co-array index. Synchronization primitives ensure that data written within a parallel section can be seen globally.

Even if CAF handles distributed datastructures in a more systematic way compared to the ad-hoc custom view exposed by message passing libraries, the programmer is still responsible for explicitly assigning data to processes and handling indexes accordingly; indexes are converted between a global view addressing the entire vector domain and a co-array fragmented view.

Chapel Chapel is a PGAS block-imperative parallel language developed by Cray, which abandons the idea of extending a sequential language such as C or Fortran. A declared goal is a better separation of algorithms and datastructures [CCDI09, CCZ07], and this language generally encourages developers to think "in parallel" from the very beginning.

Chapel provides a global view of distributed datastructures, which otherwise are partitioned among processors. Several distribution schemes are provided, including the block, cyclic, and sparse matrix decompositions. The language provides mechanisms

allowing developers to define custom datastructures and distributions, if needed. Chapel provides a nice algorithm-datastructure decoupling: changing the datastructure distribution or representation does not require changing the code that accesses the new datastructure.

This language supports task parallelism using *cobegin* and *coend* constructs, and data parallelism using *forall* loops; both types of parallelism are composable, and the *on* keyword can be used to specify task-to-process assignment. The exploration of alternative execution patterns requires modifying these assignments, and may result in a better or worse locality and performance.

While Chapel's distributed datastructure abstractions look promising, the language suffers—similar to UPC and other PGAS languages—from fine-grain remote access overheads, thereby prompting developers to explicitly copy global to local datastructure blocks to achieve a good performance [CDHW]. The success of this language depends on how well compiler technology can address these problems [CL10, CCDI09], and the extent to which the HPC community will adopt the new language and its datastructure abstractions.

X10 X10 is a PGAS object oriented parallel language developed by IBM with syntax inherited from Java; it is not a Java extension but rather a language of its own [CGS+05].

Parallelism is expressed explicitly using the *finish/async* construct, which executes multiple tasks similar to an SPMD computation, and then waits for all of the tasks to finish. Asynchronous calls can take both data and function names as parameters; execution is coordinated using clocks and lock-free synchronization primitives.

Similar to other PGAS languages, X10 distributed arrays provide a global view, and their domain is explicitly partitioned between processes that can access local and remote data elements. The keyword *here* returns the name of the process executing the

code; by default, function calls are executed by the same process as the caller, unless the keyword *at* is used to specify another process.

The X10 language is being actively developed by IBM, and preliminary results have shown it is powerful enough to allow the implementation of complex scientific problems [MGRG11]. Its performance does not currently scale as well as traditional message passing implementations, but this performance gap is expected to diminish over time as the compiler matures.

1.2.4 Dataflow

Dataflow models define an algorithm in terms of data-computation dependencies rather than making use of traditional von Newman control-flow constructs. A computation is described as a directed graph, where nodes are computational elements, and data flows through the arcs [AC86]; later dataflows define a bipartite graph with computation and data-link nodes connected by edges [KBB86]. An interesting feature of dataflows is their ability to inherently express parallelism; computations can be executed in parallel in any order as long as data dependencies are obeyed (Fig. 1.4).

Pure Dataflow The original dataflow model consists of a graph with nodes executing primitive operations and arcs between nodes holding data tokens. Arcs can be seen as unlimited First In First Out (FIFO) queues from which nodes read tokens when a node becomes fireable, and to which nodes produce tokens after the node operation is executed. Specific rules define which input arcs need to hold tokens before a node becomes fireable [AC86].

This type of dataflow graph can include special flow-control nodes that are needed to model the conditional execution and looping. Two such nodes are the *merge* and *switch* nodes, which behave like a multiplexer and demultiplexer, respectively. A *merge* node has a boolean select input, two data inputs, and one output; the node first consumes

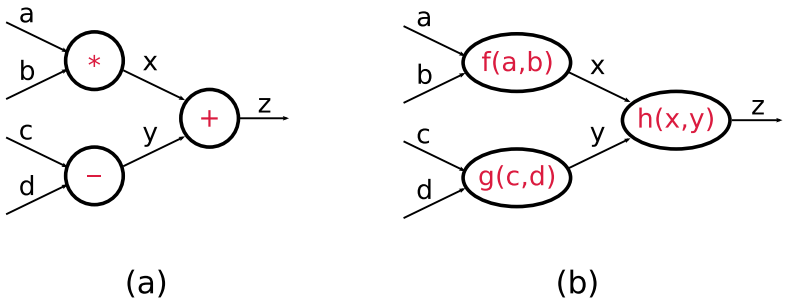


Figure 1.4: Fine- and large-grain dataflow models: (a) computation nodes are elementary operations; and (b) computation nodes or macro-actors are arbitrary imperative functions.

the input select token, and then consumes one token from the selected data input, producing it as an output. A *switch* node has a boolean select input, one data input, and two outputs; the input token is passed only to the selected output. These nodes are sufficient for modeling any arbitrarily complex algorithm.

The mathematics behind this apparently simple model is rather complex, i.e., it relies on Petri nets to deal with extreme cases when no node has the required tokens to fire, or an infinite number of tokens have accumulated on an arc [Mur89].

If a fine-grained approach provides the biggest potential for parallelization and compiler optimization, practice shows that compiling coarser-grain nodes into optimized sequential code decreases dataflow overheads leading to an improved performance. Choosing the granularity of computation nodes remains an important problem in performance fine-tuning.

Dataflow languages The dataflow model is functional in nature as its computation nodes take data tokens as inputs and produce data tokens as outputs, without having any side effects. The first dataflow languages were functional languages adapted to the dataflow model, although they had some resemblance to their original application.

Lucid Lucid started as a functional language developed for use with formal mathematical proofs [AW77]. Its functional and single assignment semantics made it a natural candidate for programming dataflow machines, within a short period of time. Iterations are elegantly handled by adding the *next* keyword that refers to the value of a variable during the next iteration, thus maintaining the single assignment functional semantics. This approach spares programmers the burden of describing iterations in a traditional functional way, i.e., using tail recursion. Lucid's background delivers the mathematical soundness of a functional language, while its non-traditional functional features such as iterations define it as a dataflow language.

Id Id was developed for the purpose of writing operating systems functionally, outside the von Neumann paradigm, based on the sequential control flow and memory cells [Nik93]. Multiple enhancements have been gradually added to the language for practical reasons; the single assignment rule that was so cherished by functional purists proved to be too restrictive for handling datastructures in complex applications.

Its functional formalism was relaxed by adding I-structures that allow lazy element evaluation. I-structures can now be produced by a computation before all I-structure elements are populated. Subsequently, computations reading I-structure elements have to wait until the evaluation of each required element is completed. The new I-structures avoid delays when propagating complex data through a dataflow; however, they do not address the issue of copying the entire datastructure when individual elements are modified.

SISAL Steams and Iteration in a Single Assignment Language (SISAL) is a structural functional language used for programming dataflow machines [GBT87]. Its conditional and iterative evaluations maintain single assignment semantics, while the language provides datastructures and explicit loop parallelism. In contrast to I-structures, SISAL datastructures are treated as single values in a purely functional way.

Large-Grain Dataflow If the dataflow model naturally expresses parallelism while maintaining the mathematical soundness of its functional semantics, the model does not enforce any fundamental limitations on the complexity of computation nodes.

The idea of collapsing small computation nodes into bigger macro-actor nodes [LH94] brings about the immediate advantage of a better performance, owing to reduced token, data handling, and scheduling overheads. Since such macro-actor nodes are executed by a single process they can be compiled into highly efficient sequential code.

Morrison has developed this idea further, and noticed that since macro-actors are executed sequentially, they can be written in an arbitrary sequential language such as C or Java [Mor94]. The "flow-based programming" concept is defined as a program consisting of coarse-grained components written in an imperative language, which are connected by data dependencies maintaining some aspects of the dataflow semantics. This technique has become the de-facto standard for a class of visual digital signal processing applications such as LabView and Simulink; however, its applicability has not been extended to other domains.

GLU Following the idea of having a coarse-grain dataflow language where computation nodes are written in a sequential imperative language, Granular Lucid (GLU) was developed as a dataflow language designed for programming conventional, rather than dataflow, computers [Jag95]. GLU represents a natural extension of Lucid, and its parallelism is therefore expressed implicitly, variables are multi-dimensional, and the single-assignment rule is enforced. GLU practically enjoys all of the features that established Lucid as a popular dataflow language.

GLU extends Lucid by allowing developers to define the functions and types in a foreign sequential language such as C. GLU claims to successfully address expressiveness and efficiency; Jagannathan reported elegant and performant solutions for several well-known application kernels [DRJ94].

Dataflow languages are an attractive choice for a programming model since they naturally expose a high degree of parallelism, do not require explicitly specifying communication and synchronization, and therefore avoid typical concurrency related failures such as data races or deadlocks. Furthermore, implementing macro-actors as sequential functions written in an imperative language can reduce the fine-grain dataflow overheads.

Unfortunately, some critical parallel computing features have not been addressed by the previously described dataflow languages, namely, the explicit datastructure and computation partition and assignment. For example, in most successful large-scale applications, large datastructures such as vectors or trees are stored and distributed among the local memories of each node, where local access is obviously much faster than remote access. For such applications, explicitly specifying datastructure layout along with computation assignment maximizes the access locality, which is essential in achieving good scalability.

We believe that a coarse-grain dataflow language allowing users to specify and change data and computation decomposition and assignment at an appropriate granularity, and in a natural way, maintains the most desired properties of dataflows while improving their performance.

1.3 This Dissertation

This dissertation proposes a hybrid SPMD – coarse grain dataflow model (CGD) that leverages the simplicity and elegance of dataflows and the good parallel performance of SPMD. It then presents a collective communication cost model that estimates the execution time of datastructure redistribution operations by naturally extending and improving the Decomposable Bulk Synchronous Parallel (DBSP) model. The most important contributions of this thesis include:

- Introduces a hybrid SPMD – coarse grain dataflow programming model, where data and task parallelism are described by dependencies between SPMD computations and datastructure distributions. Communication, synchronization, and data handling are automatically added.
- Presents a programming language for the CGD model, and describes the implementation of a CGD compiler, as well as several benchmarks implemented using the new language.
- Shows that the CGD model increases productivity for programming and optimizing these benchmarks, at the same time achieving a performance on par or better than the original pthreads or MPI implementations.
- Introduces an αDBSP hierarchical bandwidth machine model that can estimate application runtime. This model naturally extends the DBSP model by adding a bandwidth growth factor α to each message exchange, i.e., h-relations are generalized as (h, α)-relations.
- Shows that the αDBSP model is an improvement of DBSP for several common globally unbalanced problems such as PDE solvers, Fast Fourier Transforms (FFT) on grid subsets, and broadcasts. Discusses how αDBSP cost estimation can aid hierarchical bandwidth capacity planning for certain HPC application classes.

1.3.1 Discussion

MPI Most successful HPC applications deployed on large-scale systems are SPMD-centric, relying on MPI and hybrid OpenMP-MPI solutions. Such applications—which use the SPMD computation view in conjunction with low-level optimizations—achieve good performance, at the cost of high development and optimization effort, and limited

productivity. Message passing requires developers to call communication and synchronization primitives, marshall and unmarshall messages, and handle buffers; the hybrid OpenMP-MPI solution further increases complexity by adding OpenMP parallel and synchronization pragmas to codes executed within MPI processes.

The CGD model, PGAS languages, and more generally, parallel programming languages address this issue by providing higher-level abstractions and moving some of the workload to the compiler and runtime library. Furthermore, the CGD language abstraction decouples algorithm and datastructure implementation, ensures a correct parallel execution of algorithms while avoiding concurrency pitfalls, provides a few architecture-specific datastructure optimizations in the runtime, and still, achieves a performance similar to and sometimes exceeding MPI performance (Section 4).

PGAS Compared to modern PGAS languages, CGD datastructure distributions are more flexible, since the datastructure domain-to-process mapping is arbitrarily defined. CGD distributions allow domain overlapping, i.e., the replication of the same datastructure elements among processes. This feature proves useful for applications sensitive to replication, e.g., PDE solvers that overlap grid domains and hide latency by aggregating multiple communication steps (Section 2.3.1). Additionally, CGD datastructures allow multiple views or distributions, while PGAS datastructures are defined for a single distribution. This is beneficial when computations locally access several non-inclusive domains of the same underlying datastructure; accessing a new datastructure distribution requires sending only the missing elements via a redistribution in CGD, but it requires creating a full datastructure replica in PGAS [CCDI09].

The CGD model helps avoiding some common yet persistent PGAS performance bottlenecks. The CGD dependency graph and distribution rules allow compilers to schedule coarse-grain data transfers well before scheduling computations depending on such data, while at the same time exploiting communication-computation overlap when available. On the other hand, PGAS compilers determine remote data dependencies

based on the static analysis of loops and array access patterns. For certain complex codes, present-day compilers cannot pre-determine all remote data dependencies, or whether local or remote elements are being accessed. The former issue leads to fine-grain remote element access and latency overhead [CDHW, CCDI09, CL10], while the latter results in slower sequential code due to branch evaluation overhead [CHea03, PG08, EGC02]. We believe that a language supporting both explicit and implicit data dependencies enables users to choose the mechanism best suited to each application and performance objective (Section 3.5.1).

GLU The GLU language describes a dataflow of sequential functions and datastructures, therefore increasing the granularity and improving the efficiency of computation nodes vs. pure dataflow languages. However, similar to dataflow schedulers, the GLU scheduler partitions the graph and assigns its nodes to processes. Partition and assignment of data and computation is a complex problem—especially on machines without CC-SAS support that require data elements be sent explicitly between processes—and dataflow compilers have historically not achieved good results on larger systems [JHRM04]. In contrast to GLU and other dataflow models, CGD relies on SPMD computation nodes, delegating the partition and assignment problems to the user, and thus, exploiting the good parallel performance of SPMD.

In terms of cost transparency, CGD datastructure redistributions implemented as collective communication steps clearly expose communication complexity, while the GLU language keeps details such as process number and data assignment out of sight, making communication cost estimation more difficult throughout the application development process (Section 5.3).

1.3.2 Organization

The remainder of this dissertation is organized as follows. First, Chapter 2 provides an overview of the CGD model, formally defines the CGD graph, and presents the mapping of a few examples and their optimizations to CGD. Next, Chapter 3 introduces the new CGD language and shows how the same examples are implemented in CGD, while commenting on language particularities. In Chapter 4, we present the implementation of the CGD compiler, along with experimental results that evaluate CGD, MPI, pthreads, and OpenMP performance on different machine configurations. Finally, the αDBSP cost model for collective communication is presented and evaluated in Chapter 5, and the concluding remarks are provided in Chapter 6.

Chapter 2

Coarse Grain Dataflow Programming Model

This chapter introduces a new hybrid SPMD – coarse grain dataflow (CGD) programming model, which represents one of the main contributions of this dissertation. First, Section 2.1 discusses the position of the new model in the landscape of SPMD and dataflow programming languages, and points out similarities and differences in contrast with models from both worlds. The main aspects of the newly introduced model are briefly presented, followed by examples illustrating them. Next, Section 2.2 provides a formal mathematical definition of the CGD graph and its execution schedule. Finally, Section 2.3 shows how more complex problems are implemented and optimized within the CGD framework, using three familiar problems as examples: the stencil computation commonly employed by scientific PDE solvers, the NPB FT application, and the SPLASH2 Barnes-Hut N-body simulation [WOT+95, BBea91].

SPMD Models Message passing libraries such as MPI and SHMEM not only support SPMD-centric applications, but also provide good problem coverage, efficiency, and portability, being the de-facto scientific community standard (Section 1.2.2). Un-

fortunately, they require a significant amount of user involvement to explicitly specify data and computation partition and assignment, send and receive messages, synchronize processes, and fine-tune the messaging and synchronization performance of specific architectures.

CC-SAS machines support SPMD applications based on pthreads, Java threads, or OpenMP, which provide good efficiency while reducing message passing implementation effort and increasing problem coverage. However, they require explicit synchronization, and frequently need to maintain both fast locally mapped and slower globally accessible datastructures in order to achieve better efficiency. Generally, SPMD-centric programming models perform well at the cost of extra programming effort [CHea03, NR98, SS99, PG08].

Dataflow Models Large-grain dataflow models such as GLU address the above issue by requiring users to describe only data-computation dependencies (Section 1.2.4), and by delegating parallelism discovery and exploitation to the compiler and the runtime.

Compared to SPMD models, dataflow models the decrease application development workload at the cost of an increased compiler complexity; the compiler has to solve a harder scheduling and assignment problem, sometimes lacking information easily available to the programmer. For example, what represents a simple 2D domain decomposition and assignment problem for the user can become a complex optimization problem for the compiler. Historically, dataflow language implementations have not succeeded in offering the best performance on large systems due to data granularity overheads, memory locality, and scheduling [JHRM04, BL05].

In the world of programming models, especially parallel models, a trade-off exists between efficiency and simplicity. As expected, code that explicitly specifies data layouts and low-level communication and synchronization primitives outperforms the code au-

tomatically generated from a dependency graph. Skilled programmers can produce code at least as good as a compiler, and further, optimize it considering application and architecture particularities. However, considering both high- and low-level programming models, an interesting question emerges: how much low-level detail is required by the compiler to produce an efficient implementation?

Hybrid Model The answer to the previous question probably lies somewhere between the two ends of the spectrum. We do not expect a significant loss in efficiency if programmers describe what they know best, including data layouts and algorithmic optimizations, while compilers and runtimes implement repetitive predictable work such as data handling and architecture-specific optimizations.

CGD is a dataflow graph with datastructure distribution data nodes and SPMD computation nodes (Section 2.1). This model aims at achieving a good trade-off between programming effort and implementation efficiency by having the programmer specify data and computation decomposition and assignment, while having the compiler solve easier scheduling and orchestration problems. Furthermore, to simplify the dataflow specification, the model is augmented with distribution rules that automatically generate datastructure transformation links in the graph. Other benefits of the coarse-grain dataflow abstraction include architecture-specific communication, synchronization, data handling, and scheduling optimizations.

Similarities Similar to dataflow models, the CGD model defines a dependency graph between data nodes and computation or actor nodes. This graph is described as a list of computations taking data nodes as input and output arguments. The order in which computations are specified is irrelevant since only the data-computation dependencies determine the schedule. In particular, a topological ordering of the computation nodes represents a valid schedule.

The CGD language shares common features with dataflow programming languages

such as Id, Lucid and GLU [Jag95, JHRM04]. CGD is a functional language that obeys the single variable assignment rule, its functions are free of side effects, its data dependencies are equivalent to scheduling, and finally, it defines unusual iterative constructs owing to its functional nature [JHRM04].

Similar to SPMD models, the CGD model requires the programmer to explicitly decompose datastructures into domains, and assign datastructure domains and computations to processes (Section 2.1). Similar to "flow-based programming" [Mor94], computations are iterative functions executed in parallel rather than compiled dataflows; sequential computations provide excellent performance by leveraging decades-old compiler optimization technology.

2.1 Model Overview

The following definitions of n, p_i, process, datastructure, domain, and distribution will be used throughout this dissertation:

- A *process* represents a processing environment that can be implemented as a thread, process, etc. depending on the runtime.
- n is defined as the total number of parallel processing environments, and p_i represents the ith running process for $0 \leq i < n$.
- A *datastructure* represents an arbitrary distributed datastructure such as an array or a binary tree that contains data elements that are *read* or *written* by the n processes. Indexes are globally addressable, i.e., all processes use the same index to refer to a given element. However, there is no guarantee that all data elements can be accessed by all processes, i.e., datastructures may enforce access locality.
- A *domain* or *range* corresponds to a subset of data elements from a datastructure. As expected, domains can describe which elements from a datastructure

can be accessed by each process. A single process executes sequential computations that take datastructure domains as inputs and produces datastructure domains as outputs, given that the process has read-access to all input datastructure domains and write-access to all output datastructure domains.

- A domain distribution, or simply a *distribution* or *partition*, is an arbitrary assignment of domains to processes. Processes can be assigned overlapping domains, and therefore datastructure elements can be replicated among processes.

- A *distribution rule* defines how a distribution can be transformed into a new distribution. Distribution rules are generic, i.e., they are applicable to all datastructures that have matching distributions. For example, a redistribution rule is a distribution rule that defines a matrix of domains that are exchanged between processes to obtain a new distribution (Section 2.1.3).

2.1.1 Coarse Grain Dataflow Graph

We define a CGD graph as a dataflow graph where the nodes are either data nodes or computation nodes. *Data nodes* are $\langle datastructure, distribution \rangle$ pairs, and *computation nodes* are SPMD computations, dataflow graphs, iterative constructs, and conditional constructs (Section 2.2); for the sake of simplicity, throughout this section we consider that computation nodes are always SPMD computations. The dataflow graph links indicate which datastructure distributions are needed to execute an SPMD computation, and which computations produce a datastructure distribution.

Throughout this dissertation, we use the terms data node, $\langle datastructure, distribution \rangle$ pair, datastructure distribution, and datastructure decomposition to refer to the data nodes from the CGD graph. A data node composed of datastructure A and distribution X is represented by

$$\langle A, X \rangle = A[X] \tag{2.1}$$

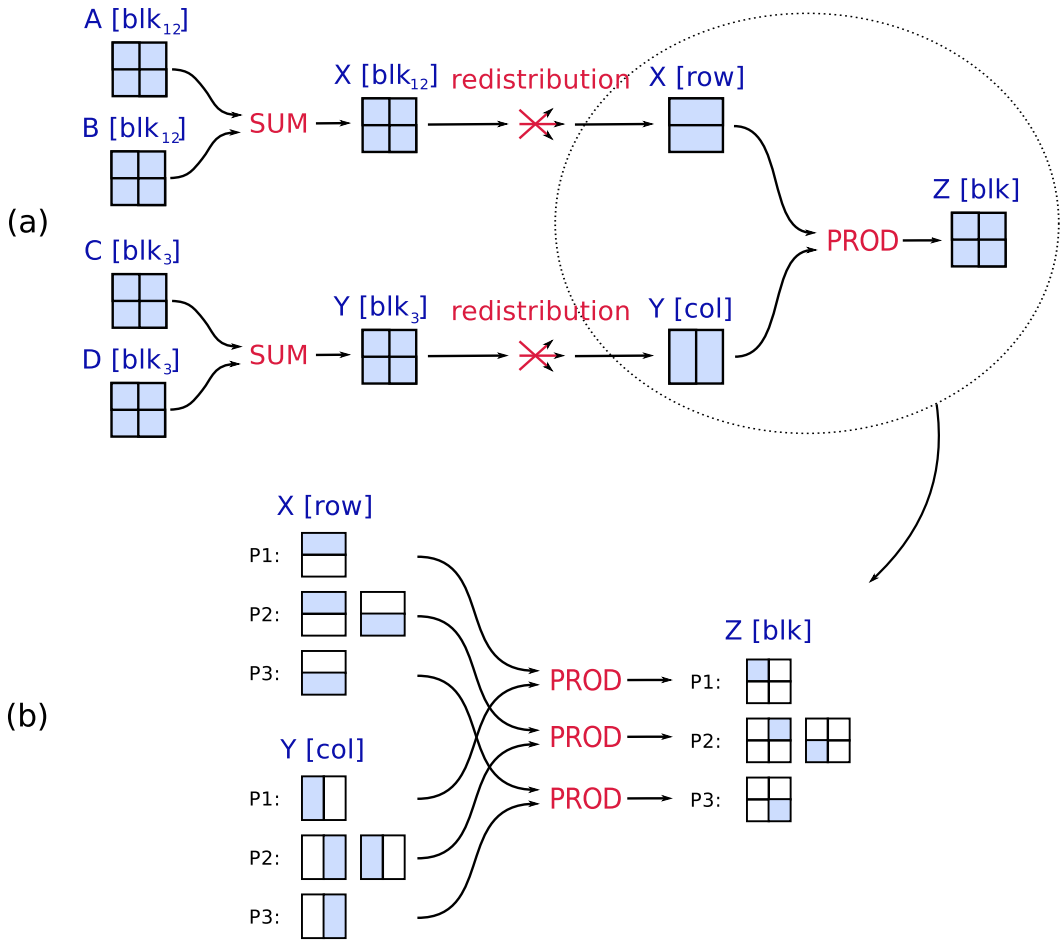


Figure 2.1: Dataflow computing $Z \leftarrow (A + B) * (C + D)$ on three processes, where A, B, C, and D are matrices: (a) the CGD graph specifies dependencies between matrix decompositions and SPMD computations; and (b) the SPMD computation $prod$ executes in parallel sequential computations acting on matrix subdomains. The row, cel, and blk distributions assign multiple domains to each process, while the $prod$ computation assigns multiple sequential computations to each process.

Similarly, we use the terms parallel computation, SPMD computation, and SPMD function to refer to the SPMD computation nodes from the CGD graph. An SPMD computation node f that takes data node $A[X]$ as an input and produces data node

$B[Y]$ as an output is represented by

$$B[Y] \leftarrow f(A[X]) \tag{2.2}$$

The remainder of this overview section introduces key CGD elements using the familiar matrix multiplication problem as a presentation device. We consider the parallel matrix multiplication implementation that uses block row, block column, and 2D decompositions. Fig. 2.1 shows how SPMD computation $Z[blk] \leftarrow prod(X[row], Y[col])$ takes matrix $X[row]$ decomposed in rows and matrix $Y[col]$ decomposed in columns as inputs, and it produces matrix $Z[blk]$ with a 2D block decomposition as output.

The CGD graph specifies both data and task parallelism: all SPMD computations exploit data parallelism, and independent SPMD computations can be executed concurrently by disjoint process sets. In Fig. 2.1a, task parallelism is exploited by executing $X[blk_{12}] \leftarrow add(A[blk_{12}], B[blk_{12}])$ on process set $P_{12} = \{p_1, p_2\}$ and $Y[blk_3] \leftarrow add(C[blk_3], D[blk_3])$ on process set $P_3 = \{p_3\}$. Fig. 2.1b shows how parallel computation $Z[blk] \leftarrow prod(X[row], Y[col])$ uses data parallelism by executing function $prod$ on process set $P = \{p_1, p_2, p_3\}$ to compute each of the four sub-matrices of Z.

Fig. 2.2 presents the CGD language implementation of the dataflow from Fig. 2.1. Here, each SPMD computation corresponds to one statement (lines 28–30): the two add computations are executed by disjoint process sets P_{12} and P_3, and the $prod$ computation is executed by all processes.

The CGD model features two easily noticeable properties: the large granularity of sequential computation chunks, and the explicit partition and assignment of both datastructures and computations. These features distance CGD from existing large- and coarse-grain dataflow models, making it a hybrid between SPMD and dataflow.

```
1  // TYPES, CONSTANTS
2  type range Range2D;
3  type partition <Range2D> PartRange2D [PartNP];
4  type mpartition <Range2D> MPartRange2D [PartNP];
5  type mswap <Range2D> MSwapRange2D [PartNP];
6  type data Vector2Dreal [PartRange2D, MPartRange2D];
7  const PartRange2D blk12[ALLn], blk3[ALLn];
8  const MPartRange2D row[ALLn], col[ALLn], blk[ALLn];
9  const MSwapRange2D blk2row[PPEn], blk2col[PPEn];
10
11 // DISTRIBUTION RULES
12 part blk2row : blk12 -> row;
13 part blk2col : blk3 -> col;
14
15 // SPMD FUNCTION Declaration
16 function prod (A, B, blk, row, col -> C)
17   Vector2Dreal A[row], B[col], C[blk],
18   Range2D row, col, blk;
19
20 function add (A, B, ra -> C)
21   Vector2Dreal A[ra], B[ra], C[ra],
22   Range2D ra;
23
24 // DATAFLOW FUNCTION Definition
25 function foo (A, B, C, D -> Z)
26   Vector2Dreal A[blk12], B[blk12], C[blk3], D[blk3], Z[blk]
27 {
28   sum <blk12> ( A[blk12], B[blk12] -> X[blk12] );
29   sum <blk3>  ( C[blk3], D[blk3] -> Y[blk3] );
30   prod ( X[row], Y[col], blk -> Z[blk] );
31 }
```

Figure 2.2: CGD language implementation of dataflow $Z \leftarrow (A + B) * (C + D)$ from Fig. 2.1. Types, constants, and SPMD computations are declared in CGD but are defined in C++; decomposition rules and dataflow functions are defined in CGD.

2.1.2 SPMD Computations

An SPMD computation is a set of sequential computations assigned to processes. Sequential computations are functions without any communication or synchronization primitives and can be written in an arbitrary iterative language; their input and output arguments are domains of distributed datastructures. Fig. 2.1b shows how computation node *prod* is represented by an SPMD computation taking matrix block decomposition arguments. SPMD computations are declared in the CGD language (Fig. 2.2, lines 15–22) but are written in C++ as sequential functions (Fig. 2.3). An exception

to this rule is datastructure redistribution SPMD computation nodes, which are automatically generated by the compiler (Section 2.1.3 and 2.2).

Sequential computations access distributed datastructure elements relying on global indexes; the CGD runtime is responsible for implementing the underlying mechanisms needed to access these elements.

SPMD computation arguments can be local or global: a local distribution argument allows sequential computations to access only the elements included in the domains assigned to each process, whereas a global distribution argument allows sequential computations to access any element from the global domain, but incurs a higher cost when accessing remote elements (Section 3.5.1).

Two sequential computations executed by an SPMD computation can produce overlapping domains of an output datastructure. In this case, the application developer needs to ensure that the replicated data elements have identical values; otherwise, the correctness of the SPMD computation may be compromised. The ability to overlap output domains adds extra flexibility to the model, e.g., parallel computations producing replicated data can avoid communication at a later stage, when remote data elements would otherwise be needed if local replicas are not available.

SPMD computations are allowed to modify their datastructure arguments to avoid unnecessary data copy; this feature is supported by in-out arguments, which are logically replicated before executing computations to maintain CGD functional semantics (Section 3.5). The translation of a CGD dataflow into C++ code avoids replication if the input data node is not required by a computation scheduled at a later time (Section 4.1.1). Replication cannot be avoided when two or more computations take the same data node as an in-out argument. Since both computations expect unmodified data as an input, and both modify the data, the only acceptable solution is copying the data; sequential dataflow languages require the same solution.

Making several modifications to the same datastructure is made possible by passing

```
1  // SPMD FUNCTION Definition
2  void prod (Vector2Dreal &A, Vector2Dreal &B,
3             Range2D &blk, Range2D &row, Range2D &col,
4             Vector2Dreal &C)
5  {
6    for (int i=blk.s1; i<=blk.e1; i++)
7      for (int j=blk.s2; j<=blk.e2; j++) {
8        double s = 0.0;
9        for (int k=col.s1; k<=col.e1; k++)
10         s += A(i,k) * B(k,j);
11       C(i,j) = s;
12     }
13 }
14
15 void add (Vector2Dreal &A, Vector2Dreal &B, Range2D &ra,
16           Vector2Dreal &C)
17 {
18   for (int i=ra.s1; i<=ra.e1; i++)
19     for (int j=ra.s2; j<=ra.e2; j++)
20       C(i,j) = A(i,j) + B(i,j);
21 }
```

Figure 2.3: C++ implementation of SPMD matrix multiplication and addition. The local domains sqr, row, col of distributed datastructures A, B, C are addressed using global indexes.

the output node of an in-out argument as an input node to the next in-out argument. In this case, several computations modify the same underlying datastructure while avoiding a deadly full datastructure replication.

2.1.3 Distribution Rules

The CGD model adds distribution rules to simplify writing dataflow graphs by automatically completing missing paths between data nodes that share the same datastructure but have different distributions. These rules are specified only once in terms of domain distributions; however, they apply to all data nodes from the graph that uses these distributions, i.e., a rule transforming distribution d_{from} to d_{to} can be triggered to transform data nodes $X[d_{from}] \rightarrow X[d_{to}]$ for any X. By defining distribution rules only once, developers are relieved from transforming data nodes every time it is required, thereby reducing the chance of human errors.

One such example is shown in Fig. 2.1a, where sum produces data node $X[blk_{12}]$ and $prod$ depends on data node $X[row]$; when rule $blk2row : blk_{12} \rightarrow col$ is specified, the compiler automatically completes the path $X[blk_{12}] \rightarrow X[row]$ by adding redistribution transformation $X[row] \leftarrow redistribute(blk2row, X[blk_{12}])$. Lines 11-13 in Figure 2.2 present the distribution rules needed to add all redistributions from Fig. 2.1a.

Distribution rules describe transformations in terms of domain inclusion, union, and intersection. Generally, a redistribution matrix specifies the domains that have to be sent and received by each process to build a target distribution (Table 3.4). Although a bit inconvenient, redistribution matrices are relatively easy to build by relying on predefined library functions that take the source and target distributions as inputs.

The model supports arbitrarily defined domains, datastructures, and distributions; to ease development, these are predefined for most common data types, and are available as off-the-shelf solutions. However, programmers can choose to define completely customized domain and datastructure representations to meet their particular needs; new domain and datastructure types are defined by providing a set of operators specific to these types (Section 3.1).

Discussion A parallel computation can access any element of a globally distributed datastructure without any user involvement if the compiler knows everything about the memory layout of each datastructure, and the mapping of its global domain to processes; this approach is indeed embraced by most PGAS languages. Assuming that PGAS compilers understand all user datastructure layouts, and fine-grain remote access is not an issue, assigning a single distribution to each datastructure has certain limitations.

In many cases, each operation performed on a datastructure works best with a different domain decomposition, e.g., the row-wise and column-wise FFT transforms executed by 2D FFT work best when the matrix has block-row and block-column distri-

butions, respectively. Since a compiler is unlikely to understand the semantics of the 2D FFT code, and assign these different layouts to the same datastructure at different points in time, optimized PGAS implementations of 2D FFT use two datastructures with explicit block-row and block-column distributions and manually copy all data elements between the two, when needed. Arguably, some of the elegance and productivity of PGAS languages is defeated when the user has to manually mimic message passing transposes to achieve good performance.

Unlike PGAS, CGD allows working with multiple distributions of the same datastructure, and relies on the developer's educated choice to specify which datastructure distribution works best for each computation, and which rules should be applied to transform datastructure distributions, when needed.

2.1.4 Orchestration

A scheduler can automatically generate a sequence of sequential computations, communication, and synchronization primitives running on each process such that all data-computation dependencies are satisfied. Compared to classic dataflow models [KBB86, JHRM04], the CGD scheduler has a simpler job as it only needs to determine the computation order and insert communication and synchronization—the partitioning and assignment of data and computation are provided by the application writer.

A scheduling solution can be as simple as traversing the graph in topological order and inserting start or end communication operations when the data become available or are needed. When an SPMD computation assigns multiple computations to a process, these computations can be executed one at a time. In this particular case, the computations scheduled for later execution can afford to receive their input data later; splitting data into multiple smaller batches can result in improved communication overlapping, as shown by the NPB FT example (Section 2.3.2).

Chapter 4 shows that for several benchmarks the CGD performance does not suffer

from dataflow overheads, being on par or even exceeding the SPMD performance; this is not surprising given that, in both cases, developers have full control over datastructure and computation decomposition and assignment.

2.2 Model Definition

In this section, we present a formal definition of the CGD model. A CGD dataflow graph is a bipartite directed graph, where the nodes are either data nodes or computation nodes. Using dataflow terminology the data nodes correspond to link nodes, and the computation nodes correspond to actor nodes.

In our dataflow model, actor nodes are pairs of input and output argument sets, and each link in the graph has a label specifying an actor argument. This augmentation is necessary since multiple data nodes can be connected to the same actor, and the order of the actor arguments is relevant. For example, specifying only that data nodes $A[row]$ and $B[col]$ are linked to computation node $prod$ is ambiguous since this graph can represent both $prod(A[row], B[col])$ and $prod(B[col], A[row])$. Accordingly, a labeling function maps every graph link to one actor input or output argument, which is possible since the graph is bipartite and each link contains exactly one actor.

For the purpose of specifying the dependencies between data and computation nodes, descriptions of how actors produce their output and how data nodes are represented are irrelevant. We will commence this section by first defining the dataflow graph, and then expanding the definition of data nodes and actors.

Definition 1. *A coarse grain dataflow graph G_D is a bipartite directed acyclic graph G specifying the dependencies between data nodes D and actor nodes A, which has input data nodes I and output data nodes O, and an edge labeling function l.*

$$G_D = \langle D, A, E; I, O, l \rangle = G(D \cup A, E) \tag{2.3}$$

where the graph nodes and edges are

$$D = \{d \mid d \textit{ is a data node }\} \tag{2.4}$$

$$A = \{f \mid f = \langle I_f, O_f \rangle \textit{ is an actor with input set } I_f \textit{ and output set } O_f\} \tag{2.5}$$

$$E = \{(d_i, f) \textit{ and } (f, d_o) \mid d_i \in D,\ d_o \in D \backslash I,\ f \in A\} \tag{2.6}$$

the labeling function has the following properties

$$\textit{for any actor } f = \langle I_f, O_f \rangle \in A, \forall i \in I_f \textit{ we have } \exists! \, e = (d, f) \in E \textit{ s.t. } l(e) = i \tag{2.7}$$

$$\textit{for any actor } f = \langle I_f, O_f \rangle \in A, \forall o \in O_f \textit{ we have } \exists! \, e = (f, d) \in E \textit{ s.t. } l(e) = o \tag{2.8}$$

and D is the set of data nodes, $I \subset D$ is the set of input data nodes, $O \subset D$ is the set of output data nodes, A is the set of actors, E is the set of links between data nodes and actor nodes, and function l maps edges corresponding to data nodes to actor inputs and outputs.

Equations (2.4), (2.5), and (2.6) define the data and actor node sets and stipulate that edges exist between data nodes and actor nodes, and actor nodes and non-input data nodes. Equations (2.7) and (2.8) ensure that for each actor input and output, there is a single edge in the graph labeled with the actor input or output.

This definition of a dataflow graph slightly diverges from the classical dataflow definition from [KBB86], given that actors are pairs of input and output arguments, and a labeling function is added to map links to these actor arguments. This approach follows more closely the solution presented by [BL05], where input and output data ports are defined for each actor, and links connect the data ports. As anticipated, (2.7) and (2.8) specify that for every single actor input and output there exists exactly one edge in the graph that is mapped to the actor argument. Generally, this means that each actor argument is linked to a single data node from the graph.

Throughout the remainder of this chapter, we use $D(G)$, $I(G)$, $O(G)$, $A(G)$, $E(G)$ and $l(G)$ to denote the corresponding elements of a dataflow graph $G(D, A, E; I, O, l)$.

For any node of the graph we can define the predecessor and successor sets in the usual manner. These definitions will prove useful when analyzing the correctness of a dataflow graph schedule.

Definition 2. *$I(x)$ and $O(x)$ represent the set of predecessors and successors of node x from dataflow graph $G_D(D, A, E; I, O, l)$*

$$I(x) = \{i \,|\, (i, x) \in E\} \tag{2.9}$$

$$O(x) = \{o \,|\, (x, o) \in E\} \tag{2.10}$$

Data nodes can be input and output nodes of the graph (Definition 1). While any data node can be an output of the graph, only data nodes that have no predecessors can be inputs of the graph i.e. $x \in I(G) \Rightarrow I(x) = \emptyset$.

Definition 3. *A distribution X is an assignment of domains X_{ij} to processes p_i*

$$X = (X_0, \ldots X_i, \ldots X_{n-1}) \tag{2.11}$$

$$X_i = \{X_{i0}, \ldots X_{ij}, \ldots\} \tag{2.12}$$

where X_i is the set of domains assigned to process p_i, and X_{ij} is the jth domain assigned to process p_i for $0 \leq j < |X_i|$

A distribution is not required to be a mathematical partition of a datastructure domain. A distribution allows overlapping domains to be assigned to different processes i.e. it is possible to have $X_{ij} \cap X_{kl} \neq \emptyset$ for $i \neq k$. The number of domains $|X_i|$ assigned to each process p_i is not required to be identical for all processes, i.e., it is possible to have $|X_i| \neq |X_k|$ for $i \neq k$.

Definition 4. *A data node $d = \langle A, X \rangle$ represents an assignment of datastructure A elements to processes p_i according to distribution X*

$$d = \langle A, X \rangle = A[X] \tag{2.13}$$

where all datastructure A elements included in any X_{ij} domain from distribution X are assigned to process p_i.

Similar to distributions, data nodes are not mathematical assignments of datastructure elements to processes; the same element can be assigned to multiple processes, i.e., element $A[x]$ can be assigned to p_i and p_k where $i \neq k$, $x \in X_{ij}$, and $x \in X_{kl}$.

Dataflow graphs are defined in terms of actors and data nodes. Actors can be defined in terms of dataflow graphs, thereby creating a graph hierarchy. Actors can be SPMD functions defined by the application writer, or they can be constructs that embed dataflow graphs.

Definition 5. *An actor $f = \langle I_f, O_f \rangle$ included in a dataflow graph G is an SPMD computation, a subgraph computation, a selection construct, or an iteration construct:*

i) f is an SPMD computation

$$b_1, b_2, \ldots \leftarrow f_{SPMD}(a_1, a_2, \ldots) \tag{2.14}$$

$$B^1[Y^1_{ij}], B[Y^2_{ij}], \ldots \leftarrow f_{SPMD}(A^1[X^1_{ij}], A[X^2_{ij}], \ldots) \tag{2.15}$$

where $I_f = \{a_i\}$, $O_f = \{b_i\}$ are the inputs and outputs of actor f, $A^k[X^k]$, $B^l[Y^l]$ are the input and output data nodes, i.e., $e_k = (A^k[X^k], f) \in E(G)$ and $l(e_k) = a_k$, $e_l = (f, B^l[Y^l]) \in E(G)$ and $l(e_l) = b_l$, and process p_i executes all computations from (2.15) for i, $0 \leq j < m$, where $m = |X^k_i| = |Y^l_i|$ for all $0 \leq k < |I_f|$, $0 \leq l < |O_f|$

ii) f is a subgraph computation

$$f_G = G_D(D, A, E; I, O, l) \tag{2.16}$$

where the actor has the same inputs and outputs as the embedded graph, $I = I_f$, $O = O_f$

iii) f is a conditional construct

$$f_{IF} = \langle G_T, G_F; I, I_C, O \rangle \tag{2.17}$$

where I_C is the input condition, G_T is a dataflow graph evaluated when the condition is true, G_F is a dataflow graph evaluated when the condition is false, and $I = I(G_T) \cup I(G_F)$ are the inputs required by both graphs; the actor takes as inputs the union of all inputs $I_f = I_C \cup I$, and both graphs produce the same outputs as the actor $O = O(G_T) = O(G_F) = O_f$

iv) f is an iterative construct

$$f_{LOOP} = \langle G_L; I, O, O_C \rangle \tag{2.18}$$

where G_L is a dataflow graph evaluated for multiple iterations until the output condition $O_C \in O(G_L)$ becomes true, and after each iteration, the outputs $O = O(G_L) \backslash O_C$ of the loop graph are logically assigned to the inputs $I \subset I(G_L)$ of the loop graph, i.e., $I \leftarrow O$, and $I_f = I(G_L)$, $O_f = O$.

Note that the inputs and outputs of actors (2.16), (2.17), and (2.18), which embed dataflow graphs, are both nodes in the embedded graph and actor arguments. When executing (2.15), the same data element may be produced more than once if the output distribution has overlapping domains. Computations are assumed to produce identical results for the same data element; otherwise, correctness is not ensured.

The CGD model recognizes three types of *SPMD computation* actors. These are classified in terms of datastructure domain access. (Section 3.5.1):

1. SPMD computations consisting of sequential computations that access only the local domains of each datastructure. Datastructure access requires no interprocess communication.

2. SPMD computations consisting of sequential computations that access the local domains of each datastructure, and the global domains of some datastructures that are not in-out arguments. Global datastructure access might require the runtime to execute interprocess communication.

3. Datastructure transformations implemented by the runtime based on distribution rules.

Accessing only the local domains results in the best performance during the computation phase, but this approach requires knowing the required distributions ahead of time. Accessing the global domains removes this constraint at the expense of some possibly higher communication cost. The CGD model provides both mechanisms, as well as a mixture of both, giving the programmer the opportunity to make an educated choice. CGD model dataflow semantics is exploited by the runtime, which implements optimizations such as *read* caching and *write* buffering while avoiding race conditions. Section 3.5.1 describes how local and global domain computations act on datastructures, and Section 3.4 illustrates how distribution rules are defined.

The *subgraph computation* actor from (2.16) is provided as a convenience method allowing a hierarchical organization of the graph, and becomes useful when expressing task parallelism. The compiler might choose to expand the subgraph into the parent graph to increase the optimization scope.

The *conditional construct* actor defined by (2.18) selectively executes the subgraph associated with the active branch. This construct is equivalent with the dataflow chart from Fig. 2.4a. The true and false subgraphs might require different input sets, i.e., the true subgraph G_T requires inputs $I_M \cup I_T$, while the false subgraph G_F requires inputs $I_M \cup I_F$. As expected, the same outputs O are produced by this construct regardless of which branch is executed.

The *iterative construct* actor defined by (2.18) addresses the well-known difficulty of dataflow programming languages to express iterations. This construct is represented

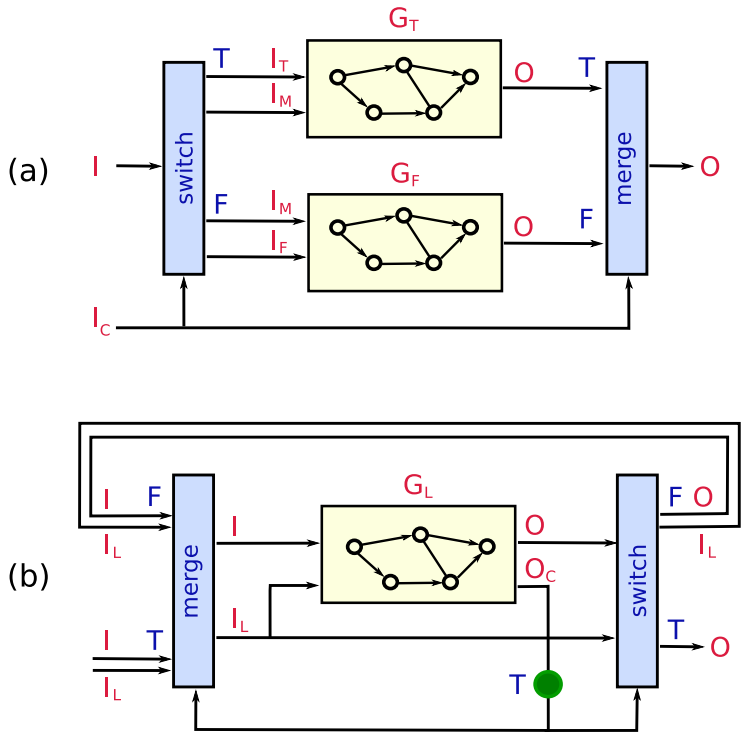


Figure 2.4: Dataflow diagrams for CGD conditional and iterative constructs. (a) The conditional construct takes $I \cup I_C$ as inputs and produces O as output; (b) The iterative construct takes $I \cup I_L$ as inputs and produces O as output. All input and output labels represent data node sets. The O_C data node is initially set to true.

by the dataflow from Fig. 2.4b. It takes a subgraph G_L as an argument, and all the outputs of subgraph O, except the output condition O_C, are assigned to their corresponding inputs I after each non-terminal iteration. Iterative constructs are defined similarly by older dataflow programming languages such as Lucid and Id [AW80, Jag95]. However, the latter define the iteration output names as the input names prepended with the *next* keyword, while CGD inputs and outputs have arbitrary names, and matching is done based on argument order (Section 3.5).

Definition 6. *The schedule $S(G)$ of dataflow graph $G(D, A, E; I, O, l)$ is an ordering of data nodes and actor executions representing a topological ordering of the nodes in the graph. The execution $E(f)$ of an actor f is represented by the actor paired with the*

schedule of the graphs, if any, embedded in the actor.

$$S(G) = (s_1, s_2, \ldots, s_i, \ldots) \tag{2.19}$$

$$E(f_{SPMD}) = f_{SPMD} \tag{2.20}$$

$$E(f_G) = \langle f_G,\, S(G(f_G))\rangle \tag{2.21}$$

$$E(f_{IF}) = \langle f_{IF},\, S(G_F(f_{IF})),\, S(G_T(f_{IF}))\rangle \tag{2.22}$$

$$E(f_{LOOP}) = \langle f_{LOOP},\, S(G_L(f_{LOOP}))\rangle \tag{2.23}$$

where

$$s_i \in D \cup \{E(f) \,|\, f \in A\},\; s_i \neq s_j \;\forall i \neq j \tag{2.24}$$

$$\forall (s_i, f) \in V,\; s_i \in D, f \in A,\; \exists!\, s_j = E(f) \textit{ and } j > i \tag{2.25}$$

$$\forall (f, s_i) \in V,\; f \in A, s_i \in D,\; \exists!\, s_j = E(f) \textit{ and } j < i \tag{2.26}$$

Equation (2.24) stipulates that no two steps of the schedule are identical, while (2.25) and (2.26) ensure that dependencies between data and computation nodes are satisfied.

A schedule is defined in terms of actor executions, and some actor executions are defined in terms of schedules. These recursive definitions allow building a hierarchical tree of schedules, where the leafs are data nodes and executions of SPMD computations.

The output data nodes $O(G)$ of graph G can be produced starting from the input data nodes $I(G)$ by executing all steps of the schedule $S = (s_i)$. Some of these steps are SPMD computations and are trivially executed, while others are executions of subgraphs, conditional constructs, and iterative constructs.

A subgraph actor execution $\langle f_G,\, S(G(f_G))\rangle$ requires simply executing the schedule of its subgraph $S(G(f_G))$. A conditional actor execution $\langle f_{IF},\, S(G_F(f_{IF})),\, S(G_T(f_{IF}))\rangle$

consists of first evaluating the condition $I_C(f_{IF})$, and then executing the schedule $S(G_T(f_{IF}))$ if the condition is true, or the schedule $S(G_F(f_{IF}))$ if the condition is false. Finally, executing a loop actor $\langle f_{LOOP},\, S(G_L(f_{LOOP}))\rangle$ requires executing the schedule $S(G_L(f_{LOOP}))$, and then evaluating $O_C(f_{LOOP})$; if the condition is true, the execution is complete; otherwise, the outputs $O(f_{LOOP})$ are assigned to the inputs $I(f_{LOOP})$, and the execution of this actor continues until the condition becomes false.

A schedule for a dataflow graph may contain both task and data parallelism. Data parallelism is trivially exposed by every SPMD computation that is executed on multiple processes. Task parallelism is exposed when two independent computations are executed on separate process sets. Let us consider the schedule corresponding to the dataflow graph shown in Fig. 2.1a:

$$\begin{aligned} S = (&A[blk_{12}], B[blk_{12}], C[blk_3], D[blk_3],\, sum_X,\, X[blk_{12}],\, redistribute(blk2row),\\ &sum_Y,\, Y[blk_3],\, redistribute(blk2col),\, X[row],\, Y[col],\, prod,\, Z[blk]\,) \end{aligned} \tag{2.27}$$

where blk_{12} domains are assigned to process set $P_{12} = \{p_1, p_2\}$; blk_3 domains to set $P_3 = \{p_3\}$; and row, col, blk domains to set $P = \{p_1, p_2, p_3\}$.

SPMD computation actors sum_X and sum_Y from schedule (2.27) are executed in parallel. This example can be generalized easily by executing in parallel two subgraph computation actors rather than two SPMD computation actors.

In the example schedule from (2.27), all distributions and redistribution matrices are assumed to be constants that are available to all computation nodes from the schedule. If this was not the case, the schedule would contain computation nodes that produce distributions and redistribution data nodes that are later required by other redistributions and computation nodes (Section 3.3).

The sequence of data nodes and actor executions from a schedule can be any traversal of the graph in topological order, thereby obeying a core principle of dataflow models.

Finding an efficient traversal represents an important problem addressed by the programming model implementation, since many traversals are typically possible. However, CGD compilers solve an easier problem, since fine-grain computation assignment is provided by users, and the topological order search space is limited to valid permutations of SPMD computation nodes.

Based on a schedule, the compiler can generate SPMD code running on the entire process set by listing all sequential SPMD computations according to the schedule order, and adding communication and synchronization for all SPMD computations that are data transformations. In the latter case, each data transformation results in two operations: the beginning of the transfer when the datastructure distribution becomes available, and the end of the transfer before the datastructure distribution is required by a computation. The original execution in the schedule of an SPMD transformation is substituted with an interval to maximize the communication-computation overlap.

2.3 Examples

This section introduces the CGD implementation of three familiar problems: the stencil computation kernel, the NPB FT, and the SPLASH2 Barnes-Hut N-body simulation. For each benchmark, we first present the problem and its importance in the scientific computing world. We then discuss the most common algorithmic and architectural challenges on present day machines. Finally, we briefly describe their CGD dataflow and potential optimizations. These problems have a relatively simple dataflow; moreover, changing the data layouts and computation assignments to implement non-trivial optimizations requires a little more than redefining a few distributions. Section 3.6 presents in detail the original and optimized CGD language implementations of these problems.

2.3.1 Stencil Computation

The stencil kernel implements a typical PDE solver that includes the "halo exchange" communication pattern so common among scientific computing codes. Many HPC labs working on fluid dynamics, plasma physics, bioinformatics, or material science make heavy use of PDE solvers to compute problems such as weather prediction, ocean current modeling, climate change, and blood flow dynamics. Although these scientific codes are rather complex systems developed by numerous teams, the equations they solve are finite differencing schemes that are similar from a communication-computation perspective with simpler equations, such as the shallow water or heat dissipation equation [BN]. More precisely, the time-forward equation solvers require the state of the physical system, as well as the spatial derivatives at time t to compute the state of the system at time $t + \Delta$. Moreover, computing the spatial derivatives of point x at time t requires reading the values adjacent to point x at time t.

Parallel implementations of such equation solvers execute iteratively computation or "domain update" steps, followed by communication or "halo exchange" steps. The 2D or 3D grids holding the discretized system state are partitioned among processes, typically using block decomposition. Inevitably, some neighbors of point x that are needed to compute the next value of x may be assigned to other processes. The communication step sends these points between the processes assigned to neighboring blocks before the computation step can be executed.

The stencil kernel captures the structure of a latency-bound parallel PDE solver by including a small "domain update" step corresponding to heat dissipation, and a "halo exchange" step produced by the 2D grid decomposition. Fig. 2.5a shows a schematic of the stencil computation steps without fully describing all of the nodes and links of the dataflow graph (Section 3.6.1). In contrast to 3D PDE solvers, 2D solvers have a poorer performance due to their larger communication-computation ratio. When the computation step is short compared to the constant communication latency, the

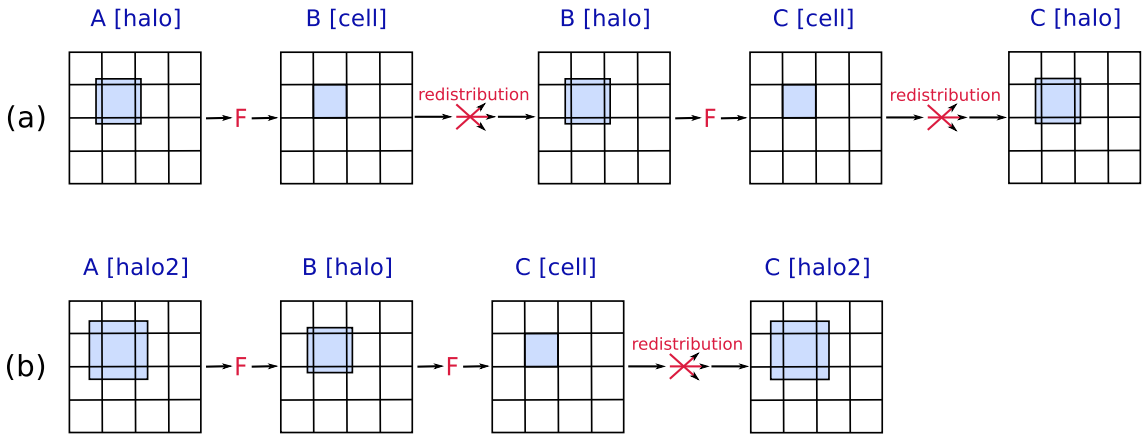


Figure 2.5: Stencil kernel "halo exchange" aggregation optimization: (a) basic stencil computation with two iterations and two communication steps; and (b) communication aggregation optimization replicating data and computation; the first iteration computes the "halo" elements needed by the second iteration.

performance is particularly poor since communication takes a large fraction of runtime regardless of the amount of data being sent over the wire. In this case, the PDE solver is latency-bound.

The parallel performance of latency-bound 2D solvers greatly depends on architecture parameters, or more precisely, on the computation unit speed and the interconnect latency. Distributed CC-SAS machines such as the SGI Altix family, and modern RDMA interconnects such as Infiniband and Cray SeaStar, have latencies in the submicrosecond range, and perform better than clusters running two-sided MPI protocols (Section 4.2). Fortunately, constant improvement is seen both in the realm of interconnects and computing units. And yet, whether newer machines will have a smaller communication latency per computation power ratio to avoid latency bottlenecks is still uncertain. The stencil computation remains an interesting benchmark as long as the latency problem remains unsolved.

An effective optimization that diminishes the overhead of communication latency is trading extra computation for communication by merging two communication steps

into one, and executing two computation steps without the intermediary "halo exchange" step. Fig. 2.5b outlines this algorithm: the first computation step relies on the larger *halo2* domain to compute the *halo* domain rather than the *cell* domain, as was the case with the original algorithm. Hence, the first communication step is avoided since the *halo* domains provided by the first iteration computation are available for the second iteration; the second computation step remains unchanged, but it sends twice as much data to build the *halo2* rather than *halo* distribution starting from the *cell* distribution. This optimization leads to improved results when communication cost is dominated by the latency and not the bandwidth cost (Section 4.2).

Application developers can implement the communication step aggregation optimization in CGD by replacing the *halo* and *cell* distributions taken by computation F with the new distributions *halo2* and *halo*, and by changing the distribution definitions and distribution rules accordingly (Fig. 2.5, Section 3.6.1).

2.3.2 NPB FT

The numerical Aerodynamic Simulation Parallel Benchmark (NPB) FT uses spectral methods to solve the following equation for a 3D vector

$$\frac{\partial u(x,t)}{\partial t} = \alpha \nabla^2 u(x,t) \tag{2.28}$$

The proposed solution first applies a Fourier transform to both sides of the equation, finds the solution analytically, and then applies an inverse Fourier transform to recover the original solution [BBea91]. The discrete solution of the equation first computes a forward FFT of the 3D grid, applies exponentials to the result, and then computes the inverse FFT of the 3D grid.

From a parallel computation perspective, the most interesting sections are the FFT transforms, which account for most of the work and expose non-trivial scalability prob-

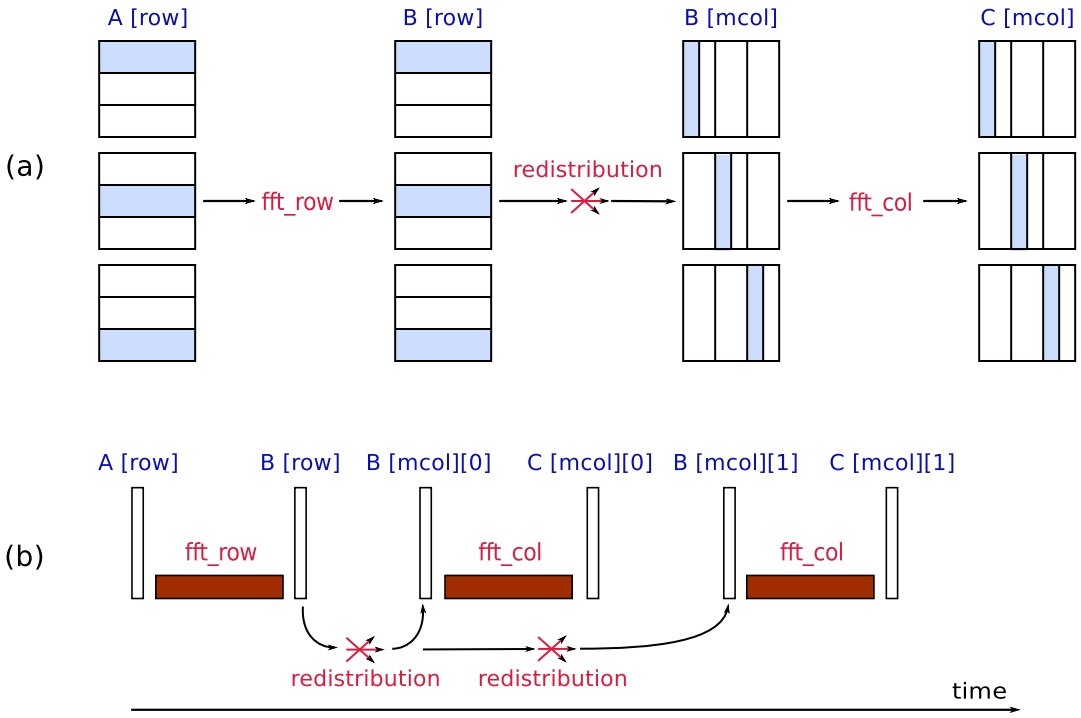


Figure 2.6: NPB FT "overlap slab" optimization: (a) 2D FFT dataflow includes a column-wise FFT SPMD computation acting on multiple "slab" domains; and (b) column-wise FFTs are executed while "slab" domains are sent between processes.

lems. The NPB MPI implementation of FFT decomposes the 3D grid into 1D or 2D layouts depending on the number of processes. For the 1D layout, the algorithm decomposes the 3D domain into Z-planes, computes the X- and Y-wise FFTs, and then transposes the vector into Y-planes and computes the Z-wise FFTs. For the 2D layout, the algorithm decomposes the domain into X-columns, computes the X-wise FFTs, and then transposes the vector into Y-columns, and computes the Y-wise FFTs. Finally, it transposes the vector into Z-columns, and computes the Z-wise FFTs. The 1D layout is faster than the 2D layout as it requires a single transpose, but this limits the available parallelism to the number of Z-planes. Subsequently, when the number of processes is smaller than the number of Z-planes the 1D layout is used, otherwise the 2D layout is used.

The NPB FT benchmark is typically bandwidth-bound since its parallel FFT transforms heavily exercise interconnect bandwidth capacity; the parallel transform includes local X-, Y-, and Z-wise FFT computations, as well as vector transpositions implemented as blocking all-to-all collective communication. This algorithm suffers from poor scalability on large systems since the transpose sends almost the entire data domain over the interconnect; on systems with limited global bandwidth, this operation represents a large fraction of the total runtime (Section 4.2). However, for large data sets scalability improves when the working set stops fitting in the aggregate cache, and the computation time per data element increases, whereas the bandwidth cost per data element remains constant.

Fortunately, communication overhead can possibly be avoided by overlapping communication and computation. The "overlap slab" optimization works by splitting each vector domain into several smaller domains called slabs, and computing the local FFTs on each slab independently [BBNY06]. Communication overlap is achieved by executing the local FFTs for a given slab at the same time transferring data needed to compute the FFTs for the next slab. Fig. 2.6 shows a simplified dataflow for the 2D version of this algorithm, illustrating how the transpose and local FFTs are concurrently executed.

Exploiting communication-computation overlap can produce code significantly faster than the original NPB FT algorithm [BBNY06]. The overlap is most effective when the architecture supports asynchronous remote memory operations, and data transfers are executed by the system while processors are free to execute computations; such systems include Infiniband clusters and the newer SGI Altix UV CC-SAS machines.

Implementing the "overlap slab" optimization for NPB FT is easily achieved in CGD: new distributions are defined to represent slab decompositions, and the FFT dataflow graph and distribution rules are modified accordingly (Section 3.6.2). The simplified 2D example from Fig. 2.6a shows how distributions are changed from block decompo-

sition *col* to slab block decomposition *mcol*; Fig. 2.6b shows how column distribution *mcol* assigns multiple slab domains to each process, and accordingly, how SPMD computation *fft_col* is executed multiple times for each slab domain. The CGD compiler automatically overlaps communication and computation when possible; hence, the execution of the 2D FFT dataflow from Fig. 2.6a takes advantage of overlap by sending the *mcol* slab domains to other processes while the *fft_col* computations are executed.

Section 3.6.2 presents the CGD language implementation of NPB FT "overlap slab" for 3D vectors; this implementation refines the 2D layout decomposition of 3D vectors by defining domains of size $a \times b \cdot fftblock \times dimm$ [BBNY06], while slab size is adjusted to ensure that interprocess messages have the same predefined size.

2.3.3 Barnes-Hut N-Body Simulation

Hierarchical algorithms that solve n-body problems are particularly useful in simulating physical systems where distances span over a large range of scales. Hence, these algorithms are commonly employed by HPC applications in the areas of astrophysics, plasma physics, and molecular dynamics. These applications simulate the evolution of a physical system by computing the interaction between n particles during each time step, and by updating the state of the system to the next time step. Unfortunately, the number of interactions grows quadratically with the number of particles, making the problem unsolvable for large systems if all pair-wise interactions are to be computed. Hierarchical n-body methods such as Barnes-Hut divide the 3D space recursively into an octotree, where each space domain, called a cell, corresponds to a node of the octotree (Fig. 2.7a) [SHG95]. The computation of the pair-wise interactions between particles can be reduced by observing that the interactions between a particle P and a group of particles contained by a distant cell C can be approximated by the interaction between particle P and cell C. For example, for the classical gravitational problem, the long-range interactions can be computed as interactions between particle

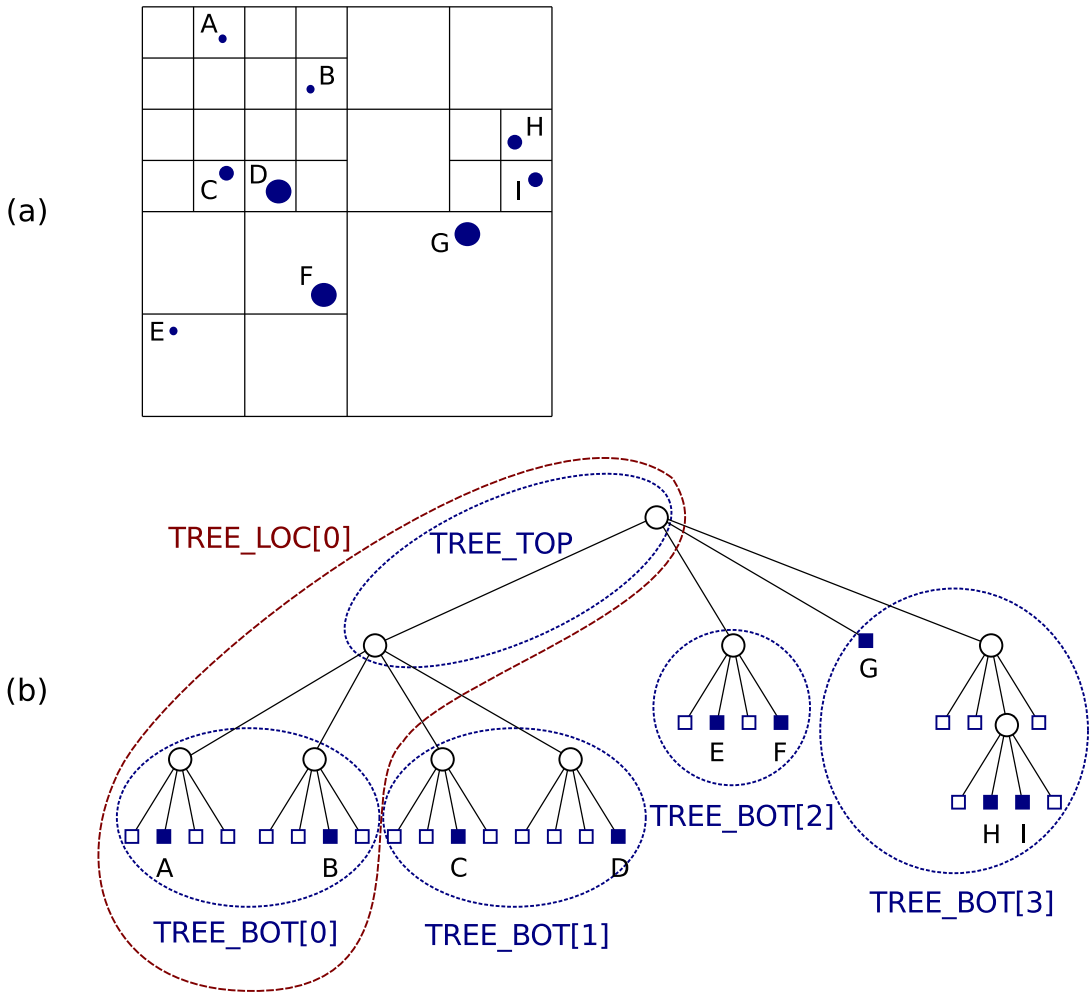


Figure 2.7: Barnes-Hut 2D particle representation: (a) spatial; and (b) quadtree. Each of the four processes, p_i, holds the $tree\_loc[i]$ nodes, which include the $tree\_bot[i]$ nodes stored only by p_i, and the $tree\_top$ nodes replicated among all processes.

P and the center of mass of cell C [GKS94, SHT+95].

The CGD parallel Barnes-Hut algorithm presented herein is based upon the SPLASH2 [WOT+95] implementation, which defines two types of tree nodes: cell nodes and leaf nodes. The cell nodes have eight children corresponding to their spatial subdomains—these children can be either cell or leaf nodes. Leaf nodes contain a list of particles assigned to their spatial domain, but no more than a predefined number of particles. This approach reduces the tree depth by avoiding the recursive cell subdivision when

only a few particles fall into the same cell. The parallel Barnes-Hut algorithm includes the following steps:

1. Building the tree by inserting all particles one by one into an initially empty octotree. When new particles are inserted into the tree root cell node, they are recursively inserted into cell nodes until a leaf node is reached. If the leaf node has too many particles, it is expanded into a cell node with eight children, and all the particles assigned to the leaf are reinserted into the newly created cell. This algorithm creates an adaptive tree that has a higher depth in areas with higher particle density.

2. Computing the center of mass of the tree nodes by first computing the center of mass of the leaf nodes, followed by recursively computing the center of mass of all cell nodes.

3. Computing the forces acting on each particle P by recursively computing the interactions between particle P and the nodes of the octotree, starting from the tree root. When the recursion reaches a cell node C or leaf node L that is distant according to a predefined measure, the recursion stops and the interaction is computed between particle P and node C or L. Otherwise, the recursion continues, and the interactions are computed between particle P and all children of cell node C, or between particle P and all particles contained by leaf node L.

4. Updating the particle properties based on the time step, coordinates, and forces computed during the previous steps.

5. Partitioning the tree between the n processes by aggregating the computational weight of each particle and assigning contiguous sections of the tree to each process, such that the total computational weight of each section is load balanced.

A CGD dataflow representing these steps is illustrated in Fig. 2.8, while Section 3.6.3 presents a detailed code implementation. The algorithm distributes the particles of

the octotree among n processes according to the *costzones* partition [SHG95]. Each process p_i locally holds the domain of the tree $tree\_loc[i]$, which represents the union of the bottom tree domain, $tree\_bot[i]$, assigned only to process p_i, and the top tree domain, $tree\_top$, replicated among all processes (Fig. 2.7b). The assignment of these tree domains to processes is described by distributions $tree\_loc$, $tree\_bot$, and $tree\_top$, while the following distribution rule applies: $tree\_loc = tree\_bot + tree\_top$ (Section 3.4).

Step (1) described above is executed in parallel by $makeTree$, with each process inserting the particles assigned to itself into a new tree partitioned according to distribution $tree\_top01$ (Fig. 2.8). This step involves inserting particles into remote subtrees since particles may be reassigned to new processors after each iteration. During step (2), each process computes $centerMass$ for all the leaf and cell nodes assigned to itself, while occasionally reading the center of mass of remote cells. For step (3), each process computes the interactions of all the particles assigned to itself by recursively descending the tree from the root. The *force* computation depends on $node[tree\_loc+]$ and $body[tree\_loc+]$, i.e., the centers of mass of each cell node of the tree and all of the particles. Although the nodes assigned locally are expected to be frequently accessed, the + sign allows the force computation to read remotely assigned centers of mass and particles. Next, step (4) is executed by computation $advance$ by updating the coordinates and speed of all particles based on the newly computed forces, and therefore the accelerations. Finally, during step (5) $score$ aggregates the computational weight

of all particles from the tree, and $balance$ then creates a new tree distribution, $tree\_top01$, by assigning a contiguous tree section to each processor while trying to evenly balance the weight of each section based on *costzones*. Both operations access both local and remote tree nodes, and each process computes the replicated top of the tree (Fig. 2.8).

Typical scalability issues exposed by the parallel Barnes-Hut algorithm include communication overhead when the problem size is too small for a given number of processes, work load imbalance, and poor scalability of the tree building phase.

During an iteration, each process computes the forces acting on local particles by reading parts of the remote essential-tree interacting with these particles [SHG95]. There are two main approaches to access the remote data needed by the local force computation: i) each process pulls remote node elements on a need-to-know basis; and ii) each process builds a list of local nodes included in the essential-tree of each other process, and then, it sends the nodes to these processes prior to the force computation phase.

In the first approach, fewer data elements are read, and implementing the algorithm is easier; however, a lower latency interconnect, such as that of CC-SAS or RDMA machines, is required to achieve good performance. For the second solution, all data elements that may be needed by remote processes are sent over the wire. Furthermore, the programmer needs to explicitly compute, send, and receive these nodes, thus conservatively estimating data dependencies. While the first approach is simpler to use and minimizes the amount of sent data, the second approach works better on higher latency machines such as MPI clusters.

As previously mentioned, the SPLASH2 Barnes-Hut algorithm solves the load balance problem using the *costzones* tree partitioning technique [SHT+95]. Nevertheless, the tree building phase can become a scalability bottleneck when the top of the tree computation takes a significant fraction of the entire tree computation time. This is the case since the top tree computation is replicated among all processes and provides no speedup. Moreover, it grows in size as the number of processors increases; the top tree size increase is needed to maintain load balance among the smaller subtrees. A trade-off naturally occurs between the load balance accuracy and the overhead from building a larger replicated top tree.

The CGD Barnes-Hut implementation uses global distribution arguments to *read* remote tree nodes during the force computation phase, and *write* remote tree nodes during the tree building phase. Although the CGD semantics is more restrictive, it enables several optimizations by ensuring that these distributed trees are accessed only

for *read* or *write* operations. Not only does this reduce the explicit fine-grain synchronization between concurrent writers to a single barrier, it also allows datastructure access optimizations such as caching remote elements into local memory and buffering writes into batches.

When processes *read* remote elements from a global distribution datastructure, the CGD runtime will use the fastest communication primitive to retrieve the elements, store them locally, and provide them to the application; on CC-SAS machines, these operations translate into remote cache line *reads* and local *writes*. When the working set size exceeds the processor cache size, the CGD mechanism becomes more efficient than simply accessing remote memory locations each time the element is needed; when incurring a cache miss, the CGD application will read elements from local memory rather than remote memory. The CGD model allows this optimization by ensuring that these distributed datastructure are accessed only for *read*.

When processes *write* remote elements to a global distribution datastructure, the CGD runtime will first queue the *write* requests, and then will send and execute them on remote processes in batches; on CC-SAS machines, these operations translate to copying data elements to local memory blocks, transferring the blocks from local to remote memory, and having remote processes copy the memory blocks to remote data elements. The CGD solution is faster than writing remote data elements one by one as it eliminates fine-grain synchronization and avoids cache line false-sharing. The CGD model allows this optimization by ensuring that these distributed datastructures are never accessed for *read*, and the time and order in which *writes* take place are irrelevant to the correctness of the algorithm. The CGD semantics requires that any sequential ordering of the local and remote *writes* that preserve the sequential ordering of the operations executed by each process leads to a correct result. In particular, inserting particles into trees in any order will produce the same correct tree during the Barnes-Hut tree building step.

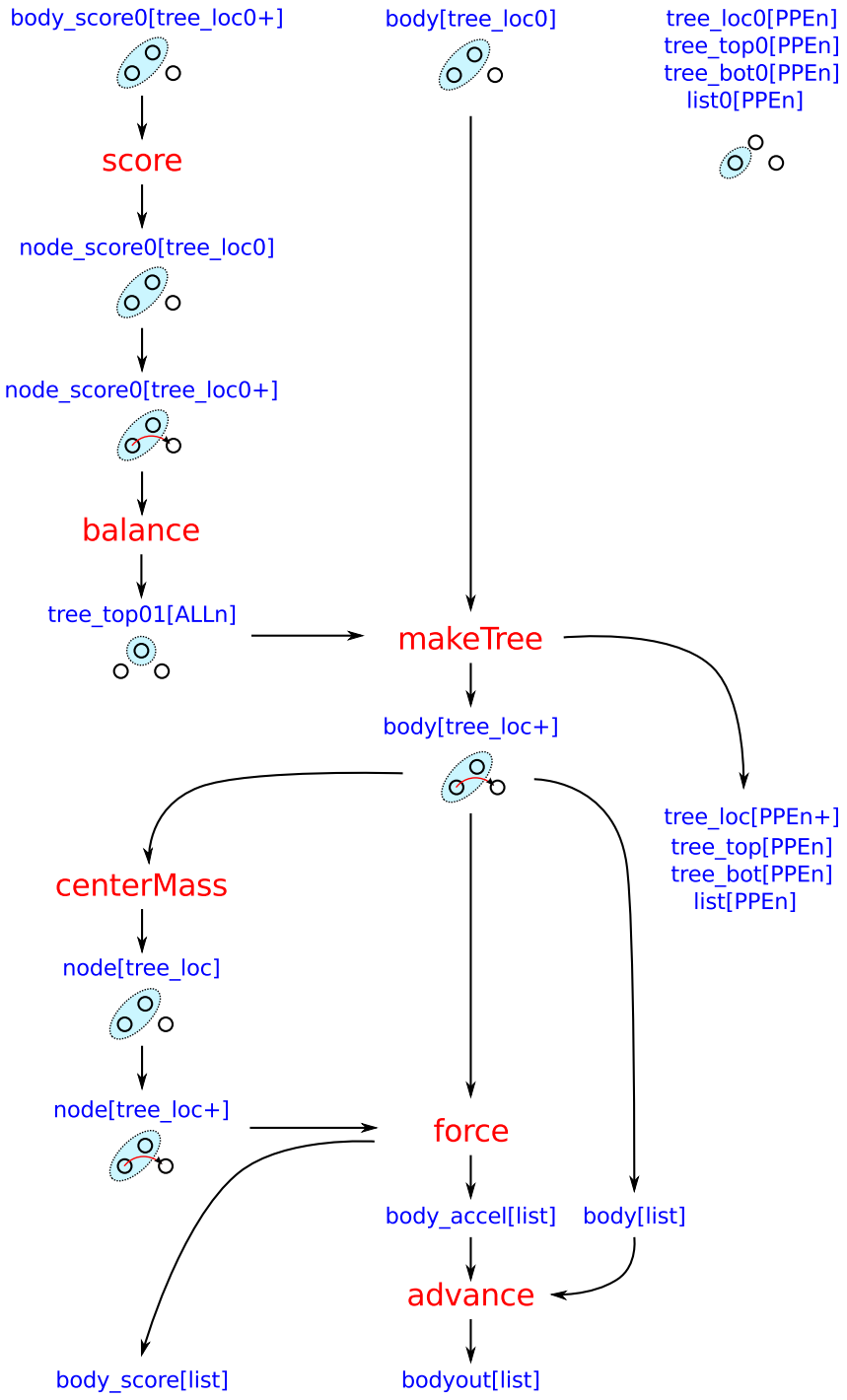


Figure 2.8: CGD dataflow for Barnes-Hut iteration: i) a tree partition $tree\_top01$ is computed by $score$ and $balance$ based on particle computation weight; ii) a tree is created by $makeTree$ by inserting all particles into the top of the tree; and iii) the particle position is computed by $advance$ after evaluating $centerMass$ for tree nodes and $force$ for particles. Data nodes with distributions followed by the $+$ sign allow for global datastructure domain access.

Chapter 3

Language Specification

This chapter introduces a programming language that implements the CGD parallel programming model presented in Chapter 2. The CGD language is a coordination language with dataflow semantics that describes parallelism at a high-level, and relies on existing languages to specify its sequential components. First, we give an overview of the essential CGD language features, and discuss some design alternatives. The following sections introduce the main language elements, starting with types, constants, and distribution rules, followed by SPMD functions, dataflow functions, and language constructs for dataflow computation nodes. Finally, Section 3.6 illustrates three CGD applications, along with an analysis of their implementation and optimization effort.

Application Structure The implementation of a CGD application requires two code sections to be provided. The top-level section is written in the CGD coordination language, and describes data and task parallelism, as well as data and computation assignment. The low-level section is written in C++, and contains sequential computations, data types, and decomposition definitions. The compiler uses the dependencies provided by the top layer to generate C++ code that includes communication, synchronization, data handling, and an ordering of sequential computations (Section 4.1).

In most cases, this code structure forces a clear separation between a shorter top-level parallel section and a lengthier low-level sequential section. This separation encourages programmers to consider and exploit parallelism during the design stage, while at the same time making the most common forms of parallelism easier to express and less prone to errors.

The advantage of a top-level parallel and low-level sequential code structure becomes more prominent in large projects, where programmers are assigned tasks based on their skill set. Structuring large-scale numerical solvers proved beneficial during our work at the Geophysical Fluid Dynamics Laboratory affiliated with Princeton University: on the one hand, sequential "kernels" were written by physicists, and on the other, higher-level "drivers" were written largely by computer scientists. The "kernels" mostly contained the numerical methods specific to each PDE solver, while the "drivers" acted as a glue, calling both "kernels" and data distribution primitives. Alternatively, allowing an arbitrary parallel code structure would typically require all programmers to understand parallel performance bottlenecks and optimize parallel codes. This would therefore demand a high-level of specialized training for all engineers and scientists.

Language Overview CGD is a functional language with dataflow semantics that inherits some of the features first introduced by the Lucid and Id programming languages [AW80, Jag95]. The CGD language includes a small set of constructs needed to declare types and sequential computations, and to define dataflow graphs and distribution rules.

If all types are declared in CGD, datastructure and domain types are arbitrarily defined by users in C++ subject to certain constraints. Subsequently, the language defines template datastructure decomposition and redistribution types based on these user types (Section 3.1). Although sequential computations are declared in CGD, they are implemented as C++ functions with standardized signatures (Section 3.5.2).

Parallelism is exposed in CGD by defining dataflow functions (Section 3.5.3). These functions represent a dataflow graph with data and computation nodes, and are written as a sequence of computations taking both input and output arguments. Each computation represents a computation node, and its input and output data node arguments represent its predecessors and successors in the dataflow graph, respectively.

Computations can be SPMD functions, dataflow functions, iterative language constructs, and conditional language constructs (Section 2.2). The semantics of these constructs is similar to other functional languages: i) iterative constructs assign the output values produced by one iteration to the inputs consumed by the following iteration; and ii) conditional constructs compute the same output datastructure distribution by evaluating one of two branches.

The language introduces the notion of distribution rules to mitigate the need for datastructure decomposition transformation (Section 3.4). These rules are defined in the global and local scopes, and are used to automatically create dataflow graph links between data nodes with identical datastructures.

Code Generation The compiler transforms a dataflow function into parallel C++ code first by topologically ordering the computation nodes, and then, by adding communication, synchronization, and other operations (Section 4.1.3). Hence, the resulting C++ code contains sequential C++ function calls corresponding to computation nodes, and collective communication and synchronization operations corresponding to redistribution operations. The generic collective communication runtime library supports several underlying communication protocols including MPI, SHMEM, and pthreads.

Discussion A dataflow coordination language such as CGD generally has several benefits vs. a pure dataflow programming language such as Lucid [JHRM04]. Particularly, CGD benefits include language simplicity and the ability to embed iterative language computations. This approach takes advantage of the programmer's familiarity

Table 3.1: Type declaration syntax

Declaration	Syntax
Domain	**type range** DomType;
Datastructure	
non-decomposable	**type data** DataType;
decomposable	**type data** DataType [DistTypeList];
Distribution	**type partition** <DomType> DistType [DistTypeList];
Multi-distribution	**type mpartition** <DomType> DistType [DistTypeList];
Redistribution of	
distribution	**type swap** <DomType> RedistType [DistTypeList];
multi-distribution	**type mswap** <DomType> RedistType [DistTypeList];

with mainstream programming languages, leverages decades-old sequential compiler optimization technology, and encourages the reuse of existing codes and off-the-shelf solutions.

This chapter starts by presenting the syntax for type declaration, type definition, constant declaration, and redistribution rule definition (Section 3.1). It then describes SPMD function declaration, and the constructs needed to define dataflow functions (Section 3.5). Finally, three applications and their CGD implementation are presented: the stencil computation, the NPB FT, and the Barnes-Hut N-body simulation (Section 3.6). Throughout this chapter, the presentation of the language syntax is aided by informal syntax tables and code fragments from the NPB FT "slab" implementation (Section 2.3.2).

3.1 Types

Domains The CGD language defines types for *domains* or *ranges*, which represent subsets of datastructure elements (Section 2.1). Domain types are declared using the *type range* keywords followed by the domain type name (Table 3.1). Domain types are arbitrary types defined by users in C++ by providing a minimum set of data-handling

operators: *equal*, *included*, *intersect*, *copy*, and *size*. Consider the NPB FT example in Fig. 3.1:

```
type range Range3D;
```

This statement declares a new domain type, *Range3D*, representing block regions of a 3D grid; this domain is defined in C++ as a structure with six elements that hold the start and end values for each $X, Y,$ and Z coordinate.

Datastructures CGD defines types for arbitrary distributed *datastructures* (Section 2.1). Datastructures are *decomposable* when their elements can be decomposed using a domain type; otherwise, datastructures are *non-decomposable*. Datastructure types are declared using the *type data* keywords followed by the datastructure type name and an optional list of compatible distribution types (Table 3.1). The latter are distribution types that can decompose the datastructure; when the list is empty, the datastructure is non-decomposable. Datastructures are defined by users in C++ by implementing a set of operators including *alloc*, *get*, *set*, *copy*, *to*, *from*, and *size*. The following example declares a decomposable 3D vector type (Fig. 3.1):

```
type data Vector3Dcplx [PartRange3D];
```

Here, the *Vector3Dcplx* type represents a 3D vector of complex numbers implemented in C++ using a template array provided by the runtime. The vector can be decomposed using *PartRange3D* distributions, which are assignments of *Range3D* domains to processes.

Distributions The language defines *distribution* or *partition* types, which represent an assignment of domains to processes (Section 2.2). Distribution types are declared using the *type partition* or *type mpartition* keywords followed by the domain type, distribution type, and a list of compatible distribution types; *distributions* assign at most one domain to each processor, while *multi-distributions* assign any number of domains

```
1  // DOMAIN Types
2  type range Range3D;
3
4  // DISTRIBUTION Types
5  type partition <Range3D> PartRange3D [PartNP];
6
7  // REDISTRIBUTION Types
8  type swap <Range3D> SwapRange3D [PartNP];
9
10 // DATASTRUCTURE Types
11 type data Setup, Checksums, Int, Bool, Cplx;
12 type data Vector3Dcplx [PartRange3D];
13 type data <Cplx> ArrayCplx [PartNP];
14
15 // Constant Datastructures
16 const Setup sp [ALL1];
17 const Int niter [ALL1], fdir [ALL1], finv [ALL1], lay1d [ALL1];
18 const PartRange3D dom0 [ALLn], dom1 [ALLn], dom2 [ALLn];
19 const SwapRange3D sw01 [PPEn], sw12 [PPEn], sw21 [PPEn];
20 const SwapRange3D sw10 [PPEn], sw02 [PPEn], sw20 [PPEn];
21
22 // DISTRIBUTION RULES
23 part sw01 : dom0 -> dom1;
24 part sw12 : dom1 -> dom2;
25 part sw21 : dom2 -> dom1;
26 part sw10 : dom1 -> dom0;
27 part sw02 : dom0 -> dom2;
28 part sw20 : dom2 -> dom0;
```

Figure 3.1: NPB FT type declarations, constants, and distribution rules

to each processor (Table 3.1). Programmers only declare distribution types in CGD; the distribution type implementation is automatically generated by the compiler. The distribution type used by the previous examples is defined in Fig. 3.1:

```
type partition <Range3D> PartRange3D [PartNP];
```

The *PartRange3D* type represents distributions assigning *Range3D* domains to processes; i.e., it represents arbitrary decompositions of 3D grids. The distribution itself is decomposed according to a predefined distribution, *PartNP,* described later in this section.

Redistributions The CGD language defines *redistribution* or *swap* types, which represent transformations between arbitrary datastructure distributions. Particu-

Table 3.2: Constant declaration syntax

Declaration	Syntax
Constant datastructure	**const**
local domain	DATATYPE CONST[CONSTDIST], ...,
global domain	DATATYPE CONST[CONSTDIST+], ...;

larly, redistribution distribution rules are defined in terms of redistribution datastructures (Section 2.1). Redistribution types are declared using the *type swap* or *type mswap* keywords followed by the domain type, redistribution type, and a list of compatible distribution types. Redistributions of distributions and multi-distributions transform the datastructures decomposed by distributions and multi-distributions, respectively (Table 3.1). Similar to distributions, redistribution types are declared in CGD, but their C++ implementation is generated automatically. Redistributions are represented as matrices of domains that are exchanged between processes to build a new datastructure decomposition (Section 3.4). The NPB FT example declares the redistribution types as follows (Fig. 3.1):

```
type swap <Range3D> SwapRange3D [PartNP];
```

Here, *SwapRange3D* is a matrix of *Range3D* domains, which are copied between processes to transpose, for example, a *Vector3Dcplx* vector from block-row to block-column organization.

3.2 Constants

The CGD language supports the definition of constant datastructures. Constants are declared by adding the *const* keyword before a list of type and datastructure distribution declarations (Table 3.2). Consider the example shown in Fig. 3.1:

```
const Setup sp [ALL1];
```

Table 3.3: Predefined distribution types

Distribution type	No of data elements	All assigned to single proc	All assigned to all procs	One element assigned per proc
PartOne	1	`ONE1`	`ALL1`	
PartNP	n	`ONEn`	`ALLn`	`PPEn`

Here, constant datastructure *sp* with type *Setup* is replicated among all processes according to distribution *ALL1*; the *sp[ALL1]* data node represents an implicit input to all dataflow functions (Section 3.5.3). Constant datastructures are set during the initialization phase before any dataflow functions are executed (Section 4.1).

3.3 Predefined Distributions

CGD defines predefined distributions to break the circular dependency between distribution data nodes and their own distribution. CGD data nodes are $\langle datastructure, distribution \rangle$ pairs, and are therefore always defined in terms of distributions. Distributions are datastructures assigning domains to processes; however, the entire assignment is not necessarily replicated among all processes, especially when the assignment changes at runtime. Subsequently, distribution datastructures are also distributed among processes according to a distribution. This circular dependency is solved by relying on predefined constant distributions that can partition distributions, redistributions, and non-decomposable datastructures.

PartOne Distributions with the predefined type *PartONE* assign one element to n processes; accordingly, non-decomposable datastructures are replicated among processes based on *PartONE* distributions (Table 3.3). Predefined distribution *ONE1* assigns the element to a single process, and distribution *ALL1* assigns the element to all processes. E.g., the constant integer *niter[ALL1]* from line 17 in Fig. 3.1 is replicated among all processes.

PartNP Distributions with the predefined type *PartNP* assign n element arrays to n processes (Table 3.3). Predefined distribution *ALLn* assigns all n elements to each process, and distribution *PPEn* assigns element i only to process p_i. Particularly, distribution assignments and redistribution matrices are decomposed using distributions with type *PartNP*. E.g., the domain assignment *dom0[ALLn]* is fully replicated among all processes, while redistribution matrix *sw01[PPEn]* is decomposed among processes by assigning the domains that are sent and received by p_i only to p_i (lines 18–19 in Fig. 3.1).

3.4 Distribution Rules

The CGD language supports distribution rules to the facilitate dataflow graph definition by reducing the number of datastructure transformations (Section 2.1). Distribution rules are applied to automatically add missing links between data nodes that have different distributions but identical datastructures. These rules are defined in the global and local dataflow function scopes (Section 3.5).

Distribution rules are generically defined in terms of distributions rather than datastructures; accordingly, they apply to all data nodes from the graph that use these distributions. This approach delegates some of the programmer's responsibilities: the programmer defines the rules once, and the compiler applies them to all relevant datastructures throughout the application. E.g., lines 23-28 in Fig. 3.1 defines the six rules needed by the entire NPB FT application to transpose 3D complex vectors between the block-X, Y, Z decompositions. The CGD language currently supports three rule types:

Inclusion The inclusion distribution rule states that distribution X is smaller than distribution Y, i.e., $(\cup X_{ij}) \subset (\cup Y_{ij})$ for $0 \leq i < n$. Subsequently, distribution Y of a

Table 3.4: Distribution rules syntax

Rule	Operation	Syntax
Inclusion	none	**part** DistX < DistY < DistZ ...;
Union	local data copy	**part** DistX = DistY + DistZ ...;
Redistribuion	local, remote data copy	**part** RedistS : DistX -> DistY;

datastructure A can be transformed to data node $A[X]$ without performing any operations. Inclusion rules are specified as an ordered sequence of distributions (Table 3.4).

Union The union distribution rule states that distribution X is the union of distributions Y, Z, ..., i.e., $\cup X_{ij} = (\cup Y_{ij}) \cup (\cup Z_{ij}) \cup \ldots$ for $0 \leq i < n$. Then, distribution X of a datastructure A can be produced by merging the local data domains from data nodes $A[Y]$ and $A[Z]$. The compiler tries to assign all $A[X]$, $A[Y]$, and $A[Z]$ data nodes to the same C++ datastructure (Section 4.1.3); when this is successful, the merge operation is avoided. Union rules are specified as a target distribution followed by a list of component distributions (Table 3.4).

Redistribution The redistribution distribution rule states that distribution X can be transformed into distribution Y by changing the assignment of subdomains according to redistribution matrix S. For each source domain X_i and target domain Y_j, S_{ij} specifies a common subdomain, if any, that has to be moved from X_i to Y_j to create the target distribution Y. Subsequently, a data node $A[X]$ can be transformed into data node $A[Y]$ using redistribution matrix S. Similar to the union rule, redistributions avoid copying the intersection between the source and target domains when both data nodes are assigned to the same C++ datastructure. E.g. the stencil computation from Section 3.4 copies only the "halo" elements instead of the entire "halo" domains during the redistribution or "halo exchange" step. Redistribution rules are specified as a redistribution, followed by the source and target distributions (Table 3.4).

Redistribution matrices are defined by the user; however, they can be computed using

Table 3.5: Dataflow and SPMD function syntax

Declaration	Syntax
Dataflow function	**function** FUNCNAME (INARG, ... -> OUTARG, ...)
Argument list	
decomposable	DATATYPE ARG [DISTARG], ...,
non-decomposable	DATATYPE ARG, ...,
global domain	DATATYPE ARG [DISTARG+], ...
Dataflow graph	{ DATAFLOWGRAPH }
SPMD function	**function** FUNCNAME (INARG, ... -> OUTARG, ...)
Argument list	
decomposable	DATATYPE ARG [DOMARG], ...,
non-decomposable	DATATYPE ARG, ...,
in-out	DATATYPE ARG* [DOMARG], ...,
global domain	DATATYPE ARG [DOMARG+], ...;

a predefined function when the source and target distributions are fully replicated, i.e., both distributions have the *ALLn* distribution. This function can be used when the intersection operator is defined by the distribution domain type.

3.5 Dataflow and SPMD Functions

The CGD language provides dataflow and SPMD functions, which represent the subgraph and SPMD computation nodes from the CGD graph (Section 2.2). The input and output arguments of these functions are data nodes representing the predecessors and successors of the computation node, respectively. Both dataflow and SPMD functions are declared in the CGD language. Dataflow functions are implemented in CGD by specifying a dataflow graph after the function declaration. SPMD functions are implemented in C++ as sequential functions taking domains of the input datastructures as input and producing domains of the output datastructures as output.

Declaration Dataflow functions are declared first by specifying the list of input and output arguments, then by defining the type and distribution of each argument (Table 3.5). The distribution corresponding to each argument can be a constant distribution, or another argument passed to the function. Input distribution arguments can be assigned to both input and output arguments, whereas output distributions are assigned only to output arguments. Moreover, the argument type and its distribution type have to be compatible; the datastructure type declaration specifies a list of compatible distribution types (Table 3.1). As a simple example, consider the declaration of function *fft* from Fig. 3.2:

```
function fft (A -> D)
   Vector3Dcplx A[dom0], D[dom2]
{ ... }
```

Here, function *fft* takes as input argument *A* with constant distribution *dom0* (*XY* planes), and produces as output argument *D* with constant distribution *dom2* (*XZ* planes). Both arguments are 3D vectors with type *Vector3Dcplx*, which can be partitioned by 3D decompositions with type *PartRange3D* (Fig. 3.1 line 12); hence, distributions *dom0* and *dom2* with type *PartRange3D* are compatible with datastructures *A* and *B*.

Similar to dataflow functions, SPMD functions are declared by specifying a list of arguments, followed by the type and domain of each argument (Table 3.5). The argument type and its domain type have to be compatible, i.e., the domain type has to be used by a distribution type compatible with the argument type. Fig. 3.2 illustrates such an example:

```
function cffts1 (sp, dir, A, dom -> A*)
   Setup sp, Int dir, Range3D dom,
   Vector3Dcplx A[dom], A*[dom];
```

Function *cffts1* takes as input arguments *sp*, *dir*, and the *dom* domain of argument *A*, and produces as output the same *dom* domain of argument *A**. *Vector3Dcplx* is compatible with distribution type *PartRange3D*, which uses domain type *Range3D*; hence, the 3D *dom* domain is compatible with 3D vector *A*. This declaration contains non-decomposable datastructures *sp* and *dir,* and in-out arguments *A* and *A**, which are addressed later in this section.

Both dataflow and SMD function declarations require all arguments to have explicit types. This requirement is needed by the compiler to infer all of the data node types from the function argument types (Section 4.1.2).

Non-Decomposable Arguments The declaration of an argument requires no explicit distribution or domain when the argument is a non-decomposable datastructure (Table 3.5). However, when the function is invoked, the data node corresponding to the non-decomposable datastructure argument requires a distribution with the *PartOne* type (Table 3.3). The *ALL1* distribution is used by default when a data node does not specify a distribution; this feature makes the dataflow graph more readable, replicating the scalars to all processes by default. Fig. 3.2 shows the declaration and invocation of function *cffts1*:

```
function cffts1 (sp, dir, A, dom -> A*)
   Setup sp, Int dir, ... ;
...
cffts1 (sp, fdir, A[dom0] -> B[dom0]);
```

When SPMD function *cffts1* is executed, the computation assigned to process p_i takes as arguments a copy of non-decomposable datastructures sp and dir, and the $dom0_i$ domain of datastructures A and B.

In-out Arguments SPMD functions support in-out arguments that are declared by specifying the same argument name as both an input and an output, and by adding

Table 3.6: Local and global data domain access examples

Domain Access	Runtime Operation	Example
All arguments local	none	`f( A[ X ] -> B[ Y ]);`
Input global, output local	remote reads, cached	`f( A[ X+ ] -> B[ Y ]);`
Input local, output global	remote writes, buffered	`f( A[ X ] -> B[ Y+ ]);`

an asterisk sign after the output argument name to differentiate between the two. An in-out argument can have different input and output distributions. A simple example is the *cffts2* function from the NPB FT benchmark (Fig. 3.2):

```
function cffts2 (sp, dir, A, dom -> A*)
   Vector3Dcplx A[dom], A*[dom], ... ;
...
cffts2 (sp, fdir, B[dom1] -> C[dom1]);
```

Here, the *cffts2* computation node takes $B[dom1]$ as an input and produces $C[dom1]$ as an output. The function declaration maps both data nodes to the same in-out argument $A[dom]$ which is *read* and *written* by the C++ implementation of *cffts2*.

A dataflow graph that contains a computation node with an in-out argument uses one input data node to represent the data before being modified, and one output data node to represent the data after modification. After the graph is ordered, the compiler replicates the input data node when required by a computation scheduled at a later time.

3.5.1 Local and Global Domain Access

Local Domain Access Dataflow and SPMD functions have arguments that support local domain access by default; when this feature is active, the corresponding sequential functions executed by a process can only access datastructure domains assigned to that particular process. In Fig 3.8, the CGD Barnes-Hut implementation uses local

domain access to compute the center of mass for the bottom of the tree nodes:

```
function ctrmassb (body, bot -> node)
   BodyPM body[bot], NodePM node[bot], Tree bot;
...
ctrmassb (body_pm [tree_bot] -> node_pm [tree_bot]);
```

Here, all $ctrmassb$ computations assigned to p_i access only the local data elements from $bodypm[treebot_i]$ and $nodepm[treebot_i]$ that are assigned to p_i.

We recall from (2.11) that distribution domains are not always disjoint, and allow data replication among processes. If local distribution arguments provide the benefits of replication and locality, they require knowing and defining data dependencies beforehand; e.g., programmers have to define which domain $treebot_i$ of datastructure $bodypm$ is needed by computation $ctrmassb$ to produce $nodepm[treebot_i]$. A good example is the optimized stencil benchmark from Section 2.3.1, which uses local domain access to considerably reduce communication cost by exploiting both locality and replication (Section 4.10).

Global Domain Access Dataflow and SPMD functions can have arguments that support global domain access; when this feature is enabled, the corresponding sequential functions executed by a process can access data domains assigned to any process. Global domain access is enabled by adding a plus sign after the distribution name when declaring a function, and when calling it from a dataflow graph (Table 3.5). The Barnes-Hut implementation uses global domain access to compute the center of mass for the top of the tree nodes (Fig 3.8):

```
function ctrmasst (node, top, bot -> node2)
   NodePM node[bot+], node2[top], Tree top, bot;
...
ctrmasst (node_pm [tree_bot+] -> node_pm2 [tree_top]);
```

The $ctrmasst$ computations assigned to p_i access any tree nodes from $nodepm$ to compute $nodepm2[treetop_i]$; if any nodes can be accessed globally, $nodepm[treebot_i]$ nodes are accessed locally. Table 3.6 summarizes the common use cases for both local and global domain access.

Several restrictions apply to functions using global arguments: global arguments cannot be in-out arguments, and datastructures using global arguments support remote element access via *set* and *get* operators implemented based on runtime primitives.

Global arguments provide greater access flexibility, and do not require knowing all data elements needed by a computation beforehand. This feature can prove useful when implementing irregular applications such as the Barnes-Hut N-body simulation presented in Section 2.3.3.

Discussion The CGD language provides both a faster local mechanism and a slower global mechanism for accessing distributed datastructures. The best solution depends on the application type and implementation complexity. Applications such as PDE solvers and other structured scientific codes can generally rely on local distribution arguments and redistribution rules to avoid using global domain access altogether. Applications such as N-body simulations and other irregular codes benefit from global domain access: they can access any data elements while maintaining fast access to local data domains. While accessing local elements is always faster than accessing remote elements, the remote access overhead highly depends on machine architecture and runtime performance.

CGD dataflow semantics allows several remote datastructure access optimizations. *Read-only* arguments benefit from local memory caching, since these datastructures are not modified until the computation is completed. On ccNUMA machines, this optimization improves the efficiency of globally shared datastructures when data sets are large, and the cumulative size of the remote reads exceeds the single-processor cache

size (Section 4.2.6). Similarly, *write-only* argument access is optimized by buffering writes and committing them in batches, since the write order is irrelevant. This solution leads to a higher write throughput by avoiding fine-grain message latency and ping-pong write-sharing pitfalls.

3.5.2 SPMD Functions

SPMD functions represent the SPMD computation nodes from the CGD graph, and they take input and output $\langle datastructure, distribution \rangle$ arguments (Section 2.2). These functions are declared in the CGD language but are implemented as a sequential C++ functions. When SPMD function f is executed in parallel, each process p_i executes sequential function f for each tuple of datastructure domains assigned to p_i by the distribution of each argument. The following example shows the declaration and invocation of function *cffts2* used by NPB FT to compute Y-wise FFTs (Fig. 3.2):

```
function cffts2 (sp, dir, A, dom -> A*)
   Setup sp, Int dir, Range3D dom,
   Vector3Dcplx A[dom], A*[dom];
...
cffts2 <ALL1> (sp[ALL1], fdir[ALL1], B[dom0] -> C[dom0]);
```

When this computation node is executed, the *cffts2* sequential computation assigned to process p_i takes as input the *sp* and *fdir* constants, as well as the $dom0_i$ domain (*XY* planes) of 3D vector B, and produces as output the $dom0_i$ domain of 3D vector C.

For simplicity's sake, distributions can be mixed with multi-distributions when calling an SPMD function (Table 3.1). All multi-distributions assign the same number of domains to each process, and all distributions assign a single domain to each process. Process p_i executes function f for each domain assigned to p_i by multi-distribution arguments; the single domain assigned to p_i by distribution arguments is used for all function calls. For example, during the execution of SPMD computation node f, which

```
1  // SPMD Function Declaration
2
3  function cffts1 (sp, dir, A, dom -> A*)                // X-WISE FFT
4    Setup sp, Int dir, Range3D dom, Vector3Dcplx A[dom], A*[dom];
5
6  function evolve (sp, idx, u0, dom -> u1)               // FORWARD TIME
7    Setup sp, Int idx, Range3D dom,
8    Vector3Dcplx u0[dom], u1[dom];
9
10 // DATAFLOW Function Definition
11
12 function mainLoop ( -> success)                        // MAIN
13   Bool success[ALL1]
14 {
15   init_checksums (sp -> cksum);                        // Initialization
16   compute_init_cond (sp -> u0[dom0]);
17
18   fft (u0[dom0] -> u1[dom2]);                          // Direct transform
19
20   loop (iter, cond; cksum[ONE1] -> cksum2[ONE1])       // Main loop
21   {
22     evolve (sp, iter, u1[dom2] -> u2[dom2]);           // Time forward
23
24     ifft (u2[dom2] -> u3[dom0]);                       // Inverse transform
25
26     checksum (iter, u3[dom0], cksum -> cksum2);        // Global checksum
27     econd (niter, iter -> cond);                       // End condition
28   }
29   verify <ONE1> (sp, cksum2[ONE1] -> success[ONE1]);
30 }
31
32 function fft (A -> D)                                  // DIRECT FFT
33   Vector3Dcplx A[dom0], D[dom2]
34 {
35   cffts1 (sp, fdir, A[dom0] -> B[dom0]);               // X-wise FFT
36
37   if (lay1d, B[dom0] -> C[dom2]) {                     // 1D layout
38     cffts2 (sp, fdir, B[dom0] -> C[dom0]);             // Y-wise FFT
39
40   } else {                                             // 2D layout
41     cffts2 (sp, fdir, B[dom1] -> C[dom1]);             // Y-wise FFT
42   }
43   cffts3 (sp, fdir, C[dom2] -> D[dom2]);               // Z-wise FFT
44 }
45
46 function checksum (iter,x,cksum, dom -> cksum2)        // GLOBAL CHECKSUM
47   Int iter[ALL1], PartRange3D dom[ALLn],
48   Vector3Dcplx x[dom], Checksums cksum[ONE1], cksum2[ONE1]
49 {
50   checksum_dom <ALL1> (x[dom] -> c1[PPEn]);
51   sumproc      <ONE1> (c1[ONEn] -> c2[ONE1]);
52   checksum_set <ONE1> (sp,c2[ONE1],iter,cksum[ONE1] -> cksum2[ONE1]);
53 }
```

Figure 3.2: NPB FT dataflow and SPMD functions

takes data nodes $A[X]$ and $B[Y]$ as inputs and produces data node $C[Z]$ as an output, process p_i executes the following sequential computations:

$$C[Z_{ij}] \leftarrow f(A[X_i], B[Y_{ij}]) \quad \text{for } 0 \leq j < |Z_i| \tag{3.1}$$

where X is a distribution, Y and Z are multi-distributions, and $|X_i| = 1$, $|Y_i| = |Z_i|$ for all $0 \leq i < n$; the same $A[X_i]$ argument is used when calling f for each $B[Y_{ij}]$ argument. A similar example is the *cffts2* function invocation from the optimized NPB FT "slab" implementation (Fig. 4.2):

```
cffts2 <mdom1> (sp[ALL1], fdir[ALL1], B[mdom1] -> C[mdom1]);
```

Here, each process p_i executes $|mdom1_i|$ sequential functions that take as arguments the "slab" domains $mdom1_{ij}$ of datastructures B and C, and the same $sp[ALL1_i]$ and $fdir[ALL1_i]$ constants.

3.5.3 Dataflow Functions

Dataflow functions represent the subgraph computation nodes from the CGD graph (Section 2.2). A dataflow function takes input and output data node arguments and embeds a dataflow graph that produces the output arguments based on the input arguments. A dataflow function definition consists of a function declaration and dataflow graph (Table 3.5).

The body of dataflow functions is specified as a sequence of computation nodes that take input and output data node arguments. Computation nodes are iterative constructs, conditional constructs, SPMD function invocations, and dataflow function invocations (Table 3.7). The order of these computations is irrelevant since they represent graph nodes, and the final execution order is generated by topologically ordering the CGD graph.

For example, dataflow function *fft* from line 32 in Fig. 3.2 computes the direct FFT

Table 3.7: Computation node and dataflow graph syntax

Construct	Syntax
Function invocation	FUNCNAME <TASK> (INARG[DIST], ... -> OUTARG[DIST], ...);
Conditional construct eval if COND eval unless COND	**if** (COND; INARG[DIST], ... -> OUTARG[DIST], ...) { TRUEDATAFLOWGRAPH } **else** { FALSEDATAFLOWGRAPH }
Iterative construct eval while COND	**loop** (IDX, COND; INARG[DIST], ... -> OUTARG[DIST], ...) { DATAFLOWGRAPH }
Dataflow graph computation node sequence	FUNCTIONINVOCATION; ... ; ITERATIVECONSTRUCT; ... ; CONDITIONALCONSTRUCT; ... ;

transform of a 3D vector; it includes three SPMD functions that compute X-wise, Y-wise, and Z-wise FFTs on local domains, and it employs a conditional construct to choose between the 1D and 2D decomposition layouts. When the 1D layout is enabled, the compiler adds a redistribution operation that transposes the 3D vector between the *dom0* (XY-plane) and *dom2 (XZ*-plane) decompositions using the *sw02* redistribution rule from line 27 in Fig. 3.1 (Section 4.1.3).

Function Invocation SPMD function invocations represent the SPMD computation nodes from the CGD graph; similarly, dataflow function invocations represent the subgraph computation nodes from the CGD graph (Section 2.2). A function invocation takes input and output data node arguments, and an optional task distribution argument specified prior to the argument list (Table 3.7). Let us consider the example in Section 3.5.2, where SPMD function *cffts2* takes multi-distribution data node arguments:

```
cffts2 <mdom1> (sp, fdir, B[mdom1] -> C[mdom1]);
```

The explicit task argument $<mdom1>$ ensures that exactly $|mdom1_i|$ computations are assigned to each process p_i.

When the task distribution is not supplied, the *ALL1* distribution is used instead; this means that, by default, a single computation is executed by each process. When a computation node takes data node arguments that do not specify a distribution, the *PPEn* distribution is used by default for datastructure arguments with distribution types, and the *ALL1* distribution is used by default for all other types; e.g., in the previous example, arguments *sp* and *fdir* are assigned the default distribution *ALL1*.

Conditional Constructs Conditional constructs correspond to conditional computation nodes in the CGD dataflow graph (Section 2.2). Being computation nodes, conditional constructs take input and output datastructure distribution arguments; however, they have an additional *condition* argument and embed two dataflow subgraphs (Table 3.7). One subgraph is evaluated when the condition argument is true, while the other subgraph is evaluated when the condition argument is false.

While both subgraphs produce the same output data nodes, they might depend on different input data nodes. The conditional computation node depends on all data nodes required by either subgraph. When the two subgraphs require different distributions for the same datastructure, an explicit distribution is specified through an input data node. This rule avoids ambiguity, ensuring that the distribution of an input datastructure is known before the condition is evaluated, and thus, data transformations and communication can be scheduled in advance.

The following example shows how dataflow function *fft* from lines 37–43 in Fig. 3.2 computes the Y-wise FFTs using either the 1D or 2D decomposition layouts:

```
if (lay1d, B[dom0] -> C[dom2]) {
   cffts2 (sp, fdir, B[dom0] -> C[dom0]);
} else {
   cffts2 (sp, fdir, B[dom1] -> C[dom1]);
}
```

This code produces the same $C[dom2]$ output regardless of which branch is evaluated; $lay1d$ determines whether the $dom0$ or $dom1$ distribution is used to compute the local FFTs; and $B[dom0]$ is provided as an explicit conditional input since the branch subgraphs take both $B[dom0]$ and $B[dom1]$ as inputs.

Iterative Constructs Iterative constructs represent the iterative computation nodes from the CGD graph (Section 2.2). Similar to other computation nodes, iterative constructs take input and output datastructure distribution arguments; however, they have additional *end condition* and *index* arguments, and include a dataflow subgraph (Table 3.7). Iterative constructs are evaluated by executing the embedded subgraph for multiple iterations until the end condition is satisfied.

The subgraph evaluation produces the subgraph outputs based on its inputs; the subgraph inputs are the loop inputs, and arbitrary data nodes from the parent graphs (Fig. 2.4b). However, only the outputs of the subgraph are assigned to its inputs when the loop evaluation continues; an assignment $A[X] \leftarrow B[Y]$ requires that A and B have the same type, and distribution X can be obtained from distribution Y based on distribution rules. The compiler tries to assign corresponding loop inputs and outputs to the same C++ datastructure to avoid local copy operations (Section 4.1.3).

The dataflow subgraph includes two special data nodes that are not visible to the parent graph: i) the index data node represents an iteration count and is provided as an input with no matching output; and ii) the end condition data node is produced as an output with no matching input. The index input is only provided for convenience and can be ignored if not needed.

Consider the main loop from the stencil example in Fig. 3.4, which updates all of the grid values during each iteration:

```
loop ( idx, cond ; A [cell] -> C [cell] ) {
   ftcs_step ( cfg, A[halo] -> B[cell] );
```

```
        ftcs_step ( cfg, B[halo] -> C[cell] );
        end_cond ( cfg, idx -> cond );
    }
```

Each loop iteration produces the $cell$ domain of grid C after applying two update steps to the larger $halo$ domains of grid A, followed by grid B. Here, the loop subgraph has inputs $A[cell]$ and $idx[ALL1]$ provided by the iterative construct, and $cfg[ALL1]$ provided by the parent graph. If $cond[ALL1]$ is false after the subgraph is evaluated, $C[cell]$ is assigned to $A[cell]$ and the process is repeated; otherwise, $C[cell]$ is produced as an output of the iterative construct computation node.

3.6 Examples

This section presents the implementation of the three examples introduced in Section 2.3: i) the stencil computation and two optimizations that aggregate communication steps and overlap communication and computation; ii) the NPB FT benchmark and its "slab" optimization similarly overlapping communication and computation; and iii) the SPLASH2 Barnes-Hut hierarchical N-body simulation.

These examples are first explored by analyzing relevant types and dataflow function structure. We then showcase interesting language features exercised by selected code fragments, and finally present a few high-level optimizations and discuss their implementation effort and performance impact.

3.6.1 Stencil Computation

This section illustrates the CGD implementation of the stencil kernel introduced in Section 2.3.1. The application types and basic algorithm dataflow are outlined in Fig. 3.3; additionally, two optimizations aggregating communication steps and overlapping computation with communication are shown in Figs. 3.4 and 3.5, respectively.

```
1  // DOMAIN Types
2  type range Range2D;
3  type partition <Range2D> PartRange2D [PartNP];
4  type mpartition <Range2D> MPartRange2D [PartNP];
5  type swap <Range2D> SwapRange2D [PartNP];
6
7  // DATASTRUCTURE Types
8  type data Vector2Dreal [PartRange2D, MPartRange2D];
9  type data Int, Config;
10
11 // CONSTANT Datastructures
12 const PartRange2D cell[ALLn], halom[ALLn], halo[ALLn];
13 const MPartRange2D boundary[ALLn];
14 const SwapRange2D sw_chm[PPEn];
15 const Config cfg[ALL1];
16
17 // DISTRIBUTION RULES
18 part cell < halom < halo;
19 part boundary < halo;
20 part halo = halom + boundary;
21 part sw_chm : cell -> halom;
22
23 // DATAFLOW Functions
24 function mainLoop ()
25 {
26   // Initialize: A, BD
27   // Main loop: two updates, two redistributions
28   loop ( idx, cond ; A [cell] -> C [cell] )
29   {
30     copy ( BD[boundary] -> A[boundary] );
31     copy ( BD[boundary] -> B[boundary] );
32     ftcs_step ( cfg, A[halo] -> B[cell] );
33     ftcs_step ( cfg, B[halo] -> C[cell] );
34     end_cond (cfg, idx -> cond);
35   }
36   // Produce results: C
37 }
```

Figure 3.3: Stencil computation types and dataflow

The stencil kernel types are typical for applications working with grid decompositions: i) *Range2D* declares a 2D block region domain type; ii) *PartRange2D* and *MPartRange2D* define distributions and multi-distributions assigning *Range2D* domains to processes; iii) *SwapRange2D* defines redistributions between 2D block decompositions; and iv) *Vector2Dreal* declares 2D arrays of real numbers that allow *PartRange3D* and *MPartRange3D* type distributions. The *cell* and *halo* distributions are 2D block decompositions, as depicted in Fig. 2.5. While *cell* has disjoint domains, *halo* repli-

cates the "halo" elements to neighbor processes; the *boundary* distribution contains the enclosing boundary elements, and the *halom* distribution represents the *halo* domains excluding the boundary elements. Accordingly, the distribution rule $halo = halom + boundary$ describes how boundary elements are added to create the *halo* distribution, and the sw_chm redistribution represents the "halo exchange" characteristic of any stencil computation.

The loop computation from line 28 in Fig. 3.3 takes as input the system state in grid *A[cell]*, advances the system by two time steps, and produces as output the system state in grid *C[cell]*; the iteration is complete after a predefined number of time steps. Each *ftcs_step* computation requires the larger *halo* domains and produces the smaller *cell* domains. Consequently, each iteration executes two $cell \rightarrow halom$ redistributions along with two computation steps. The compiler moves the computations that copy the boundary elements outside the loop, to avoid unnecessary work; such optimizations are possible when computations do not depend on the loop inputs (Section 4.1.3).

The stencil kernel communication can be optimized by replicating computation and aggregating communication steps (Section 2.3.1). The "merged step" optimization computes the first *ftcs_step* update on larger *halo2* domains and, produces *halom* rather than *cell* domains (line 20 in Fig. 3.4). Subsequently, the two $cell \rightarrow halom$ redistributions from each iteration are replaced by a single $cell \rightarrow halom2$ redistribution. This optimization avoids the constant overhead of one extra communication step, trading computation for latency.

The "merged step" optimization can be added to the basic stencil implementation with a few easy steps (Fig. 3.3 vs. Fig. 3.4): i) new distributions are defined for *halo2*, *halom2*, and *boundary2* to represent larger domains; ii) the distribution inclusion graph is augmented with new distribution rules; iii) the *sw_chm* redistribution is changed accordingly; and iv) the main loop data nodes are changed to use the larger domains.

```
1  // DISTRIBUTION RULES
2  part cell < halom < halo < halo2;
3  part cell < halom2 < halo2;
4  part boundary < boundary2;
5  part boundary < halo;
6  part boundary2 < halo2;
7  part halo = boundary + halom;
8  part halo2 = boundary2 + halom2;
9  part sw_chm : cell -> halom2;
10
11 // DATAFLOW Functions
12 function mainLoop ()
13 {
14   // Initialize: A, BD
15   // Main loop: two updates, one redistribution
16   loop (idx, cond ; A [cell] -> C [cell])
17   {
18     copy (BD [boundary2] -> A [boundary2]);  // larger domain
19     copy (BD [boundary]  -> B [boundary]);
20     ftcs_step (cfg, A [halo2] -> B [halom]);  // larger domain
21     ftcs_step (cfg, B [halo]  -> C [cell]);
22     end_cond (cfg, idx -> cond);
23   }
24   // Produce results: C
25 }
```

Figure 3.4: Stencil computation aggregating two communication steps

```
1  // DISTRIBUTION RULES
2  part cell2 < cell < halom < halo;
3  part border < cell;
4  part boundary < halo;
5  part halo = boundary + halom;
6  part cell = cell2 + border;
7  part sw_chm : cell -> halom;
8
9  // DATAFLOW Functions
10 function mainLoop ()
11 {
12   // Initialize: A, BD
13   // Main loop: two updates, two overlapped redistributions
14   loop (idx, cond ; A [cell] -> C [cell])
15   {
16     copy (BD[boundary] -> A[boundary]);
17     copy (BD[boundary] -> B[boundary]);
18     ftcs_step (cfg, A[cell] -> B[cell2]);  // overlapped
19     ftcs_step (cfg, A[halo] -> B[border]);
20     ftcs_step (cfg, B[cell] -> C[cell2]);  // overlapped
21     ftcs_step (cfg, B[halo] -> C[border]);
22     end_cond (cfg, idx -> cond);
23   }
24   // Produce results: C
25 }
```

Figure 3.5: Stencil computation exploiting communication overlap

Although the stencil kernel is computationally simple, this optimization is generally applicable to PDE codes that exhibit similar computation-communication patterns, i.e., "domain update" computations followed by "halo exchanges". Let us consider the process of adding the "merged step" optimization to a large scale application, which includes PDE solvers developed by several scientists. In CGD, the optimization requires adding distributions and distribution rules (steps i-iii above), and trivially modifying dataflow distributions. While the distribution rules are redefined once in the top-level section, they allow the optimization of all "halo exchanges" throughout the entire application. On the other hand, in message passing, this optimization requires reorganizing the computation loops, data buffers, send and receive calls, and synchronization primitives in each code section that employs a PDE solver.

Another stencil kernel optimization improves performance by overlapping the "halo exchange" communication with the "domain update" computation (Fig. 3.5). Particularly, each *ftcs_step* computation is split into a computation acting on the *cell* domains that does not depend on the *halo-cell* elements, and the computation of the remaining elements, which requires the *halo* elements. The compiler overlaps the first computation with the redistribution operation, since the computation does not depend on the redistribution. Similar to the "merged step" optimization, implementing the stencil overlap optimization only requires new distributions and distribution rules to be added, and the main loop dataflow to be slightly altered.

3.6.2 NPB FT

This section presents the CGD implementation of the NPB FT benchmark first introduced in Section 2.3.2. The application types and distribution rules were presented earlier in this chapter in Fig. 3.1, and a few relevant dataflow functions were shown in Fig. 3.2. These types are typical for an application working with distributed vectors: i) *Range3D* declares a 3D region domain type; ii) *PartRange3D* and *SwapRange3D* define

```
1  function fft (A -> D)
2    Vector3Dcplx A[dom0], D[dom2]
3  {
4    cffts1 <ALL1>  (sp, fdir, A[dom0] -> B[dom0]);
5    if (lay1d, B[dom0] -> D[dom2])
6    {                                                          // 1D layout
7      cffts2 <ALL1>  (sp, fdir, B[dom0]  -> C[dom0]);
8      cffts3 <mdom2> (sp, fdir, C[mdom2] -> D[mdom2]);         // overlapped
9    }
10   else
11   {                                                          // 2D layout
12     cffts2 <mdom1> (sp, fdir, B[mdom1] -> C[mdom1]);         // overlapped
13     cffts3 <mdom2> (sp, fdir, C[mdom2] -> D[mdom2]);         // overlapped
14   }
15 }
```

Figure 3.6: NPB FT "slab" optimization

distributions and redistributions based on *Range3D*; and iii) *Vector3Dcplx* declares 3D vectors of complex numbers allowing distributions of type *PartRange3D*. Distributions *dom0*, *dom1*, and *dom2* are the *X*-, *Y*-, and *Z*-block decompositions used by NPB FT, whereas *sw01*, *sw12*, etc. are the redistribution rules transforming these distributions.

The main loop computation takes one checksum as an input and produces another as an output (line 20 in Fig. 3.2); during each iteration it executes a time-forward step, then computes the inverse Fourier transform, and finally incorporates the result into checksum *cksum[ONE1].* The *checksum* function first computes the local domain checksums, and then aggregates them globally; *n* local checksums are produced by all processes in *c1[PPEn]*, which is later aggregated by a single process from input *c1[ONEn]* to output *c2[ONE1]* (lines 50–51). Array transformations between the predefined distributions *ONE1*, *ONEn*, and *PPEn* are automatically supported by predefined redistribution rules (Section 3.3).

The *fft* dataflow function calculates the direct Fourier transform of a 3D complex vector (line 32 in Fig. 3.2). The parallel 3D FFT is computed by executing *X-, Y-*, and *Z*-wise uni-dimensional sequential transforms that take local vector domains as arguments. Subsequently, the corresponding *cffts1*, *cffts2*, and *cffts3* SPMD functions take

arguments with *dom0*, *dom1*, and *dom2* distributions that assign an entire *X*, *Y*, and *Z* coordinate to a single process.

Vector distributions are chosen to optimize communication according to the original NPB algorithm: the 1D layout is used when the number of processors is less than the *Z*-dimension; otherwise, the 2D layout is chosen (Section 2.3.2). The conditional construct from line 37 computes the *Y*-wise FFT starting from vector decomposition *B[dom0]*, and producing vector decomposition *C[dom2]*: i) for 1D layouts the *Y*-wise FFT is executed on the same *dom0* decomposition as the *X*-wise FFT since *dom0* contains *XY*-planes; ii) for 2D layouts the *Y*-wise FFT is executed on the *dom1* decomposition containing *Y*-block columns, since *dom0* contains *X*-block rows and it is no longer compatible with *Y*-wise FFTs. The compiler uses distribution rules to automatically add one transpose, $dom0 \rightarrow dom2$, for the 1D layout, and two transposes, $dom0 \rightarrow dom1$ and $dom1 \rightarrow dom2$, for the 2D layout (line 23 in Fig. 3.1).

The original NPB FT algorithm can be optimized by overlapping communication with computation (Section 2.3.2). The "overlap slab" optimization works by splitting each vector domain into smaller domains called slabs, and assigning multiple FFT computations acting on these smaller domains to each process; overlap is achieved by executing the local FFTs on a given slab, while at the same time transferring data needed to compute the FFTs on the next slab (Fig 2.6).

Adding the "overlap slab" optimization to the CGD NPB FT implementation requires a few simple changes (Fig. 3.2 vs. Fig. 3.6): i) new multi-distributions *mdom0*, *mdom1*, and *mdom2* are defined to represent slab decompositions; ii) distribution rules are changed to allow multi-distributions; and iii) *fft* and *ifft* dataflow function data nodes are changed to use multi-distributions when invoking *Y*-wise and *Z*-wise FFT computations. The *cffts2* and *cffts3* SPMD computations are then executed on vector multi-distributions, resulting in communication-computation overlap and a better cache behavior. Section 2.3.2 describes in more detail how this optimization exploits communi-

Table 3.8: NPB FT programming effort and scalability comparison

Version	Layout	Language	Line count				Speedup
Serial	0D	Fortran				704	1.00
OpenMP	1D	C				752	17.21
MPI	1D, 2D	Fortran				1261	99.12
			C++	CGD func.	CGD decl.	Total	
CGD	1D, 2D	C++, CGD	703	45	38	786	111.46
CGD slab	1D, 2D slabs	C++, CGD	740	47	43	830	125.72

cation overlap.

The parallel implementation and optimization of NPB FT requires less effort in CGD compared to message passing models such as MPI; moreover, the CGD performance matches or exceeds the MPI performance (Section 4.2.2). Table 3.8 presents the amount of code measured as a SLOC count for five NPB FT implementations: sequential Fortran, MPI Fortran, OpenMP, CGD, and CGD "overlap slab". The Fortran versions are the original NPB implementations, the OpenMP version was derived from the Fortran version by the Omni compiler project, and the CGD versions were derived from the OpenMP version. Each of these implementations shares a common base of about 700 lines of code implementing sequential functions that were first translated by Omni from Fortran to C, and then incorporated in a virtually unchanged state into CGD. The table reveals that the parallel implementation of the original sequential algorithm adds about 50 lines of OpenMP code, 100 lines of CGD code, and 500 lines of Fortran MPI code. The MPI implementation requires a significantly higher effort without achieving a performance improvement over CGD, and the OpenMP implementation lacks 2D layout support and shows poor scalability on large CC-SAS machines (Table 3.8, Section 4.2.2).

The "overlap slab" NPB FT implementation requires minimal dataflow and sequential code changes in CGD; however, it does require a non-trivial remanufacturing of the message passing code. In CGD, the optimized and unoptimized versions have roughly

the same SLOC count; adding the optimization changes about 15 lines of CGD code and 30 lines of C++ code (Table 3.8). These modifications represent localized dataflow changes in CGD, and add straightforward sequential code for computing new distributions in C++. On the other hand, an MPI implementation of the same algorithm requires adding—on top of 500 lines of existing parallel code—new code that allocates vectors and buffers, issues multiple send and receive operations, changes the loop structure, and reorders computations to allow overlap. Subsequently, the CGD optimization process for NPB FT—which relies on automatic issue of communication and synchronization operations, and automatic computation ordering and overlap—requires significantly less effort than manually remanufacturing functionally equivalent message passing code.

3.6.3 Barnes-Hut N-Body Simulation

This section analyzes the CGD implementation of the Barnes-Hut algorithm presented in Section 2.3.3. The application types and the main loop dataflow are presented in Fig. 3.7, and two aggregation functions are presented in Fig. 3.8. The *Tree* and *List* domain types represent octotree regions, and *TreePart* and *ListPart* are their corresponding distribution types. A *ListPart* distribution assigns lists of particles to processes, while *TreePart* distributions assign subtree domains to processes; subtrees can be traversed to enumerate their particles. List distributions do not expose the tree structure, but they do describe particle properties such as position and mass (*BodyPM*), and acceleration (*BodyAccel*). On the other hand, tree distributions maintain the tree context, and describe cell and leaf node properties such as position and mass (*NodePM*), and the computation weight (*NodeScore*). Note that the *BodyPM* datastructure type allows both *TreePart* and *ListPart* distributions (line 7 in Fig. 3.7).

The tree domain, list domain, and datastructure types are opaque to CGD, as they have been defined specifically for this application. Although the compiler has no un-

```
1  // DOMAIN Types
2  type range Tree, List;
3  type partition <Tree> TreePart [PartNP];
4  type partition <List> ListPart [PartNP];
5
6  // DATASTRUCTURE Types
7  type data BodyPM [TreePart, ListPart];
8  type data BodyScore [TreePart, ListPart], BodyAccel [ListPart];
9  type data NodePM [TreePart], NodeScore [TreePart];
10 type data <Box> BoxArray [PartNP];
11 type data <Stats> StatsArray [PartNP];
12 type data Int, Bool, File, Vector, Box, Params, Stats;
13
14 function mainLoop ( pari -> stats )
15   Params pari[ALL1], StatsArray stats[PPEn]
16 {
17   // DISTRIBUTION RULES
18   part list = tree_loc;
19   part list0 = tree_loc0;
20
21   // Compute initial tree and partition
22
23   // Main loop
24   loop (idx, cond ;
25         body0 [tree_loc0], body_sc0 [tree_loc0], stats0 [PPEn],
26         tree_loc0 [PPEn], tree_top0 [PPEn], tree_bot0 [PPEn], list0 [PPEn]
27         ->
28         bodyout [tree_loc], body_sc [tree_loc], stats [PPEn],
29         tree_loc [PPEn], tree_top [PPEn], tree_bot [PPEn], list [PPEn])
30   {
31     // Build partitions
32     Score (body_sc0[tree_loc0], tree_top0, tree_bot0 -> node_sc0[tree_loc0]);
33     balance (par, node_sc0 [tree_loc0+] -> tree_top01);
34
35     // Build top tree
36     MinMax (body0 [list0] -> M);
37     makeTree (body0 [list0], tree_top01, M ->
38               body [tree_loc+], tree_loc [PPEn+]);
39     aliasPart (tree_loc, tree_top01 -> tree_bot, tree_top, list);
40     endCond (par, idx -> cond);
41
42     // Compute center of mass
43     CtrMass (body [tree_loc], tree_top, tree_bot -> node [tree_loc]);
44
45     // Force calculation
46     force (par, M, body [tree_loc+], node [tree_loc+], stats0 [PPEn] ->
47            body_sc [list], body_ac [list], stats [PPEn]);
48
49     // Move particles
50     advance (par, idx, body [list], body_ac [list] -> bodyout [list]);
51   }
52   // Write particle data
53 }
```

Figure 3.7: Barnes-Hut type declarations and main dataflow function

derstanding of how these *Tree*, *List*, and *BodyPM* types are represented, it relies on user defined operators to apply merge and redistribution distribution rules, and to handle remote element access (Section 4.1.3). The application uses a single C++ tree to encode multiple distributions; each tree node contains a bit set, and distributions are defined as the set of nodes with one of the bits set. Hence, all *tree_top*, *tree_bot*, *tree_loc*, and *list* distributions can be stored within the same tree. Other tree datastructure and tree domain implementations are possible, subject on the user providing the necessary datastructure and domain operators.

The main loop from line 24 in Fig. 3.7 takes as input and produces as output several tree distributions and particle properties; these include the particle position, mass, computational weight, and pairwise interaction statistics. During each iteration, the iterative construct body computes these outputs starting from these inputs. Most importantly, the distributed tree is rebalanced during each iteration based on computation weight; the Barnes-Hut CGD application dynamically recomputes the tree distributions using the *makeTree* and *aliasPart* functions (lines 37–39). The loop executes the following computations: i) *Score* aggregates the computational weight of each subtree; ii) *balance* computes a new top tree trying to load balance the computational weight; iii) *makeTree* creates a new tree by inserting particles into the top tree; iv) *CtrMass* computes the center of mass of each subtree; v) *force* computes the forces acting on each particle by recursively descending the distributed tree; and vi) *advance* finally moves the particles into their new position (Fig. 2.8). These computations and the Barnes-Hut algorithm are described in more detail in Section 2.3.3.

The *CtrMass* dataflow function computes the center of mass for each tree node relying on distribution rules and remote domain access (Fig. 3.8 line 2). This function uses local domain access to compute the $tree\_bot_i$ domains of the tree bottom (line 10), and global domain access to compute the replicated $tree\_top_i$ domains of the tree top (line 11). The $tree\_loc_i$ domains contain the local regions of the tree bottom and the replicated tree top region (Fig. 2.7); hence, the $tree\_loc$ distribution is produced

```
1  // Compute the center of mass for all cell nodes
2  function CtrMass (body_pm, tree_top, tree_bot, tree_loc -> node_pm)
3    BodyPM body_pm [tree_loc],
4    NodePM node_pm [tree_loc],
5    TreePart tree_top [PPEn], tree_bot [PPEn], tree_loc [PPEn]
6  {
7    // DISTRIBUTION rules
8    part tree_loc = tree_top + tree_bot;
9
10   ctrmassb (body_pm [tree_bot]  -> node_pm [tree_bot]);
11   ctrmasst (node_pm [tree_bot+] -> node_pm2 [tree_top]);
12   copy     (node_pm2 [tree_top]  -> node_pm [tree_top]);
13 }
14
15 // Compute the bounding box of all bodies
16 function MinMax (body, list -> M)
17   BodyPM body [list],
18   Box M [ALL1],
19   ListPart list [PPEn]
20 {
21   minmaxlt <ALL1> (body [list] -> Mnp [PPEn]);
22   minmaxnp <ONE1> (Mnp [ONEn] -> M [ONE1]);
23 }
```

Figure 3.8: Barnes-Hut center of mass and bounding box dataflow functions

by merging the $tree\_bot$ and $tree\_top$ distributions using the local distribution rule $tree\_loc = tree\_top + tree\_bot$ (line 8). The top of the tree is replicated among all processes, and the top of the tree computation is also deliberately replicated; the *ctrmassb* and *ctrmasst* function invocations use $ALL1$ as a task and argument distribution, and thus, they replicate both computations and their arguments. Alternatively, a gather-scatter algorithm could build the tree top once, and then broadcast it to all processes. Overall, the computation replication algorithm is the better choice, sending less data and avoiding an extra communication step.

The *MinMax* dataflow function computes the bounding box over all particles by first computing the bounding box locally, and then aggregating the results globally (line 16 in Fig. 3.8). This function takes the local *body[list]* particles, and produces the *Mnp[PPEn]* array containing the bounding boxes for each domain of distribution *list* (line 21). The bounding boxes from *Mnp[ONEn]* are then aggregated by one of the processes to produce the global bounding box, *M[ONE1]*, which is later broadcasted

to every process (line 22). Here, the compiler knows how to convert the predefined *PPEn* distribution that assigns one element to each process to the *ONEn* distribution that assigns all n elements to a single process (Section 3.3). Similarly, the *ONE1* distribution is converted into the *ONEn* distribution by replicating the bounding box M among all processes. CGD allows changing computation and data assignments easily; in this case, the gather-scatter global bounding box aggregation can be transformed into one all-to-all communication step and one computation step replicated among all processes by simply replacing *ONE1* with *ALL1*, and *ONEn* with *ALLn* in line 22.

Chapter 4

Implementation and Evaluation

This chapter presents the implementation of a CGD language compiler, and describes its performance evaluation based on several benchmarks (Section 3.6). First, Section 4.1 gives an overview of the implementation, and outlines key compiler aspects such as the graph ordering and datastructure assignment. The front- and back-end compiler steps are then described in more detail, using the FT "slab" algorithm from Section 3.6.2 as an example. Next, experimental results are presented in Section 4.2, which gives an account of machine setup and methodology, and analyzes the performance of each benchmark for multiple implementations and machine configurations. The CGD benchmarks rely on the MPI, SHMEM, and pthreads runtimes, while the original benchmark implementations are written for MPI, pthreads, and OpenMP.

The CGD model permits a compiler implementation that produces efficient parallel code. Furthermore, optimizations such as communication-computation overlap, caching of global reads, buffering of global writes, and datastructure reuse, are supported automatically at the same time being effective (Section 4.1.3). Experimental results show that CGD performance matches and sometimes exceeds original implementation performance; the performance boost is partly due to high-level algorithmic refinements, and architecture-specific runtime optimizations (Section 4.2).

4.1 Compiler Implementation

4.1.1 Overview

A CGD application consists of a CGD code file, and C++ header and code files. Figs. 4.1 and 4.2 show that the CGD code file, *FT.pd*, contains type declarations, distribution rule definitions, global constants, SPMD function declarations, and dataflow function definitions. On the other hand, the C++ header file, *FT.h*, defines user types, and the *FT.cc* file provides C++ implementations for all SPMD functions (Fig. 3.2). The CGD compiler reads the *FT.pd* file and produces C++ files *FT_auto.h* and *FT_auto.cc*; the header contains user type declarations, automatic type definitions, and function signatures, whereas the code file contains the parallel implementation of all dataflow functions (Fig. 4.3). Dataflow function implementations call user defined sequential functions, and include CGD distributed datastructure operations that may execute communication and synchronization. These operations are provided by a CGD runtime library implemented based on the SHMEM, MPI, and pthreads libraries. The final application is built by linking together the automatically generated C++ code, the C++ code provided by the user, and the CGD runtime library.

The CGD compiler takes a CGD code file as input and generates C++ header and code files as output. The compiler front-end first performs lexical and syntax analysis and generates an abstract syntax tree using the *flex* and *bison* GNU tools. Then, the semantic analysis step completes missing computation node arguments using default values and rules, determines the type of all data nodes from each dataflow function graph, and checks for type inconsistencies. The syntax tree is converted into an intermediate representation that holds symbol information, distribution rules, and a dataflow graph structure for each dataflow function; graph structures can recursively include other graph structures (Section 4.1.2).

The compiler back-end starts by inlining dataflow functions to expand optimization

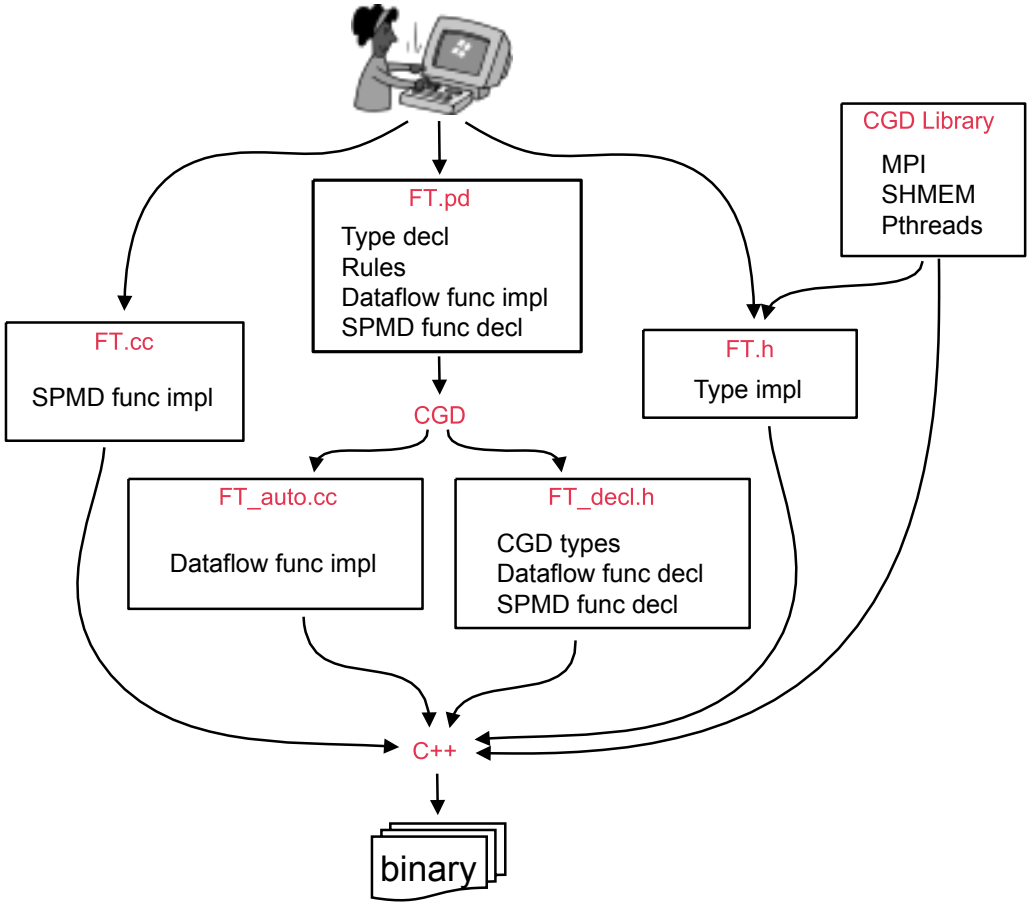


Figure 4.1: CGD application components: *FT.pd* – CGD declarations and dataflow functions; *FT.h* – C++ type definitions; *FT.cc* – C++ sequential computations.

scope. The compiler generates a topological ordering of dataflow functions by recursively ordering graph structures. The following datastructure assignment step assigns the data nodes from each dataflow function to C++ datastructure names; this step tries to avoid local copy operations and minimize the number of allocated C++ datastructures. Finally, C++ code is generated from the intermediate representation for all relevant dataflow functions. These steps are presented in greater depth in Section 4.1.3.

The graph ordering step is particularly important for a dataflow language. CGD graph ordering features a few particularities: i) distribution rules are used to automatically

add computations and links in the graph, whereas redistribution rules are avoided when other rules can be applied; ii) computation nodes are moved from subgraphs to their enclosing graphs to reduce computation replication; iii) computation nodes are removed from the graph if they are not needed to produce the output data nodes; and iv) when multiple graph orderings are possible, the original computation ordering is used as a tie breaker.

The compiler adds a redistribution rule by inserting *setup*, *begin*, and *end* operations. The CGD runtime typically initializes a shared datastructure during the *setup* operation, starts sending and receiving data during the *begin* operation, and completes sending and receiving data during the *end* operation. E.g., lines 13, 21, and 29 in Fig. 4.2 show how redistribution rule *sw01* is employed to transform datastructure B from multi-distribution *mdom0* to multi-distribution *mdom1;* lines 38-41 in Fig. 4.2 show the C++ code that implements this redistribution operation.

Communication-computation overlap is exploited by scheduling the *begin* operation as early as possible, and the *end* operations as late as possible while maintaining data dependencies. Particularly, for multi-distributions this results in sending and receiving data for smaller domains and achieving better overlap; one such example is the NPB FT "slab" algorithm presented in Section 2.3.2. In line 25 of Fig. 4.2, the invocation of SPMD computation *cffts3* is translated into the loop from line 30 of Fig. 4.3; this loop calls the *cffts3* sequential function for all Z-wise slabs and redistributes data from the *mdom1* multi-distribution to the *mdom2* multi-distribution by inserting a *swapBegin* operation at line 29 and a *swapEndAM* operation at line 32. Here, *swapBeginAM* sends data using asynchronous messaging and *swapEndAM* is called multiple times to receive data for each "slab" domain. The *cffts3* computation from line 33 is executed after data for the next iteration have been sent but not yet received, and the *cffts3* execution is therefore overlapped with the asynchronous Y-Z transposition.

The assignment step maps multiple compatible data nodes to the same C++ datastruc-

```
1  type range Range3D;                                              // types
2  type partition <Range3D> PartRange3D [PartNP];
3  type mpartition <Range3D> MPartRange3D [PartNP];
4  type mswap <Range3D> MSwapRange3D [PartNP];
5  type data Vector3Dcplx [PartRange3D, MPartRange3D];
6  type data Setup, Int;
7  ...
8  const Setup sp[ALL1], Int fdir[ALL1], lay1d[ALL1];              // constants
9  const PartRange3D  dom0[ALLn], dom1[ALLn], dom2[ALLn];
10 const MPartRange3D mdom0[ALLn], mdom1[ALLn], mdom2[ALLn];
11 const MSwapRange3D sw01[PPEn], sw12[PPEn], sw02[PPEn];
12 ...
13 part sw01 : dom0 -> dom1; part sw12 : dom1 -> dom2;             // distribution
14 part sw02 : dom0 -> dom2;                                       // rules
15 part dom0 = mdom0; part dom1 = mdom1; part dom2 = mdom2;
16 ...
17
18 function fft (A -> D)
19   Vector3Dcplx A[dom0], D[dom2]
20 {
21   cffts1 <ALL1>  (sp, fdir, A[dom0] -> B[dom0]);                // X FFT
22   if (lay1d, B[dom0] -> D[dom2])
23   {                                                            // 1D layout
24     cffts2 <ALL1>  (sp, fdir, B[dom0]  -> C[dom0]);            // Y FFT
25     cffts3 <mdom2> (sp, fdir, C[mdom2] -> D[mdom2]);           // Z FFT
26   }
27   else
28   {                                                            // 2D layout
29     cffts2 <mdom1> (sp, fdir, B[mdom1] -> C[mdom1]);           // Y FFT
30     cffts3 <mdom2> (sp, fdir, C[mdom2] -> D[mdom2]);           // Z FFT
31   }
32 }
```

Figure 4.2: NPB FT "slab" CGD code: dataflow function *fft*, and relevant global types, constants, and distribution rules.

ture; the assignment algorithm maps to the same C++ datastructure in-out argument data nodes, iteration input and output data nodes, data nodes with the same underlying datastructure, and arbitrary compatible data nodes. The first three cases reduce the number of local copy operations between nodes representing the same datastructure, and all of the above cases reduce the total number of allocated C++ datastructures. Two data nodes are compatible if they have the same type, one distribution is included in the other distribution, and either both data nodes have the same datastructure, or the final use of one data node comes before the first use of the other.

Let us consider the optimized FT "slab" example from Fig. 4.2. Its dataflow graph

contains eleven data nodes corresponding to 3D vectors, including *A[dom0]*, *B[dom0]*, *B[mdom1]*, *C[dom0]*, *C[mdom1]*, *C[mdom2]*, *D[mdom2]*. If the handwritten Fortran MPI implementation uses three 3D vectors by relying on reshapable arrays, the automatically generated C++ code from Fig. 4.3 uses just four vectors: *u0_dom2*, *u2_dom2*, *u2_dom1*, and *u2_dom0*. The CGD compiler not only generates code that uses about the same amount of memory as the original handwritten implementation, but it also avoids copy operations inherent to dataflow languages.

In general, an automatic datastructure assignment may give better results than a manual assignment, especially for large-scale projects. The automatic assignment is easily applied to the entire project scope after dataflow functions are inlined; on the other hand, the manual assignment requires programmers to coordinate code development globally, and manually change assignments and check for use conflicts each time a change is made. Note that sequential compiler optimizations are unlikely to map full distinct datastructures such as arrays or trees to the same memory location; however, the CGD compiler is able to reuse full datastructures by taking advantage of dataflow semantics and CGD data nodes that represent entire datastructures.

4.1.2 Front End

Lexical Analysis The first compiler step is lexical analysis, where the source code is broken up into tokens; lexical analysis is implemented using the GNU *flex* tool to generate a CGD language scanner. The scanner produces 25 token types, including identifiers, operators, delimiters, and the eight CGD reserved words: *function*, *loop*, *if*, *else*, *type*, *const*, *part*, and *as*.

Syntax Analysis The next step is syntax analysis, where the token sequence is parsed into an abstract syntactic tree using the GNU *bison* tool. CGD has a context-free grammar that is specified by 30 rules; these rules recursively define language

```
1   Alias (PartRange3D, dom0, env.dom0);                              // constants
2   Alias (PartRange3D, dom1, env.dom1);
3   Alias (PartRange3D, dom2, env.dom2);
4   Alias (MPartRange3D, mdom1, env.mdom1);
5   Alias (MPartRange3D, mdom2, env.mdom2);
6   Alias (MSwapRange3D, sw01, env.sw01);
7   Alias (MSwapRange3D, sw12, env.sw12);
8   Alias (MSwapRange3D, sw02, env.sw02);
9   Alias (Setup, sp, env.sp);
10  Alias (Int, fdir, env.fdir);
11  ...
12  Data (Vector3Dcplx, u0_dom2);                                     // declare vars
13  Data (Vector3Dcplx, u2_dom0);
14  Data (Vector3Dcplx, u2_dom1);
15  Alloc (u0_dom2, dom2[pe]);                                        // allocate vars
16  Alloc (u2_dom0, dom0[pe]);
17  Alloc (u2_dom1, dom1[pe]);
18  ...
19  swapSetupAM (sw02, u2_dom0, u0_dom2, 0, 0, pe);                   // init swaps
20  swapSetupAM (sw01, u2_dom0, u2_dom1, 0, 1, pe);
21  swapSetupAM (sw12, u2_dom1, u0_dom2, 0, 2, pe);
22  ...
23
24  cffts1 (sp, fdir, u2_dom0 /* mod */, dom0[pe]);                   // X FFT
25
26  if (lay1d)                                                        // 1D layout
27  {
28    cffts2 (sp, fdir, u2_dom0 /* mod */, dom0[pe]);                 // Y FFT
29    swapBeginAM (sw02, u2_dom0, u0_dom2, 0, 0, pe);                 // start Z swap
30    for (int _mi=0; _mi<mdom2.getNo(pe); _mi++)                     // loop Z slabs
31    {
32      swapEndAM (sw02, u2_dom0, u0_dom2, _mi, 0, 0, pe);            // end Z swap
33      cffts3 (sp, fdir, u0_dom2 /* mod */, mdom2.idx(pe,_mi));      // Z FFT
34    }
35  }
36  else                                                              // 2D layout
37  {
38    swapBeginAM (sw01, u2_dom0, u2_dom1, 0, 1, pe);                 // start Y swap
39    for (int _mi=0; _mi<mdom1.getNo(pe); _mi++)                     // loop Y slabs
40    {
41      swapEndAM (sw01, u2_dom0, u2_dom1, _mi, 0, 1, pe);            // end Y swap
42      cffts2 (sp, fdir, u2_dom1 /* mod */, mdom1.idx(pe,_mi));      // Y FFT
43    }
44    swapBeginAM (sw12, u2_dom1, u0_dom2, 0, 2, pe);                 // start Z swap
45    for (int _mi=0; _mi<mdom2.getNo(pe); _mi++)                     // loop Z slabs
46    {
47      swapEndAM (sw12, u2_dom1, u0_dom2, _mi, 0, 2, pe);            // end Z swap
48      cffts3 (sp, fdir, u0_dom2 /* mod */, mdom2.idx(pe,_mi));      // Z FFT
49    }
50  }
```

Figure 4.3: NPB FT “slab” generated C++ code: relevant code fragments implementing dataflow function *fft*; variable *env* points to the global scope that defines global constants including partitions and redistributions.

constructs such as types, constants, distribution rules, SPMD function declarations, dataflow function definitions, function invocations, iterative constructs, and conditional constructs (Chapter 3).

Semantic Analysis During this step, the compiler creates the global and local scopes, determines all data node types, fills in the default arguments and distributions, and checks the program for inconsistencies.

A CGD program defines a global scope, and each dataflow function defines a local scope. Scopes contain distribution rules, types, and $\langle datastructure, distribution, type \rangle$ tuples representing data nodes and their type. As expected, when parsing a dataflow function, the local scope overwrites the global scope. The global scope has two particularities: data nodes can only be constants, and types are declared only in the global scope.

Dataflow function definitions and SPMD function declarations fully specify the type and distribution of their arguments. However, the type of data nodes that are not arguments or global constants is not specified. Similarly, function invocations are not required to specify all arguments and distributions. Fortunately, data node types can be inferred from function declarations; extra arguments and distributions can also be inferred from function declarations, or using default values.

The CGD syntax allows both a simplified and a full function invocation syntax. The aim of simplified syntax is to make programs more readable and easier to write, while that of full syntax is to provide better verbosity. The full syntax requires that all arguments have distributions, and all non-global distributions are provided as arguments (Section 3.5.3). E.g., the function invocation from line 21 in Fig. 4.2 is not fully specified; the first two arguments are missing their distribution, and the *dom0* distribution is not provided as an argument.

During semantic analysis, the compiler reconstructs the full function invocation syn-

tax from the simplified function invocation syntax. Missing argument distributions are inferred based on argument type: *PPEn* is used for datastructures allowing *PartNP* distribution types, and *ALL1* is used for datastructures allowing *PartOne* distribution types; in particular, distribution arguments lacking distributions take the PPEn distribution, and non-decomposable datastructures take the *ALL1* distribution. Moreover, non-global distributions used by data node arguments but not provided as stand-alone arguments are automatically added as an extra input argument. E.g., the invocation *"cffts1 (sp, fdir, A[dom0] -> B[dom0])"* is converted into *"cffts1 <ALL1> (sp[ALL1], fdir[ALL1], A[dom0], dom0[PPEn] -> B[dom0])"* by adding the default task distribution <*ALL1*>, adding the default distribution <*ALL1*> to the first two arguments, and adding the *dom0* distribution used by data node *A[dom0]* as a separate argument with default distribution PPEn.

Dataflow functions only specify the type of input and output arguments; all other data nodes types are inferred automatically. When a function graph is parsed, the compiler generates a list of type relations that are later used to determine datastructure types and check the type consistency. These relations are built by matching actual arguments against function declarations, by matching datastructures with their distributions, and by analyzing distribution rules. Type relations can be assignments or consistency checks.

Assignment type relations have the following form:

$$t(A) \quad \leftarrow \quad \begin{cases} t(B) \\ part2range(B) \\ part2swap(B) \end{cases} \tag{4.1}$$

The type of datastructure A can be computed when the type of datastructure B is known using relation (4.1). The right-hand side of the relation is computed as follows: i) $t(B)$ is the type of datastructure B; ii) $part2range(B)$ is the range type corresponding

to distribution type B; and iii) $part2swap(B)$ is the redistribution type corresponding to distribution type B.

Similarly, consistency check type relations have the following form:

$$check\ T \quad = \quad \begin{cases} t(B) \\ part2range(B) \\ part2swap(B) \end{cases} \tag{4.2}$$

These relations check whether type T is identical to the right-hand side of the relation. Types are considered identical only if they correspond to the same global symbol.

The type relations from a dataflow function are built into a graph that is traversed level by level to determine all types and complete all checks. A single relation establishes a link between B and A; once the type of B is evaluated, the type of A can be evaluated as well. For a simple graph that doesn't include subgraphs, the relation graph can be traversed in one pass; the type of all function invocation arguments is known from the start. For nested graphs, the relation graph is traversed in several steps, propagating types from the inner to the outer subgraphs. When the relation graph traversal is completed, all datastructures—and therefore all data nodes—have a type, and the compatibility of distribution types is checked.

Intermediate Representation Generation After semantic analysis is complete, an intermediate format is generated from the abstract parse tree. The intermediate format has the following representation: a program contains the global scope, SPMD function declarations, and dataflow function definitions; the global scope contains types, constant data nodes, and distribution rules; dataflow function definitions contain a local scope and a dataflow graph; local scopes contain only data nodes and distribution rules. As an example, the intermediate representation of a dataflow function, *fft*, is illustrated in Fig. 4.4.

```
1  0   0  0  IN <ALL1> (  -> A[dom0], ALL1[ALLn], ONE1[ALLn], ALLn[ALLn], ONEn[ALLn],
2                PPEn[ALLn], SwONE2ALL1[PPEn], SwONE2ALLn[PPEn], SwPPE2ALLn[PPEn],
3                SwPPE2ONE n[PPEn], SwONE2PPEn[PPEn], sp[ALL1], niter[ALL1], fdir[ALL1],
4                finv[ALL1], lay1d[ALL1], mdom0[ALLn], mdom1[ALLn], mdom2[ALLn], dom0[ALLn],
5                dom1[ALLn], dom2[ALLn] , sw01[PPEn], sw12[PPEn], sw21[PPEn], sw10[PPEn],
6                sw02[PPEn], sw20[PPEn] )
7  2   2  0  cffts1 <ALL1> ( sp[ALL1], fdir[ALL1], A[dom0], dom0[PPEn] -> B[dom0] )
8  3   3  0  IF 0 <ALL1> ( lay1d[ALL1], B[dom0] -> D[dom2] )
9  0   0  0      IN <ALL1> (  -> lay1d[ALL1], B[dom0] )
10 2   0  0      cffts2 <ALL1> ( sp[ALL1], fdir[ALL1], B[dom0], dom0[PPEn] -> C[dom0] )
11 3   0  0      cffts3 <mdom2> ( sp[ALL1], fdir[ALL1], C[mdom2], mdom2[PPEn] -> D[mdom2] )
12 4   0  0      OUT <ALL1> ( D[dom2] ->  )
13             ELSE
14 0   0  0      IN <ALL1> (  -> lay1d[ALL1], B[dom0] )
15 2   0  0      cffts2 <mdom1> ( sp[ALL1], fdir[ALL1], B[mdom1], mdom1[PPEn] -> C[mdom1] )
16 3   0  0      cffts3 <mdom2> ( sp[ALL1], fdir[ALL1], C[mdom2], mdom2[PPEn] -> D[mdom2] )
17 4   0  0      OUT <ALL1> ( D[dom2] ->  )
18             ENDIF
19 4   4  0  OUT <ALL1> ( D[dom2] ->  )
```

Figure 4.4: NPB FT "slab": intermediate representation of function block *fft* after semantic analysis

A scope structure contains the following elements:

1. A list of data nodes and their type. This is represented as a map between datastructures and a list of distributions, and a map between datastructures and their type.

2. The declaration or definition of types. Custom types are only declared, whereas automatic types are defined for both distributions and redistributions (Section 3.1).

3. The distribution rules for inclusion. These are represented as a directed acyclic graph, where nodes are distributions, and links between nodes represent the inclusion relation. A distribution X is considered smaller than distribution Y if there is a path from X to Y in this directed graph; in this case, if a data node $A[X]$ is needed as an input but only $A[Y]$ is available, $A[Y]$ can be used as an input instead of $A[X]$ (Section 3.4).

4. The distribution rules for union. These are represented as a list describing how a distribution can be created by merging a list of smaller distributions, e.g., $A = B + C$ indicates that distribution A can be created by merging the domains of

distributions B and C (Section 3.4).

5. The distribution rules for redistribution. These specify a redistribution matrix for every redistribution transformation, e.g., $M_{c2h} : cell \rightarrow halo$ specifies how the $cell$ distribution can be transformed into the $halo$ distribution (Section 3.4).

A dataflow graph structure is represented as follows:

1. A list of input and output data nodes. These represent the inputs that are required and the outputs that are produced by the graph.

2. A list of computations. Each computation has a list of input data node arguments and a list of output data node arguments. These computations represent computation nodes in the dataflow graph; their arguments and the input and output data nodes from (1) represent data nodes in the dataflow graph.

3. Computations can be SPMD functions, dataflow functions, conditional language constructs, and iterative language constructs. Dataflow functions and language constructs can recursively include graph structures (Section 3.5.3).

4.1.3 Back End

Function Inlining The internal representation generated by the compiler is optimized by replacing dataflow function invocations with their dataflow graphs. The goal of this step is not avoiding the function call overhead, but expanding the scope of optimizations such as datastructure assignment and communication-computation overlap.

By default, all dataflow function invocations are recursively replaced with their graphs; a pragma directive is used to avoid the expansion of specific functions.

Expanding dataflow function F is first conducted by expanding all included dataflow functions, I_n, and then, by merging the scope and graph structures of expanded functions I_n into the scope and graph structure of function F. Merging function I_i starts

```
1  4  20  4  cffts1 <ALL1> ( sp[ALL1], fdir[ALL1], u0[dom0], dom0[PPEn] -> Be2[dom0] )
2  5  21  5  IF 0 <ALL1> ( lay1d[ALL1], Be2[dom0], ALL1[PPEn], sp[ALL1], fdir[ALL1], dom0[PPEn],
3               PPEn[PPEn], mdom2[PPEn], sw02[PPEn], dom2[PPEn], mdom1[PPEn], sw01[PPEn],
4               PPEn[PPEn], dom0[PPEn], sp[ALL1], ALL1[PPEn], fdir[ALL1], mdom2[PPEn],
5               sw12[PPEn], dom1[PPEn], dom2[PPEn] -> u1[dom2] )
6  0  12  0       IN <ALL1> (  -> lay1d[ALL1], Be2[dom0] )
7  1  13  1       DEP <ALL1> (  -> ALL1[PPEn], sp[ALL1], fdir[ALL1], dom0[PPEn], PPEn[PPEn],
8                    mdom2[PPEn], sw02[PPEn], dom2[PPEn] )
9  2  14  2       cffts2 <ALL1> ( sp[ALL1], fdir[ALL1], Be2[dom0], dom0[PPEn] -> Ce2[dom0] )
10 3   5  3       SWAP_Beg_AM -1 <ALL1> ( sw02[PPEn], Ce2[dom0] -> Ce2[mdom2] )
11 4  30  4       SWAP_End_AM -1 <mdom2> ( sw02[PPEn], Ce2[dom0] -> Ce2[mdom2] )  Ct
12 5  15  5       cffts3 <mdom2> ( sp[ALL1], fdir[ALL1], Ce2[mdom2], mdom2[PPEn] -> u1[mdom2] )
13 6  16  6       OUT <ALL1> ( u1[dom2] ->  )
14              ELSE
15 0  14  0       IN <ALL1> (  -> lay1d[ALL1], Be2[dom0] )
16 1  15  1       DEP <ALL1> (  -> mdom1[PPEn], sw01[PPEn], PPEn[PPEn], dom0[PPEn], sp[ALL1],
17                    ALL1[PPEn], fdir[ALL1], mdom2[PPEn], sw12[PPEn], dom1[PPEn], dom2[PPEn] )
18 2   5  2       SWAP_Beg_AM -1 <ALL1> ( sw01[PPEn], Be2[dom0] -> Be2[mdom1] )
19 3  35  3       SWAP_End_AM -1 <mdom1> ( sw01[PPEn], Be2[dom0] -> Be2[mdom1] )  Ct
20 4  16  4       cffts2 <mdom1> ( sp[ALL1], fdir[ALL1], Be2[mdom1], mdom1[PPEn] -> Ce2[mdom1] )
21 5   6  5       SWAP_Beg_AM -1 <ALL1> ( sw12[PPEn], Ce2[dom1] -> Ce2[mdom2] )
22 6  36  6       SWAP_End_AM -1 <mdom2> ( sw12[PPEn], Ce2[dom1] -> Ce2[mdom2] )  Ct
23 7  17  7       cffts3 <mdom2> ( sp[ALL1], fdir[ALL1], Ce2[mdom2], mdom2[PPEn] -> u1[mdom2] )
24 8  18  8       OUT <ALL1> ( u1[dom2] ->  )
25              ENDIF
```

Figure 4.5: NPB FT "slab" function *fft*: intermediate representation after function expansion and graph ordering

by inserting the scope of I_i into the scope of F, while at the same time renaming the I_i datastructures and distributions to avoid name conflicts; scope merging includes datastructure and distribution rule merging.

A rename map is created to translate old names into new ones; this map adds entries for I_i function arguments and for I_i data nodes that have name conflicts. A copy of the I_i graph structure is created, and all data nodes are renamed using the rename map. Finally, the expanded I_i graph structure is inserted into F to replace the initial dataflow function invocation. After this process is completed for all functions I_n, the expanded function F is stored into a table, and will be used to expand the subsequent invocations of F.

Topological Order Generation Here, the compiler recursively generates a topological order for every graph structure contained by expanded dataflow functions generated during the previous step. Additionally, distribution rules are employed to add

datastructure transformation computations when needed. Each graph structure is ordered first by ordering its included subgraphs, such as graphs defined by conditional and iterative constructs, and then, by generating a sequence or schedule of computations that produces the graph structure outputs based on the graph structure inputs while maintaining the required data dependencies. Several optimizations are applied during this process, including moving computation nodes from iterative construct subgraphs to their enclosing graphs when possible, and maximizing communication-computation overlap.

The ordering of a graph structure—containing a list of computations and their arguments as an intermediate representation—first creates a dataflow graph where links are added between computation nodes and their output data nodes, and between data nodes and computation nodes that require these data nodes as inputs. E.g., the *cffts2* computation node from line 10 in Fig. 4.4 depends on data node *B[dom0],* which in turn depends on the *cfft1* computation from line 7.

After the graph representation is created, it is traversed starting from the output data nodes, by following predecessor links; predecessors of input data nodes are not followed. When reaching a data node *A[X]* that has no predecessors and is not an input data node, the compiler tries to add links and datastructure transformations using distribution rules that can create distribution X from other distributions already available for datastructure A. This algorithm employs a heuristic that aims at reducing the number of redistributions and local copy operations. Missing data nodes are constructed by applying the minimum number of distribution rules in the following order:

1. Inclusion rules
2. Union and inclusion rules
3. Redistribution and inclusion rules
4. Union, redistribution and inclusion rules

After successfully completing this traversal, the graph contains all of the nodes and all the links needed to compute the output nodes, starting from the input nodes.

The ordering of a graph structure can now be completed by generating a topological ordering of the graph built during the previous step; this ordering contains only the nodes needed to compute the output nodes of the graph. When multiple orderings are possible, the compiler breaks any ties by giving priority to the node appearing first in the original code; effectively, the line number acts a second ordering key, transforming a partial order relation into a total order relation. This algorithm produces deterministic results that remain unchanged when other unrelated sections of the graph are modified.

Next, redistribution operations are split into *begin*, *end*, and *setup* operations (Fig. 4.5). When a *begin* operation is executed, the datastructure is ready to be redistributed; after an *end* operation is completed, the datastructure has been successfully redistributed. Communication-computation overlap is maximized by moving the *setup* and *begin* operations as early as possible, and the *end* operations as late as possible, while maintaining data dependencies at all times.

When a dataflow function graph is ordered, some computation nodes are moved from subgraphs to their parent graphs, or are removed altogether. We recall that a graph structure is ordered recursively first by ordering its subgraphs, and then, by ordering its nodes. When a graph ordering is complete, new nodes can be inserted based on distribution rules, and nodes are moved to the parent graph if they are not dependent on the input nodes, or are not needed to compute the output nodes. E.g., the boundary copy needed by the stencil example from line 18 in Fig. 3.5 is moved outside the loop subgraph since its input arguments do not depend on the loop input arguments; this optimization avoids unnecessarily copying these values during each iteration. When the recursive ordering process finishes at the top-level, computations that are not dependent on input nodes are kept in the main dataflow function graph, and computa-

tions not needed to produce output nodes are discarded. Particularly, if two distinct computations produce a single identical output data node, then at most one of these computations is included in the graph ordering.

Datastructure Assignment For each ordered graph structure the compiler maps its data nodes to C++ datastructures; the mapping algorithm uses a heuristic that tries to minimize the number of C++ datastructures, and tries to avoid local copy operations. After the mapping is complete, datastructure copy and datastructure allocation operations are added to the graph structure; the relative ordering of graph structure computation nodes is maintained throughout this process.

The assignment problem consists of creating a map between data nodes and C++ datastructure names. Multiple data nodes can be mapped to the same datastructure name if they are compatible and they have no use conflicts. Data nodes $A[X]$ and $B[Y]$ are compatible if datastructures A and B have the same type, and there is an inclusion relation between distributions X and Y. $A[X]$ and $B[Y]$ have no use conflict if $A = B$, or the last use of$A[X]$ is before the first use of $B[Y]$, or the last use of $B[Y]$ is before the first use of $A[X]$.

Datastructure assignment starts by computing a directed graph representing distribution inclusion relations; the inclusion graph is generated using the union and inclusion distribution rules. The compiler then generates a topological ordering of all distributions, and it computes the creation time and the expiry times for each data node. Creation time is represented by the position of the computation node that first uses a data node, whereas expiry time is represented by the position of the computation node that uses a data node last. Creation and expiry times are computed for each distribution, since distributions are data nodes, too. Data node compatibility is checked relying on both the inclusion graph, and the creation and expiry times.

The assignment algorithm uses a heuristic to minimize the number of C++ datas-

tructures and local copy operations. It maps compatible data nodes to the same C++ datastructure name in the following order:

1. In and out data nodes corresponding to in-out SPMD function arguments
2. Loop in data nodes and out data nodes
3. Compatible data nodes with identical datastructures
4. Compatible data nodes with distinct datastructures

The compiler executes step (3) for each datastructure A by selecting all data nodes $A[Y_i]$ from the graph and building an ordered list of distributions Y_n; distributions are ordered according to the distribution topological ordering built during the previous steps. Next, the largest distribution Y_n is selected and all data nodes $A[Y_i]$ where $Y_i \leq Y_n$ are assigned to the same C++ datastructure, and their distributions are removed from the list. This step is repeated until the distribution list becomes empty; during every step, a new C++ datastructure is created. As expected, a single C++ datastructure is allocated when Y_n is larger than all distributions of A.

Step (4) starts by recomputing the distribution, creation time, and expiry time for each group of data nodes assigned to the same C++ datastructure by steps (1), (2), and (3). Then, compatible groups with identical distributions are merged together using the following algorithm: i) the ordered graph structure is traversed from the beginning to the end while maintaining a list of target datastructures; ii) when a data node group is first used, it is assigned to a target datastructure that is not active, which activates the target datastructure; if all target datastructures are active, a new target datastructure having the name of the data node group is added; iii) when a data node group reaches its expiry time, its target datastructure becomes inactive. This algorithm minimizes the number of C++ datastructures that are needed to assign all data node groups with identical distributions.

Next, a C++ variable name is assigned to each data node from the graph structure. The naming convention tries to merge the original datastructure name with the distribution name to create a data node name; e.g., *u0_dom2* from line 24 in Fig. 4.3 represents the *dom2* distribution of datastructure *u0*. Maintaining this naming convention is not always possible since multiple distributions, and even multiple datastructures, can be mapped to the same C++ datastructure.

Before datastructure assignment is complete, the compiler makes a few more additions. First, data allocations are inserted before the first computation that uses each C++ datastructure. Synchronization operations are then added between the producer and consumer of data nodes used as global access datastructures.

Code Generation During code generation the compiler converts the latest internal representation into C++ code by generating a header file and a code file; the header file defines all types and declares all SPMD functions, and the code file contains the body of all expanded dataflow functions.

The header file declares all user defined types, and includes a user header file defining them (Fig. 4.1). Next, automatic CGD types are defined for distributions, multi-distributions, redistributions, multi-redistributions, and the global environment that holds all constant data nodes (Section 3.1). All dataflow functions and user defined SPMD functions are then declared; the function signature uses the following convention: i) input arguments are followed by in-out arguments, then by out arguments; ii) within each list datastructure arguments are followed by distribution arguments; iii) all arguments are passed by reference; and iv) the last argument is a pointer to the processing environment that holds a pointer to the global scope.

The compiler writes the body of each dataflow function to the code file executing the following steps: i) the function header is written following the convention used by code generation; ii) constants from the global scope are aliased to local variables; iii) vari-

ables are declared for all C++ datastructures; and iv) the graph structure computations are written one by one, adding auxiliary operations when needed.

Computation nodes are written as C++ function calls that take as arguments all datastructures and distributions referenced by the data node arguments. Computations are translated to *for* loops that iterate over multiple domains when data node arguments use multi-distributions, or the task-set argument is a multi-distribution. Computations that take as arguments global access datastructures are surrounded by auxiliary operations that handle synchronization as well as setup read caching and finalize buffered writes.

The code generated for copy and redistribution computation nodes does not follow exactly the same pattern as regular computation nodes. Copy operations require domain rather than distribution arguments. Redistribution operations require a few extra arguments, including an id that points to a temporary structure maintaining the communication state; these structures can be reused after the operation is completed. The compiler generates the minimum number of ids such that any concurrent redistributions use different ids.

The iterative and conditional constructs are translated into *for* and *if* statements; the loops are augmented with operations that copy the output datastructures to input datastructures when datastructure assignment cannot assign both the input and the output to the same C++ datastructure.

4.2 Experimental Results

This section first describes the two experimental platforms used to run the experiments, as well as the measurement methodology. An evaluation of the relative performance of the original, CGD, and optimized CGD implementations is then presented for the following problems: stencil computation kernel, NPB FT, and SPLASH2 Barnes-Hut N-body simulation.

4.2.1 Machine Setup

Opteron SMP The first experimental platform is a small CC-SAS SMP machine with two quad-core 2.3 GHz Shanghai Opteron processors, and 8 GB of memory. Each CPU has a split 128KB 2-way L1 cache, a 512KB 16-way L2 cache per core, and a 6MB L3 cache shared between the four cores. The L1 TLB has 48 entries and the L2 TLB has 512 entries for data and instructions; both TLBs are 4-way set associative. The system runs a Linux kernel, version 3.5.0-17, and uses the version 4.7.2 GCC compilers, and version 1.4.1 MPICH2 library. The MPI library is compiled with shared memory support providing zero copy intra-node communication.

Altix 4700 The second experimental platform is an Altix 4700, which is a larger CC-SAS distributed memory machine produced by SGI. It has 1.6 GHz Itanium 2 processors, a 6.4 Gb/sec NUMAlink4 interconnect, and 1024 processor cores organized as 256 boxes with two dual-core processors each. The CPU cores have 16KB 4-way data and instruction L1 caches, and 256KB 8-way L2 caches; the 24MB L3 cache is shared by the two cores. The L1 has 64-byte cache lines, while the L2 and L3 have 128-byte cache lines. The TLBs are fully associative, and have 32 L1 entries and 128 L2 entries, both for data and instructions. The system uses the version 4.3.3 GCC compilers. The SHMEM and MPI implementations rely on *fast_bcopy* to transfer data between nodes

via direct memory access. Short messages have very low latency, but the amount of communication overlap allowed by asynchronous communication is limited.

The experiments presented here were executed on machines deployed at the Geophysical Fluid Dynamics Laboratory, which is affiliated with Princeton University. These systems run jobs submitted by multiple scientific teams, and a scheduling system assigns jobs to machine partitions. However, the NUMAlink interconnect is shared by all machine partitions, and thus, the communication executed by a job indirectly influences the performance of other jobs when contention occurs. To attenuate this issue, all experiments that are compared against each other were executed together as a single job running on the same machine partition.

Methodology Throughout this section, all speedups presented in the same chart are computed as $S_n = T_1/T_n$, where T_1 is the same single-processor runtime for all implementations, and T_n is the n processor runtime of each implementation. In addition to the speedup charts, we present efficiency charts, where efficiency is defined as $E_n = nT_n/T_1$, and T_1 is the same runtime for all implementations. Efficiency represents a fraction of linear speedup, e.g., an efficiency of 0.8 on n processors corresponds to a speedup of $0.8n$.

The experiments were run three times for each configuration, and all measured times were aggregated and recorded into a file; large problems running on small processor counts were executed only twice. The time breakdowns presented in this chapter are taken from the run with the shortest runtime; this approach was designed to filter out measurement variability due to machine load. Particularly, for the Altix 4700 machine, independent jobs running on the computer farm can hurt bandwidth-intensive experiments that require a large machine partition. In practice, the runtime distribution is grouped within 5% of the minimum runtime, occasionally containing an outlier.

The CGD experiments measure the overall runtime, as well as the time taken by a

Table 4.1: Single-processor runtime for NPB FT implementations

Problem	Domain Size	Language	Implementation	Machine	Runtime (seconds)
FT A	$256 \times 256 \times 128$	Fortran	Sequential, NPB	Opteron SMP	6.69
		OpenMP	OpenMP, Omni	Opteron SMP	6.76
		Fortran	MPI, NPB	Opteron SMP	6.70
		C++	CGD	Opteron SMP	7.40
		C++	CGD slab	Opteron SMP	7.02
FT B	$512 \times 256 \times 256$	Fortran	Sequential, NPB	Altix 4700	306.98
		OpenMP	OpenMP, Omni	Altix 4700	314.39
		Fortran	MPI, NPB	Altix 4700	309.20
		C++	CGD	Altix 4700	329.53
		C++	CGD slab	Altix 4700	307.76
FT C	$512 \times 512 \times 512$	Fortran	Sequential, NPB	Altix 4700	1341.07

few relevant SPMD computations, datastructure redistribution operations, and other runtime library calls. These time intervals are measured using *gettimeofday* clocks that add an overhead of 0.5 microseconds.

Experiments running on either machine were compiled with the GCC compilers for C++, Fortran, and OpenMP, i.e., *"gfortran", "gcc",* and *"gcc -fopenmp"* enabling the flags *"-O3 -funroll-loops*".

4.2.2 NPB FT

The NPB FT benchmark is a spectral PDE solver that computes a 3D FFT transform during each iteration; this is a popular benchmark that exercises bisection-width bandwidth and floating point performance. The problem and its CGD implementation are described in more detail in Sections 2.3.2 and 3.6.2.

This section evaluates the performance of our CGD NPB FT implementation for both the original and optimized algorithms. We compare four implementations:

1. The "OpenMP" implementation was developed by the Omni OpenMP compiler project by converting the sequential NPB 2.3 Fortran implementation into C. This implementation uses only a 1D layout to decompose the 3D vector.

2. The "NPB MPI" implementation is the original NPB 2.3 Fortran version. It uses a 1D layout when the number of processors is small; otherwise, a 2D layout is utilized.

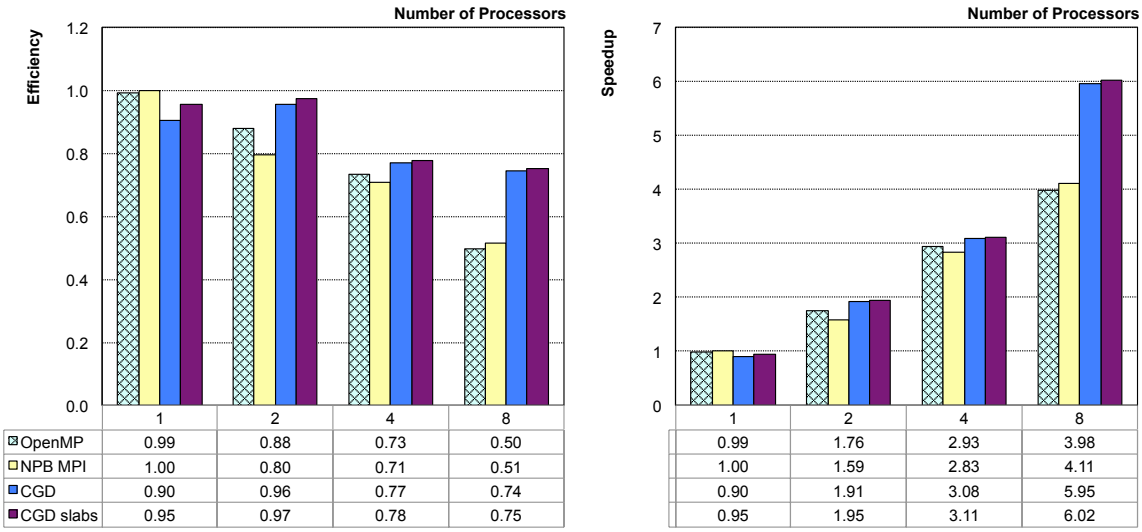
3. The "CGD" implementation has SPMD computations written as C++ functions that are based on the OpenMP C code. The lower-level C functions called from these computations are virtually unchanged and unoptimized, allowing a fair comparison between implementations. The decomposition algorithm is identical with the NPB MPI algorithm.

4. The "CGD slab" implementation adds a decomposition optimization to the CGD version, but otherwise uses the same codebase. This algorithm uses both 1D and 2D layouts; however, it decomposes data domains into smaller slices allowing communication-computation overlap and better cache utilization.

Both CGD experiments were compiled on the Altix machine with a mixed SHMEM and MPI runtime, where smaller messages are sent via SHMEM for improved latency; for the Opteron SMP, these experiments were compiled using the pthreads runtime.

The speedup of all NPB FT implementations is computed based on the sequential NPB Fortran runtime (Fig. 4.1). However, the runtimes presented herein are represented by $total - setup$, where $total$ and $setup$ are the times defined and measured by the NPB benchmark. We believe this approach makes the comparison more meaningful given the large setup time of OpenMP (Fig. 4.2)

Opteron SMP Figure 4.6 shows the NPB FT results for the smallest domain size for the SMP machine configuration. The efficiency is good for up to two processors,



	1	2	4	8
OpenMP	0.99	0.88	0.73	0.50
NPB MPI	1.00	0.80	0.71	0.51
CGD	0.90	0.96	0.77	0.74
CGD slabs	0.95	0.97	0.78	0.75

	1	2	4	8
	0.99	1.76	2.93	3.98
	1.00	1.59	2.83	4.11
	0.90	1.91	3.08	5.95
	0.95	1.95	3.11	6.02

Figure 4.6: Efficiency and speedup for NPB FT class A on Opteron SMP

after which it decreases for all implementations. Here, memory bandwidth becomes the main bottleneck; if the transpose time depends on per process domain size and ideally should scale linearly, the transpose time is 0.37 sec and 0.19 sec on two and eight processors, respectively. Unfortunately, these results are expected to occur since two processes running on two quad-cores have the same available bandwidth as eight processes running on the same two quad-cores.

Although the C++ codes have a 5–10% sequential performance handicap compared to Fortran codes (Table 4.1), CGD C++ recovers the handicap and becomes faster than NPB MPI Fortran on eight processors (1.126 vs. 1.628 sec). For this configuration, the sequential performance is CPU-bound and depends on language specific optimizations, while the parallel performance is mostly memory-bound. The CGD redistribution operation copies data elements directly between datastructures avoiding unnecessary vector-to-buffer data copy; hence, the slower sequential C++ code is able to achieve a better absolute speedup by reducing memory traffic.

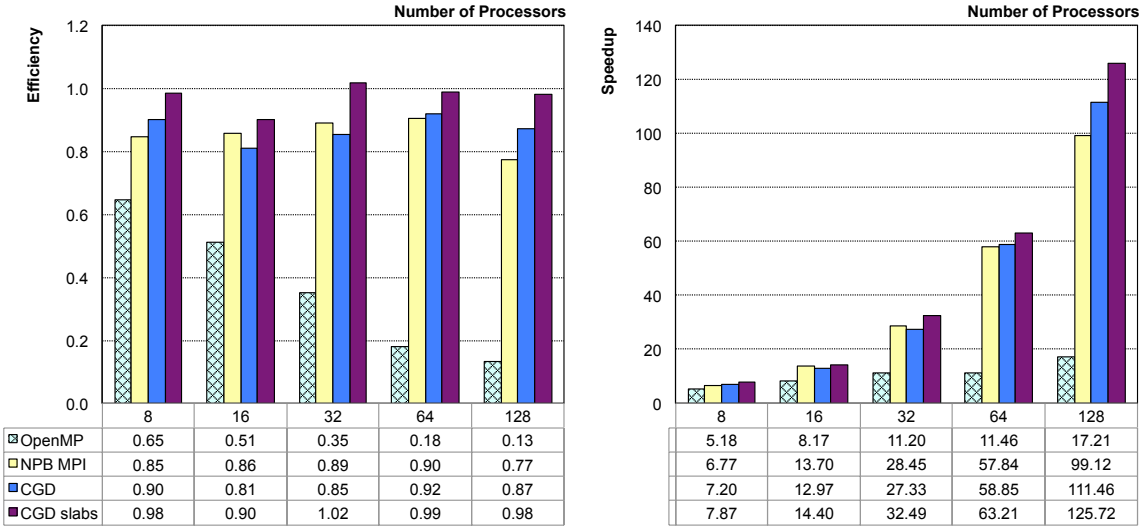
The original and optimized CGD versions have a very similar performance (1.13 vs.

1.11 sec on eight processors). This result is expected, since the overlap optimization is not effective on small SMPs where communication consists merely of reading from and writing to local memory.

CGD improves the OpenMP performance by a solid 51%. While both OpenMP and CGD represent 3D vectors as globally addressable arrays, CGD uses block copy to transpose the vectors, while OpenMP relies on finer-grain per element access. While CGD explicitly assigns vector domains to processes, the OpenMP compiler makes this assignment by relying on programmer hints. These results indicate that explicit data assignment allows a CGD runtime implementation that optimizes data transfers better than OpenMP for problems such as NPB FT, even on small SMPs.

Figure 4.6 shows that CGD is 46% faster than NPB MPI. The CGD transpose copies data directly between 3D vectors, while the NPB MPI transpose copies the same amount of data twice: i) it copies data from 3D vectors to buffers, then it sends and receives the buffers via MPI at no extra cost, relying on the zero-copy MPI implementation; and ii) it copies data from buffers into 3D vectors. Sending twice as much data to the memory negatively affects the MPI performance given the memory bandwidth limitation. In this case, the higher-level CGD datastructure abstraction allows a runtime optimization for SMPs—copying data directly between vectors—that was not available to the message passing implementation, which could only exchange contiguous memory blocks between processors.

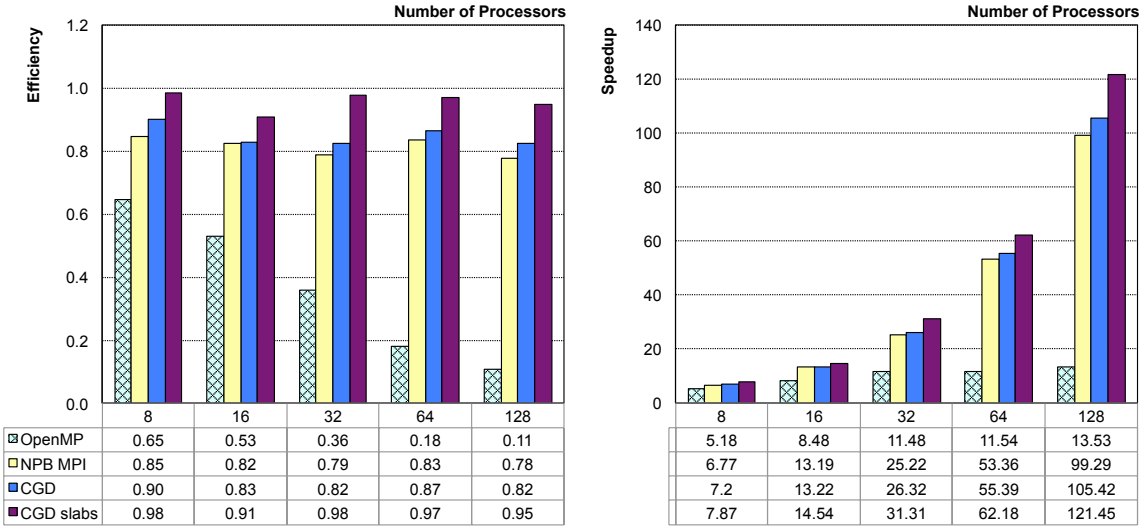
Altix 4700 The single-processor performance of the four NPB FT implementations is not exactly identical, despite the fact that all implementations are derived from the same Fortran code (Table 4.1). The sequential Fortran runtime is shorter than the single-processor CGD runtime (306.98 vs. 329.53 sec for FT class B). Similarly, the performance of the sequential 1D FFT function *fftlow* is slightly poorer for C++ than for Fortran (CGD vs. MPI in Table 4.2), and *fftcopy* exhibits the same performance disparity when copying data locally between a 3D vector and a smaller buffer; we



	8	16	32	64	128
OpenMP	0.65	0.51	0.35	0.18	0.13
NPB MPI	0.85	0.86	0.89	0.90	0.77
CGD	0.90	0.81	0.85	0.92	0.87
CGD slabs	0.98	0.90	1.02	0.99	0.98

8	16	32	64	128
5.18	8.17	11.20	11.46	17.21
6.77	13.70	28.45	57.84	99.12
7.20	12.97	27.33	58.85	111.46
7.87	14.40	32.49	63.21	125.72

Figure 4.7: Efficiency and speedup for NPB FT class B on Altix 4700



	8	16	32	64	128
OpenMP	0.65	0.53	0.36	0.18	0.11
NPB MPI	0.85	0.82	0.79	0.83	0.78
CGD	0.90	0.83	0.82	0.87	0.82
CGD slabs	0.98	0.91	0.98	0.97	0.95

8	16	32	64	128
5.18	8.48	11.48	11.54	13.53
6.77	13.19	25.22	53.36	99.29
7.2	13.22	26.32	55.39	105.42
7.87	14.54	31.31	62.18	121.45

Figure 4.8: Efficiency and speedup for NPB FT class C on Altix 4700

assume these are compiler-related issues.

The single-processor runtime handicap can bias the speedup comparison towards the implementation with the fastest single-processor code, considering that all speedups are computed using the same single-processor runtime. In Figs. 4.7 and 4.8, we present

both the speedup and efficiency of each implementation. We notice that the efficiency of the CGD implementation is already better than NPB MPI for eight processors, despite its single-processor runtime handicap, and this difference is maintained all the way up to 128 processors.

The speedups of the MPI and CGD implementations are nearly linear for FT class B and C (Figs. 4.7, 4.8). The good efficiency is partly explained by the fast all-to-all block communication of the Altix machine, and the super-linear effect of the larger aggregate multi-processor cache partially offsetting parallel overhead.

CGD is slightly faster than the original NPB MPI version, which is impressive considering that the parallel code is automatically generated, and that the C++ sequential code has a performance handicap. This performance gap is closed by the CGD runtime, which handles data faster than its NPB MPI counterpart. For example, the 1D layout transposition takes 2.58 sec in CGD vs. 5.05 sec in NPB MPI for FT class C running on 128 processors (Fig. 4.2).

The handwritten NPB MPI transpose includes some lengthy memory copy optimizations but its measured performance on the Altix 4700 machine is poorer than CGD. Let us investigate one of these transpose functions: *transpose_x_z* calls *transpose_x_z_local*, *transpose_x_z_global*, and *transpose_x_z_finish*; the first one moves vector elements into buffers, the second executes collective communication via an *mpi_alltoall* call, and the third copies data from buffers into vectors. All of these functions are optimized to reduce memory copy costs, and the local transpose tries to achieve better caching by executing the copy operation into two stages: first it copies the vector elements to a small buffer, and then, it copies the small buffer into the destination buffer.

The NPB MPI transpose optimizations take about 600 lines of code that are executed by all architectures. While these optimizations will probably lead to improved results on a particular machine, it is not clear whether they are relevant to other present-day or future architectures. It is not feasible to expect an application be optimized for all

Table 4.2: Time breakdown (sec) for NPB FT on 128 processor Altix 4700

Problem	Code section	OpenMP	MPI	CGD	CGD slab
FT B	fft	15.02	2.66	2.26	1.99
	fftlow		1.19	1.23	1.27
	fftcopy		.175	.258	.231
	swap		1.295	.794	.489
	setup	9.81	.068	.060	.060
	total - setup	17.84	2.98	2.75	2.44
	speedup	17.21	103.10	111.46	125.72
FT C	fft	81.38	12.44	10.70	9.31
	fftlow		5.40	5.73	5.74
	fftcopy		1.99	2.39	1.49
	swap		5.05	2.58	2.08
	setup	37.79	0.21	0.22	0.22
	total - setup	99.12	13.88	12.72	11.04
	speedup	13.53	96.61	105.42	121.45

target architectures; a partial approach might report good numbers in some particular cases, while possibly operating poorly in the general case. Rather than encouraging developers to spend time over-optimizing their code, a higher-level abstraction such as CGD moves these low-level architecture-specific optimizations from the application into the compiler and runtime, while making higher-level algorithmic optimizations easier to write.

Such an optimization is “CGD slab”, which reports speedups significantly exceeding the original NPB MPI results; for FT class B on 128 processors, the improvement is 22% (Fig. 4.7). This gain is due to communication scheduling, cache locality, and runtime optimizations. For example, the transposition is executed as a sequence of small chuck data transfers that fit into the cache; when local FFT computations are executed, they are likely to find these chunks in the cache. We expect the performance benefits of the “slab” algorithm to be greater on machines with improved asynchronous messaging support.

Table 4.3: Single-processor iteration time for Stencil

Domain Size	Language	Implementation	Machine	Iteration Time T_1 (usec)	Element Comp. T_e (usec)
128×128	C++	MPI, manual	Opteron SMP	51.45	.0031
256×256	C++	MPI, manual	Opteron SMP	204.35	.0031
512×512	C++	MPI, manual	Opteron SMP	872.30	.0033
128×128	C++	MPI, manual	Altix 4700	281.24	.0171
256×256	C++	MPI, manual	Altix 4700	1123.89	.0171
512×512	C++	MPI, manual	Altix 4700	4480.65	.0171
1024×1024	C++	MPI, manual	Altix 4700	31377.40	.0299

4.2.3 Stencil Computation

The stencil micro-kernel solves the equation of heat dissipation on a 2D grid by implementing a forward time centered space differencing scheme. This PDE solver computes new grid values during each iteration as a function of the old grid values, and their spatial derivatives (Section 2.3.1); hence, the solver requires nearest-neighbor communication to compute the derivatives. 2D PDE solvers—including heat dissipation—are generally latency-bound, having a poor communication-computation ratio. The stencil computation and its CGD implementations are presented in Sections 2.3.1 and 3.6.1.

Herein, we evaluate and compare both the original and optimized version of stencil computation. Three implementations are considered:

1. The “MPI manual” implementation is written by hand relying on asynchronous communication. The goal of this example is to estimate the performance of typical user implementations that include common optimizations. The “halo” exchange uses eight neighbor pair-wise communication, while at the same time handling the boundary conditions. The handwritten implementation is comprised of the following steps:

```
i.   forall (messages to receive)
        MPI_Irecv (buffer)
ii.  forall (messages to send)
        marshall message to buffer
        MPI_Isend (buffer)
iii. MPI_Waitany (messages to receive)
        unmarshall message from buffer
iv.  Compute grid update
v.   MPI_Waitall (messages already sent)
```

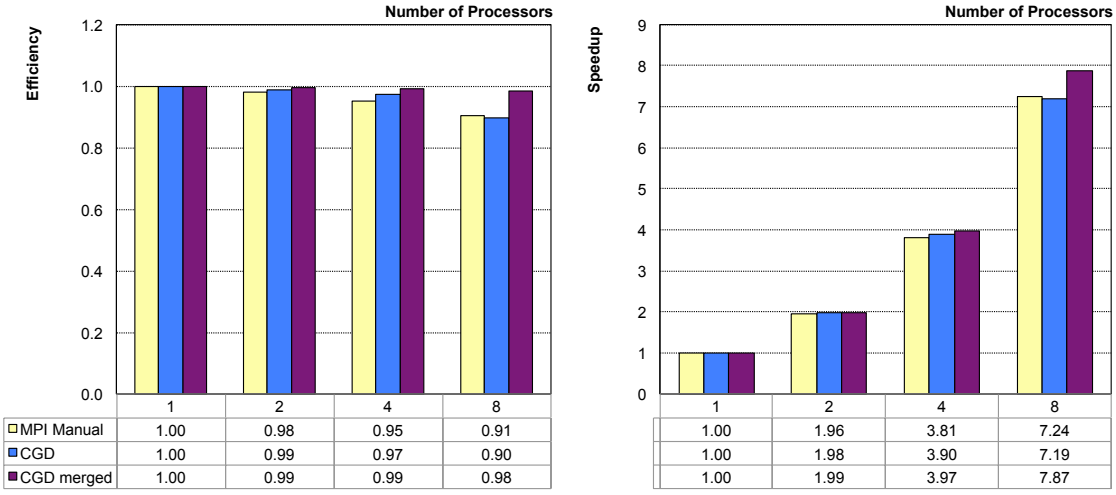
2. The "CGD" implementation represents the above algorithm in CGD, with each iteration containing a redistribution operation and a computation update.

3. The "CGD merged step" implementation optimizes the original algorithm by merging two redistributions of two consecutive iterations into a single redistribution. Two iterations require two computation updates and a single larger redistribution operation. As mentioned in Section 2.3.1, this technique reduces communication latency at the expense of slightly more computation.

The speedup of all implementations is computed based on the single-processor MPI "manual" runtime (Table 4.3). However, the single-processor runtime is virtually identical among all experiments, which are written in the same language and rely on the same update computation. Table 4.3 presents both the single-processor runtime, and the average time needed to process a single grid element, $T_e = T_1/domainsize$.

Opteron SMP Both CGD experiments use the pthreads runtime for this machine configuration; hence, the redistribution or "halo" exchange operations are executed by reading the "halo" elements directly from memory allocated by each thread, thereby, minimizing latency overhead.

Figure 4.9 shows that efficiency is very good on this small SMP for a problem size of



	1	2	4	8
MPI Manual	1.00	0.98	0.95	0.91
CGD	1.00	0.99	0.97	0.90
CGD merged	1.00	0.99	0.99	0.98

1.00	1.96	3.81	7.24
1.00	1.98	3.90	7.19
1.00	1.99	3.97	7.87

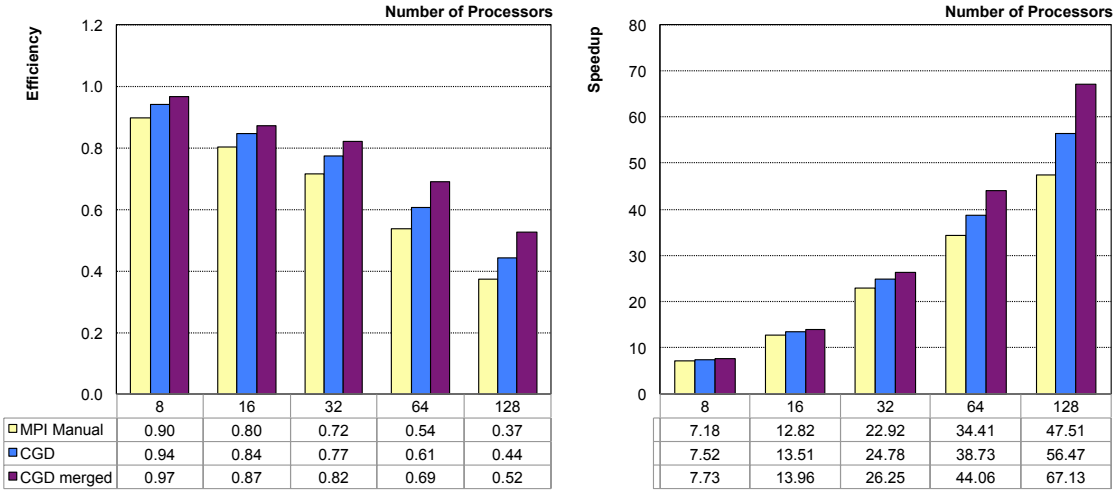
Figure 4.9: Efficiency and speedup for Stencil 512×512 on Opteron SMP

512×512. For four processors, all implementations have an efficiency greater than 0.95, and for eight processors, the MPI and CGD efficiency decreases to approximately 0.90. However, the CGD "merged step" implementation does not suffer from this performance degradation; here, the constant communication cost, which includes synchronization, is reduced by executing a single communication step rather than two communication steps.

Overall, the MPI and CGD performance is very similar on this machine, and the CGD "merged step" optimization slightly outperforms both original implementations.

Altix 4700 On this machine, both CGD experiments use the SHMEM rather than the MPI runtime to reduce messaging latency. The MPI "manual" implementation uses the standard MPI software stack.

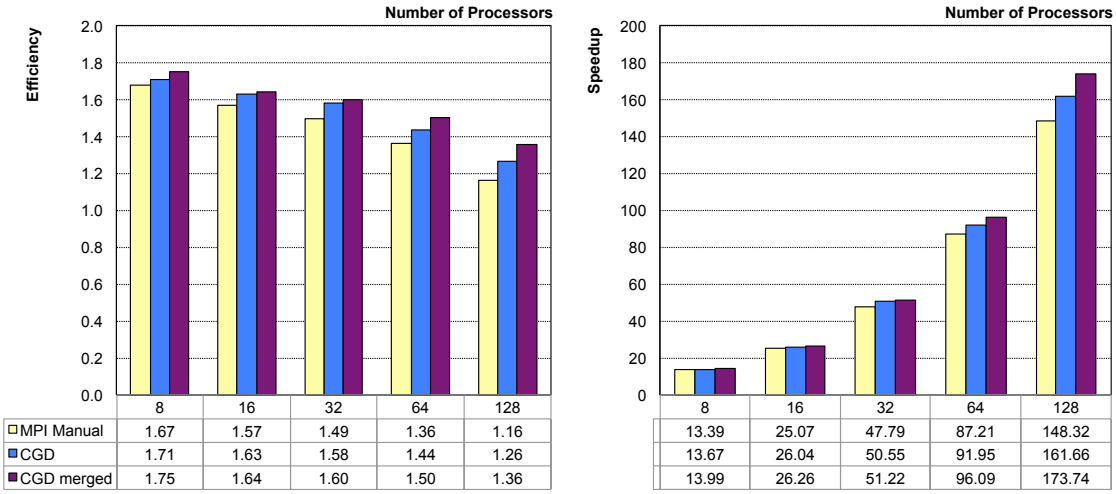
Table 4.3 reveals that the element computation time T_e remains constant for problem sizes of up to 512×512, and increases more than 75% (0.0171 vs. 0.0299 usec) when the working set stops fitting into the aggregate cache. This particularity leads to a super-linear speedup, visible in Fig. 4.13, where the "merged step" 64-processor speedup jumps from 44.06 for problem size 512×512 to 96.09 for problem size 1024×1024.



	8	16	32	64	128
MPI Manual	0.90	0.80	0.72	0.54	0.37
CGD	0.94	0.84	0.77	0.61	0.44
CGD merged	0.97	0.87	0.82	0.69	0.52

8	16	32	64	128
7.18	12.82	22.92	34.41	47.51
7.52	13.51	24.78	38.73	56.47
7.73	13.96	26.25	44.06	67.13

Figure 4.10: Efficiency and speedup for Stencil 512×512 on Altix 4700



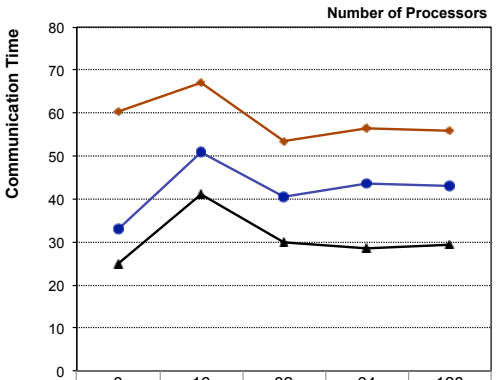
	8	16	32	64	128
MPI Manual	1.67	1.57	1.49	1.36	1.16
CGD	1.71	1.63	1.58	1.44	1.26
CGD merged	1.75	1.64	1.60	1.50	1.36

8	16	32	64	128
13.39	25.07	47.79	87.21	148.32
13.67	26.04	50.55	91.95	161.66
13.99	26.26	51.22	96.09	173.74

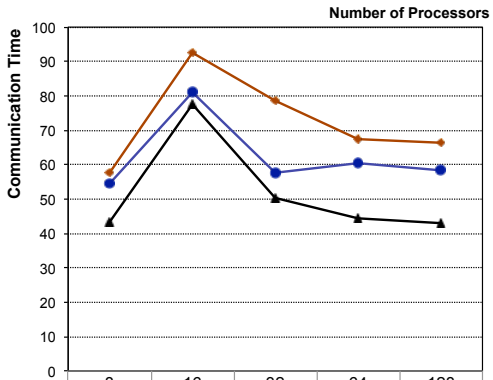
Figure 4.11: Efficiency and speedup for Stencil 1024×1024 on Altix 4700

The efficiency chart in Fig. 4.10 shows that the 2D stencil is a hard problem even for a low latency machine such as the Altix 4700; the MPI implementation efficiency drops from about 0.9 for eight processors, to 0.4 for 128 processors, while its maximum speedup reaches only 47.5.

Even when the CGD version is algorithmically identical to the handwritten MPI version, the former outperforms the latter considerably by 19%. This improvement is



	8	16	32	64	128
MPI manual	60.5	67	53.5	56.5	56
CGD	33	51	40.5	43.5	43
CGD merged	25	41	30	28.5	29.5



	8	16	32	64	128
	57.5	92.5	78.5	67.5	66.5
	54.5	81	57.5	60.5	58.5
	43.5	77.5	50.5	44.5	43

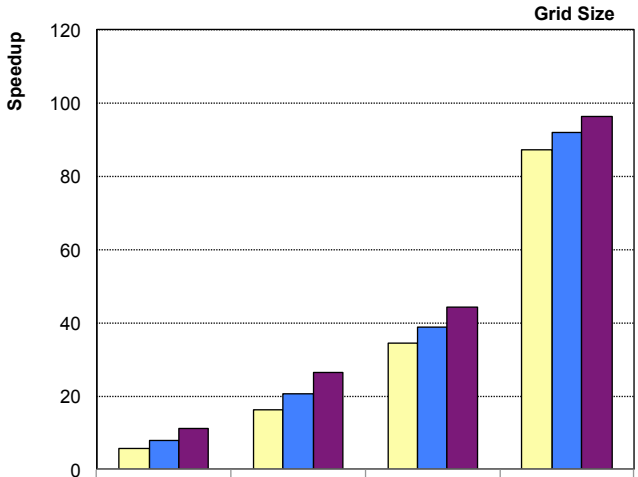
Figure 4.12: Communication time per iteration (microseconds) for Stencil 512×512, 1024×1024 on Altix 4700

mainly due to the SHMEM runtime being slightly faster than MPI for small messages; for the 512×512 problem, CGD communication improves the MPI time from 56 usec to 43 usec on 128 processors (Fig. 4.12). As was the case with NPB FT (Section 4.2.2), the CGD distributed datastructure abstraction allows architecture-specific optimizations that are otherwise unavailable to MPI applications.

The CGD "merged step" optimization outperforms both the MPI and CGD implementations, improving the MPI speedup by more than 40% (67.1 vs. 47.5). Here, the slightly more expensive update computation is offset by the shorter aggregated communication time, which drops from 43 usec for MPI to 29.5 usec for CGD "merged step" (Fig. 4.12).

Moreover, communication time does not really increase with the processor count, being mostly determined by the constant latency cost; on the other hand, the CGD optimizations, which reduce latency or avoid a communication step altogether, show improved results across the board (Fig. 4.12). This experiment shows that algorithmic changes taking advantage of latency hiding techniques can effectively mitigate machine limitations.

Figure 4.11 presents improved results for all implementations for the larger 1024×1024



	128	256	512	1024
MPI Manual	5.48	16.36	34.41	87.21
CGD	7.83	20.45	38.73	91.95
CGD merged	11.16	26.48	44.06	96.09

Figure 4.13: Speedup for Stencil 128×128, 256×256, 512×512, 1024×1024 on 64-processor Altix 4700

problem size: efficiency decreases from approximately 1.7 to 1.2, and speedups on 128 processors range from 148 to 173. As previously discussed, these numbers show super-linear speedups due to the larger aggregate cache.

Figure 4.13 plots speedups for different problem sizes running on 64 processors; a super-linear bump is easily visible between 512 and 1024. We recall that T_e—the sequential processing time of each element—is significantly higher for the 1024×1024 problem size, since the full dataset does not fit into a single processor's cache (Table 4.3). The parallel code splits the dataset among multiple processors, reducing the local working set to a fraction of the full dataset. When the number of processors is large enough the local working set fits into the processor cache, and the application benefits from a large computation speedup. Fig. 4.11 already shows a super-linear speedup of 13-14 for eight processors. The efficiency then gradually decreases with processor count since the computation decreases in size, communication starts to become more latency-bound, and the overall communication-computation ratio increases.

Table 4.4: Single-processor runtime for Swaptions and Blackscholes (largesim)

Benchmark	Language	Implementation	Machine	Runtime (seconds)
Swaptions	C++	PARSEC, pthreads	Opteron SMP	9.085
Swaptions	C++	CGD, pthreads	Opteron SMP	8.853
Swaptions	C++	CGD, MPI	Opteron SMP	8.866
Blackscholes	C++	PARSEC, pthreads	Opteron SMP	1.614
Blackscholes	C++	CGD, pthreads	Opteron SMP	1.473
Blackscholes	C++	CGD, MPI	Opteron SMP	1.469

4.2.4 Swaptions

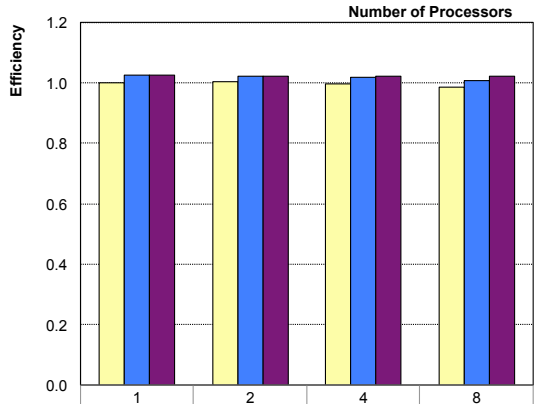
This benchmark computes the price of a swaptions portfolio using the Heath-Jarrow-Morton (HJM) framework [BL09]. The HJM framework models the interest rate evolution, taking into consideration the relationship between the drift and volatility of the forward-rate dynamics. There is no analytical approach to solving this non-Markovian model, and the price computation is therefore based on Monte Carlo (MC) simulation.

This section evaluates the performance of Swaptions for both the CGD and PARSEC [BL09] versions. We compare three implementations:

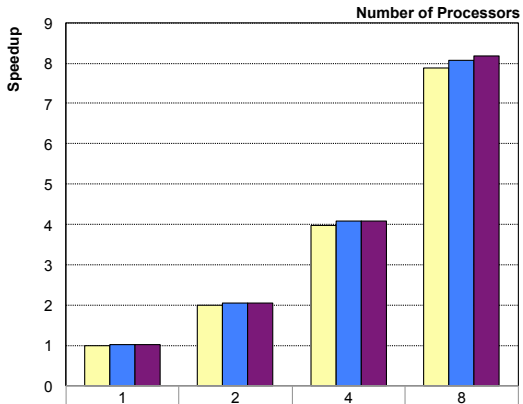
1. The “PARSEC pthreads” implementation is the original application provided by the PARSEC 2.1 benchmark suite. This implementation is compiled with pthreads support.

2. The “CGD mpi” implementation includes SPMD computations that are virtually unchanged PARSEC code snippets; datastructures and top level functions are slightly modified to fit into the CGD framework. This implementation uses the MPI CGD runtime.

3. The “CGD pthreads” implementation is identical to “CGD mpi”; however, it uses the pthreads runtime.

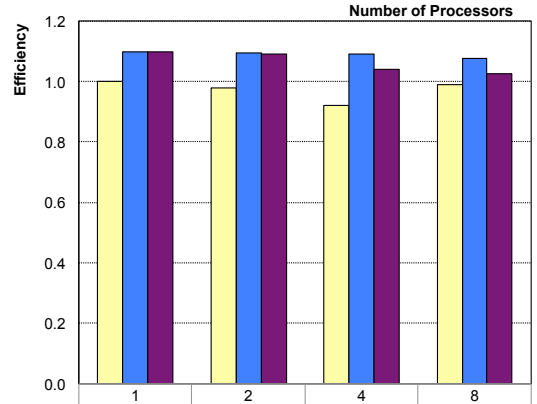


	1	2	4	8
PARSEC pthreads	1.00	1.00	1.00	0.99
CGD pthreads	1.03	1.02	1.02	1.01
CGD mpi	1.02	1.02	1.02	1.02

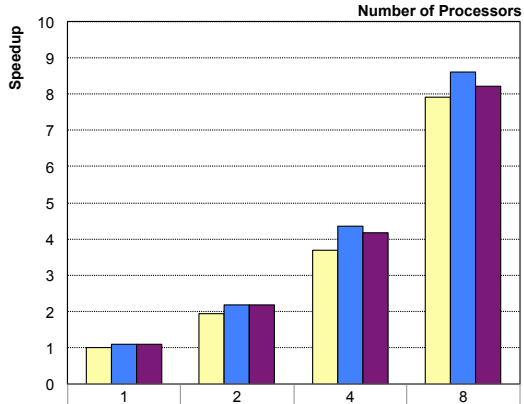


1	2	4	8
1.00	2.01	3.98	7.89
1.03	2.04	4.08	8.07
1.02	2.05	4.08	8.17

Figure 4.14: Efficiency and speedup for Swaptions largesim on Opteron SMP



	1	2	4	8
PARSEC pthreads	1.00	0.98	0.92	0.99
CGD pthreads	1.10	1.09	1.09	1.08
CGD mpi	1.10	1.09	1.04	1.03



1	2	4	8
1.00	1.96	3.68	7.91
1.10	2.19	4.37	8.62
1.10	2.18	4.16	8.20

Figure 4.15: Efficiency and speedup for Blackscholes largesim on Opteron SMP

Figure 4.14 shows that on the Opteron SMP machine, efficiency is very good for both PARSEC and CGD; similarly, speedup is almost linear. This application is computationally intensive, runs on a small machine with fast inter-processor communication, and therefore features a low communication overhead and good scalability. CGD single-processor performance exceeds PARSEC by only 2% (Table 4.4), and this gain is maintained for all processor counts.

Compared to PARSEC, the CGD Swaptions implementation has both a simpler structure, and a better performance portability. While the PARSEC implementation mixes pthreads, Intel Threading Building Blocks (TBB), and OpenMP function calls in the top-level functions, the CGD code describes parallelism in a simple top-level dataflow function. Furthermore, the CGD code can be compiled with the MPI, pthreads, and SHMEM runtimes without any code changes. The PARSEC implementation runs only on CC-SAS machines, while the CGD code has better portability, running on both CC-SAS and message-passing cluster machines.

4.2.5 Black-Scholes

This kernel is provided by the PARSEC benchmark suite, and calculates the value of a portfolio of European options relying on the Black-Scholes partial differential equation:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \tag{4.3}$$

The option prices are computed numerically, since there is no closed-form expression for the Black-Scholes equation [BL09]. This is a computation-bound kernel typical for computational finance applications.

Similar to the Swaptions application from Section 4.2.4, the Black-Scholes performance analysis compares both the MPI and pthreads CGD implementations, and the original PARSEC 2.1 implementation. Moreover, for the same reasons stated in Section 4.2.4, the CGD implementation has a simpler parallel structure and a better performance portability compared to PARSEC.

On the Opteron SMP machine, the single-processor runtime of CGD is faster than PARSEC by approximately 10% (Table 4.4); this improvement is partly explained by the more efficient use of CGD distributed datastructures.

The Black-Scholes speedups are roughly linear for both PARSEC and CGD (Figure 4.15).

Table 4.5: Single-processor runtime for Barnes-Hut

No. Bodies	Language	Implementation	Machine	Runtime (seconds)
32K	C	SPLASH2, pthreads	Opteron SMP	10.86
	C++	CGD, pthreads	Opteron SMP	10.39
32K	C	SPLASH2, pthreads	Altix 4700	37.06
	C++	CGD, shmem	Altix 4700	30.87
256K	C	SPLASH2, pthreads	Altix 4700	390.74
	C++	CGD, shmem	Altix 4700	362.59
1M	C	SPLASH2, pthreads	Altix 4700	1728.03
	C++	CGD, shmem	Altix 4700	1801.40

Efficiency remains largely constant, and "CGD pthreads" maintains the 10% performance gain vs. "PARSEC pthreads" for all processor counts. The good overall performance of all implementations is accounted for by the small communication-computation ratio of this computationally intensive application.

4.2.6 Barnes-Hut N-Body Simulation

This benchmark implements the Barnes-Hut hierarchical n-body algorithm; during each iteration, it computes the interactions between every particle and selected particles or groups of particles contained by an adaptive subdivision spatial tree (Section 2.3.3).

Parallelizing the Barnes-Hut algorithm exhibits irregular fine-grain communication when accessing the nodes of the distributed octotree, and requires constantly repartitioning the tree among processors to balance the computational load. The irregular access pattern of this application makes it a good candidate for the shared memory model, while message passing implementations need to rework the algorithm to aggregate data access into larger chunks. Sections 2.3.3 and 3.6.3 present these issues and the CGD Barnes-Hut implementation in more detail.

This section evaluates the performance of the CGD and SPLASH2 Barnes-Hut implementations for different problem sizes. The following versions are compared:

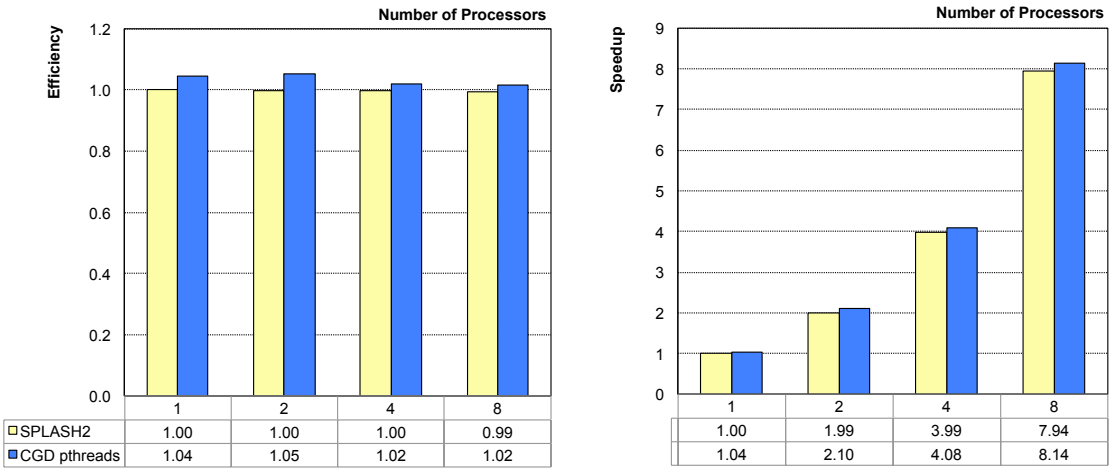
1. The "SPLASH2" implementation is the original Barnes-Hut application provided by the SPLASH2 benchmark suite, and includes the University of Delaware patches. This version is compiled with pthread support, using pthread barriers for global synchronization.

2. The "CGD pthreads" implementation was derived from the SPLASH2 implementation, sharing with it most of the sequential C code. However, a few essential modifications were made: (i) tree nodes are accessed via a distributed datastructure API rather than directly by pointers; (ii) cell nodes encode children as 4-byte indexes rather than as 8-byte pointers; and (iii) the tree partitioning algorithm is slightly modified, such that the replicated tree top contains only cell nodes. Otherwise, the algorithm remains unchanged, and the results match accordingly. This experiment uses the CGD pthreads runtime.

3. The "CGD shmem" implementation is identical to "CGD pthreads"; however, it uses the CGD SHMEM runtime.

The CGD adaptation of the SPLASH2 algorithm relies on global domain access during the force computation, tree building, and score aggregation steps (Section 3.6.3). Accessing remote tree cells and leafs translates into direct memory access for the pthreads version, and *shmem_get* calls for the SHMEM version. The CGD scheduler ensures that there is at least one barrier between the point where a datastructure distribution pair is created using a global distribution argument, and the point where it is consumed.

Table. 4.5 shows the single-processor runtimes measured for the SPLASH2 and CGD implementations for both machine configurations; all speedups presented in this section, except for those in Table 4.6, are based on the SPLASH2 runtimes. For all ex-



	1	2	4	8
SPLASH2	1.00	1.00	1.00	0.99
CGD pthreads	1.04	1.05	1.02	1.02

1	2	4	8
1.00	1.99	3.99	7.94
1.04	2.10	4.08	8.14

Figure 4.16: Efficiency and speedup for Barnes-Hut 32K on Opteron SMP

periments, the first four main loop iterations were not measured to avoid cold start noise.

Opteron SMP The CGD experiments use only the CGD pthreads runtime for this machine configuration. Figure 4.16 reveals that both CGD and SPLASH2 implementations have good efficiencies, even for eight processors. The sequential CGD C++ code is a bit faster (Table 4.5), leading to an increase in efficiency and speedup of about 4%.

Overall, the SPLASH2 and CGD pthreads implementations have both very similar and solid results on the small SMP machine. The next paragraph presents how this problem scales on larger ccNUMA machines.

Altix 4700 On this machine, we measure the CGD implementation performance for the pthreads and SHMEM CGD runtimes. CGD improves the SPLASH2 sequential performance by 20% for the smallest problem size (Table 4.5), mostly due to the new distributed tree implementation, which uses smaller indexes rather than pointers; this advantage is lost for larger problems that are dominated by numerical computation time. Consequently, the speedup comparison gives an unfair advantage to CGD by adding the benefit of a faster sequential code to absolute speedup.

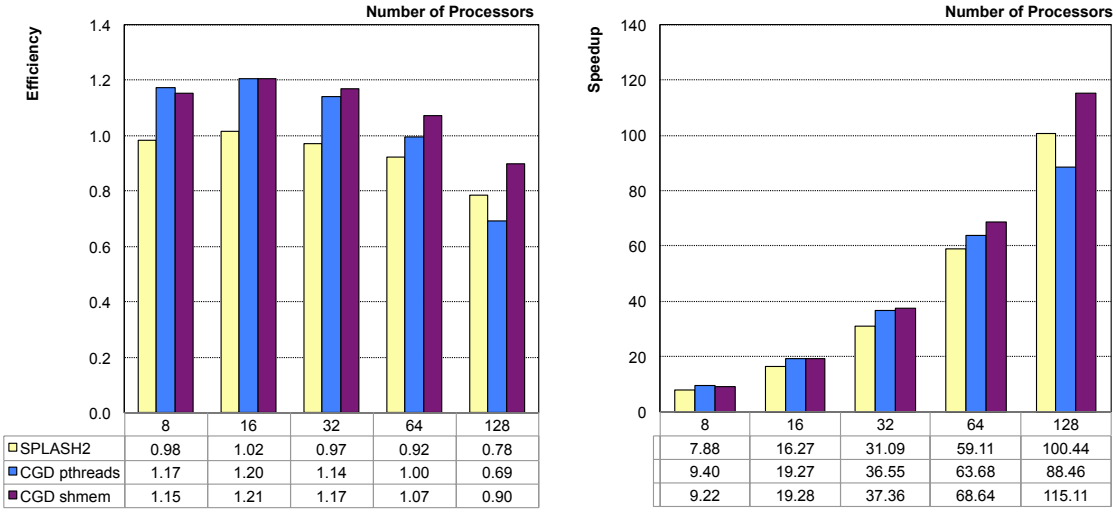
Table 4.6: Relative and absolute speedup for Barnes-Hut on Altix 4700

No. Bodies	Implementation	Sequential Runtime (seconds)	Speedup			
			64 proc		128 proc	
			relative	absolute	relative	absolute
32K	SPLASH2	37.06	59.11		100.44	
	CGD	30.87	57.17	68.64	95.88	115.11
256K	SPLASH2	390.74	64.59		125.88	
	CGD	362.59	74.79	80.60	143.37	154.50
1M	SPLASH2	1728.03	64.90		128.03	
	CGD	1801.40	83.95	80.53	165.10	158.38

Table 4.6 shows both the absolute speedups based on the SPLASH2 single-processor runtime, and the relative speedups based on the single-processor runtime of each implementation. Unfortunately, the relative speedup comparison is also not fair: the implementation with the faster sequential code includes less computation but pays the same constant communication cost as the implementation with the slower sequential code. Hence, relative scalability becomes a harder problem for the implementation with the faster sequential code. Not only are speedups compounded with sequential code performance, but also, for larger problems they are also compounded with the computation speed boost due to the larger aggregate multi-processor cache. These issues are reviewed and discussed later in this section.

Both CGD versions have a very similar performance for the larger problems (Figs. 4.18 and 4.19); however, for the 32K problem, "CGD shmem" improves the speedup of "CGD pthreads" by nearly 30% for 128 processors (Fig. 4.17). This reveals that for the smallest problem size, the pthreads latency advantage vs. SHMEM is offset by the slower pthreads synchronization. Moreover, this shows that SHMEM one-sided communication is almost as fast as direct memory access, which is an expected result given that SHMEM is implemented as a thin layer on top of CC-SAS memory.

Figure 4.17 shows that for the smallest problem size, "CGD shmem" improves the



	8	16	32	64	128
SPLASH2	0.98	1.02	0.97	0.92	0.78
CGD pthreads	1.17	1.20	1.14	1.00	0.69
CGD shmem	1.15	1.21	1.17	1.07	0.90

8	16	32	64	128
7.88	16.27	31.09	59.11	100.44
9.40	19.27	36.55	63.68	88.46
9.22	19.28	37.36	68.64	115.11

Figure 4.17: Efficiency and speedup for Barnes-Hut 32K on Altix 4700

SPLASH2 speedup by 15% on 128 processors, although having a steeper efficiency drop between eight and 128 processors (1.15 to 0.90 vs. 0.98 to 0.78). The same observation is confirmed by Table 4.6, where the absolute CGD speedup is greater than the SPLASH2 speedup, while the relative speedup is less (115.11 and 95.88 vs. 100.44). As previously discussed, these numbers show how a shorter sequential computation can lead to greater scalability limitations while still achieving a better overall runtime. However, both implementations are highly competitive, exceeding a relative speedup of 95 for a small problem running on 128 processors.

The CGD speedups reach super-linear territory at 16 processors for all problem sizes, ranging from 19.28 to 20.64 (Figs. 4.17, 4.18, and 4.19). While efficiency is relatively flat for all Barnes-Hut experiments, there is a small efficiency bump at 16 processors due to the larger multi-processor aggregate cache size. For fewer processors, efficiency deceases due to cache misses, whereas for more processors, efficiency slowly decreases as the number of processors increases and the communication-computation ratio increases: i) 1.21 to 0.90 (32K for 16 to 128 processors); ii) 1.29 to 1.21 (256K for 16 to 128 processors); iii) 1.26 to 1.21 (1M for 32 to 128 processors).

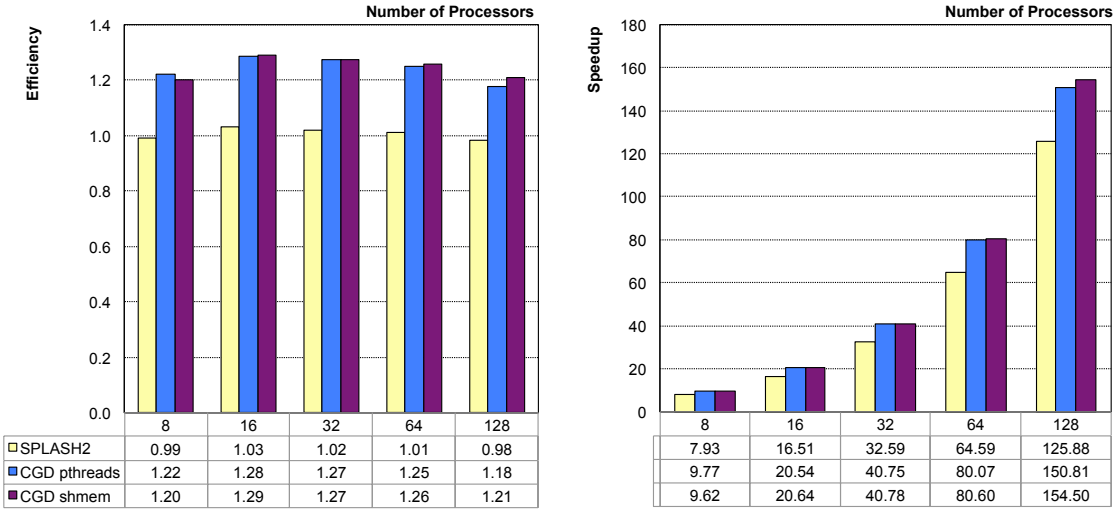
For the larger 256K and 1M problem sizes, "CGD shmem" outperforms SPLASH2 in terms of both absolute and relative speedup (Table 4.6). E.g., CGD improves the SPLASH2 absolute speedup by 22%, and relative speedup by 14% on 128 processors (154.50 and 143.37 vs. 125.88).

The improved CGD scalability for larger Barnes-Hut problems can be explained by the distributed octotree implementation. As described in Section 2.3.3, there are two CGD optimizations that apply to this problem: first, remote element reads are cached in local memory, avoiding repeated communication when data stops fitting into the processor cache; and second, writes are clumped together into larger batches to improve throughput. Additionally, remote element access translates into block reads for the entire size of a cell or leaf node, therefore increasing the granularity of remote memory access.

Similar results were reported by [SS99], where a comparison of CC-SAS, SHMEM, and MPI Barnes-Hut implementations shows that the SHMEM implementation improves the CC-SAS implementation results for the 1M problem running on a 64-processor SGI Altix 2000. The SHMEM and MPI implementations use a slightly modified algorithm that builds the essential tree for each processor during a communication step, and then it proceeds to the force computation step. While the SHMEM communication step sends more data than CC-SAS, its force calculation accesses only local memory.

For the largest problem size, the tree building step is dominated by the force computation, and the latter is significantly faster for SHMEM and MPI. The time breakdowns show that CC-SAS memory access takes longer than SHMEM and MPI local memory access, and it is argued that accessing the remote top of the tree nodes during the force computation leads to TLB misses, thereby hurting performance. A new CC-SAS version that replicates the top of the tree shows good improvements, matching the SHMEM and MPI performance.

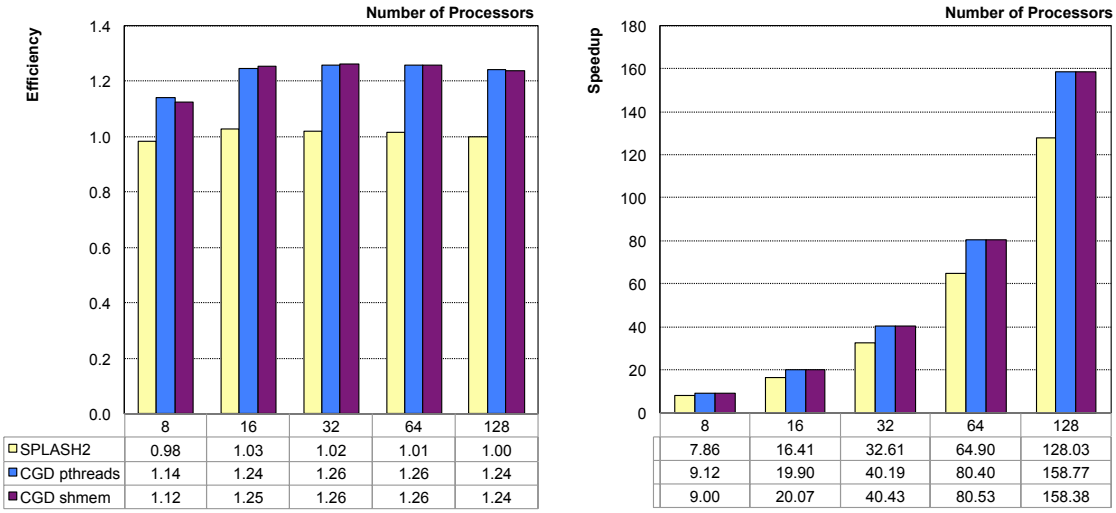
Maintaining access locality by replicating the tree top and caching the locally accessed



	8	16	32	64	128
SPLASH2	0.99	1.03	1.02	1.01	0.98
CGD pthreads	1.22	1.28	1.27	1.25	1.18
CGD shmem	1.20	1.29	1.27	1.26	1.21

8	16	32	64	128
7.93	16.51	32.59	64.59	125.88
9.77	20.54	40.75	80.07	150.81
9.62	20.64	40.78	80.60	154.50

Figure 4.18: Efficiency and speedup for Barnes-Hut 256K on Altix 4700



	8	16	32	64	128
SPLASH2	0.98	1.03	1.02	1.01	1.00
CGD pthreads	1.14	1.24	1.26	1.26	1.24
CGD shmem	1.12	1.25	1.26	1.26	1.24

8	16	32	64	128
7.86	16.41	32.61	64.90	128.03
9.12	19.90	40.19	80.40	158.77
9.00	20.07	40.43	80.53	158.38

Figure 4.19: Efficiency and speedup for Barnes-Hut 1M on Altix 4700

remote tree nodes is automatically provided by the CGD runtime and compiler without user involvement (Section 3.6.3). The CGD model easily allows such optimizations since globally addressable distributed datastructure arguments are guaranteed to be either read-only or write-only (Section 3.5.1).

Chapter 5

Cost Model

Building scalable parallel applications remains a difficult task, owing to implementation complexity, inability to easily understand costs and bottlenecks, and unpredictable communication performance. The machine architecture further influences both application efficiency and performance transparency. An accurate and general cost model may become a bridge between application development and machine design; such a model would allow application writers to optimize their algorithms and datastructure decompositions, while helping system designers build interconnects and tailor bandwidth capacity to application needs.

CGD datastructure redistribution operations, and in general, scientific computing codes, are developed using custom-built collective communication primitives; these can be satisfactorily analyzed using the family of Bulk Synchronous Parallel (BSP) machine models [Val90, BFPP01]. However, modeling the effect of bandwidth limitations for globally unbalanced communication and estimating the hierarchical bandwidth used by applications remain key challenges. This chapter introduces a hierarchical bandwidth machine model ($\alpha DBSP$) that naturally extends the Decomposable BSP (DBSP) model by associating a bandwidth growth factor α to each collective communication pattern.

The chapter is organized as follows. First, we discuss the main contributions of the $\alpha DBSP$ model in the context of BSP-family models in Section 5.1. Section 5.2 summarizes related work and presents the meaning of the h, l, and g parameters of BSP-family models in more depth. Section 5.3 provides a more precise definition of DBSP and introduces the αDBSP model. In Section 5.4, we present two routing algorithms that are later used to prove the upper bounds on executing (h, α)-relations on various topologies. Section 5.5 analyzes three computation kernels that have been selected to illustrate the differences between αDBSP and DBSP in terms of time complexity and analysis effort. Finally, Section 5.6 summarizes the results presented in this chapter.

5.1 Model Overview

Cost Model Implementing applications that rely on structured collective communication steps is a common trend in the scientific community [BN]. This programming paradigm uses custom-built data redistribution libraries, making large-scale codes easier to write and maintain. At the same time, this programming model allows the analysis of application performance using the Bulk Synchronous Parallel (BSP) machine model introduced by Valiant [Val90], or the Decomposable BSP (DBSP) machine extension that models submachine locality [dlTK96, Nic06].

On the BSP machine, a globally synchronized superstep, which includes w work and an h-relation, runs in $w+hg+l$ time. Hence, an h-relation is a message exchange in which each processor sends and receives at most h packets, while l and g are machine parameters that correspond to global latency and bandwidth. The upper bounds of these parameters are computed for theoretical topologies by solving a routing problem using store and forward switches [Val90]. Experimental results have the capacity to approximate the same parameters for real machine configurations [HMS+98, BJvOR03]. The DBSP model extends BSP by executing synchronized supersteps on predefined partitions of processors [dlTK95, dlTK96, Ber99]. This model exploits locality by computing

its parameters for each machine size i; running an i-superstep takes $w + hg(i) + l(i)$ time. Similar to BSP, DBSP is an effective but more accurate tool for cost analysis.

While DBSP overestimates the runtime of unbalanced h-relations, the results are improved by routing each h-relation as $O(\log(p))$ supersteps. A routing algorithm presented in [HPP01] achieves optimal results for (k_1, k_2)-routing problems[1] running on DBSP($n, \mathrm{g}^{(\alpha)}, \mathrm{l}^{(\beta)}$) machines such as nD meshes. It follows as a corollary that (h, m)-relations[2] can be routed in $O(\lceil m/n \rceil^{\alpha} h^{1-\alpha} n^{\alpha} + n^{\beta})$ time on such machines, matching the results of the Extended BSP [JW96] without adding a new machine primitive [BFPP01, FPP06, BPP07]. While this bound is optimal when considering all routing problems that are (h, m)-relations, there remain problems in which DBSP overestimates the runtime. E.g., the 2D nearest-neighbor single-packet exchange executes in constant time on a 2D mesh and $O(\log(p))$ time on a pruned butterfly, yet DBSP estimates its runtime as $O(\sqrt{p})$ for both topologies.

This chapter presents a hierarchical bandwidth machine model (αDBSP) [SS12] that naturally extends the DBSP model. A new exponential factor α is augmented to each h-relation, describing its growth in terms of required hierarchical bandwidth, i.e., $H(i) \leq \frac{1}{2^{(k-i-1)\alpha}} h$, where $H(i)$ roughly estimates the number of packets per processor reaching machine level i (Definition 10).

On αDBSP, a superstep including w work and an (h, α)-relation runs in $w + hg(i, \alpha) + l(i, \alpha)$ time, where $l(i, \alpha)$ and $g(i, \alpha)$ are the new machine parameters. We present two routing algorithms based on (k_1, k_2)-routing [HPP01], which improve the (h, α)-relation execution upper bound compared to DBSP h-relations. The modified algorithms redistribute fewer packets and compute the execution time using an upper bound on the number of packets traveling up to each level in the machine hierarchy.

The αDBSP model is at least as powerful as the DBSP model; any DBSP algorithm pro-

[1] A (k_1, k_2)-routing is a routing instance where each processor sends at most k_1 packets and receives at most k_2 packets

[2] A (h, m)-relation is a routing instance where each processor sends and receives at most h packets, and the total number of packets is m

vides identical results on αDBSP when h-relations are replaced with $(h, 0)$-relations. Additionally, there are unbalanced h-relations for which αDBSP provides better results. For example, for the 2D nearest-neighbor exchange αDBSP gives $\Omega(\log(p))$ and $O(\log^3(p))$ on the pruned butterfly, improving the DBSP (k_1, k_2)-routing upper bound of $O(\sqrt{p})$. Similarly, for the North-South 1D FFT problem (Section 5.5.2) αDBSP requires $O(\log^4(p))$, whereas DBSP requires $O(\sqrt{p})$. Section 5.5 presents a cost analysis for the three more general computation kernels commonly included in scientific codes.

The αDBSP results on pruned butterflies are relevant today if we consider that most modern large-scale supercomputers have their switches organized as meshes or partial fat-trees. Large Infiniband deployments such as the Nebulae supercomputer[3] and higher-end cache-coherent machines such as the LRZ SGI Altix 4700[4] employ meshes or federations of fat-trees at the top of their router hierarchy. For the class of machines that rely on such fat-trees, the αDBSP model brings a significant improvement when analyzing common scientific computing kernels such as the 2D nearest-neighbor exchange, or multi-pole FFT filtering.

αDBSP slightly increases analysis effort by adding an α factor to each h-relation to improve the analysis accuracy. This trade-off may lead to a faster runtime, or to a simpler cost analysis.

For the generalized broadcast problem, αDBSP requires $O(np)$, whereas DBSP requires $O(np\sqrt{p})$ on the pruned butterfly. Hence, the (k_1, k_2)-routing DBSP algorithm runs this problem as fast as αDBSP, but executes $O(\log(p))$ supersteps, making the analysis more complex on arbitrary machines. Based on the examples from Section 5.5, αDBSP provides a reasonable balance between cost accuracy and analysis effort, improving the DBSP results without increasing difficulty.

[3]TOP 500 supercomputer number four at the time of writing, 120,640 nodes
[4]Munich University supercomputing center, 9,728 nodes

System Design Architectural advances over the years have led to a relatively standardized modern parallel computer design. Fat-tree and mesh interconnects have a high bandwidth and are easy to build, being commonly used with Myrinet, Infiniband, NumaLink, and SeaStar switches [Lei85]. While the advent of wormhole routing has made the in-transit time insignificant compared to end-point overhead, bandwidth capacity planning remains an important issue [NGM97].

Unfortunately, bandwidth availability may vary dramatically at various levels of the memory-network hierarchy [FPP06, DDHM08]. For example, inter-core bandwidth is higher than intra-socket bandwidth, which is higher than the networking chip bandwidth. Moreover, large-scale machines are unable to provide full bisection-width bandwidth[5] at the top of their router hierarchy[6].

This unbalance becomes more prevalent as more cores are packed into a box, and system designers are forced to make a trade-off between the number of cores, per-core bandwidth, and switch size. When thousand-core chips become available, we can expect the bandwidth gap to become even larger.

It is possible to build parallel machines that avoid inherent bandwidth bottlenecks for a class of applications without requiring a full bisection-width bandwidth. An αDBSP application exposes communication as (h, α)-relations, and its α bandwidth growth factor allows the computation of a lower bound on the required hierarchical bandwidth and link capacity. E.g., a 2D PDE solver that employs FFT filtering at the top and bottom of its grid has $\alpha = 1/2$. A large-scale machine organized as a fat-tree with $\alpha_M = 1/2$, or a 2D mesh are able to execute this problem without an inherent bandwidth bottleneck (Section 5.5).

[5] Minimum aggregate bandwidth linking two equal partitions of the interconnect

[6] The LRZ SGI Altix 4700 machine has nodes organized as a mesh of fat-trees; the RedStorm Cray XT4 system is a 3D torus

5.2 Related Work

The PRAM model provides a framework for algorithm analysis extending the RAM concept to parallel architectures by assuming constant latency remote memory access. The technical infeasibility of this model has been addressed by PRAM variants that aim to adapt the cost model to real machine performance [HR91].

First in a family of bridging models, Valiant's BSP [Val90] describes a machine that executes synchronized supersteps consisting of local computation, global communication, and synchronization. The BSP machine executes a superstep with w work and an h-relation in $w + gh + l$ time, where the latency and bandwidth machine parameters l and g are computed globally without modeling locality.

The LogP model proposed by Culler et al. relaxes the synchronization requirement, and models interconnect performance more accurately [CKP+93, AISS95]. The fine-grain cost model accounts for the local bandwidth limitation G, global interconnect capacity L, and local overhead incurred by sending and receiving packets o. The efficiency of LogP implementations may be higher than BSP, in part due to its asynchronicity; however, the simplicity of BSP may be desirable. The two models are equivalent in terms of complexity [BHP+96].

The BSP machine overestimates the execution time for globally unbalanced h-relations. The EBSP model addresses this issue by adding a new (M, k_1, k_2)-relation that incorporates the total number of packets, as well as the maximum number of sent and received packets [JW96]. While this machine improves the BSP results, it increases analysis effort by adding a relatively complex primitive.

The Decomposable BSP machine is a BSP extension that captures submachine locality by executing supersteps on predefined partitions of processors [dlTK95, dlTK96, BPP07]. The l and g parameters are vectors whose elements correspond to each partition size; these costs become smaller with a decreasing partition size.

If DBSP overestimates the runtime of unbalanced h-relations, the results are improved by routing each relation as $O(\log(p))$ supersteps using the DBSP (k_1, k_2)-routing algorithm from [HPP01]. Subsequently, on DBSP($n, \mathbf{g}^{(\alpha)}, \mathbf{l}^{(\beta)}$) machines (h, m)-relations take $O(\lceil m/n \rceil^{\alpha} h^{1-\alpha} n^{\alpha} + n^{\beta})$ routing time, matching the results obtained by EBSP [BFPP01, FPP06].

While this bound is optimal when considering all routing problems within a class, the solution is not optimal for all such problems. There are unbalanced routing problems for which αDBSP improves the DBSP bounds, or allows a simpler analysis (Section 5.5). Furthermore, the αDBSP machine abstraction allows for more than an estimation of time complexity; for tight relations, the α factor provides a lower bound on the required hierarchical link bandwidth, which is useful for capacity planning.

5.3 Definitions

Applications that rely on structured collective communication and computation steps may be analyzed using the Bulk Synchronous Parallel (BSP) family of machine models. The Decomposable BSP (DBSP) model is an extension of BSP that exploits locality by relying on submachine decomposition [dlTK95]. In this dissertation, we consider only recursive DBSP machines that allow recursive binary machine decompositions (Fig. 5.1) [dlTK96].

It is assumed that $p = 2^k$ is the total number of processors of a DBSP machine. For each $0 \leq i \leq k$, the machine is partitioned into 2^i disjoint i-clusters $C(i, j)$ containing processors $j2^{k-i}$ to $(j+1)2^{k-i}-1$. Note that each $C(i, j)$ cluster has 2^{k-i} processors, and the clusters form a binary decomposition hierarchy, i.e., $C(i, j) = C(i + 1, 2j) \cup C(i + 1, 2j + 1)$.

A DBSP machine is parametrized by two vectors $g(i)$ and $l(i)$, where $0 \leq i \leq k$. These parameters describe the bandwidth and latency characteristics of submachines, i.e., a

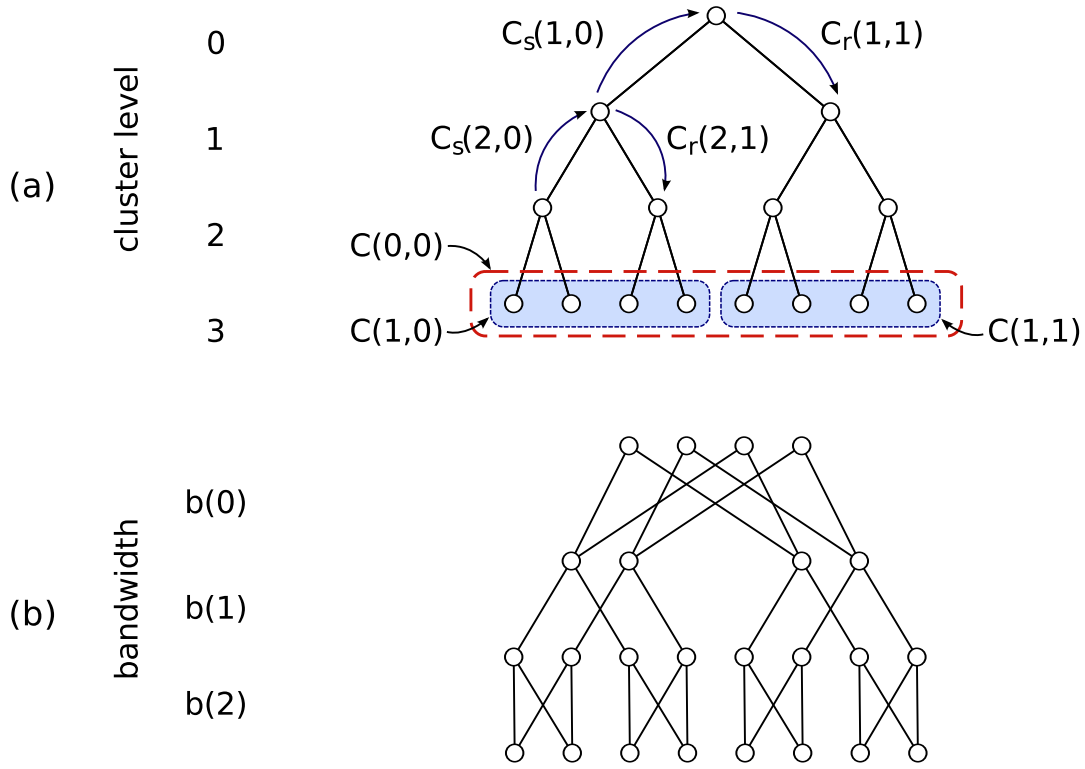


Figure 5.1: (a) DBSP cluster hierarchy for a 16 processor fat-tree, and top-level $C_{s/r}(i,j)$ packet sets; (b) Pruned butterfly implementing a fat-tree with link bandwidths $b(i) = 2^{\lceil k-i \rceil \alpha_M}$ for $\alpha_M = 1/2$.

h-relation is executed on a $C(i,j)$ cluster in $hg(i) + l(i)$ time; a h-relation is a routing problem where each processor sends and receives at most h packets.

A DBSP program is executed as a sequence of labeled supersteps. An i-superstep is executed in parallel on $C(i,j)$ clusters and takes $w + hg(i) + l(i)$ time, where w is the maximum number of operations or work performed by each processor during this superstep [BFPP01].

The above definitions of p, k, $g(i)$, $l(i)$, and $C(i,u)$ will be used throughout the rest of the dissertation.

Definition 7. **R**-*routing is a routing problem where processor* x *sends* $R(x, y)$ *packets to processor* y*.*

Definition 8. *For a* **R**-*routing problem,* $C_s(i, u)$ *is defined as the set of packets sent by the processors of cluster* $C(i, u)$ *to all other processors, and* $C_r(i, u)$ *is the set of packets received by the processors of cluster* $C(i, u)$ *from all other processors. These parameters are represented by the following equations*

$$|C_s(i,u)| = \sum_{x \in C(i,u),\ y \notin C(i,u)} R(x,y)$$

$$|C_r(i,u)| = \sum_{x \notin C(i,u),\ y \in C(i,u)} R(x,y)$$

A DBSP h-relation may be generalized by computing a value $H(i)$ corresponding to each level i of the machine hierarchy. $H(i)$ represents the maximum number of packets sent to or from any i-cluster divided by the number of processors in that cluster. The **H** vector is used as a tool for cost analysis rather than as a machine primitive.

Definition 9. *An* **R**-*routing problem is an* **H**-*relation if*

$$H(i) = \frac{1}{2^{k-i-1}}\ max_{0 \le u < 2^{i+1}}\ \{|C_s(i+1,u)|,\ |C_r(i+1,u)|\}$$

for all $0 \le i < k$

Next, an exponential factor α is defined to capture the growth of **H**. The $\log(p)$ values of vector **H** are compressed to a single value pair (h, α) that enforces an upper bound on all $H(i)$.

Definition 10. *A* **H**-*relation executed during an* i-*superstep is a* (h, α)-*relation if*

$$H(j) \le \frac{1}{2^{(k-j-1)\alpha}}\ h$$

for all $i \le j < k$*.*

Note that for $\alpha = 0$ we have $H(i) = H(k-1)$, which means that all packets are routed all the way to the top of the hierarchy, while for $\alpha = 1$ we have $H(i-1) = H(i)/2$ which means that half of the packets reaching level i will be routed up one level to level $i-1$.

The α factor provides an upper bound on the number of packets sent to each level of the hierarchy, and it facilitates the computation of an execution upper bound. For example, this factor allows determining the λ load factor [Lei85] and its corresponding upper bound for a fat-tree with known bandwidths [BB95]. For some patterns, α also provides a lower bound on packet count. When this is the case, a lower bound on execution time may be proved by making a bandwidth argument.

Definition 11. *An algorithm executing an i-superstep* H*-relation is said to execute a tight (h, α)-relation if $\exists c > 0$ s.t. for any problem size and machine size we have*

$$c \frac{1}{2^{(k-j-1)\alpha}} h \leq H(j) \leq \frac{1}{2^{(k-j-1)\alpha}} h$$

for all $i \leq j < k$.

We define an αDBSP machine as an extension of an DBSP machine that executes (h, α)-relations during each i-superstep. The bandwidth $g(i, \alpha)$ and latency $l(i, \alpha)$ parameters are defined as a function of both machine size i, and bandwidth growth factor α.

Definition 12. *An αDBSP machine executes an i-superstep (h, α)-relation and w work in time*

$$T_{\alpha DBSP}(i, w, h, \alpha) = w + hg(i, \alpha) + l(i, \alpha)$$

where $g(i, \alpha)$, $l(i, \alpha)$ are the αDBSP machine parameters.

DBSP machines and algorithms are particular cases of αDBSP for $\alpha = 0$. Any DBSP machine is an αDBSP machine with $g(i, \alpha) = g(i)$ and $l(i, \alpha) = l(i)$ since any (h, α)-

relation is a h-relation; any DBSP algorithm becomes an αDBSP algorithm when h-relations are replaced with $(h, 0)$-relations.

Definition 13. *For a machine, $C_{bw}(i, u)$ is defined as the total bandwidth of the links connecting cluster $C(i, u)$ to any other cluster.*

Machines with symmetric topologies such as butterflies and meshes have the same number of links connecting any i-cluster to the rest of the machine. For these machines it is possible to estimate the bandwidth $C_{bw}(i, u)$ in terms of i and a bandwidth growth factor α_M.

Definition 14. *A machine has an α_M bandwidth growth factor if $\exists c > 0$ s.t. for any machine size we have:*

$$c\frac{1}{2^{(i-1)\alpha_M}}C_{bw}(1,0) \leq C_{bw}(i,u) \leq \frac{1}{2^{(i-1)\alpha_M}}C_{bw}(1,0)$$

for all $0 < i < k,\ 0 \leq u < 2^{i+1}$

Note that for fat-trees and meshes $C_{bw}(1, 0) = C_{bw}(1, 1)$ represents the bisection-width bandwidth. An nD mesh has a bandwidth growth factor of $\alpha_M = \frac{1}{n}$, while a pruned butterfly has a bandwidth growth factor of $\alpha_M = \frac{1}{2}$ (Table 5.1).

The next section analyzes communication cost and presents the αDBSP machine parameters for most common topologies.

5.4 Bounds

An upper-bound on (h, α)-relation execution time may be computed by routing the relation as a sequence of redistribution steps. This method is similar to the algorithm for (k_1, k_2)-routing [FPP03] that redistributes all packets evenly among all processors during a bottom up phase, then routes each packet to its destination during a top down phase.

The (h, α)-relation routing algorithm improves the (k_1, k_2)-routing by redistributing only the packets with destinations outside of a cluster during the bottom up phase, i.e., $C_s(i, j)$ for machines of size i. The number of packets that are redistributed at each level is upper-bounded by $\frac{h}{2^{(k-i)\alpha}}$ (Definition 10) providing a bandwidth cost of $O\left(\lceil H(i) \rceil \, g(i)\right)$.

Theorem 1. *An s-superstep* **H***-relation is executed by a DBSP machine in time*

$$O\left(\sum_{i=s}^{k-1} \lceil H(i) \rceil g(i) + \sum_{i=s}^{k-1} \left(l(i) + Pf(i)\right)\right)$$

where $g(i)$, $l(i)$ are the DBSP machine parameters; $Pf(i)$ is the parallel prefix execution time on an i-cluster.

Proof. Let $C_x(i, j)$ be the set of packets sent from $C(i+1, 2j)$ processors to $C(i+1, 2j+1)$ processors, and from $C(i + 1, 2j + 1)$ processors to $C(i + 1, 2j)$ processors. $C_x(i, j)$ are packets exchanged between the children of cluster $C(i, j)$ and travel up to level i in the hierarchy. We have $C_s(i + 1, 2j) \cup C_s(i + 1, 2j + 1) = C_x(i, j) \cup C_s(i, j)$ and $C_r(i + 1, 2j) \cup C_r(i + 1, 2j + 1) = C_x(i, j) \cup C_r(i, j)$.

It is assumed that n packets are evenly distributed among 2^a processors when each processor holds at most $\lceil \frac{n}{2^a} \rceil$ packets. The following algorithm may be applied to execute an **H**-relation:

1. For each $i = k-1$ downto s execute step (1a)

1a. For each processor cluster $C(i, j)$ with $0 \leq j < 2^i$ execute step (1b)

Invariant: $C_s(i + 1, x)$ and $C_x(i + 1, x)$ have packets evenly distributed among $C(i + 1, x)$ for all x.

1b. Redistribute evenly the packets from $C_s(i, j)$ among the processors of $C(i, j)$. Repeat the process for the packets from $C_x(i, j)$.

2. For each $i = s$ to $k-1$ execute step (2a)

2a. For each processor cluster $C(i,j)$ with $0 \leq j < 2^i$ execute step (2b)

Invariant: $C_r(i,x)$ have packets evenly distributed among $C(i,x)$ for all x.

2b. Evenly redistribute the packets from $C_r(i+1,2j)$ among the processors of $C(i+1,2j)$, and the packets from $C_r(i+1,2j+1)$ among the processors of $C(i+1,2j+1)$

The invariants after step (1a) and (1b) are easily verified by induction. During the bottom up phase the packets from the $C_x(i,j)$ sets are evenly distributed among $C(i,j)$ processors, level by level. Step (1b) may be executed as a parallel prefix and two $\lceil H(i) \rceil$-relations. Note that for $H(i) < 1$ execution time is still $g(i)$, even when $H(i)g(i)$ may be significantly smaller. □

Proof. Similarly, during the top down phase the packets from the $C_r(i,j)$ sets are distributed evenly, and step (2b) is executed as a parallel prefix followed by two $\lceil H(i) \rceil$-relations. The theorem follows after adding up the execution time for steps (1b) and (2b). □

The results from Theorem 1 may be improved on the pruned butterfly, in the event that redistributions of $n < p$ packets run faster than a 1-relation. Such redistributions are executed as $\left\lceil \frac{n}{\sqrt{p}} \right\rceil$ waves \cite{PBFLY} take $O\left(\left\lceil \frac{n}{\sqrt{p}} \right\rceil + \log(p)\right)$ time, improving the $O\left(\sqrt{p}\right)$ bound for a 1-relation.

Theorem 2. *An s-superstep* H*-relation is executed by a pruned butterfly in time*

$$O\left(\sum_{i=s}^{k-1} \lceil H(i)g(i) \rceil + \sum_{i=s}^{k-1} (l(i) + Pf(i))\right)$$

where $l(i) = k-i$, $g(i) = 2^{(k-i)\alpha_M}$, $\alpha_M = 1/2$, and $Pf(i)$ is the parallel prefix execution time on an i-cluster.

Table 5.1: Bounds on i-superstep (h, α)-relation execution time and αBSP parameters for common machine topologies with p processors. The relation is executed on a partition of 2^i submachines with $p' = p/2^i$ processors each, i.e., on all i-clusters. α_M is a constant defined for each machine type described in this table. The lower bounds apply only to tight relations.

Machine	α_M	(h, α)	DBSP	αDBSP		$g(i, \alpha)$	$l(i, \alpha)$
Butterfly	0	$\alpha = 0$	$O(h\,log(p'))$	$O(h\,log(p'))$	$\Omega(h + log(p'))$	$O(log(p'))$	$O(1)$
		$\alpha > 0$		$O(h + log^3(p'))$	$\Omega(h + log(p'))$	$O(log(p'))$	$O(log^3(p'))$
Pruned nD Butterfly	$1/2$	$\alpha < \alpha_M$	$O(h\,p'^{\alpha_M})$	$O(h\,p'^{\alpha_M - \alpha})$	$\Omega(h\,p'^{\alpha_M - \alpha})$	$O(p'^{\alpha_M - \alpha})$	$O(log(p'))$
		$\alpha = \alpha_M$		$O(h\,log(p') + log^3(p'))$	$\Omega(h + log(p'))$	$O(log(p'))$	$O(log^3(p'))$
	$1/n$	$\alpha > \alpha_M$		$O(h + log^3(p'))$	$\Omega(h + log(p'))$	$O(1)$	$O(log^3(p'))$
nD Mesh	$1/n$	$\alpha < \alpha_M$	$O(h\,p'^{\alpha_M})$	$O(h\,p'^{\alpha_M - \alpha} + p'^{\alpha_M})$	$\Omega(h\,p'^{\alpha_M - \alpha})$	$O(p'^{\alpha_M - \alpha})$	$O(p'^{\alpha_M})$
		$\alpha = \alpha_M$		$O(h\,log(p') + p'^{\alpha_M})$	$\Omega(h)$	$O(log(p'))$	$O(p'^{\alpha_M})$
		$\alpha > \alpha_M$		$O(h + p'^{\alpha_M})$	$\Omega(h)$	$O(1)$	$O(p'^{\alpha_M})$

Proof. The algorithm presented by Theorem 1 can be modified to prove Theorem 2. It is assumed that n packets are evenly distributed among 2^a processors when: i) for $n \geq 2^a$ each processor holds at most $\lceil \frac{n}{2^a} \rceil$ packets; and ii) for $n < 2^a$ processor x holds one packet if $x = 0 \ (mod\ 2^{a-m})$, otherwise it holds no packets, where $2^{m-1} < n \leq 2^m$.

Steps (1b) and (2b) are executed as $H(i)$-relations for $H(i) \geq 1$, and $\lceil H(i)/g(i) \rceil$ waves for $H(i) < 1$ [BB95].

Let's consider the redistribution of $C_s(i,j)$ during step (1b) for $H(i) < 1$. The $C_s(i,j)$ packets are evenly distributed i) by computing a parallel prefix for the placement of $C_s(i+1,2j) \cup C_s(i+1,2j+1)$ also in $C_s(i,j)$; ii) by subsequently routing these packets as waves to their $C(i,j)$ destinations.

The routing of packets $C_s(i+1,2j) \cap C_s(i,j)$ to processors $C(i,j)$ is considered. Cluster $C(i+1,2j)$ has 2^{k-i-1} processors; let $s = k-i-1$, and let's choose m s.t. $2^{m-1} < H(i)2^s \leq 2^m$.

Processors $x = 0 \, (mod\ 2^{s-m})$ from $C(i+1,2j)$ hold at most one packet since $C_s(i+1,2j)$ has at most $H(i)2^s \leq 2^m$ packets distributed among 2^s processors. Similarly, processors $x = 0 \, (mod\ 2^{s-m})$ from $C(i,j)$ hold at most one packet.

When $m \geq \lceil (k-i)/2 \rceil$ the routing is executed as $2^{m-\lceil (k-i)/2 \rceil}$ waves, with each wave j involving processors $x = j \, (mod\ 2^{s-m-\lceil (k-i)/2 \rceil})$; otherwise it is executed as a single wave. This gives a routing time of $O\left(\left\lceil 2^{m-\lceil (k-i)/2 \rceil} \right\rceil + (k-i)\right) = O\left(\lceil H(i)g(i) \rceil + l(i)\right)$.

The $C_s(i+1,2j+1) \cap C_s(i,j)$ packets are routed to $C(i,j)$ using the same algorithm. Overall, the redistribution of $C_s(i,j)$ during step (1b) takes $O(\lceil H(i)g(i) \rceil + l(i) + Pf(i))$ time. The same argument may be repeated for redistributing $C_x(i,j)$ during step (1b), and redistributing $C_r(i+1,2j)$, $C_r(i+1,2j+1)$ during step (2b). The theorem follows after adding up these terms. □

We may compute an upper bound on i-superstep (h,α)-relation execution time by inserting the machine parameters into Theorem 1 for the butterfly and mesh topologies,

and into Theorem 2 for the pruned butterfly topologies. The lower bounds may be deduced by making a bandwidth argument about the links connecting $i+1$ level machines and enforcing the minimum packet travel time within i level machines. Here, we briefly calculate the results shown in Table 5.1.

On a butterfly, for $\alpha = 0$ the h-relation BSP routing algorithm provides an execution time of $O(h(k-i))$. For $\alpha > 0$ Theorem 1 provides an execution time of

$$O\left(\sum_{j=i}^{k-1}\left(\frac{1}{2^{(k-j)\alpha}}h+1\right)(k-j)+\sum_{j=i}^{k-1}(k-j)^2+(k-i)^2\right)=O\left(h+(k-i)^3\right) \tag{5.1}$$

On a nD mesh with $\alpha_M = \frac{1}{n}$ the execution time is

$$\begin{aligned}
&O\left(\sum_{j=i}^{k-1}\left(\frac{1}{2^{(k-j)\alpha}}h+1\right)2^{(k-j)\alpha_M}+\sum_{j=i}^{k-1}2^{(k-j)\alpha_M}+n2^{(k-i)\alpha_M}\right)= \\
&O\left(\sum_{j=i}^{k-1}\frac{1}{2^{(k-j)\alpha}}\,h\,2^{(k-j)\alpha_M}+2^{(k-i)\alpha_M}\right)= \\
&O\left(h\sum_{j=i}^{k-1}2^{(k-j)(\alpha_M-\alpha)}+2^{(k-i)\alpha_M}\right)
\end{aligned} \tag{5.2}$$

Expression (5.2) evaluates to

$$O\left(h\,2^{(k-i)(\alpha_M-\alpha)}+2^{(k-i)\alpha_M}\right) \qquad \text{for } \alpha < \alpha_M \tag{5.3}$$

$$O\left(h(k-i)+2^{(k-i)\alpha_M}\right) \qquad \text{for } \alpha = \alpha_M \tag{5.4}$$

$$O\left(h+2^{(k-i)\alpha_M}\right) \qquad \text{for } \alpha > \alpha_M \tag{5.5}$$

On a pruned butterfly with $\alpha_M = \frac{1}{2}$ the run time is

$$O\left(\sum_{j=i}^{k-1}\left(\frac{1}{2^{(k-j)\alpha}}\,h\,2^{(k-j)\alpha_M}+1\right)+(k-i)^3+(k-i)^2\right)=$$

$$O\left(h\sum_{j=i}^{k-1}2^{(k-j)(\alpha_M-\alpha)}+(k-i)^3\right) \tag{5.6}$$

Similarly, (5.6) evaluates to

$$O\left(h2^{(k-i)(\alpha_M-\alpha)}\right) \qquad \text{for } \alpha<\alpha_M \tag{5.7}$$

$$O\left(h(k-i)+(k-i)^3\right) \qquad \text{for } \alpha=\alpha_M \tag{5.8}$$

$$O\left(h+(k-i)^3\right) \qquad \text{for } \alpha>\alpha_M \tag{5.9}$$

The pruned butterfly topology presented in [BB95] may be generalized by excluding one set of nodes from a regular butterfly every n levels (Fig.~2). The resulting topology has a bisection-width of $p^{(n-1)/n}$. While both an nD mesh and an nD butterfly have the same bisection width, the butterfly has $\log(p)$ maximum latency vs. $\sqrt[n]{p}$ for the mesh. Note that for $n=1$ the topology is a binary tree, for $n=2$ it is a pruned butterfly, and for $n=\infty$ it is a butterfly.

Definition 15. *An nD butterfly is a graph $G(V,E)$ that is a subgraph of a butterfly with $p=2^k$ processors. More precisely*

$$\begin{aligned}V&=\left\{\langle i,j,u\rangle \mid 0\le i\le k,\, 0\le j<2^i,\, 0\le u<2^m\right\}\\ E&=\Big\{\left(\langle i,j,u\rangle,\langle i+1,2j,u'\rangle\right),\left(\langle i,j,u\rangle,\langle i+1,2j+1,u'\rangle\right)\mid\\ &\qquad 0\le i<k,\, 0\le j<2^i,\, 0\le u<2^m\Big\}\end{aligned}$$

where $m=\left\lceil(k-i)\frac{n-1}{n}\right\rceil$, and $u'=u$ for $k-i=0\,(mod\ n)$; $u'=u-2^{m-1}\left\lfloor\frac{u}{2^{m-1}}\right\rfloor$ for $k-i\neq 0\,(mod\ n)$

An upper bound on executing an i-superstep (h,α)-relation on the nD butterfly may be computed by relying on a generalization of Theorem 2 where $\alpha_M=\frac{1}{n}$. The results from Table 5.1 follow as corollaries. Proving the generalization of Theorem 2 relies on the ability of nD butterflies to route m waves of $2^{\lceil k(n-1)/n\rceil}$ evenly distributed packets

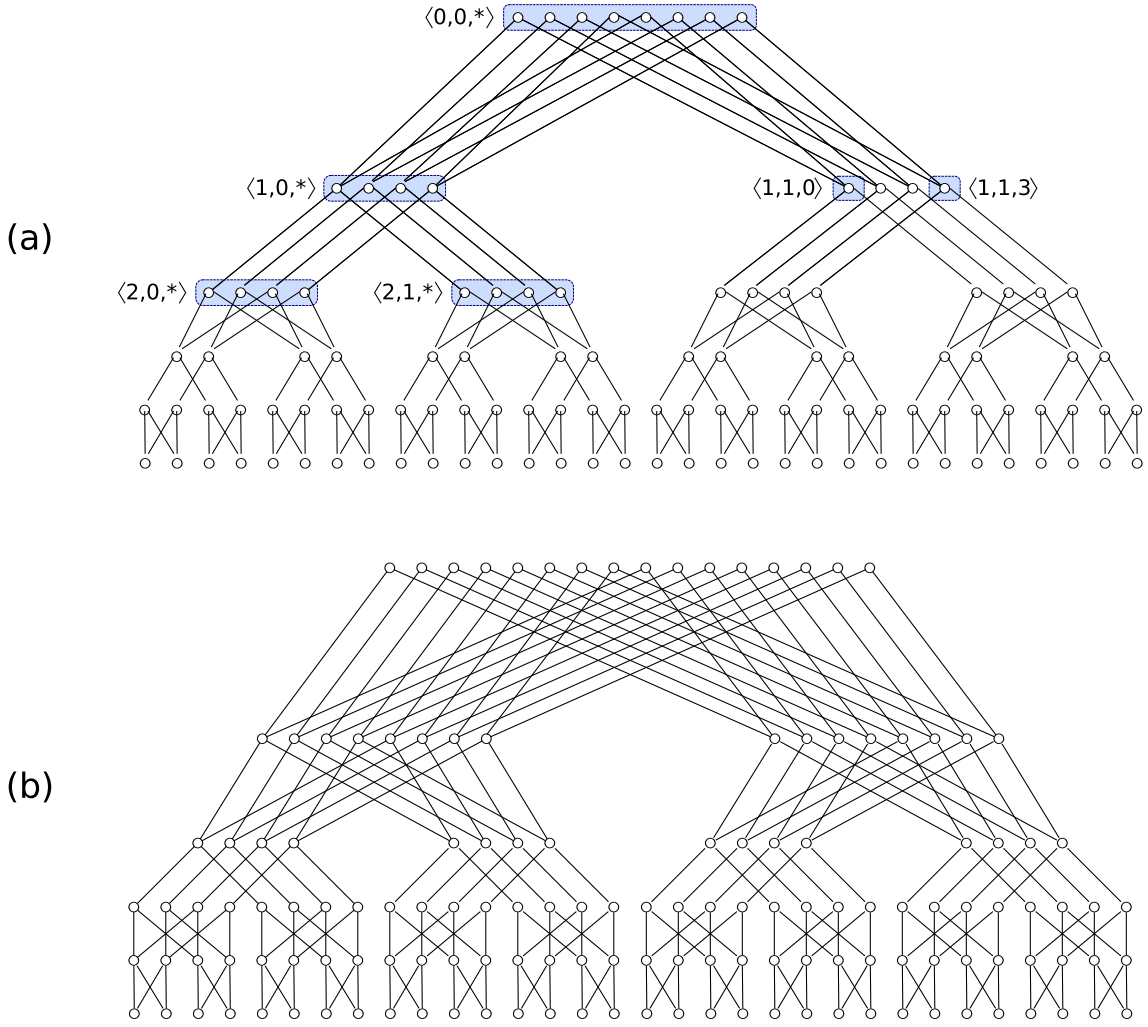


Figure 5.2: Pruned butterfly examples: (a) 2D butterfly for $p = 32$; (b) 3D butterfly for $p = 32$

in $O(m + \log(p))$ time. This property may be proved by building a routing algorithm similar to the pruned butterfly wave routing algorithm presented in [BB95].

5.5 Examples

This section compares the αDBSP model presented in this dissertation with DBSP using three simple kernels common in scientific computing applications.

Message patterns are executed on DBSP using two routing algorithms:

1. DBSP : an h-relation is executed as a single i-superstep
2. DBSP+ : a (k_1, k_2) K_{12}-routing problem is executed as $2(\log(p) - i)$ supersteps as described in [FPP03, BFPP01]

A (h, m) hm-routing problem is executed as a $(h, \lceil \frac{m}{p} \rceil)$ K_{12}-routing followed by another $(\lceil \frac{m}{p} \rceil, h)$ K_{12}-routing. To simplify this analysis we use the following notation to represent K_{12}-routing time:

$$K_{12}route(k_1, k_2, i) = O\left(\sum_{j=0}^{\log(p)-1-i} \left(min\{k_1, k_2 2^j\} + min\{k_1 2^j, k_2\})g(j+i) + l(j+i) + Pf(j+i)\right)\right) \quad (5.10)$$

where $Pf(i)$ is the time taken to execute a parallel prefix on an i-level machine.

Our examples are selected to illustrate the difference between the BSP family models in terms of time complexity and analysis effort (Table~\ref{tab:complex}). We decided to evaluate the results of the pruned butterfly since our focus is based on estimating the required hierarchical bandwidth. For the full bisection-width topologies the results are straight-forward, while for the meshes the bandwidth cost may be dominated by the latency cost.

5.5.1 Generalized Broadcast

The generalized broadcast algorithm sends n data elements from one processor to all other processors in a single step. This problem is an example of unbalanced communi-

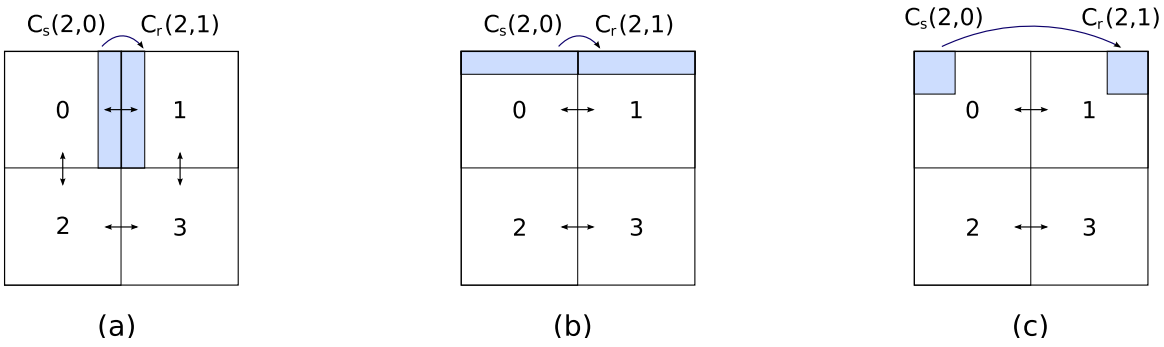


Figure 5.3: Message patterns corresponding to a $(n, \frac{\sqrt{p}}{2}n)$ hm-routing: (a) nearest-neighbor exchange between level 2 clusters; (b) North-South FFT for $s = \log(\sqrt{p}) - 1$; (c) $\frac{\sqrt[4]{p}}{\sqrt{2}} \times \frac{\sqrt[4]{p}}{\sqrt{2}}$ corner area exchange

cation for which αDBSP provides a result similar to K_{12} routing on DBSP+ and better than DBSP.

The routing problem is a np-relation for DBSP, a tight $(np, 1)$-relation for αDBSP, and a K_{12} (np, n)-routing for DBSP+. The αDBSP runtime is:

$$T_{\alpha BSP}(n) = np\, g(0,1) + l(0,1) \tag{5.11}$$

While for DBSP we have:

$$T_{DBSP}(n) = np\, g(0) + l(0) \tag{5.12}$$

$$\begin{aligned} T_{DBSP+}(n) &= K_{12} route(np, n, 0) \\ &= O\left(\sum_{i=0}^{\log(p)-1} \left(n(2^i + 1) g(i) + l(i) + Pf(i) \right) \right) \end{aligned} \tag{5.13}$$

On the pruned butterfly $g(i) = \sqrt{p}/2^{i/2}$, $l(i) = \log(p) - i$, $Pf(i) = (\log(p) - i)^2$, and $g(i,1) = 1$, $l(i,1) = (\log(p) - i)^2$ (Table 5.1). By entering these parameters we evaluate the execution time for the three machine models:

$$
\begin{aligned}
T_{\alpha BSP}(n) &= O\left(np + \log^2(p)\right) = O(np) && (5.14)\\
T_{DBSP}(n) &= O\left(np\sqrt{p} + \log(p)\right) = O(np\sqrt{p}) && (5.15)\\
T_{DBSP+}(n) &= O\left(\sum_{i=0}^{\log(p)-1} \left(n2^i\sqrt{p}\,2^{-i/2} + (\log(p) - i)^2\right)\right)\\
&= O\left(n\sqrt{p}\sum_{i=0}^{\log(p)-1} 2^{i/2} + \log^3(p)\right)\\
&= O\left(np + \log^3(p)\right) = O(np) && (5.16)
\end{aligned}
$$

As anticipated, the αDBSP model presented in this dissertation provides a cost analysis that is both simple and accurate for the generalized broadcast problem.

5.5.2 North-South FFT

Many scientific codes discretize the space using grids that have singularity points. For example, problems such as atmospheric and ocean simulations use bipolar or tripolar grids that are mapped to the surface of the Earth. In such instances, most numerical methods become unstable in the neighborhood of grid poles and use FFTs to filter undesired frequencies.

The North-South FFT problem computes an $n\sqrt{p}$ element FFT on the top and bottom rows of a 2D matrix. We summarize the algorithm for the top row FFT in terms of computation and communication steps:

1. At the beginning each top row processor is assigned n data elements
2. Each processor computes a local FFT that takes $n\ \log(n)$ computation time
3. For each $s = 0..\log(\sqrt{p}) - 1$ execute steps (4) and (5)
4. Processor x obtains n data elements from processor y, where x, y are top row processors and their binary representation is identical except for bit $2s$

5. Each top row processor recomputes its local n data elements taking n computation time

6. At the end the n data elements assigned to the top row processors contain the Fourier transform

The 2D processor grid may be recursively divided vertically, then horizontally to create a binary tree that is mapped to the hierarchical DBSP machine.

Each communication step (4) is executed up to level $i = \log(p) - 1 - 2s$. For every $0 \leq s' < s$ and intermediate level $i' = \log(p) - 1 - 2s'$ there are $2^{s'}$ packets sent from a size $2^{2s'}$ submachine to the higher levels. Step (4) has a growth factor of $\alpha = 1/2$ and represents a n-relation for DBSP, a $(n, 1/2)$-relation for αDBSP, and a $(n, n\sqrt{p})$ hm-routing for DBSP+.

Hence, the αDBSP runtime is:

$$T_{\alpha BSP}(n) = n\ \log(n\sqrt{p}) + \sum_{s=0}^{\log(p)/2-1} (ng(2s+1, 1/2) + l(2s+1, 1/2)) \tag{5.17}$$

DBSP is represented by:

$$T_{DBSP}(n) = n\ \log(n\sqrt{p}) + \sum_{s=0}^{\log(p)/2-1} (ng(2s+1) + l(2s+1)) \tag{5.18}$$

DBSP+ is represented by:

$$T_{DBSP+}(n) = n\ \log(n\sqrt{p}) \sum_{s=0}^{\log(p)/2-1} 2\,K_{12} route\left(n, \left\lceil \frac{2^s n}{2^{2s}} \right\rceil, \log(p) - 1 - 2s\right) \tag{5.19}$$

$$= O\left(n\ \log(n\sqrt{p}) + \sum_{s=0}^{\log(p)/2-1} \sum_{j=0, i=\log(p)-1-2s+j}^{2s} \left(Pf(i)l(i) + min\left\{n, \left\lceil \frac{n}{2^s} \right\rceil 2^j\right\} g(i)\right)\right)$$

On the pruned butterfly $g(i) = \sqrt{p}/2^{i/2}$, $l(i) = \log(p) - i$, $Pf(i) = (\log(p) - i)^2$, and $g(i, 1/2) = \log(p) - i$, $l(i, 1/2) = (\log(p) - i)^2$ (Table 5.1).

We therefore evaluate:

$$T_{\alpha BSP}(n) = O\left(n\ \log(np) + n\ \sum_{i=0}^{\log(p)/2} (2i + (2i)^3)\right)$$
$$= O\left(n\ \log(n) + n\ \log^2(p) + \log^4(p)\right) \quad (5.20)$$
$$T_{DBSP}(n) = O\left(n\ \log(np) + n\ \sum_{i=0}^{\log(p)/2} (2^i + 2i)\right)$$
$$= O\left(n\ \log(n) + n\sqrt{p}\right) \quad (5.21)$$

To compute $T_{DBSP+}(n)$ we generate an argument similar to [FPP03]. We observe that the inner sum is dominated by the largest term that is reached when $n/2^{j/2} = \left\lceil \frac{n}{2^s} \right\rceil 2^{j/2}$. The resulting equation is:

$$T_{DBSP+}(n) = O\left(n\ \log(np) + \sum_{s=0}^{\log(p)/2} \left(\left\lceil \frac{n}{2^s} \right\rceil^{1/2} n^{1/2} 2^s + s^3\right)\right)$$
$$= O\left(n\ \log(np) + \sum_{s=0}^{\log(p)/2} \left(n2^{s/2} + n^{1/2}2^s\right) + \log^4(p)\right) \quad (5.22)$$
$$= O\left(n\ \log(n) + n\sqrt[4]{p} + n^{1/2}\sqrt{p}\right)$$

Minimal effort is required for the αDBSP analysis to lead to a generic parametrized expression, while entering the machine parameters is straightforward.

The K_{12} routing algorithm used by DBSP+ improves the single superstep DBSP runtime while αDBSP improves the DBSP+ bounds: for $n = 1$ αDBSP gives $O(\log^4(p))$ vs. $O(\sqrt{p})$ on DBSP; for $n = \sqrt{p}$ αDBSP gives $O(n\ \log^2(p))$ vs. $O\left(n\sqrt[4]{p}\right)$ on DBSP (Table 5.1).

5.5.3 Nearest-Neighbor Exchange

Scientific codes commonly use differencing schemes to solve partial differential equations. To approximate the local derivatives during iterations, these solvers require data that is computed by the neighbor processors during the previous iteration. The nearest-neighbor problem sends n data elements from each processor to its direct neighbors considering that processors are mapped to a 2D grid.

The 2D processor grid is mapped to a binary tree by recursive horizontal and vertical division as in Section 5.5.2. For a vertical split, a level $2s$ subtree sends up $2^s n$ data; for a horizontal split a level $2s+1$ subtree sends up $2^s n$ data. This problem is a n-relation for DBSP, a tight $(n, 1/2)$-relation for αDBSP, and a sum of two $(n, \lceil \frac{2^s n}{2^{2s}} \rceil)$ K_{12}-routings for DBSP+. For αDBSP we obtain:

$$T_{\alpha BSP}(n) = ng(0,1/2) + l(0,1/2) \tag{5.23}$$

While for DBSP we obtain:

$$T_{DBSP}(n) = ng(0) + l(0) \tag{5.24}$$

$$T_{DBSP+}(n) = O\left(\sum_{s=0}^{\log(p)/2} K_{12} route\left(n, \left\lceil \frac{n}{2^s} \right\rceil, \log(p) - 1 - s\right)\right) \tag{5.25}$$

On the pruned butterfly the computations are similar to the analysis from Section 5.5.2. By substitution we obtain $T_{\alpha BSP}(n) = O(n\log(p) + \log^3(p))$ and $T_{DBSP}(n) = O(n\sqrt{p})$. Evaluation of the $K_{12} route$ term for DBSP+ produces $T_{DBSP+}(n) = O\left(n\sqrt[4]{p} + n^{1/2}\sqrt{p}\right)$.

The nearest-neighbor and the North-South FFT examples represent a simplification of the steps executed by scientific codes running on 2D grids mapped to the surface of the Earth. By adding up the numbers we compute a cost of $O\left(\sum_{i=0}^{\log(p)-1}(ng(i,1/2) + l(i,1/2))\right)$ for communication, and a required bandwidth growth factor of $\alpha = 1/2$ for the entire application.

Table 5.2: Bounds on algorithm execution time for three BSP family machines modeling a pruned butterfly topology.

Model	Generalized Broadcast	North-South FFT	Nearest-Neighbor Exchange
DBSP	$O(np\sqrt{p})$	$O(n\,log(n) + n\sqrt{p})$	$O(n\sqrt{p})$
DBSP+	$O(np)$	$O(n\,log(n) + n\sqrt[4]{p} + n^{1/2}\sqrt{p})$	$O(n\sqrt[4]{p} + n^{1/2}\sqrt{p})$
αDBSP	$\Omega(np)$	$\Omega(n\,log(np) + log^2(p))$	$\Omega(n + log(p))$
	$O(np)$	$O(n\,log(n) + nlog^2(p) + log^4(p))$	$O(n\,log(p) + log^3(p))$

5.5.4 Discussion

The DBSP+ results from Sections 5.5.2 and 5.5.3 use the hm and K_{12} routing algorithms for DBSP presented in [FPP03, BFPP01]. The K_{12} algorithm is not optimal for all problems classified as $\left(n, \left\lceil \frac{n}{\sqrt{p}} \right\rceil\right)$ K_{12}-routings. For example, if the cases shown in Fig. 5.3b and Fig. 5.3c have identical K_{12}-routings, the first example could be executed faster.

For the case from Fig. 5.3b the DBSP algorithm introduced by Theorem 1 improves the K_{12} results being as good as αDBSP for $n \geq \sqrt{p}$. For $n < \sqrt{p}$ this is no longer the case. In particular, for $n = 1$ no DBSP algorithm can give a bound better than $O(\sqrt{p})$ since at least one 0-superstep 1-relation is executed. This observation in parallel with the results from Theorem 2 explain why αDBSP has the capability to improve the highly optimized DBSP+ results.

Table 5.2 summarizes the execution bounds for the three cases analyzed in this section; the αDBSP results include lower bounds since all relevant (h, α)-relations are tight. For these examples αDBSP is an improvement of DBSP+ in terms of execution complexity, and observing the detailed analysis it is possible to argue that αDBSP is no harder to use than DBSP.

5.6 Summary

This chapter presents the αDBSP model—an extension of the DBSP model that augments h-relations with an exponential bandwidth growth factor α. This additional factor allows for improved cost analysis by providing an upper bound on the number of packets traveling up to each machine level.

We describe two routing algorithms for (h, α)-relations that improve DBSP results. Theorem 1 describes an algorithm that may be executed on any DBSP machine but sends fewer packets compared to the (k_1, k_2)-routing algorithm, while Theorem 2 further exploits the ability of the pruned butterfly to route sparse relations faster than $O(\sqrt{p})$. Table 5.1 shows the αDBSP machine parameters that were computed using the above routing algorithms. On the mesh and pruned butterfly topologies the bandwidth parameter $g(i, \alpha)$ improves the DBSP $g(i)$ parameter by p^{α_M}, p^{α} or $p^{\alpha_M}/\log(p)$, depending on the α_M machine factor and the α relation factor. These results are applied in Section 5.5 to analyze scientific computing kernels representative for ocean and atmospheric global Earth models. This exercise shows that αDBSP improves DBSP for both the FFT and PDE kernels, at the same time maintaining a virtually unchanged analysis complexity (Table 5.2).

A bandwidth argument may be generated by computing the lower bound on (h, α)-relation execution time (Table 5.1). A heuristic routing algorithm that runs on a real world interconnect would most likely approach this bound if contention is avoided by the distribution of messages over multiple paths, i.e., the interconnect implements an efficient adaptive routing algorithm. Subject to this assumption, a (h, α)-relation executes in linear time on a real world machine with $\alpha_M \leq \alpha$. To avoid bandwidth bottlenecks for a class of applications, machine interconnects could be designed to meet at least this criterion.

Chapter 6

Conclusions

This dissertation introduced a hybrid SPMD – coarse grain dataflow model that describes high-level data and task parallelism, argued that this model achieves a good trade-off between programming simplicity and implementation efficiency by presenting the implementation and performance of several benchmarks, and showed how the αDBSP model estimates collective communication cost on parametrized machines while improving DBSP accuracy on globally unbalanced patterns. Section 1.3 summarizes the main contributions of this dissertation.

A prerequisite of a productive and efficient programming model design is finding the proper balance between low- and high-level programming abstractions. If low-level communication libraries and memory access primitives can potentially provide the best performance, they require a substantial development effort and may exhibit limited portability. Highly abstract languages require that fewer implementation details be specified by developers, thus increasing productivity; however, their compilers are faced with solving complex optimization problems, in many instances underperforming hand-written code or requiring programming workarounds that defeat the stated high productivity goals (Section 1.3.1).

The CGD model (Section 2.1) aims to solve the above trade-off by requiring programmers to describe what they know best—including data layouts, computation assignments, and algorithmic optimizations—and allowing compilers to handle repetitive work and well-understood optimizations such as communication and synchronization, datastructure access, communication overlap, and machine specific low-level optimizations (Section 4.1). Accordingly, instead of trying to address each hard problem automatically, the CGD model provides abstractions that aid developers to solve these problems with less effort.

The language support for datastructure decomposition and distribution rules (Section 3.4) makes application implementation and layout optimization easier, as shown through the examples presented in Sections 2.3 and 3.6. The explicit dataflow data dependencies allow compilers to generate C++ code that is very similar to hand-written code in terms of collective communication structure. Hence, the performance is very good compared to the reference MPI or pthreads manual implementations, sometimes outperforming them by relying on optimizations such as communication overlap, avoiding data serialization, and caching and buffering remote reads and writes (Section 4.2).

Both the CGD model and PGAS languages provide a distributed datastructure abstraction that hides the communication and synchronization behind the datastructure implementation, elegantly expressing SPMD structured parallelism. In contrast to PGAS languages, a single CGD datastructure can have multiple distributions, and distributions allow domain overlap and replication. E.g., X- and Y-wise FFTs access locally the row and column block decompositions of a single vector datastructure; finite differencing scheme PDE solvers access locally the overlapping "halo" domains needed to compute spacial derivatives (Sections 1.3.1 and 2.3). Hence, algorithms represented as a sequence of SPMD computations can maximize access locality by executing each computation relying on a different view of the same datastructure, and by replicating data elements relying on distributions with overlapping domains.

While only approximating the real hardware performance, a theoretical communication model provides a mental map between communication and hardware operations and their complexity. This dissertation introduces the CGD programming model, which exposes collective communication as redistribution operations (Section 2.1.3), and the αDBSP machine model, which estimates the collective communication cost on parametrized architectures (Section 5.1). These models allow developers to understand communication and computation complexity during the development stage, therefore helping them to make an educated choice among the implementation options.

The αDBSP model defines (h, α)-relations and improves DBSP for several common globally unbalanced problems such as PDE solvers and FFTs on grid subdomains. Additionally, αDBSP estimates the hierarchical bandwidth required by an application, helping system architects design interconnects that avoid bandwidth bottlenecks for their target applications. For example, a 2D PDE solver that employs FFT filtering at the top and bottom of its grid has a bandwidth growth factor of $\alpha = \frac{1}{2}$, and may be executed without inherent bandwidth bottlenecks on a machine with $\alpha_M = \frac{1}{2}$ such as a 2D fat-tree or 2D mesh (Section 5.5).

Bibliography

[AC86] Arvind and David E. Culler. Dataflow architectures. In *Annual review of computer science*, volume 1, pages 225–253. Annual Reviews Inc., November 1986.

[AISS95] Albert Alexandrov, Mihai Ionescu, Klaus Schauser, and Chris Scheiman. LogGP: incorporating long messages into the LogP model, one step closer towards a realistic model for parallel computation. In *Proceedings of the 7th Symposium on Parallel Algorithms and Architectures (SPAA '95)*. ACM, July 1995.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[AW80] Ed Ashcroft and Bill Wadge. Some common misconceptions about Lucid. *SIGPLAN Notices*, 15(10):15–26, October 1980.

[BB95] Paul Bay and Gianfranco Bilardi. Deterministic on-line routing on area-universal networks. *Journal of the ACM (JACM)*, 42(3):614–640, May 1995.

[BBea91] D. H. Bailey, E. Barszcz, and J. T. Barton et al. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[BBNY06] Christian Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*. IEEE Computer Society Press, April 2006.

[BCBY04] C. Bell, W. Chen, D. Bonachea, and K. Yelick. Evaluating support for global address space languages on the Cray X1. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '04)*. ACM, June 2004.

[Ber99] Martin Beran. Decomposable bulk synchronous parallel computers. In *Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM '99)*. Springer-Verlag, November 1999.

[BFPP01] G. Bilardi, C. Fantozzi, A. Pietracaprina, and G. Pucci. On the effectiveness of D-BSP as a bridging model of parallel computation. In *Proceedings of International Conference on Computational Science (ICCS '01)*, pages 579–588. Springer-Verlag, May 2001.

[BHP+96] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs LogP. In *Proceedings of the 8th Symposium on Parallel Algorithms and Architectures (SPAA '96)*. ACM, June 1996.

[BJvOR03] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) library. *Journal of Parallel Computing*, 29(2):187 – 207, February 2003.

[BL05] Shawn Bowers and Bertram Ludascher. Actor-oriented design of scientific workflows. In *Proceedings of the 24st International Conference on Conceptual Modeling (ER'05)*. Springer-Verlag, October 2005.

[BL09] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[BN] V. Balaji and Robert W. Numrich. A uniform memory model for distributed data objects on parallel architectures. In *Use of High-Performance Computing in Meteorology*. World Scientific Publishing Co.

[Bon02] Dan Bonachea. GASNet specification, v1.1. University of California at Berkeley, 2002.

[BPP07] G. Bilardi, A. Pietracaprina, and G. Pucci. *Decomposable BSP: A Bandwidth-Latency Model for Parallel and Hierarchical Computation*. CRC Press, Inc., 2007.

[But97] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., May 1997.

[BVZ+07] Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*, pages 69–84. IEEE Computer Society, 2007.

[CCDI09] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and David Iten. Global HPC challenge benchmarks in Chapel. Technical report, Cray Inc., November 2009.

[CCZ07] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing*, 21:231–312, August 2007.

[CDHW] Bradford L Chamberlain, Steven J Deitz, Mary Beth Hribar, and Wayne A Wong. Technical report.

[CGS+05] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, October 2005.

[CHea03] Wei Yu Chen, P. Husbands, and D. Bonachea et. al. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*. ACM, June 2003.

[cil04] Cilk 5.4.1 – reference manual, 2004.

[CKP+93] D. E. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E.Santos, R. Subramonian, and T. Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming (PPOPP '93)*. ACM, August 1993.

[CL10] Marcelo Cintra and Diego R Llanos. Preliminary evaluation of chapel capabilities with nas parallel benchmarks. 2010.

[CLZ+11] Xuhao Chen, Jiawen Li, Zhong Zheng, Li Shen, and Zhiying Wang. Evaluating scalability of emerging multithreaded applications on commodity multicore server. *International Conference of Information Technology, Computer Engineering and Management Sciences*, pages 332–335, 2011.

[CSG98] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, August 1998.

[DDHM08] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory of Computing Systems*, 35, 2008.

[dlTK95] Pilar de la Torre and C. P. Kruskal. A structural theory of recursively decomposable parallel processor-networks. In *Proceedings of the 7th Symposium on Parallel and Distributed Processing (SPDP '95)*. IEEE Computer Society Press, October 1995.

[dlTK96] Pilar de la Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In *Proceedings of the second International Euro-Par Conference on Parallel Processing (Euro-Par '96)*, pages 352–358. Springer-Verlag, August 1996.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January 1998.

[DRJ94] Pushpa Rao Dartmouth, Pushpa Rao, and R. Jagannathan. Developing scientific applications in GLU. In *Proceedings of the Seventh International Symposium on Lucid and Intensional Programming*, pages 45–52, 1994.

[EGC02] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study. In *Proceedings of the Supercomputing Conference (SC '02)*, pages 1–26. IEEE Computer Society Press, November 2002.

[Fei95] K Feind. Shared memory access (SHMEM) routines. In *Proceedings of Cray User Group (CUG '95)*, pages 303–308. Cray Inc., 1995.

[FG06] K. FÃŒrlinger and M. Gerndt. Analyzing overheads and scalability characteristics of OpenMP applications. In *Proceedings of International Meeting on High Performance Computing for Computational Science (VECPAR '06)*, 2006.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee Knoxville, 1994.

[FPP03] C. Fantozzi, A. Pietracaprina, and G. Pucci. A general PRAM simulation scheme for clustered machines. *International Journal of Foundations of Computer Science*, 14, 2003.

[FPP06] Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. Translating submachine locality into locality of reference. *Journal of Parallel Distributed Computing*, 66:633–646, May 2006.

[GBT87] J. Gurd, W. Bohm, and Y. M. Teo. Performance issues in dataflow machines. *Future Generation Computer Systems*, 3(4):285–297, December 1987.

[GKS94] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulations of the Barnes-Hut method for n-body simulations. In *Proceedings of Supercomputing Conference (SC '94)*, pages 439–448. IEEE Computer Society Press, 1994.

[GS99] E. L. W. Gropp and A. Skjellum. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, November 1999.

[HMS+98] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Journal of Parallel Computing*, 24(14):1947–1980, 1998.

[HPP01] Kieran T. Herley, Andrea Pietracaprina, and Geppino Pucci. Implementing shared memory on mesh-connected computers and on the fat-tree. *Journal of Information and Computation*, 165, 2001.

[HR91] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation. In *Proceedings of the 1991 Third Symposium on Parallel and Distributed Processing (SPDP '91)*. IEEE Computer Society Press, December 1991.

[HSH96] Chris Holt, Jaswinder Pal Singh, and John Hennessy. Application and architectural bottlenecks in large scale distributed shared memory machines. In *Proceedings of the 23rd annual International Symposium on Computer Architecture (ISCA '96)*, pages 134–145. ACM, 1996.

[Ift98] Liviu Iftode. Home-based shared virtual memory. *Princeton University, Princeton, NJ*, 1998.

[Jag95] R. Jagannathan. Coarse-grain dataflow programming of conventional parallel computers. In *Advanced Topics in Dataflow Computing and Multithreading*, pages 113–129. IEEE Computer Society Press, 1995.

[JHRM04] Wesley M. Johnston, J. R. Paul Hanna, Richard, and J. Millar. Advances in dataflow programming languages. *ACM Computing Survey (CSUR)*, March 2004.

[JKP+12] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Speculative separation for privatization and reductions. In *Proceedings of the 33rd Conference on Programming Language Design and Implementation (PLDI '12)*, pages 359–370. ACM, June 2012.

[JS99] D. Jiang and J.P. Singh. Scaling application performance on a cache-coherent multiprocessors. In *Proceedings of the 26th annual International Symposium on Computer Architecture (ISCA '99*, pages 305–316. ACM, 1999.

[JSS97] Dongming Jiang, Hongzhang Shan, and Jaswinder Pal Singh. Application restructuring and performance portability on shared virtual memory and

hardware-coherent multiprocessors. In *Proceedings of the 6th Symposium on Principles and Practice of Parallel Programming (PPOPP '97)*, pages 217–229. ACM, August 1997.

[JW96] Ben H. H. Juurlink and Harry A. G. Wijshoff. The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model. In *Proceedings of International Euro-Par Conference on Parallel Processing (Euro-Par '96)*. Springer-Verlag, 1996.

[KBB86] K. M. Kavi, B. P. Buckles, and U. N. Bhat. A formal definition of data flow graph models. *IEEE Transactions on Computers*, 35:940–948, 1986.

[KL89] Paul Hudak Kai Li. Memory coherence in shared virtual memory systems. pages 321–359, November 1989.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. The c programming language, 1988.

[Kus06] Bradley C. Kuszmaul. A cilk response to the hpc challenge (class 2). Technical report, Massachusetts Institute of Technology. Boston, MA, 2006.

[Lei85] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, 1985.

[LH94] Ben Lee and A. R. Hurson. Dataflow architectures and multithreading. *Computer*, 27(8):27–39, August 1994.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA '90)*, pages 148–159. ACM, 1990.

[MAB94] Richard Alpert Cezary Dubnicki Edward W Felten Jonathan Sandberg Matthias A Blumrich, Kai Li. Virtual memory mapped network interface for the shrimp multicomputer. pages 142–153, April 1994.

[MGRG11] Josh Milthorpe, V. Ganesh, Alistair P. Rendell, and David Grove. X10 as a parallel language for scientific computation: Practice and experience. In *Proceedings of the 25th International Parallel and Distributed Processing Symposium (IPDPS '11)*, pages 1080–1088. IEEE Computer Society Press, May 2011.

[Mor94] J. Paul Morrison. *Flow Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, 1994.

[MTT+09] Damián A. Mallón, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Andrés Gómez, Ramón Doallo, and J. Carlos Mouriño. Performance evaluation of mpi, upc and openmp on multicore architectures. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 174–184. Springer-Verlag, 2009.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[NC99] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy libray for ditributed array libraries and compiler run-time systems. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS/SPDP '99)*, 1999.

[NGM97] L. M. Ni, Y. Gui, and S. Moore. Performance evaluation of switch-based wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):462–474, 1997.

[Nic06] Virginia Niculescu. Cost evaluation from specifications for BSP programs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS '06)*. IEEE Computer Society Press, April 2006.

[Nik93] Rishiyur S. Nikhil. An overview of the parallel language Id (a foundation for pH, a parallel dialect of Haskell). Technical report, Digital Equipment Corporation, Cambridge Research Laboratory, 1993.

[NR98] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[OP] Stephen Olivier and Jan Prins. Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming*.

[PG08] I. Patel and J.R. Gilbert. An empirical study of the performance and productivity of two parallel programming models. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS '08)*, pages 1–7. IEEE Computer Society Press, April 2008.

[PGZ+11] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. Commutative set: a language extension for implicit parallel programming. In *Proceedings of the 32nd Conference on Programming Language Design and Implementation (PLDI '11)*, pages 1–11. ACM, June 2011.

[PL] Karin Petersen and Kai Li. Cache coherence for shared memory multiprocessors based on virtual memory support.

[RLR98] Keith H. Randall, Charles E. Leiserson, and H. Randall. Cilk: Efficient multithreaded computing. Technical report, Massachusetts Institute of Technology. Boston, MA, 1998.

[RSG93] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th annual International Symposium on Computer Architecture (ISCA '93)*, pages 14–26. ACM, 1993.

[SHG95] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of hierarchical N-body methods for multiprocessor architectures. *Transactions on Computer Systems (TOCS)*, 13(2):141–202, May 1995.

[SHT+95] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John L. Hennessy. Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27:118–141, June 1995.

[SJHG93] J. P. Singh, T. Joe, J. L. Hennessy, and A. Gupta. An empirical comparison of the Kendall Square Research KSR-1 and Stanford DASH multiprocessors. In *Proceedings of Supercomputing Conference (SC '93)*, pages 214–225. IEEE Computer Society Press, 1993.

[SOW+95] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference*. MIT Press, 1995.

[SRG94] Jaswinder Pal Singh, Edward Rothberg, and Anoop Gupta. Modeling communication in parallel algorithms: a fruitful interaction between theory and systems? In *Proceedings of the 6th Symposium on Parallel Algorithms and Architectures (SPAA '94)*, pages 189–199. ACM, July 1994.

[SS99] Hongzhang Shan and Jaswinder Pal Singh. A comparison of MPI, SHMEM and cache-coherent shared address space programming models on the SGI Origin 2000. In *Proceedings of the 13th International Conference on Supercomputing (ICS '99)*, pages 329–338. ACM, June 1999.

[SS09] Adrian Soviani and Jaswinder Pal Singh. Optimizing communication scheduling using dataflow semantics. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP '09)*, pages 301–308. IEEE Computer Society Press, 2009.

[SS10] Caitlin Sadowski and Andrew Shewmaker. The last mile: Parallel programming and usability. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*, pages 309–314. ACM, 2010.

[SS12] Adrian Soviani and Jaswinder Pal Singh. Estimating application hierarchical bandwidth requirements using BSP family models. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW '12)*, pages 914 – 923. IEEE Computer Society Press, 2012.

[SSOB00] Hongzhang Shan, Jaswinder Pal Singh, Leonid Oliker, and Rupak Biwas. A comparison of three programming models for adaptive applications on the Origin 2000. *Journal of Parallel and Distributed Computing*, June 2000.

[SSOB03] Hongzhang Shan, Jaswinder P. Singh, Leonid Oliker, and Rupak Biswas. Message passing and shared address space parallelism on an SMP cluster. *Parallel Computing*, 29(2):167–186, February 2003.

[SSOG93] Jaspal Subhlok, James M. Stichnoth, David R. O'hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming (PPOPP '93)*, pages 13–22. ACM, August 1993.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[VECGS92] T. Von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. *Active Messages: a mechanism for integrated communication and computation*, volume 20. ACM, 1992.

[WMT08] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS '08)*, pages 1–8. IEEE Computer Society Press, 2008.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 24–36. ACM, 1995.