AUTOMATIC COMPUTATION AND DATA DECOMPOSITION FOR MULTIPROCESSORS

Jennifer-Ann Monique Anderson

Technical Report: CSL-TR-97-719
March 1997

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-9040

Abstract

Memory subsystem efficiency is critical to achieving high performance on parallel machines. The memory subsystem organization of modern multiprocessor architectures makes their performance highly sensitive to both the distribution of the computation and the layout of the data. A key issue in programming these machines is selecting the *computation decomposition* and *data decomposition*, the mapping of the computation and data, respectively, across the processors of the machine.

A popular approach to the decomposition problem is to require programmers to perform the decomposition analysis themselves, and to communicate that information to the compiler using language extensions. This thesis presents a new compiler algorithm that automatically calculates computation and data decompositions for dense-matrix scientific codes. The core of the algorithm is based on a linear algebra framework for expressing and calculating the computation and data decompositions. Using the linear algebra model, the algorithm generates a system of equations that specifies the conditions the desired decompositions must satisfy. The decompositions are then calculated systematically by solving the system of equations. Since the best decompositions may change as different phases of the program are executed, the algorithm also considers re-organizing the data dynamically. The analysis is performed both within and across procedure boundaries so that entire programs can be analyzed.

We have incorporated our techniques into the SUIF parallelizing compiler system. We evaluated the effectiveness of the algorithm by applying the compiler to a suite of benchmark programs, and compared the performance of the resulting code to the performance obtained without using our techniques. We found that our decomposition analysis and optimization can lead to significant increases in program performance.

Key Words and Phrases: parallelization, compiler optimization, data decomposition, computation decomposition

Acknowledgements

I would like to thank my advisor Monica Lam for her enthusiasm and support. She dedicated a great deal of time and energy to this work, and she always encouraged me to do my very best. I also feel fortunate to have had John Hennessy and Anoop Gupta on my thesis committee. I thank them for their insightful comments and guidance.

I had the privilege of working with many smart and talented people during my time at Stanford. Saman Amarasinghe wrote a lot of code that was integral to getting the SUIF parallelizer used in this thesis up and running. Also, the interprocedural decomposition algorithm described in Chapter 6 is built on top of Saman's linear inequality framework for parallelizing compilers. Saman is insightful, motivated and creative, and I always enjoyed working with him. The compiler-directed page coloring optimization described in Chapter 2 is joint work with Ed Bugnion, who is both very talented and great fun to work with. I also wish to thank all the members of the SUIF compiler group, including Saman Amarasinghe, Robert French, Mary Hall, David Heine, Shih-Wei Liao, Amy Lim, Dror Maydan, Todd Mowry, Brian Murphy, Jason Nieh, Jeff Oplinger, Karen Pieper, Martin Rinard, Patrick Sathyanathan, Dan Scales, Brian Schmidt, Mike Smith, Steve Tjiang, Chau-Wen Tseng Bob Wilson, Chris Wilson and Michael Wolf. In particular, Bob Wilson and Chris Wilson wrote much of the code that forms the base of the SUIF system, and Chris has done an amazing job of keeping the entire SUIF system running smoothly.

I would also like to thank Digital's Western Research Lab for awarding me a graduate fellowship. Anita Borg was my mentor at WRL, and she is a great source of advice and moral support.

I am fortunate to have a wonderful set of family and friends that encouraged me during my time in graduate school. I thank my parents for instilling in me a love a learning that led me to pursue this degree. Above all, I would like to thank my husband Grant for his constant love, understanding and support. I consider myself extremely fortunate to have such a wonderful husband.

This work was supported in part by ARPA contracts DABT63-91-K-0003 and DABT63-94-C-0054, and a fellowship from Digital's Western Research Lab.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Processor speeds in modern computer systems are outpacing the memory speeds. The result is that the latency of memory accesses is a bottleneck to achieving high processor utilization. To bridge the gap between processor and memory speeds, computer designers typically use a hierarchy based on different speeds and sizes of memories. Smaller, faster memories or caches are placed close to the processors while successively larger, slower memories are placed farther away from the processors.

The gap between processor speeds and memory speeds is expected to widen. In the last decade, microprocessor performance has been steadily improving at a rate of 50% to 100% every year[39]. Meanwhile, the access times for DRAMs, the components that comprise computer main memories, have been improving at the rate of only 7% per year[39]. As a result, memory subsystem performance will continue to have a great impact on overall computer system performance.

Making effective use of a memory hierarchy relies on a program's *locality of reference*: if a data item is accessed then that data item, or data near it, is likely to be accessed again in the near future. Performance benefits if we are able to keep recently accessed data items in the fastest memories. However, it has been notoriously difficult for scientific codes to make effective use of a memory hierarchy. In fact, various past machines built for scientific computations such as the Cray C90, Cydrome Cydra-5[65] and the Multiflow Trace[24] were all built without caches.

Realizing good memory hierarchy performance on multiprocessors is even more important to application performance than on uniprocessors, but also more difficult. Not only are multiprocessor memory hierarchies supersets of the uniprocessor hierarchy, but the application must also ensure that uniprocessor locality is not compromised by multiprocessor considerations. Any benefits to program performance due to increased parallelism can be quickly outweighed if the program ends up stalled waiting for data. Thus to achieve good performance on parallel systems, programs must take into account both parallelism and locality.

1.1 Multiprocessor Memory Hierarchies

There are two principal classes of memory architectures for parallel machines: distributed address space machines and shared address space machines. Distributed address space machines are built from a set of processors connected by a communication network. Each processor has its own local memory that it can access directly; messages have to be sent across the network for a processor to access data stored in another processor's memory. Examples of distributed address space machines include the Intel Paragon, IBM SP-2 and clusters of workstations. In contrast, shared address space machines present the programmer with a single memory space that all processors can access. Many of these machines support hardware cache coherence to keep data consistent across the processors. Shared address space machines may either have a single shared memory that can be uniformly accessed by all the processors, or the memories may be physically distributed across the processors. The former class of machines are often called *centralized* shared address space machines, while the latter class of machines are called *distributed* shared address space machines. Distributing the memories across the processors removes the need for a single shared bus going from all the processors' caches to the memory and thus makes it easier to scale to more processors. Today many computer vendors (e.g. Sun, Digital Equipment, Sequent and Compaq) offer centralized shared address space machines. Examples of distributed shared address space systems include the Stanford DASH[55] and FLASH[54] multiprocessors and MIT ALEWIFE[2] research machines, and the Convex Exemplar, Cray T3D/E and Silicon Graphics Origin commercial machines.

There are two key reasons for why it is more difficult to get good memory subsystem performance on multiprocessors over uniprocessors: long latencies due to inter-processor communication, and multiprocessor-specific cache misses on machines with coherent caches. We discuss each of these reasons in more detail in the following two sections.

1.1.1 Communication Latencies

On current parallel machines, it can cost anywhere from 50 to over 10,000 processor clock cycles to communicate data between processors[39]. The exact communication cost depends on the memory architecture of the machine, the type of interconnection network and the size of the system.

On machines with a centralized memory, cache coherence is typically maintained using an invalidation-based protocol. Data written by one processor is invalidated in the other processors' caches; subsequent reads by the other processors miss in the cache and have to be read from memory. Communication can thus result in cache misses, and the data must be fetched from main memory.

On machines with physically distributed memories, the memory access times are non-uniform. The memory latency to access another processor's memory can be significantly higher than the latency to local memory. In effect, the remote memories on these machines form yet another level in the memory hierarchy. Distributed shared address space machines that support cache coherence in hardware also typically use invalidation-based cache-coherence protocols (e.g. SGI Origin). On these machines, communication results in cache misses where the data must be fetched from the other processor's remote memory. On non-coherent distributed shared address space machines (e.g. Cray T3D/E) remote data is typically not cached, and all remote references result in accesses to another processor's memory.

On distributed address space machines, processors must send explicit messages to communicate data. The cost of communication is the time to send a message to the remote processor and for the receiving processor to process the message. Explicit communication through message-passing involves software and is much more expensive than implicit communication through cache-coherence hardware. For example, Table 1.1 shows the typical remote memory latencies for a variety of machines with physically distributed memories. The first column lists the machines and the second column shows whether the machines have a shared or distributed address space. The third and fourth columns list the processors used at each node of the machine and the processor cycle times, respectively. The final column gives typical remote access times. For the shared address space machines, the value is the remote load latency; for the distributed address space machines, the value is the round-trip message time (for one word messages). For comparison, accesses that hit in the cache typically take one processor cycle. All the remote access time data is from [39], unless otherwise noted.

Machine	Address Space Organization	Processor	Processor Cycle Time (ns)	Remote Access Time (ns)
SGI Origin[69]	shared	R10000	5.0	500 - 1100
Cray T3E[66]	shared	21164	3.3	1000 - 2000
Convex Exemplar	shared	PA8000	5.5	2000
TMC CM-5	distributed	Sparc	30.0	10000
Intel Paragon	distributed	i860	20.0	10000 - 30000
IBM SP-2	distributed	RS6000	13.0	30000 - 100000

Table 1.1: Typical remote memory access times. For the shared address space machines, the value is the remote load latency; for the distributed address space machines, the value is the round-trip message time.

The long memory latencies mean that the amount of inter-processor communication in the program is a critical factor for performance. Thus it is important for computations to have good *temporal locality*. A computation has temporal locality if it re-uses much of the data it has been accessing; programs with high temporal locality tend to require less communication.

It is important to take communication and temporal locality into consideration when deciding how to parallelize a loop nest and how to assign the iterations to processors. Consider the code shown in Figure 1.1(a). This code is representative of the tomcatv benchmark from the SPEC92 and SPEC95 benchmark suites. While all the iterations in the first two-deep loop nest can run in parallel, only the inner loop of the second loop nest

```
real x[N,N], y[N,N], z[N,N]
for time = 1 to nsteps do
  ...
  for $i_2$ = 1 to N do          /* doall */
    for $i_1$ = 1 to N do        /* doall */
      x[$i_1$,$i_2$] = y[$i_1$,$i_2$] + z[$i_1$,$i_2$]

  for $i_2$ = 2 to N-1 do
    for $i_1$ = 1 to N do        /* doall */
      x[$i_1$,$i_2$] = 0.333 * (x[$i_1$,$i_2$] + x[$i_1$,$i_2$-1] + x[$i_1$,$i_2$+1])
  ...
end for
```

(a)

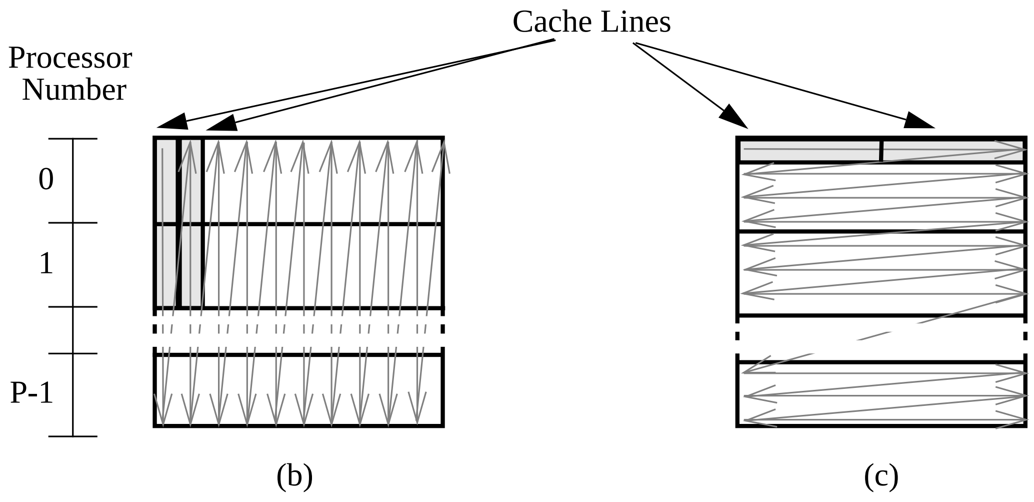


Figure 1.1: An example to illustrate multiprocessor memory hierarchy issues: (a) sample code, (b) original data mapping and (c) optimized data mapping. The light grey arrows show the memory layout order.

is parallelizable (parallel loops are also known as *doall* loops and are annotated with the comment */* doall */* in the figure). Consider what happens to the data access patterns when we distribute the parallel loops in this example. Assume that when a loop is distributed, each processor executes equal-sized blocks of consecutive iterations. If we distribute both parallel loops in the first loop nest, each processor accesses a two-dimensional block of array elements for each of the arrays. If we distribute the one parallel loop in the second loop nest, each processor accesses a block of rows for each array. There is communication between the two loop nests since the data accessed by each processor differs. However, we can eliminate the communication by distributing only the inner loop in the first loop nest. Now, each processor accesses the same data in both of the loop nests – a block of contiguous rows for each array. In this way, no inter-processor communication is necessary throughout the entire computation. Figure 1.1(b) shows the data accessed by each processor when only the inner parallel loops in both of the loop nests are distributed across the processors. The light grey arrows in the figure show the memory layout order.

1.1.2 Multiprocessor Cache Effects

On multiprocessors that support a shared address space via cache coherence, it is not sufficient to just minimize the essential cache misses caused by communication. Due to characteristics found in typical data caches, multiprocessors also experience non-essential cache misses that can significantly hurt performance[28, 29].

In cache-coherent machines, the cache line size is the default coherence unit. When a processor performs a write, the entire cache line is invalidated in all other processor's caches. In today's machines, cache lines are typically 16 to 128 bytes long. A computation has *spatial locality* if it uses multiple words in a cache line before the line is displaced from the cache. While spatial locality is a consideration for both uni- and multiprocessors, *false sharing* is unique to multiprocessors. False sharing results when different processors use different data that happen to be co-located on the same cache line. Even if a processor re-uses a data item, the item may no longer be in the cache due to an intervening access by another processor to another word in the same cache line.

Assuming the FORTRAN convention that arrays are allocated in column-major order, there is a significant amount of false sharing in the example from Figure 1.1(b). If the number of rows accessed by each processor is smaller than the number of array elements in a cache line, then every cache line is shared by at least two processors. Each time one of these lines is accessed, unwanted data are brought into the cache. Also, when one processor writes part of the cache line, that line is invalidated in the other processor's cache. This particular combination of computation mapping and data layout will result in poor cache performance.

Another problematic characteristic of data caches is that they typically have a small set-associativity; that is, each memory location can only be cached in a small number of cache locations. *Conflict misses* occur whenever different memory locations contend for the same cache location. Since each processor only operates on a subset of the data, the addresses accessed by each processor may be distributed throughout the shared address space.

Consider what happens to the example in Figure 1.1(b) if the arrays are of size 256×256 and the target machine has a direct-mapped data cache of size 8 KB. Assuming that each element is 4 bytes, the elements in every 8th column will map to the same cache location and cause conflict misses. This problem exists even if the caches are set-associative, given that existing caches usually only have a small degree of associativity.

In general, the memory subsystem performance of multiprocessor code depends on how the computation is distributed as well as how the data are laid out. Instead of simply obeying the data layout convention used by the input language (e.g. column-major in FORTRAN and row-major in C), we can improve the cache performance by customizing the data layout for the specific program. Once we determine the data that each processor is going to access, we can further optimize multiprocessor cache performance by making the data accessed by each processor contiguous in the shared address space. Such a layout enhances spatial locality, minimizes false sharing and also minimizes conflict misses. Customizing the data layout benefits all cache-coherent machines, including both centralized and distributed shared address space systems. An optimized data layout that makes each processor's data contiguous for the example from Figure 1.1(b) is shown in Figure 1.1(c).

Figure 1.2 illustrates the impact of multiprocessor cache effects on program performance for the full tomcatv benchmark from the SPEC95 benchmark suite. The figure shows the speedup over the best sequential version for three versions of the benchmark running on a centralized shared address space machine, an 8-processor 300 MHz Digital AlphaServer 8400. The curve labeled *base* shows the speedup obtained without any analysis to eliminate communication across loop nests. In this version of the program, we simply distribute iterations of the outermost parallel loop in each loop nest across the processors. The curve labeled *comp sched* corresponds to Figure 1.1(b), and shows the speedup when the computation is scheduled so as to eliminate communication across loop nests. Finally, the curved labeled *comp sched + data transform* shows the speedup obtained when the computation is scheduled to eliminate communication and the data layout is customized to make each processor's data contiguous. This last curve corresponds to Figure 1.1(c). Even though the *comp sched* program has essentially no communication, the performance is still quite poor due to poor spatial locality and false sharing. Only after the data has been transformed does the program achieve scalable performance. Whereas the speedup for the *base* program is only 2.9 on eight processors, the speedup for the *comp sched + data transform* version is 7.5.

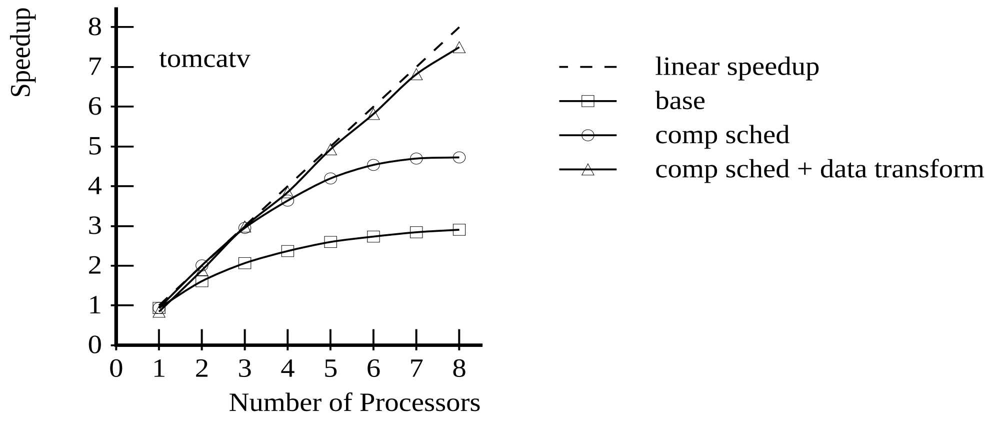


Figure 1.2: Speedups for the SPEC95 benchmark tomcatv on the AlphaServer 8400.

1.2 Problem Statement

To achieve good performance on parallel systems, programs must make effective use of the computer's memory hierarchy as well as its ability to perform computation in parallel. If we can co-locate the data with the processor that accesses the data, then we can minimize the problems specific to multiprocessor memory hierarchies. This means that we have to find a mapping of the data to the processors of the parallel machine, as well as a mapping of the computation to the processors of the machine. In this thesis we use the term *data decomposition* to refer to the data-to-processor mapping and the term *computation decomposition* to refer to the computation-to-processor mapping.

Selecting a good computation and data decomposition for a program is a difficult problem. First, there are many possible ways to assign the data and computation across the processors of the machine. The data decomposition may need to change dynamically as the program executes, for example, if different phases of the program operate on the data in different ways.

Second, the choices of data and computation decomposition are inter-related; it is important to examine both the opportunities for parallelism and the locality of data to determine the decompositions. For example, if the only available parallelism in a computation lies in operating on different elements of an array simultaneously, then allocating those elements to the same processor renders the parallelism unusable. The data decomposition dictated by the available parallelism in one loop nest affects the decision of how to parallelize the next loop nest, and how to distribute the computation to minimize communication.

Third, decomposition analysis must be performed across the entire program. It is not sufficient to just analyze one procedure at a time. If the data decompositions of the variables do not match across procedures, then the program could potentially incur large amounts of communication at every procedure call entry and call return.

A popular approach to the decomposition problem is to solicit the programmer's help in determining the data decompositions. A number of parallelizing compiler systems have been developed that rely on the user to specify the data-to-processor mapping using language extensions[21, 31, 49, 64, 74, 82]. The compiler then infers the computation mapping using the *owner-computes rule*[20, 64, 82]: the processor that is assigned a data element performs all computation that writes that element. The High Performance FORTRAN (HPF) standard developed by a consortium of academic and commercial organizations also relies upon user-specified data decompositions[48]. These compiler systems are geared towards scientific codes and primarily target distributed address space machines. While the languages provide significant benefit to the programmer by eliminating the tedious job of managing the different address spaces explicitly, the programmer is still faced with a very difficult programming problem. The tight coupling between the mapping of data and computation means that the programmer must, in effect, also analyze the parallelization of the program when specifying the data decompositions. As the best decomposition may change based on the architecture of the machine, the programmer must fully master the machine details. Furthermore, the data decompositions may need to be modified to make the program run efficiently on a different architecture.

Parallelizing compilers for shared address space machines currently do not perform any decomposition analysis; they make no attempt to schedule computation that accesses the same data onto the same processor. These compilers start with a sequential program, but they only analyze one loop nest at a time and will typically always distribute the outermost parallel loop in a loop nest. Of course, the resulting programs still run correctly; however, they may not run as efficiently as possible due to communication, false sharing and poor spatial locality.

In short, current parallelizing compiler systems for distributed address space machines require programmers to perform the decomposition analysis themselves and to communicate that information to the compiler using language extensions. For shared address space machines, current parallelizing compilers do not implement decomposition analysis at all. A compiler that automatically finds the computation and data decompositions for a program relieves programmers from doing the complex analysis themselves and leads to more efficient code. The compiler can then take as input a sequential program – without language extensions for decompositions – and generate high-performance code for both distributed and shared address space machines.

1.3 Thesis Overview and Contributions

The computation and data decomposition of a program is critical to performance on a wide variety of parallel machines. Yet, selecting a good decomposition for a program on a particular machine is a complex optimization problem. This thesis presents a compiler algorithm that automatically calculates computation and data decompositions for dense matrix computations that result in efficient, high-performance code. The contributions of this thesis are as follows:

Decomposition Framework. We have developed a linear algebra framework for expressing and calculating decompositions. We model decompositions in two steps: first an affine function[1] maps the computation and data onto a virtual processor space, and second a folding function maps the virtual processor space onto the physical processors of the target machine. Using this framework, our algorithm generates a system of equations that specifies the conditions that the desired decompositions must satisfy. We then calculate the decompositions systematically by solving the systems of equations. Our mathematical model allows a rich set of target decompositions and is not limited to an arbitrary set of possible decompositions.

Decomposition Algorithm. Our compiler algorithm for finding decompositions is the first complete algorithm based on a linear algebra framework. Our decomposition algorithm operates by partitioning the program into regions that have the same data assignment for all the computation in the region. Within each region, we use the mathematical model to generate a system of equations that describes the decompositions. Communication occurs across regions as the data must be reorganized. The regions are found incrementally by gathering constraints on the data and computation that must be assigned to the same processor in order for a solution to exist to the set of equations. The constraints are gathered starting with the most frequently executed loops in the program. This approach builds regions of the program that have no data reorganization that are as large as possible, and places any necessary communication in the least executed sections of the code. Within

[1] An affine function is a linear function plus a constant offset.

the regions, the decompositions we find are optimal in that they are guaranteed to have the largest degree of parallelism with no data reorganization. The algorithm also handles replication of data and explicit synchronization within loop nests.

Our decomposition algorithm has several key features. It is the first algorithm that calculates decompositions directly while simultaneously modeling the benefits of parallelization and the cost of communication. It allows data to be reorganized, if necessary, to benefit the overall execution time of the program. Our algorithm finds decompositions incrementally, and as a result it scales to handle complete programs. Finally, our algorithm also incorporates replication and synchronization which are often needed to generate good code for realistic programs.

Interprocedural Decompositions. Our decomposition algorithm is the first to analyze both within and across procedure boundaries so that consistent decompositions can be found throughout the entire program. The interprocedural decomposition algorithm visits each procedure twice, once in a bottom-up traversal of the call graph and once in a top-down traversal of the call graph. The bottom-up pass uses the base decomposition algorithm to incrementally gather constraints on the decompositions and the top-down pass records the final decompositions at each procedure. Our mathematical representation of decompositions, coupled with the incremental solving method, allows us to succinctly summarize all the necessary information on decompositions within a procedure. Thus when the algorithm analyzes a procedure, it does not have to re-solve for the decompositions for any procedures that are called by the current procedure.

Implementation and Evaluation. The algorithms described in this thesis have all been implemented as part of the SUIF parallelizing compiler system[76]. To evaluate the effectiveness of our proposed algorithm, we applied the compiler to a suite of benchmark programs. We ran the compiler-generated code on the Stanford DASH multiprocessor[55] and a Digital AlphaServer 8400, and present a comprehensive study of the resulting performance. Our experiments show that on both of these machines, our decomposition analysis and optimization improves program performance by as much as a factor of four.

1.4 Thesis Organization

In this thesis we describe our model and algorithm for finding computation and data decompositions. We evaluate the effectiveness of the algorithm by looking at the performance achieved by the compiler-optimized code. We begin in Chapter 2 by describing how the decomposition analysis fits into the context of a complete parallelizing compiler system. We also show that once the decompositions are calculated, the compilation techniques needed for both shared and distributed address space machines are similar. Chapter 3 describes our assumptions about the target architectures and presents the mathematical framework we use to represent decompositions. Chapters 4 and 5 present the details of the intraprocedural decomposition algorithm. In Chapter 6 we then describe the interprocedural version of the decomposition algorithm. In Chapter 7 we compare the performance of the optimized code against un-optimized code and analyze the results. We conclude in Chapter 8 by summarizing the contributions of this thesis.

Chapter 2

Compiler Overview

A parallelizing compiler takes a sequential program as input and automatically translates it into parallel code for the target machine. In this chapter we describe how the decomposition analysis described in this thesis fits into the context of a complete parallelizing compiler system. We found that there are many similarities in the compiler techniques needed for both shared and distributed address space machines. This led us to design a unified compiler framework for both kinds of architectures.

Our implementation platform is the SUIF compiler system[76]. The SUIF compiler takes as input sequential FORTRAN-77 or C programs. The source programs are first translated into the SUIF compiler's intermediate representation. All program analysis and optimization passes operate on the SUIF representation. The optimized and parallelized SUIF program is then converted into a combination of C and/or FORTRAN and is compiled by the native compiler on the target machine. The SUIF output contains calls to a portable run-time library, which is linked in by the native compiler.

The design of a complete compiler framework that incorporates decomposition analysis is shown in Figure 2.1. The compiler first runs pre-parallelization analyses to gather information needed by the subsequent passes. The parallelization phase transforms the code to find the maximum degree of loop-level parallelism, and these loops are then passed on to the decomposition phase. The computation and data decompositions are used to generate parallel code for both distributed and shared address space machines. Finally, the compiler further optimizes the uniprocessor code that runs on each individual processor. In

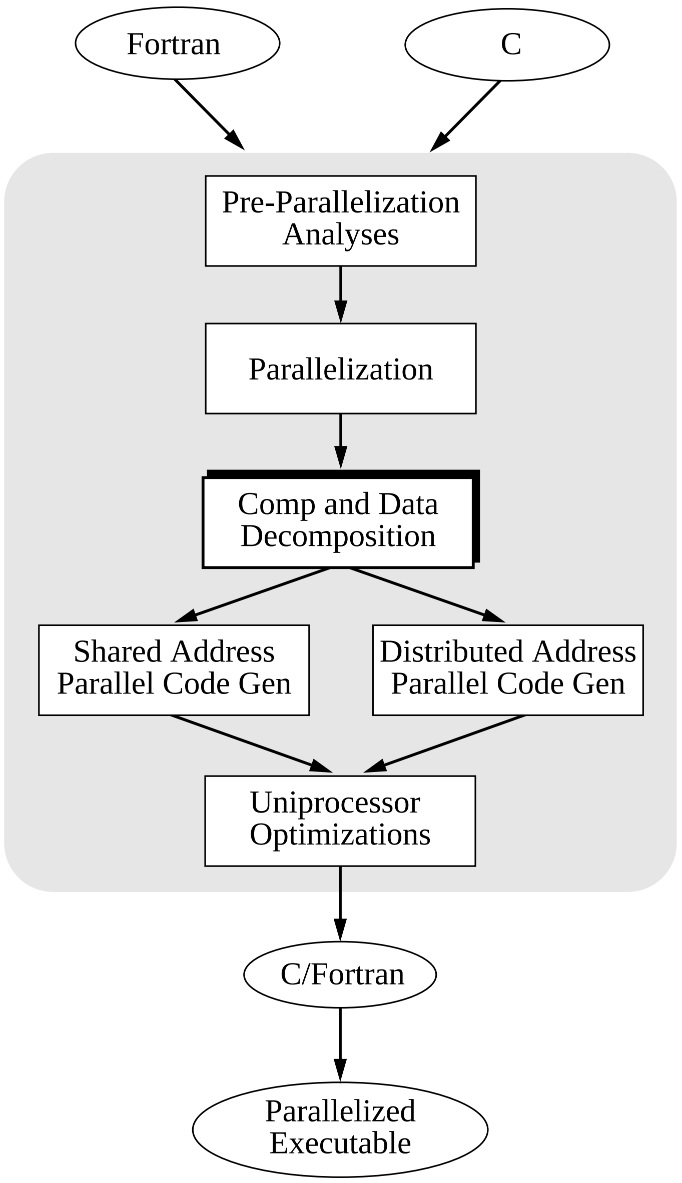

Fortran
C
Pre-Parallelization Analyses
Parallelization
Comp and Data Decomposition
Shared Address Parallel Code Gen
Distributed Address Parallel Code Gen
Uniprocessor Optimizations
C/Fortran
Parallelized Executable

Figure 2.1: The design of a parallelizing compiler system for both shared and distributed address space machines.

the remainder of this chapter, we first define the scope of input programs that are amenable to automatic parallelization and decomposition analysis. We then discuss the key compiler phases in more detail, with an emphasis on how the decomposition analysis interfaces with each of the phases.

2.1 Domain of Applications

Scientific codes are currently the most common domain for parallelizing compilers. These programs tend to make heavy use of loops to operate over array data structures. Loops can be parallelized by executing different iterations concurrently. In order to parallelize a loop, the parallel version must have the same semantics as the original sequential version. This condition imposes ordering constraints among the iterations in a loop nest. If two iterations may access the same memory location, and at least one of the accesses is a write, then it can change the semantics to reorder the iterations. Thus there is an ordering constraint, or *data dependence*, between the iterations and they must execute in the original serial order. This analysis is called data dependence analysis and is based on integer programming theory[14, 59, 62]. A loop can execute in parallel if and only if there are no data dependences carried by that loop. Data dependence analysis, and thus automatic parallelization, is typically restricted to the domain of affine functions. Specifically, we analyze loop nests of the following general form (the compiler normalizes the step sizes to 1[5]):

```
for $i_1$ = $L_1$ to $U_1$ do
  X[$f_1(i_1)$, $f_2(i_1)$, ..., $f_m(i_1)$] = ...
  for $i_2$ = $L_2(i_1)$ to $U_2(i_1)$ do
    X[$f_1'(i_1, i_2)$, $f_2'(i_1, i_2)$, ..., $f_m'(i_1, i_2)$] = ...
    ...
    for $i_l$ = $L_l(i_1, \ldots, i_{l-1})$ to $U_l(i_1, \ldots, i_{l-1})$ do
      X[$f_1''(\vec{\imath})$, $f_2''(\vec{\imath})$, ..., $f_m''(\vec{\imath})$] = ...
    end for
    ...
  end for
end for
```

In this loop nest format, $L_1 \ldots L_l$ are affine functions that compute the lower bound of each loop, and $U_1 \ldots U_l$ are affine functions that compute the upper bounds. The access functions for each dimension of the arrays, $f_1 \ldots f_m$, ${f_1}' \ldots {f_m}'$ and ${f_1}'' \ldots {f_m}''$ are also affine functions. In all cases, the loop bounds and array accesses are affine functions of outer nested loop indices and symbolic constants.

2.2 Pre-Parallelization Analyses

The compiler runs pre-parallelization symbolic analyses to extract information necessary for subsequent parallelization and optimization passes. These analyses include scalar variable analyses such as constant propagation, induction variable identification and forward propagation, as well as reduction recognition on scalar and array variables.

At this point, it is also desirable for the compiler to transform the code so that each loop nest has as few array accesses as possible. Having fewer array accesses per loop nest reduces the likelihood that the accesses will cause conflicting requirements on the computation decomposition for the loop nest. The *loop fission* transformation can be used to split a single loop into multiple loops that have the smallest number of statements possible. Each of the new loops has the same loop bounds as the original, but contains a subset of the statements[11, 51]. After the decomposition analysis, *loop fusion* can be used to regroup compatible loop nests[19]. Loop fission and loop fusion are not currently implemented in the SUIF compiler.

2.3 Parallelization Analysis

The parallelization analysis transforms the code using unimodular transformations to expose the maximum degree of loop-level parallelism, while minimizing the frequency of synchronization. It tries to generate the *coarsest granularity* of parallelism by placing the largest degree of parallelism in the outermost positions of the loop nest. Since no synchronization is needed between iterations of a parallel loop, pushing the parallel loops outermost reduces the frequency of synchronization. Algorithms for analyzing and maximizing parallelism within a loop nest have been well-studied, for example see [5, 77, 79, 80, 83]. The following two subsections describe the interface between the parallelization analysis and the decomposition analysis in more detail.

2.3.1 Choices of Parallelism

The SUIF compiler uses the algorithm developed by Wolf and Lam[77, 79] to put the loop nests in a canonical form consisting of a nest of the largest possible *fully permutable* loop nests. A loop nest is fully permutable if any arbitrary permutation of the loops within the nest is legal. A fully permutable loop nest of depth l has the property that it can always be transformed to make $l - 1$ degrees of parallelism, that is, $l - 1$ parallel loops[43, 79]. In the special case where the loop nest has no loop-carried dependences, it has l degrees of parallelism.

The compiler transforms the code to create the largest possible fully permutable nests, starting from the outermost loops. This form exposes the maximum degree of parallelism within the loop nest[77]. The compiler also marks those loops in the nest that are *doall* loops and moves them to the outermost possible position within each fully permutable nest. A *doall* loop is simply a parallel loop and can thus execute in parallel with no synchronization. The maximum degree of parallelism for the entire loop nest is the sum of the degree of parallelism contained in each of the fully permutable subnests.

In a fully permutable loop nest of depth l, the $l - 1$ degrees of parallelism can be exploited in many ways. One possibility is to transform the code to have $l - 1$ *doall* loops[77], and then distribute iterations of the *doall* loops across the processors. However, it is also possible to distribute iterations of a loop with loop-carried dependences. In this case, explicit synchronization and communication are required to enforce the dependences within the computation of the loop. Loops that are distributed across processors, but require explicit synchronization between iterations, are called *doacross* loops. Opportunities for exploiting parallelism with *doacross* loops occur when a fully permutable loop nest contains at least two loops (if there is only a single loop then the synchronization between iterations makes the loop run sequentially and there is no parallelism). To illustrate the different kinds of parallelism that are available within fully permutable loop nests, consider the following example representative of an ADI (Alternating Direction Implicit) integration:

```
real x[N,N]
/* Loop nest 1 */
for $i_1$ = 1 to N do              /* doall */
  for $i_2$ = 2 to N do
    x[$i_1$,$i_2$] = $f_1$(x[$i_1$,$i_2$], x[$i_1$,$i_2$-1])

/* Loop nest 2 */
for $i_1$ = 1 to N do              /* doall */
  for $i_2$ = 2 to N do
    x[$i_2$,$i_1$] = $f_2$(x[$i_2$,$i_1$], x[$i_2$-1,$i_1$])
```

The outer loops are *doall* loops and both loop nests are fully permutable. When iterations of the *doall* loops are run in parallel, neither communication nor synchronization is required within each of the loop nests.

Figure 2.2(a) shows the original *iteration space* for the loop nests. In the figure, the vertical axis corresponds to the outer i_1 loop, the horizontal axis corresponds to the inner i_2 loop and each node represents one iteration in the loop nest. The arrows represent the data dependences between the iterations; the data dependences show the ordering constraints among the nodes that are required to guarantee the same semantics as the original sequential version. The figure applies to both loop nests 1 and 2 since they have the same iteration space. Figure 2.2(b) shows the parallel execution of loop i_1 for both loop nests. The shaded regions in the figure show iterations that are assigned to the same processor. Since the iterations that are ordered by data dependences all execute sequentially on the same processor, no synchronization is needed within the loop nest.

In addition to the *doall* parallelism in the ADI example, *doacross* parallelism is also available in both loop nests. Figure 2.2(c) shows the parallel execution of the i_2 loop using *doacross* parallelism. Let i_1' be an iteration of the i_1 loop, and let i_2' be an iteration of the i_2 loop. If processor p executes $i_2 = i_2'$ and processor $p + 1$ executes $i_2 = i_2' + 1$, then p must communicate all the array elements it writes to $p + 1$. Synchronization is needed because processor $p + 1$ cannot execute iteration $(i_1, i_2) = (i_1', i_2' + 1)$ until p has executed iteration $(i_1, i_2) = (i_1', i_2')$. Parallelism is available along a diagonal or wavefront in the

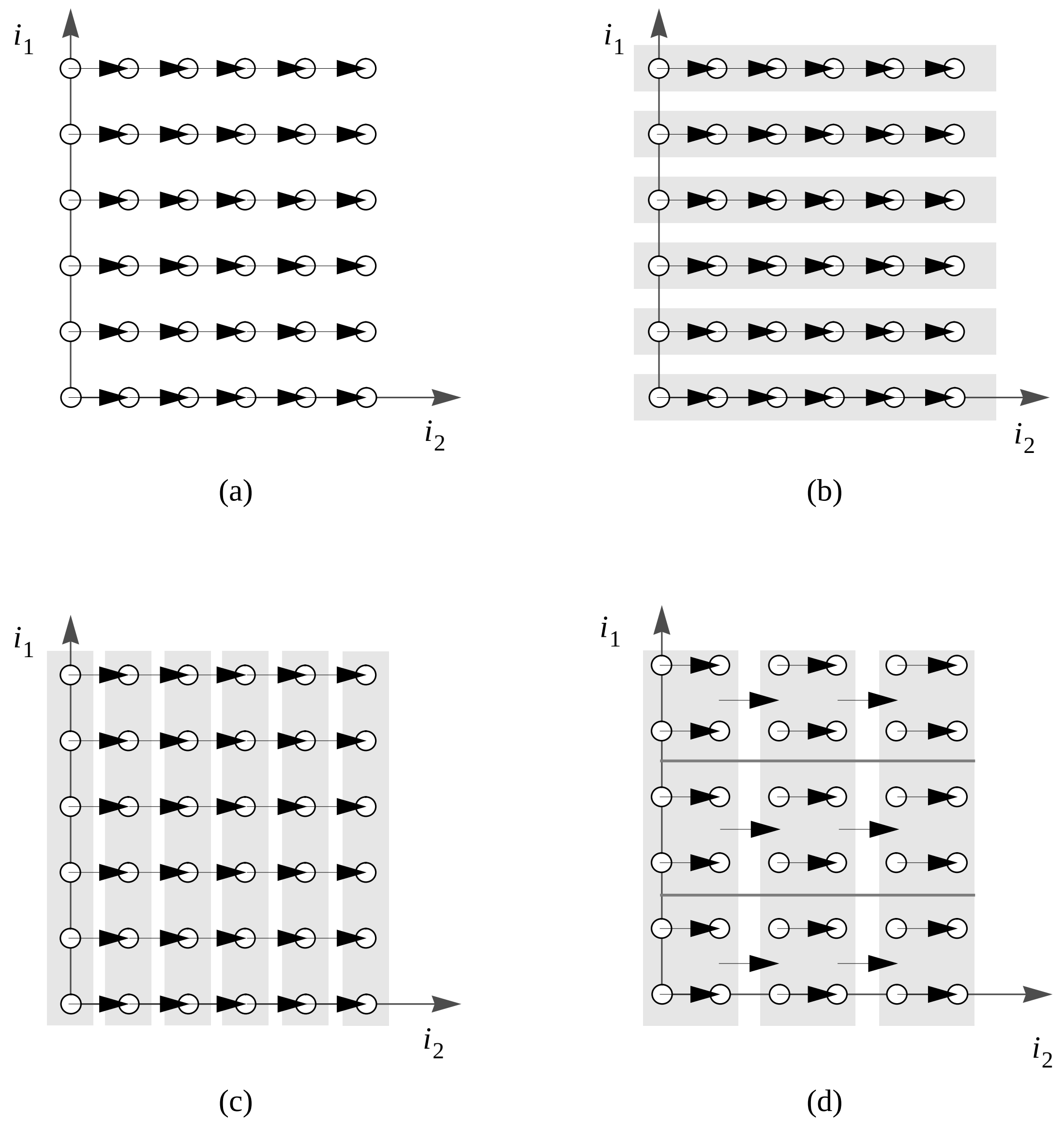


Figure 2.2: (a) Original iteration space for the loop nests in the ADI example. (b)–(d) Iteration spaces showing different ways to execute the loop nests in parallel. The arrows represent data dependences, and the iterations in each shaded region are assigned to the same processor.

iteration space, i.e. p can execute iteration $(i_1, i_2) = ({i_1}' + 1, {i_2}')$, while $p + 1$ executes $(i_1, i_2) = ({i_1}', {i_2}' + 1)$.

Another way to look at the parallelism in this example is to say that in Figure 2.2(b) we allocated iterations along the direction $(0, 1)$ to each processor, that is, all pairs of iterations that differ by $(0, 1)$ are assigned to the same processor. In Figure 2.2(c) we allocated iterations along the direction $(1, 0)$ to each processor. In fact, it is possible to exploit parallelism by allocating to each processor iterations along any direction within the two axes.

When using *doacross* parallelism, it is not very efficient to synchronize and communicate for every iteration of the loop. A general technique that is used to reduce the synchronization frequency and communication volume of parallel loops is *loop blocking*[78, 80] (also known as tiling, unroll-and-jam and stripmine-and-interchange). Blocking transforms a loop nest of depth l into a loop nest of depth $2l$. The inner l loops iterate over a fixed number of iterations (given by the block size), while the outer loops iterate across the inner blocks of iterations.

By blocking and then only parallelizing the outer loops, the synchronization frequency is reduced by the size of the block. The reduction in communication volume from blocking is a function of the loop's data dependences. Two array accesses are dependent within a loop i if there is a data dependent pair of iterations i' and i''. The references are said to be dependent with *distance* $d_i = i' - i''$. If the array accesses in a loop have small, finite dependence distances such that $0 \leq d_i < b$, where b is the block size, then only d_i elements at each block boundary need to move.

Fully permutable loop nests have several properties that are important for efficiently exploiting *doacross* parallelism. First, the fact that the loop nests are fully permutable means that *doacross* parallelism is legal, and the loop nest can be completely blocked[43, 77]. Also, for a fully permutable loop nest of depth l, when the dependences are distances then we know that $d_i \geq 0$ for all loops $i = 1 \ldots l$[77]. The result of this is that *doacross* parallelism and blocking can be applied along any dimension in the iteration space of such fully permutable loop nests.

Since the compiler first transforms the loop nests into the canonical form of nests of fully permutable subnests, the blocking transformation is easily applied. Figure 2.2(d) shows the parallel execution of the i_2 loop from the on-going ADI example using *doacross* parallelism and blocking.

2.3.2 Global Considerations

If we look at each loop nest individually, then distributing the iterations in the direction of the *doall* loops is preferable, as neither communication nor synchronization is necessary within the loop. However, this is not always the case when we analyze multiple loop nests together. Going back to the ADI example, consider what happens if we only try to exploit the parallelism in the *doall* loops. The *doall* loop in the first loop nest accesses rows of array x, whereas iterations of the *doall* loop in the second loop nest accesses columns of array x. Communication will occur between the loop nests because the data must be completely reorganized as the data decomposition switches between rows and columns.

We can avoid reorganizing the arrays between the two loop nests in the ADI example by using *doacross* parallelism in one of the loop nests. For example, in the second loop nest the loop with the loop-carried dependence (the i_2 loop), accesses rows of the array. If iterations of this loop are distributed across the processors, then in both loop nests processors access rows of the arrays. Communication and synchronization are now required within the second loop nest. However, since we use loop blocking to reduce the communication volume and synchronization frequency, the overhead incurred within the one loop nest is typically much less than the overhead to reorganize the data between the loop nests.

As this example illustrates, only exploiting the parallelism in the *doall* loops may not result in the best overall performance. In general, there may be tradeoffs between the best *local* loop-level decompositions, and the best overall *global* decompositions. Thus the loop-level analysis in our compiler transforms the code to expose the maximum degree of loop-level parallelism, but does not make decisions as to how that parallelism is to be implemented. The loop-level analysis leaves the code in a canonical format of nests of fully permutable loop nests, from which the coarsest degree of parallelism can be easily derived.

2.4 Decomposition Analysis Overview

The decomposition analysis takes as input the loop nests in the canonical form of nests of fully permutable loop nests. It analyzes the array accesses within the loop nest to calculate the mappings of data and computation onto the processors of the target machine. For each loop nest and for each array accessed in each loop nest, the decomposition analysis outputs a system of linear inequalities that describes the processor mappings.

The decomposition analysis only examines affine array accesses, and any non-affine accesses are ignored. All affine array access within a loop nest are examined, regardless of control-flow within the loop nest. Any non-perfectly nested accesses are treated as if they were perfectly nested, but with conditional guards (this is the model used by the SUIF compiler's parallelization analysis[77]). For example, for the following generalized two-deep loop nest with non-perfectly nested accesses:

```
for $i_1 = L_1$ to $U_1$ do
  x[$f_1(i_1)$] = ...
  for $i_2 = L_2(i_1)$ to $U_2(i_1)$ do
    y[$f_1{}'(i_1, i_2)$, $f_2{}'(i_1, i_2)$] = ...
  end for
  z[$f_1{}''(i_1)$] = ...
end for
```

The compiler models the code as:

```
for $i_1 = L_1$ to $U_1$ do
  for $i_2 = L_2(i_1)$ to $U_2(i_1)$ do
    if $i_2 = L_2(i_1)$ then x[$f_1(i_1)$] = ...
      y[$f_1{}'(i_1, i_2)$, $f_2{}'(i_1, i_2)$] = ...
    if $i_2 = U_2(i_1)$ then z[$f_1{}''(i_1)$] = ...
  end for
end for
```

Of course, this model is only legal if and only if the innermost i_2 loop in the original code executes at least one iteration whenever the outermost i_1 loop does. If we cannot prove this condition statically, then we must insert explicit checks. Since the decomposition analysis does not take control-flow within the body each loop nest into account, it ignores the guards and treats the non-perfectly nested accesses as if they were perfectly nested.

2.5 Parallel Code Generation and Optimization

The parallel code generator takes as input the linear inequalities representing the computation and data decompositions, and emits SPMD (single-program, multiple-data) parallelized code. The generated code is parameterized by the number of processors; each processor gets the total number of processors and its own processor identifier from calls to the run-time library. In this section, we describe parallel code generation for both distributed address space machines and shared address space machines. In the current SUIF compiler, only shared address space code generation is supported. Here we discuss the necessary steps to generate parallel code for both types of machines to show how the decomposition information is used, and to underscore the similarities between the techniques.

2.5.1 Distributed Address Space Machines

A parallel code generator for distributed address space machines is responsible for the following three main tasks:

1. Distribute the computation according to the computation decomposition. The bounds of the distributed loops are generated by applying a series of projection transformations to the polyhedron represented by the computation decomposition's system of linear inequalities[7, 8]. The resulting loop bounds are parameterized by the processor identifier so that each processor executes only the iterations that it has been assigned.

2. Allocate memory locally in each processor's address space for its portion of the distributed arrays. The global array addresses in the original program are then translated into local addresses. Each processor's portion of the arrays is given by the data decomposition.

3. Generate communication code (i.e. *send* and *receive* messages) whenever a processor accesses remote data. The decompositions are used to identify accesses to non-local elements. The data and computation decompositions, along with the array accesses, are composed into a single system of linear inequalities. The projection transformation is applied to the resulting system of inequalities to calculate the non-local accesses and the identity of the sending and receiving processors[7].

2.5.2 Shared Address Space Machines

When using decomposition information, the parallel code generator for shared address space machines is similar to the parallel code generator for distributed address space machines. It also performs three main tasks, described below. The items listed as *optional* are strictly optimizations and are not required for generating a correct parallel program.

1. Schedule the computation according to the computation decomposition. This step is performed in exactly the same way as for distributed address space machines, described in Step 1 of Section 2.5.1.

2. (*optional*) Make each processor's data contiguous in the shared address space. This improves the spatial locality of the application and eliminates cache conflicts and false sharing. The compiler manages the data placement both within a single array and across multiple arrays.

   To make the data of a single array accessed by each processor contiguous, the compiler transforms the data layout of the array in the virtual address space, and translates the array addresses in the original program into new addresses[9]. The data each processor accesses for each array is given by the array's data decomposition. These data transformations are analogous to global to local address translation for distributed address space machines, described in Step 2 of Section 2.5.1.

   To make the data across multiple arrays contiguous, we developed a technique called *compiler-directed page coloring*[18]. The operating system's page mapping policy determines the location of memory pages in physically indexed caches (external caches for most current processors are physically indexed). The compiler supplies the data decompositions of all the arrays to a run-time library. The run-time library then uses that information to direct the operating system's page allocation policy into making each processor's data contiguous in the physical address space. The operating system uses these hints to determine the virtual-to-physical page mapping when the pages are allocated.

3. (*optional*) Eliminate unnecessary synchronization. On shared address space machines, communication is performed by the hardware and explicit communication code (as described in Step 3 of Section 2.5.1) is unnecessary. Although communication analysis to identify accesses to non-local data is not required to ensure correctness, the same analysis can be used to optimize synchronization. Without any optimization, synchronization is implemented by inserting barriers at the end of each parallel loop to prevent potential data races. These barriers, however, cause overhead and can inhibit parallelism. By using the computation and data decompositions to identify exactly when and where accesses to non-local data occur, the compiler can eliminate unnecessary barrier synchronization or replace them with efficient point-to-point synchronization[73].

2.6 Uniprocessor Locality Optimizations

After generating the SPMD parallelized code, the compiler further optimizes the code that runs on each of the individual processors. In particular, the compiler optimizes for the memory hierarchy in a single processor. The parallelization and decomposition analysis are run before the uniprocessor locality optimizations because interprocessor communication is the most expensive form of data movement in the memory hierarchy and minimizing such communication is the most critical locality optimization.

The compiler's uniprocessor data locality optimizing algorithm uses the same loop transformation framework as the parallelization analysis[77, 78]. We apply the uniprocessor locality algorithm to the subnest consisting of the distributed parallel loops and their inner loops. The original loop structure differs from this subnest by having additional sequential loops outside the parallel loops. Since these sequential loops must be placed outermost due to legality reasons, the uniprocessor data locality is not compromised by parallelization.

Finally, to avoid a common source of cache conflicts in the on-chip cache, the compiler inserts padding between arrays in the virtual processor space[11]. This causes the starting addresses of the arrays to map to different locations in the cache.

2.7 Summary

This chapter described the context of the decomposition analysis within a complete parallelizing compiler system. The current domain of parallelizing compilers is scientific codes; in particular, when calculating computation and data decompositions, we only analyze loop bounds and array accesses that are affine functions of outer nested loop indices and symbolic constants.

The decomposition analysis relies on a pre-pass to optimize the parallelism within each individual loop nest. The parallelization analysis pre-pass puts the loop nests in a canonical form consisting of a nest of the largest possible fully permutable loop nests. This format exposes the maximum degree of loop-level parallelism, but does not constrain how the parallelism is to be implemented.

Once the computation and data decompositions are calculated, the parallel code generator uses the decompositions to create SPMD parallelized code. The transformations needed to generate optimized code for both shared and distributed address space machines are very similar. A unified compiler framework can thus be used for both kinds of architectures.

Chapter 3

Computation and Data Decomposition Basics

Decomposition analysis maps a program's computation and data onto the processors of a target machine. This chapter lays the foundation for the decomposition algorithm by describing our parallel architecture model, and by defining our representation of computation and data decompositions. In Section 3.1 we present the basic machine model that our decomposition algorithm targets. There are many different parallel architectures, and it is important to understand how the choice of machine model impacts the quality of the resulting decompositions. We also categorize the types of communication based on the data movement patterns, and discuss the communication costs on the target machine. Then in Section 3.2 we describe the mathematical framework we have developed for representing and calculating decompositions. We also discuss the key properties of the mathematical decomposition model that are fundamental to the decomposition algorithm.

3.1 Machine and Communication Models

The decomposition analysis models a machine with physically distributed memories, and each memory is associated with one processor. This architecture is shown in Figure 3.1. The decompositions calculated by the compiler map the computation onto the processors, and map the data onto the local memories associated with the processors. The decomposition analysis always models a separate memory for each processor, despite the fact that multiple processors may actually share memory on the true target machine. For example, this is the case for centralized shared address space machines, or for machines where each node is itself a centralized shared address space machine (e.g. Stanford DASH, SGI Origin). Regardless of the memory configuration of the true target machine, the fundamental problems we are solving are how to allocate the computation so as to minimize communication while maintaining sufficient parallelism, and how to allocate the data for good memory hierarchy performance. By targeting a machine with separate memories for each processor, we calculate decompositions such that each individual processor accesses the same data as much as possible. The resulting computation decompositions thus minimize inter-processor communication, independent of the memory architecture. The resulting data decompositions specify exactly which data are accessed by each individual processor. This information is then used to tailor the data allocation for the specific target architectures. For example, as described in Section 2.5.2, on shared address space machines the parallel code generator uses the data decompositions to transform the data layout so that each processor's data are contiguous in the shared address space.

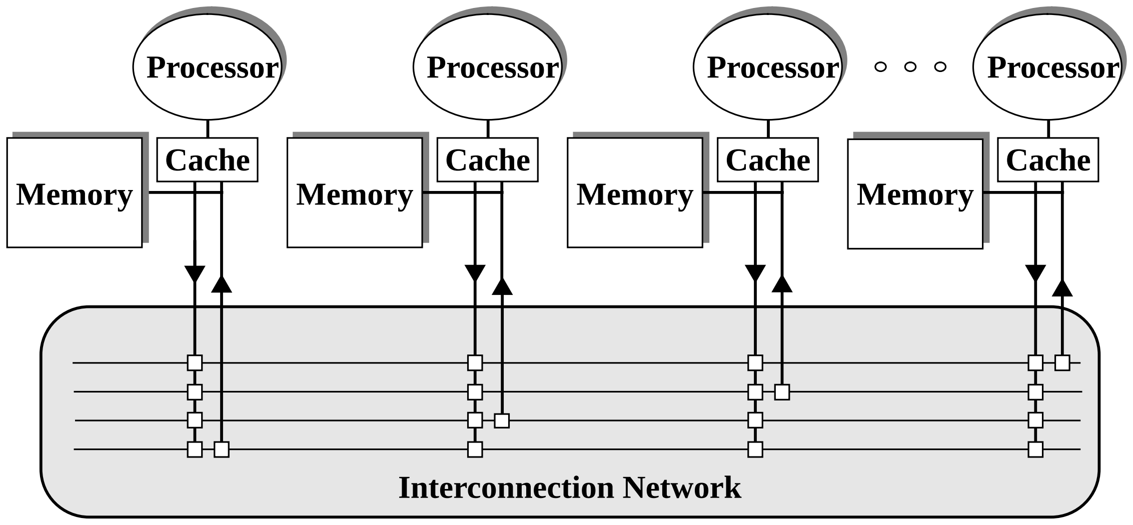


Figure 3.1: Basic machine model used by the decomposition analysis.

The interconnection network of the target machine is a key factor in determining the cost of communication. There are many different network topologies that can be used to build the interconnection network for parallel architectures. We model a *fully-connected* interconnection network, where all processors are directly connected to one another. Examples of fully-connected networks include crossbars and buses. The decomposition analysis assumes that the communication time between any two processors is the same, regardless of the specific processors involved. Thus, the main factor in determining the cost to access a data element is whether that access is to the local processor or to a remote processor. Hot-spotting effects can cause the communication time to a particular processor to become disproportionately large, making the communication costs between different pairs of processors non-uniform. In this case, the remote access times continue to dominate the local access times, and the decomposition analysis will still optimize to eliminate the remote accesses. In general, our communication model is reasonable for most parallel machines today, since the ratio of local access time to remote access time is typically much greater than the ratio of the maximum remote access time to the minimum remote access time. The decomposition analysis models local accesses as having zero cost and models all remote accesses as having equal cost. As a result, the communication costs calculated by the decomposition algorithm are directly proportional to the *amount* of data accessed by remote processors.

The communication patterns in applications determine the amount of data that is being moved. We make the distinction between two communication patterns, *nearest-neighbor communication* and *data-reorganization communication*. When the communication pattern is nearest-neighbor shifts of data, then the amount of data transferred can be significantly reduced by assigning blocks of contiguous iterations of a distributed loop to the same processor. In this way, any nearest-neighbor communication between the iterations in the same block is eliminated. We describe our use of blocking for reducing the cost of nearest-neighbor communication in more detail in Section 4.2. Data-reorganization communication is unstructured and requires general movement of the entire data structure, for example, transposing a distributed array. We thus consider nearest-neighbor communication to be inexpensive compared to data-reorganization communication.

3.2 Mathematical Decomposition Model

This section describes our mathematical framework for expressing and calculating decompositions. We represent decompositions as two separate components. First, the computation and data are mapped onto a virtual processor space. The virtual processor space has as many processors as is needed to fit the number of loop iterations and the sizes of the arrays. Second, the processors of the virtual processor space are mapped onto the physical processors of the target machine. This representation is general enough to express a broad class of decompositions, including all the decompositions available to HPF programmers. Section 3.2.1 defines the data and computation mappings onto the virtual processor space, and Section 3.2.2 describes the virtual processor mapping onto the physical processor space.

3.2.1 Virtual Processor Mapping

A loop nest of depth l defines an iteration space $\mathcal{I}$. Each iteration of the loop nest is identified by its index vector $\vec{\imath} = (i_1, i_2, \ldots, i_l)$. An array of dimension m defines an array space $\mathcal{A}$, and each element in the array is accessed by an index vector $\vec{a} = (a_1, a_2, \ldots, a_m)$. Similarly, an n-dimensional processor array defines a processor space $\mathcal{P}$. We consider affine array access functions $\vec{f} : \mathcal{I} \rightarrow \mathcal{A}$, $\vec{f}(\vec{\imath}) = F\vec{\imath} + \vec{\zeta}$, where F is a linear transformation and $\vec{\zeta}$ is a constant vector. The mappings of computation and data onto the virtual processor space are represented by affine functions and are called *affine decompositions*.

Definition 3.2.1 *Let* $\vec{a} = (a_1, a_2, \ldots, a_m)$ *be an index vector for an* m*-dimensional array. The* **affine data decomposition** *of the array onto an* n*-dimensional processor space is an affine function* $\vec{d} : \mathcal{A} \rightarrow \mathcal{P}$*, where*

$$\vec{d}(\vec{a}) = D\vec{a} + \vec{\delta}$$

D *is an* $n \times m$ *linear transformation matrix and* $\vec{\delta}$ *is a constant vector.*

Definition 3.2.2 *Let* $\vec{\imath} = (i_1, i_2, \ldots, i_l)$ *be an index vector for a loop nest of depth* l*. The* **affine computation decomposition** *of the loop nest onto an* n*-dimensional processor space* *is an affine function $\vec{c} : \mathcal{I} \to \mathcal{P}$, where*

$$\vec{c}(\vec{\imath}) = C\vec{\imath} + \vec{\gamma}$$

C is an $n \times l$ linear transformation matrix and $\vec{\gamma}$ is a constant vector.

We also define the *linear decomposition* as the linear transformation part of the affine decomposition and the *offset decomposition* as the constant part of the affine decomposition. Mathematically, the linear data and computation decompositions are represented by the matrices D and C from the above definitions, and the offset decompositions are represented by the constant vectors $\vec{\delta}$ and $\vec{\gamma}$, respectively.

In our model, all the statements within a loop nest are treated as a single unit. For each iteration $\vec{\imath}$ of a loop nest, the affine computation decomposition function $\vec{c}(\vec{\imath})$ specifies the virtual processor which executes all statements of iteration $\vec{\imath}$. We do not consider finding separate affine functions for each statement within the loop nest.

3.2.1.1 Properties of Affine Decompositions

In this section we describe several key mathematical properties of affine decompositions. The range of an array access function $\vec{f}(\vec{\imath}) = F\vec{\imath} + \vec{\zeta}$ is the subspace of the array space accessed by that reference, and is denoted by S:

$$S = \text{range}\left(\vec{f}\right) \tag{3.1}$$

For an array of dimension m, whenever rank$(F) < m$, then S is a proper subset of the array space, $S \subset \mathcal{A}$.

Let $\vec{d}$ be the affine data decomposition from Definition 3.2.1. Two array elements $\vec{a}_1, \vec{a}_2 \in S$ are allocated to the same virtual processor if and only if

$$\begin{aligned} \vec{d}(\vec{a}_1) &= \vec{d}(\vec{a}_2) \\ D\vec{a}_1 + \vec{\delta} &= D\vec{a}_2 + \vec{\delta}, \end{aligned}$$

that is,

$$D(\vec{a}_1 - \vec{a}_2) = 0 \text{ or } \vec{a}_1 - \vec{a}_2 \in \mathcal{N}(D).$$

Here $\mathcal{N}(D)$ is *nullspace* of the matrix D, where the nullspace of a matrix D is the vector space consisting of all vectors $\vec{a}$ such that $D\vec{a} = \vec{0}$. Conversely, any two array elements such that $(\vec{a}_1, \vec{a}_2 \in S) \wedge ((\vec{a}_1 - \vec{a}_2) \notin \mathcal{N}(D))$ are assigned to different virtual processors and are considered distributed.

Let $\vec{c}$ be the affine computation decomposition from Definition 3.2.2. Two iterations $\vec{\imath}_1, \vec{\imath}_2 \in \mathcal{I}$ are executed on the same virtual processor if and only if

$$\begin{aligned} \vec{c}(\vec{\imath}_1) &= \vec{c}(\vec{\imath}_2) \\ C\vec{\imath}_1 + \vec{\gamma} &= C\vec{\imath}_2 + \vec{\gamma}, \end{aligned}$$

that is,

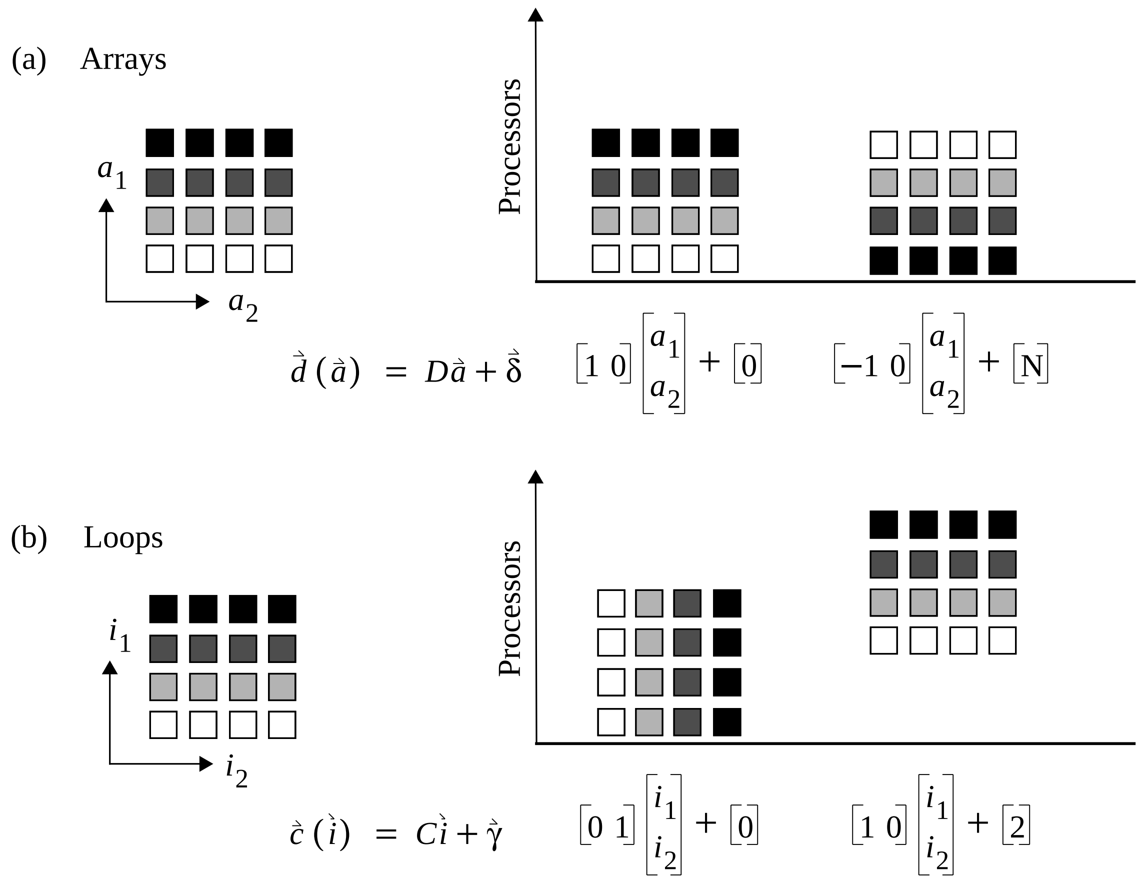
$$C(\vec{\imath}_1 - \vec{\imath}_2) = 0, \text{ or } \vec{\imath}_1 - \vec{\imath}_2 \in \mathcal{N}(C).$$

where $\mathcal{N}(C)$ is nullspace of the matrix C. Any two iterations $\vec{\imath}_1, \vec{\imath}_2 \in \mathcal{I}$ such that $(\vec{\imath}_1 - \vec{\imath}_2) \notin \mathcal{N}(C)$ are said to be distributed and are run on different virtual processors.

Definition 3.2.3 *Given an affine computation decomposition function $\vec{c}(\vec{\imath}) = C\vec{\imath} + \vec{\gamma}$ for a loop nest, the* **degree of parallelism** *is the rank of the linear computation decomposition matrix C. This is equivalent to $l - \dim(\mathcal{N}(C))$, where l is the depth of the loop nest.*

Figure 3.2 shows sample affine decompositions onto a virtual processor space and the corresponding affine functions. Figure 3.2(a) shows affine data decompositions for a two-dimensional array onto a one-dimensional virtual processor space. Figure 3.2(b) shows affine computation decompositions for a two-deep loop nest onto a one-dimensional virtual processor space. In the figure, the elements are shaded to identify their positions.

Given a loop iteration or array element, the affine decomposition assigns that iteration or array element to a specific virtual processor. The data and computation that are assigned to the same processor are represented mathematically by the nullspaces of the matrices D and C, respectively. For example, for both affine data decompositions shown in Figure 3.2(a), elements along the direction $(0, 1)$ (i.e. each row) are assigned to the same processor and thus $\mathcal{N}(D) = \text{span}\,\{(0, 1)\}$. In the first affine computation decomposition shown in



(a) Arrays
a_1
a_2
Processors
$\vec{d}(\vec{a}) = D\vec{a} + \vec{\delta}$
$\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}a_1\\a_2\end{bmatrix} + \begin{bmatrix}0\end{bmatrix}$
$\begin{bmatrix}-1 & 0\end{bmatrix}\begin{bmatrix}a_1\\a_2\end{bmatrix} + \begin{bmatrix}N\end{bmatrix}$
(b) Loops
i_1
i_2
Processors
$\vec{c}(\vec{i}) = C\vec{i} + \vec{\gamma}$
$\begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}i_1\\i_2\end{bmatrix} + \begin{bmatrix}0\end{bmatrix}$
$\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}i_1\\i_2\end{bmatrix} + \begin{bmatrix}2\end{bmatrix}$

Figure 3.2: Sample affine decompositions onto a virtual processor space: (a) affine data decompositions and (b) affine computation decompositions. The elements are shaded to identify their positions.

Figure 3.2(b), iterations along the direction $(1, 0)$ (i.e. iterations of i_1) are on the same processor and $\mathcal{N}(C) = \text{span}\{(1, 0)\}$. In the second affine computation decomposition iterations along $(0, 1)$ (i.e. iterations of i_2) are assigned to the same processor and $\mathcal{N}(C) = \text{span}\{(0, 1)\}$.

3.2.1.2 Relationship between Computation and Data

The computation and data in a program are related by the array access functions. Let the affine computation decomposition for a loop nest j be represented by $\vec{c}_j(\vec{\imath}) = C_j(\vec{\imath}) + \vec{\gamma}_j$, and let the affine data decomposition for an array x be represented by $\vec{d}_x(\vec{a}) = D_x(\vec{a}) + \vec{\delta}_x$. Furthermore, let $\vec{f^k_{xj}}$ be the kth array access function for array x in loop nest j. No communication is required if it is possible to define an affine computation decomposition for each loop nest j and an affine data decomposition for each array x such that the following equation holds for all array access functions k:

$$D_x(\vec{f^k_{xj}}(\vec{\imath})) + \vec{\delta}_x = C_j(\vec{\imath}) + \vec{\gamma}_j \tag{3.2}$$

A trivial solution that guarantees zero communication is to place all the data on a single processor by setting the affine data decompositions such that $\forall x, \vec{d}_x = 0$. By Equation 3.2, this means that the affine computation decompositions are such that $\forall j, \vec{c}_j = 0$, and all the computation executes sequentially. Therefore for all loop nests j, $\mathcal{N}(C_j)$ would span the entire iteration space $\mathcal{I}$, and for all arrays x, $\mathcal{N}(D_x)$ would span the entire array space $\mathcal{A}$. The objective, however, is to maximize parallelism while incurring as little communication as possible. Maximizing parallelism means finding affine data and computation decompositions such that for all loop nests j, rank (C_j) is as large as possible, or equivalently, $\mathcal{N}(C_j)$ is as small as possible.

If it is possible to find a single non-trivial, affine decomposition with no communication, then there exist many equivalent affine decompositions with the same degree of parallelism and no communication. For example, given a communication-free affine decomposition we could always transpose all the data and computation and still have no communication.

Communication due to mismatches in the linear transformation part of an affine decomposition are expensive since they require data reorganization for entire arrays. In contrast, communication at the offset level is typically inexpensive nearest-neighbor communication. We thus also consider the version of Equation 3.2 that omits the offsets. Only nearest-neighbor communication is required if it is possible to define a linear computation decomposition for each loop nest j and a linear data decomposition for each array x such that the following equation holds for all array access functions k. Letting $\vec{f^k_{xj}}(\vec{\imath}) = F^k_{xj}(\vec{\imath}) + \vec{\zeta^k_{xj}}$,

$$D_x F^k_{xj}(\vec{\imath}) = C_j(\vec{\imath}) \tag{3.3}$$

3.2.2 Physical Processor Mapping

The virtual processors in each dimension are mapped onto the physical processors of the target machine via one of the following folding functions: BLOCK, CYCLIC or BLOCK-CYCLIC(b), where b is the block size. A BLOCK folding function means that $\left\lceil \frac{P_v}{P_p} \right\rceil$ contiguous virtual processors are assigned to each physical processor, where P_v is the number of virtual processors and P_p is the number of physical processors. With a CYCLIC folding function each virtual processor is mapped to a physical processor using a round-robin assignment. Similarly, with BLOCK-CYCLIC(b), b contiguous virtual processors are assigned round-robin to the physical processors.

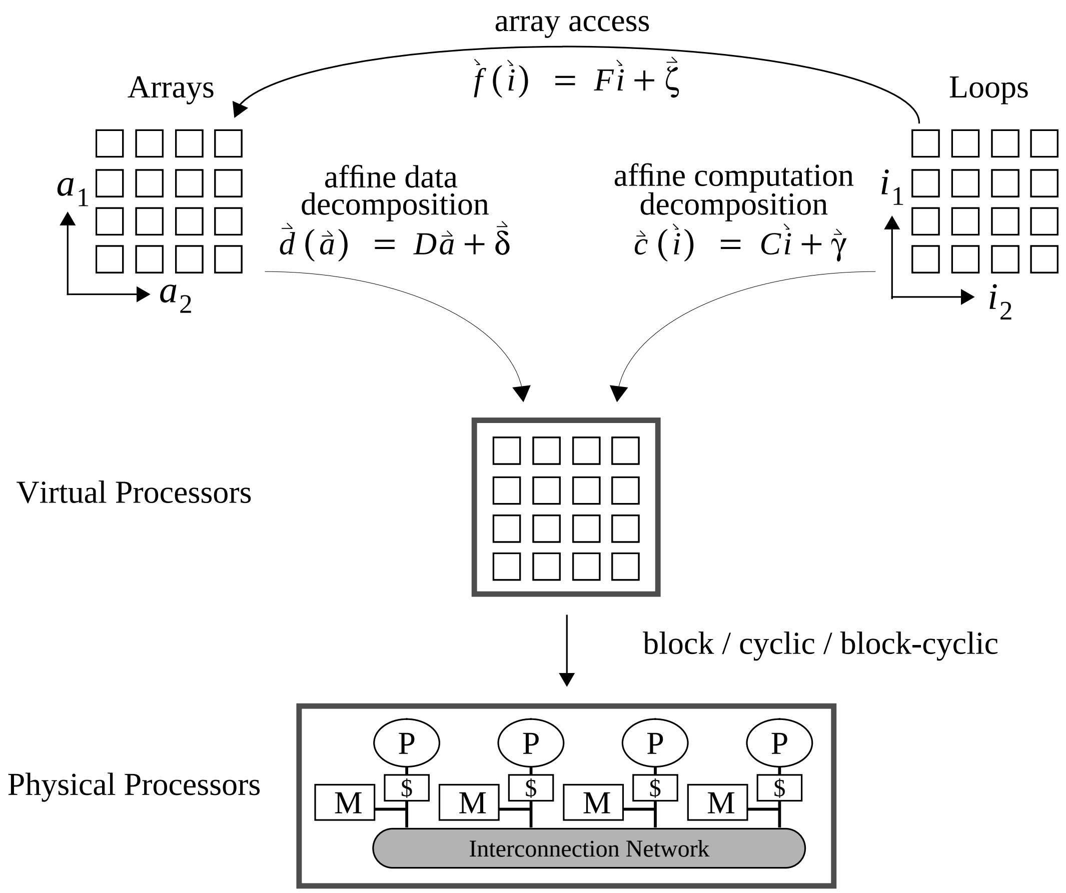
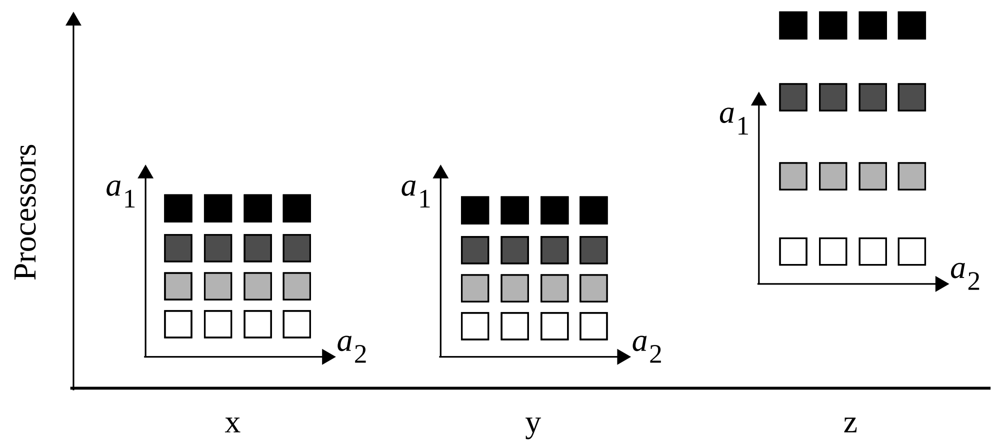

array access
$\vec{f}(\vec{i}) = F\vec{i} + \vec{\zeta}$
Arrays
Loops
a_1
a_2
affine data decomposition
$\vec{d}(\vec{a}) = D\vec{a} + \vec{\delta}$
affine computation decomposition
$\vec{c}(\vec{i}) = C\vec{i} + \vec{\gamma}$
i_1
i_2
Virtual Processors
block / cyclic / block-cyclic
Physical Processors
P
$
M
Interconnection Network

Figure 3.3: Two-step model of computation and data decompositions.

```
real x(N,N), y(N,N), z(N,N)
template T(2*N+3,3*N), distribute(block, *)
align x(I,J) with T(I,J)
align y(I,J) with T(I,3*J)
align z(I,J) with T(2*I+3,J)
```



affine function: $\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}a_1 \\ a_2\end{bmatrix} + \begin{bmatrix}0\end{bmatrix}$ $\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}a_1 \\ a_2\end{bmatrix} + \begin{bmatrix}0\end{bmatrix}$ $\begin{bmatrix}2 & 0\end{bmatrix}\begin{bmatrix}a_1 \\ a_2\end{bmatrix} + \begin{bmatrix}3\end{bmatrix}$

folding function: block block block

Figure 3.4: Example of HPF decompositions and the corresponding virtual and physical processor mappings under our decomposition model.

A complete decomposition thus consists of an affine function for the virtual processor mapping plus a folding function for the physical processor mapping. This is shown in Figure 3.3.

The two-step model used by our compiler can represent a superset of the decompositions available to HPF programmers. The affine decompositions in our model determine the array alignments in HPF. The rank of the linear transformation part of the affine function specifies the number of distributed dimensions – this corresponds to the dimensions in the HPF `distribute` statement that are not marked as "`*`". The folding functions (BLOCK, CYCLIC and BLOCK-CYCLIC) correspond directly to those used in the `distribute` statement in HPF. For example, Figure 3.4 shows sample HPF distributions with the corresponding virtual and physical processor mappings under our model. The `template` directive declares a two-dimensional template of size $(2N+3) \times 3N$ and the `distribute` directive distributes the first dimension across a one-dimensional processor space. The array x is aligned directly onto the template. The first dimension of array y is aligned directly onto the template, and the second dimension is aligned with a stride of 3. Note that even though x and y have different alignments in the second dimension, their decompositions in our model are identical since the second dimension is local to each processor. The first dimension of array z is aligned with a stride of 2 and an offset of 3. The stride translates directly to the linear transformation matrix of the affine decomposition, and the offset becomes the offset decomposition.

3.3 Summary

In this chapter we first described the machine model used by the decomposition analysis. We model a machine with physically distributed memories, and each memory is associated with a single processor. We assume that the communication time between any two processors is the same, regardless of the specific processors involved. Thus the main factor in determining the cost to access a data element is whether that access is to the local processor or to a remote processor.

Next we described our mathematical model of decompositions. We model decompositions in two steps: first an affine function maps the computation and data onto a virtual processor space, and second a folding functions maps the virtual processor space onto the physical processors of the target machine. We also presented a system of equations that describes the relationship between the computation and data decompositions such that there is no communication.

Chapter 4

A Static-Decomposition Algorithm

We begin the presentation of our decomposition algorithm in this chapter by considering the restricted problem of calculating *static decompositions*. A decomposition is static when there is a single data decomposition for each array throughout the entire program region being analyzed. We present an algorithm that finds static computation and data decompositions with the maximum degree of parallelism. In the following chapter, we will use the static-decomposition algorithm as a component of the algorithm for finding *dynamic decompositions*, in which the data decompositions are allowed to change across different loop nests.

The static-decomposition algorithm is based on the two-step mathematical model of decompositions presented in the previous chapter. In Section 4.1 we present the algorithm for finding static decompositions onto the virtual processor space. Then in Section 4.2 we describe how the virtual processor space is mapped onto the physical processor space.

4.1 Finding Virtual Processor Mappings

Our algorithm for mapping computation and data onto the virtual processor space generates affine decompositions. The algorithm sets up a system of equations that the desired affine decompositions must satisfy, and then solves for the decompositions. The key issues we address are how to formulate the system of equations, and then how to solve the equations efficiently. We begin in Section 4.1.1 by giving an overview of the strategy behind the algorithm, and the subsequent sections present the details of the algorithm.

4.1.1 General Approach for Finding Affine Decompositions

From Section 3.2.1.2, we know that the data are always local to the processor accessing the data when Equation 3.2 holds for all arrays x, loop nests j and array accesses k:

$$D_x(\vec{f^k_{xj}}(\vec{\imath})) + \vec{\delta}_x = C_j(\vec{\imath}) + \vec{\gamma}_j$$

We can always find a strictly communication-free decomposition by creating a system of these equations, and then solving directly for the affine computation decompositions, $C_j(\vec{\imath}) + \vec{\gamma}_j$, and the affine data decompositions, $D_x(\vec{a}) + \vec{\delta}_x$.

Often, however, the only solution with strictly no communication is the trivial solution that maps all the data and computation onto a single processor. Since each equation in the system represents a requirement on the data and computation decompositions, it is possible that the only solution that satisfies all these requirements is the trivial solution. Thus rather than requiring that Equation 3.2 strictly holds for all accesses to every array in every loop nest, we relax the equations to allow limited types of communication. As described in Section 3.2.1.2, communication due to mismatches in the linear transformation part of an affine decomposition are more expensive than mismatches in the offset part. Our strategy is to split the affine decomposition problem into two main steps: we first solve for the linear decompositions – this determines the linear transformation part of the affine function that represents an affine decomposition, and then we solve for the offset decompositions.

To calculate the linear decompositions, we use the version of Equation 3.2 that omits the offsets, Equation 3.3. This will lead to a solution where the linear decompositions have no data-reorganization communication, but may still have nearest neighbor communication due to offsets. Then after finding the linear computation and data decompositions C_j and D_x, respectively, the algorithm finds the offset decompositions $\vec{\gamma}_j$ and $\vec{\delta}_x$ to form the complete affine decompositions.

When the decompositions are truly communication-free, no synchronization is necessary. However, linear decompositions that allow nearest-neighbor communication can require synchronization within and/or across loop nests. Since the parallelism is not as effective in loop nests that require synchronization as in loop nests with no synchronization, we would prefer to find a linear decomposition where synchronization is only needed outside the loop nest, rather than a linear decomposition where synchronization is needed within the loop nest. Based on this observation, we first restrict our attention to problem of calculating linear decompositions that have no synchronization within the loop nests. In other words, at this point we only consider distributing iterations of outermost *doall* loops. We refer to these decompositions as *basic linear decompositions*. The loop nests in the program have been transformed into nests of fully permutable loop nests by the previous parallelization phase, and the parallel loops have been moved to the outermost possible position within each fully permutable nest (Section 2.3.1). The algorithm for calculating basic linear decompositions is presented in Section 4.1.2. We describe how to generate a system of equations that represents basic linear decompositions, and discuss how to solve for the decompositions.

Next, we consider additional forms of communication and synchronization. Section 4.1.3 shows how we modify the system of equations for basic linear decompositions to allow replication communication. Then, in Section 4.1.4 we again modify the system of equations to add regular synchronization within fully permutable loop nests. Section 4.1.5 gives a summary of the full linear decomposition algorithm. Finally, in Section 4.1.6, we describe how to find the offset decompositions that complete the affine decompositions.

4.1.2 Basic Linear Decompositions

Our algorithm for finding basic linear decompositions has the important property that it finds basic linear decompositions that have the maximum degree of parallelism possible (we prove this property in Theorem 4.1.8). We require that the bounds of the loops are sufficiently large to keep all the processors of the target machine busy. In other words, we only care about how many loops are parallelized, not which individual loops. If it can be determined statically that the bounds of a loop are small, then that loop is not considered parallelizable.

In Section 4.1.2.1 we describe how to formulate the system of equations that describes basic linear decompositions, and show some examples in Section 4.1.2.2. We then present our method for solving the system of equations in Section 4.1.2.3.

4.1.2.1 Formulating the Equations

Basic linear decompositions have no data-reorganization communication and no synchronization within the loop nests. No synchronization within a loop nest means that iterations of loops that are not outermost parallel loops must be assigned to the same processor. These equations are called the *synchronization equations*. No data-reorganization communication within and across loop nests means that using Equation 3.3, we must ensure that for all loop nests j, for all access functions k to arrays x in the loop nests, $D_x F^k_{xj}(\vec{\imath}) = C_j(\vec{\imath})$. The equations that result from this requirement are called the *communication equations*.

Synchronization Equations. The synchronization equations describe the loops that are assigned to the same processor. As we are only considering the case where there is no synchronization within a loop nest, any loops that are not outermost parallel loops must be assigned to the same processor. Formally, for a loop nest j of depth l, let loops $1 \ldots s$ be the outermost parallel loops in the loop nest. Then for all loops $q = (s+1) \ldots l$, iterations $\vec{\imath}$ and $\vec{\imath} + \vec{e_q}$ must be assigned to the same processor, where $\vec{e_q}$ is the qth elementary vector of dimension l [1]:

$$\begin{aligned} C_j(\vec{\imath} + \vec{e_q}) &= C_j(\vec{\imath}) \\ C_j((\vec{\imath} + \vec{e_q}) - \vec{\imath}) &= \vec{0} \\ C_j(\vec{e_q}) &= \vec{0} \end{aligned}$$

Thus for each loop nest j, and for each loop $q = (s+1) \ldots l$ we generate the following:

$$C_j(\vec{e_q}) = \vec{0} \tag{4.1}$$

We also generate synchronization equations for loops with small bounds to guarantee that they will execute on the same processor.

Communication Equations. The communication equations must be satisfied for there to be no data-reorganization communication. We simplify Equation 3.3, $D_x F^k_{xj}(\vec{\imath}) = C_j(\vec{\imath})$,

[1] The qth elementary vector, written $\vec{e_q}$, has a 1 in the qth position and zero in all other positions.

by eliminating the iteration space vector $\vec{\imath}$ on both sides of the communication equation to give $D_x F^k_{xj} = C_j$. The equation must now hold for all integer vectors of length l, where l is the depth of the loop nest, whether or not they are actually within the bounds of the loops. For each loop nest j, for each array x and for each array access function k, we generate:

$$D_x F^k_{xj} \quad = \quad C_j \tag{4.2}$$

Together, the synchronization and communication equations represent the necessary conditions for a linear decomposition that allows only nearest-neighbor communication and no synchronization within a loop nest. There can be many possible solutions to this system of equations, including the trivial solution that assigns all the computation and data to a single processor. The objective, however, is to find a solution with the maximum degree of parallelism. Mathematically, this corresponds to finding linear data and computation decompositions such that for all loop nests j, rank (C_j) is as large as possible.

4.1.2.2 Examples of Basic Linear Decompositions

For the single loop nest shown in Figure 4.1(a), the array access functions for the four references `x[`i_1`,`i_2`,`i_3`]`, `x[`i_1`,`i_2`,`i_3`-1]`, `y[`i_1`,N-`i_2`+1,`i_3`]` and `y[`i_1`,`i_2`,`i_3`]` are

$$\begin{aligned}
\vec{f}^{\,1}_{x1}(\vec{\imath}) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \\
\vec{f}^{\,2}_{x1}(\vec{\imath}) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}, \\
\vec{f}^{\,1}_{y1}(\vec{\imath}) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} 0 \\ N+1 \\ 0 \end{bmatrix} \quad \text{and} \\
\vec{f}^{\,2}_{y1}(\vec{\imath}) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}
\end{aligned}$$

respectively. These array accesses are affine functions of the form $\vec{f}(\vec{\imath}) = F\vec{\imath} + \vec{\zeta}$. Since here we are only concerned with the linear transformation part of the affine functions, we consider only the array access matrices:

$$F^1_{x1} = F^2_{x1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad F^1_{y1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad F^2_{y1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
real x[N,N,N], y[N,N,N]
/* Loop nest 1 */
for $i_1$ = 1 to N do              /* doall */
  for $i_2$ = 1 to N do            /* doall */
    for $i_3$ = 1 to N do
      x[$i_1$,$i_2$,$i_3$] = x[$i_1$,$i_2$,$i_3$-1] + y[$i_1$,N-$i_2$+1,$i_3$] + y[$i_1$,$i_2$,$i_3$]
```

(a)

```
real x[N,N], y[N,N], z[N,N]
/* Loop nest 1 */
for $i_1$ = 1 to N do              /* doall */
  for $i_2$ = 1 to N do            /* doall */
    x[$i_1$,$i_2$] = y[$i_1$,$i_2$] + z[$i_1$,$i_2$]

/* Loop nest 2 */
for $i_1$ = 1 to N do              /* doall */
  for $i_2$ = 1 to N do            /* doall */
      y[$i_1$,$i_2$] = y[$i_1$,$i_2$] + x[$i_2$,$i_1$]
```

(b)

Figure 4.1: Code examples used to illustrate linear decompositions: (a) single loop nest and (b) multiple loop nests.

The innermost i_3 loop is not parallel, which results in the following synchronization equation:

$$C_1(\vec{e_3}) = \vec{0}$$

This equation ensures that all iterations of the i_3 loop are assigned to the same processor. The communication equations are as follows:

$$\begin{aligned} D_x F^1_{x1} &= C_1 \\ D_x F^2_{x1} &= C_1 \\ D_y F^1_{y1} &= C_1 \\ D_y F^2_{y1} &= C_1 \end{aligned}$$

These equations ensure that for each iteration of the loop nest, the processor that executes that iteration must also be assigned elements $F^{1,2}_{x1}$ of array x and elements F^1_{y1} and F^2_{y1} of array y. The complete set of equations for the example are as follows, after eliminating all redundant equations:

$$\begin{aligned} C_1 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} &= \vec{0} \\ D_x \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} &= C_1 \\ D_y \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} &= C_1 \\ D_y \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} &= C_1 \end{aligned}$$

A solution with the maximum degree of parallelism for these equations is:

$$C_1 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}, \ D_x = D_y = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

The degree of parallelism is $\text{rank}(C_1) = 1$. This solution corresponds to distributing iterations of the i_1 loop across the processors.

To find basic linear decompositions across multiple loop nests, the equations for each individual loop nest are simply merged into a single system of equations. For example, the complete set of communication equations for the two loop nests in Figure 4.1(b) is:

$$\begin{array}{rclcrcl} D_x \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & = & C_1 & \qquad & D_x \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & = & C_2 \\ D_y \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & = & C_1 & \qquad & D_y \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & = & C_2 \\ D_z \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & = & C_1 & & & & \end{array}$$

There are no synchronization equations since there are no loops with dependences. This example also illustrates why the system of equations describes a static decomposition, and not a dynamic decomposition. For each array x there is only one decomposition variable D_x across all loop nests, and thus the solution will have a single linear data decomposition for each array. A solution with the largest possible degree of parallelism for these equations is

$$C_1 = C_2 = \begin{bmatrix} 1 & 1 \end{bmatrix}, \;\; D_x = D_y = D_z = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

The degree of parallelism for this example is rank $(C_1) = $ rank $(C_2) = 1$.

4.1.2.3 Solving the Equations

This section describes how we solve the system of equations that describes a linear decomposition. The objective is to find a solution with the maximum degree of parallelism. It is also important that the solution is computed *incrementally*: we need to be able to efficiently build up the solution and avoid re-solving the equations unnecessarily as we examine larger and larger regions of the program. Even though in this chapter we are considering the sub-problem of finding static decompositions for a single region in the program, later we rely on the incremental nature of the algorithm in Chapter 5 for finding dynamic decompositions and in Chapter 6 for finding interprocedural decompositions.

Given a linear computation decomposition matrix C_j for a loop nest j, the degree of parallelism is represented mathematically by rank (C_j) (Definition 3.2.3). Let l be the depth of loop nest j, then rank $(C_j) = l - \dim(\mathcal{N}(C_j))$. Thus maximizing the degree of parallelism means finding linear decompositions such that rank (C_j) is as large as possible, or equivalently, the dimensionality of $\mathcal{N}(C_j)$ is as small as possible.

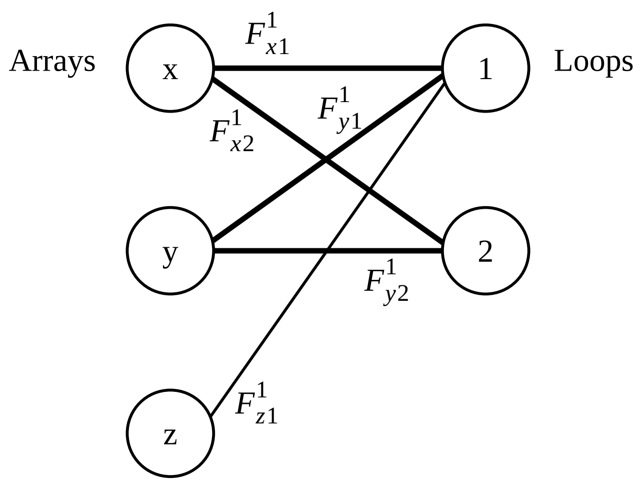
In general, for arrays x and loop nests j, the nullspaces of the linear decomposition matrices D_x and C_j represent the data and computation that are assigned to the same processor, respectively. We observe that the data and computation that go on the same processor is the major factor that determines the amount of parallelism and communication, not the individual processor to which the data and computation are assigned. There are many different, yet equivalent, linear decompositions with the same nullspaces. An important characteristic of our algorithm is that we first find the nullspaces that are guaranteed to lead to the desired linear decompositions. We find the smallest possible nullspaces for which a solution to the system of equations exists. Then a simple calculation is used to find the corresponding linear transformation matrices. The nullspaces serve as a succinct representation of the constraints on what data and computation must be assigned to the same processor. When a new equation is added to the system, we need only update the nullspaces, not re-solve the entire system of equations.

Based on our mathematical framework, presented in [10], Bau et al. have developed an alternative method for solving a system of equations to calculate maximum rank affine decompositions[15]. Their solution, however, requires re-solving the entire system of equations whenever a new equation is added.

Solver Representation

To solve for the linear decompositions, we represent the computation and data in the program region by a bipartite *interference graph*, $G_s = (V_c, V_d, E)$. The loop nests form one set of vertices V_c, and the arrays form the other set of vertices V_d. There is an undirected edge between an array and a loop nest for each array access function for the array in the loop nest. For example, the interference graph for the code in Figure 4.1(b) is shown in Figure 4.2.

An edge $e^k_{xj} \in E$ corresponds to the kth array access function, F^k_{xj}, for array x in loop nest j. The linear decompositions for array x and loop nest j are related by the communication equation, $D_x F^k_{xj} = C_j$. An important property of the interference graph representation is that if there is an edge between array x and loop nest j, we can calculate a linear computation decomposition C_j given the linear data decomposition D_x, and vice-versa. We use this property later in subsequent sections to calculate the nullspaces of the



$$F^1_{x1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$F^1_{y1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$F^1_{z1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$F^1_{x2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$F^1_{y2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figure 4.2: The interference graph corresponding to the code in Figure 4.1(b). The bold edges show a cycle in the graph.

linear decomposition matrices and also to calculate the final linear decompositions.

Mapping from the linear data decomposition to the linear computation decomposition is straightforward. Given the $m \times l$ array access matrix F^k_{xj} and $n \times m$ linear data decomposition matrix D_x, we find the $n \times l$ linear computation decomposition matrix C_j by setting $C_j = D_x F^k_{xj}$. Mapping from the linear computation decomposition to the linear data decomposition requires solving for the $n \times m$ linear data decomposition matrix D_x, given the $m \times l$ array access matrix F^k_{xj} and the $n \times l$ linear computation decomposition matrix C_j. To show that a solution always exists for D_x, we must consider the following three cases:

1. The system $D_x F^k_{xj} = C_j$ has a single solution for D_x.

2. The system $D_x F^k_{xj} = C_j$ is under-constrained and has infinitely many solutions for D_x. This occurs when the rank of the $m \times l$ matrix F^k_{xj} is less than the number of rows, rank $\left(F^k_{xj}\right) < m$. In this case a solution does exist, but there are free variables in the solution which are completely arbitrary. We solve for D_x, and then fill in the free variables afterwards. For example, the following loop nest results in an

under-constrained system:

```
real x[N,N]
/* Loop nest 1 */
for $i_1$ = 1 to N do
  x[$i_1$,1] = ...
```

The array access matrix for array x in loop nest 1 above is $F_{x1}^1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. Here rank $\left(F_{x1}^1\right) = 1$ and $m = 2$. If we let $C_1 = \begin{bmatrix} 1 \end{bmatrix}$, then

$$\begin{aligned} D_x F_{x1}^1 &= C_1 \\ D_x \begin{bmatrix} 1 \\ 0 \end{bmatrix} &= \begin{bmatrix} 1 \end{bmatrix} \\ D_x &= \begin{bmatrix} 1 & d_{12} \end{bmatrix} \end{aligned}$$

where d_{12} can be any arbitrary value. Informally, what happens in this case is that the second dimension of array x is not completely accessed by the given array access function, F_{x1}^1. The linear data decomposition of the second dimension of x can thus be anything – it still will not cause data-reorganization communication when the array access is executed.

3. The system $D_x F_{xj}^k = C_j$ is over-constrained and has no solution for D_x. This could potentially happen if the rank of the $m \times l$ matrix F_{xj}^k is less than the number of columns, rank $\left(F_{xj}^k\right) < l$. Specifically, the system $D_x F_{xj}^k = C_j$ is equivalent to $(F_{xj}^k)^T {D_x}^T = {C_j}^T$ and only has a solution when range $\left({C_j}^T\right) \subseteq$ range $\left((F_{xj}^k)^T\right)$[72]. Our algorithm ensures that a solution exists by generating C_j such that this condition is always met (see Equation 4.4 and Lemma 4.1.2 below). For example, consider the following code:

```
real x[N]
/* Loop nest 2 */
for $i_1$ = 1 to N do
  for $i_2$ = 1 to N do
    ...  = x[$i_1$]
```

The array access matrix for array x in loop nest 2 above is $F^1_{x2} = \begin{bmatrix} 1 & 0 \end{bmatrix}$. Here rank $\left(F^1_{x2}\right) = 1$ and $l = 2$. If we let $C_2 = \begin{bmatrix} 1 & 0 \end{bmatrix}$, then we have

$$\begin{aligned} D_x F^1_{x2} &= C_2 \\ D_x \begin{bmatrix} 1 & 0 \end{bmatrix} &= \begin{bmatrix} 1 & 0 \end{bmatrix} \\ D_x &= \begin{bmatrix} 1 \end{bmatrix} \end{aligned}$$

In this case, range $\left({C_2}^T\right)$ = range $\left((F^1_{x2})^T\right)$ = span $\{(1,0)\}$ and a solution exists. Informally, what happens is that all iterations of the i_2 loop access the same element of array x. Thus the i_2 loop must be assigned to the same processor, that is, $\mathcal{N}(C_2)$ must contain span $\{(0,1)\}$, or equivalently, range $\left({C_2}^T\right)$ must be in span $\{(1,0)\}$. Otherwise, the elements of array x would have to be allocated to multiple processors at the same time, and this is not possible without communication (in Section 4.1.3 we relax the model to allow replication communication in some such cases).

The mapping between linear computation and data decompositions are shown in algorithm form in Figure 4.3.

Calculating the Nullspaces

The first step in solving the system of equations is to calculate the nullspaces of the linear decomposition matrices. This corresponds to calculating the data and computation that must be assigned to the same processor so as to guarantee a solution to the synchronization and communication equations.

What data and computation must be assigned to the same processor? There are four *constraints* that we place on the nullspaces of the linear decomposition matrices. The

```
algorithm Linear_Comp_Decomp
            ($F^k_{xj}$ : matrix,
            $D_x$ : matrix)
   return
            ($C_j$ : matrix)

   $C_j$ = $D_x F^k_{xj}$;
   return $C_j$;
end algorithm;

algorithm Linear_Data_Decomp
            ($F^k_{xj}$ : matrix,
            $C_j$ : matrix)
   return
            ($D_x$ : matrix)

   $D_x$ = solve for $D_x$ in $D_x F^k_{xj} = C_j$;
      /* a solution is guaranteed to exist, see page 48 */
   return $D_x$;
end algorithm;
```

Figure 4.3: Algorithms for mapping between linear data decompositions and linear computation decompositions.

constraints on the nullspaces are represented as a set of basis vectors. In the rest of this section we describe each of the four constraints and present an algorithm to calculate them. The first constraint is derived from the synchronization equations and the remaining constraints are derived from the communication equations. We show that if a solution to the system of synchronization and communication equations exists, then the nullspaces of the resulting linear decomposition matrices must contain the space spanned by the constraints (Theorem 4.1.6).

(1) Synchronization Constraints. The synchronization equations (Equation 4.1) generate synchronization constraints on the nullspaces of the linear computation decomposition matrices. For each loop nest j, we have a synchronization equation $C_j(\vec{e_q}) = \vec{0}$, for all loops q that are not outermost parallel loops. Each synchronization equation leads directly to a constraint on the nullspace of C_j:

$$\vec{e_q} \in \mathcal{N}(C_j) \tag{4.3}$$

The following lemma shows that Equation 4.3 is a necessary condition for solving the synchronization equations.

Lemma 4.1.1 *For all loop nests j with loops q that are not outermost parallel loops, if a solution to the synchronization equation $C_j(\vec{e_q}) = \vec{0}$ (Equation 4.1) exists, then the synchronization constraint $\vec{e_q} \in \mathcal{N}(C_j)$ (Equation 4.3) is satisfied.*

Proof: This follows directly from Equation 4.1. □

(2) Computation Communication Constraints. The communication equations (Equation 4.2) generate computation communication constraints on the nullspaces of the linear computation decomposition matrices. If two iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ of loop nest j access the same element of array x, then $\vec{\imath}_1$ and $\vec{\imath}_2$ must be mapped onto the same processor. Consider an array access function F^k_{xj} in loop nest j. Iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ access the same element of array x when $F^k_{xj}(\vec{\imath}_1) = F^k_{xj}(\vec{\imath}_2)$, that is, $F^k_{xj}(\vec{\imath}_1 - \vec{\imath}_2) = \vec{0}$. Letting $\vec{t} = \vec{\imath}_1 - \vec{\imath}_2$, $F^k_{xj}(\vec{t}) = \vec{0}$ and $\vec{t} \in \mathcal{N}(F^k_{xj})$. Using Equation 4.2, if $\vec{t} \in \mathcal{N}(D_x F^k_{xj})$ then $\vec{t} \in \mathcal{N}(C_j)$. Since $\mathcal{N}(D_x F^k_{xj}) \supseteq \mathcal{N}(F^k_{xj})$, then $\vec{t} \in \mathcal{N}(F^k_{xj})$ implies that $\vec{t} \in \mathcal{N}(C_j)$. This leads to the following constraint on the nullspace of C_j:

$$\forall \vec{t} \in \mathcal{N}(F^k_{xj}), \vec{t} \in \mathcal{N}(C_j) \tag{4.4}$$

The following lemma shows that Equation 4.4 is a necessary condition for solving the communication equations.

Lemma 4.1.2 *For all arrays x, loop nests j and array accesses k, if a solution to the communication equation $D_x F^k_{xj} = C_j$ (Equation 4.2) exists, then the computation communication constraint $\forall \vec{t} \in \mathcal{N}(F^k_{xj}), \vec{t} \in \mathcal{N}(C_j)$ (Equation 4.4) is satisfied.*

Proof: To prove this lemma, we show that if a solution to $D_x F^k_{xj} = C_j$ exists, then $\mathcal{N}(F^k_{xj}) \subseteq \mathcal{N}(C_j)$. We know from linear algebra that a solution only exists when range $\left({C_j}^T\right) \subseteq$ range $\left((F^k_{xj})^T\right)$[72]. Also for any matrix A, $\mathcal{N}(A) \perp$ range $\left(A^T\right)$, that is, $\mathcal{N}(A)$ is the orthogonal subspace to range $\left(A^T\right)$. Thus $\mathcal{N}(F^k_{xj}) \subseteq \mathcal{N}(C_j)$. $\Box$

(3) Data Communication Constraints. The communication equations (Equation 4.2) also generate data communication constraints on the nullspaces of the data decomposition matrices. If two array elements $\vec{a}_1$ and $\vec{a}_2$ of array x are accessed by the same iteration of loop nest j, then $\vec{a}_1$ and $\vec{a}_2$ must be mapped onto the same processor. Furthermore, if elements of different arrays are accessed by the same iteration, then they must be assigned to the same processor. This can also cause constraints on the nullspaces of the data decomposition matrices if different loop nests place conflicting requirements on the arrays.

To calculate the data communication constraints, we first look at the interference graph $G_s = (V_c, V_d, E)$. For two vertices $v_x, v_y \in (V_d \cup V_c)$, if there is a cycle in the graph $(v_x, \ldots, v_y, \ldots, v_x)$, then there are multiple distinct paths from v_x to v_y. If there are multiple paths between two vertices, then it is possible for the loop nests to cause conflicting requirements on the decompositions of the arrays.

We now show how a cycle in the interference graph can lead to a constraint on the nullspaces of the linear data decomposition matrices. For each simple cycle in the interference graph, let v_x and v_y be two vertices in the cycle. Then there are two distinct paths from vertex v_x to vertex v_y, $V_{xy_1} = (v_x, \ldots, v_s, \ldots v_y)$ and $V_{xy_2} = (v_x, \ldots, v_t, \ldots v_y)$, where $v_s \neq v_t$. Assume without loss of generality that v_x and v_y are data vertices, i.e. $v_x, v_y \in V_d$. All cycles in a bipartite graph contain an even number of vertices, since the graph only contains edges between vertices in the set V_c and the set V_d. Thus any cycle will contain at least two data vertices and the assumption is valid. We define a *path access function*, H_{xy}, such that for a path V_{xy} we have $D_y = D_x H_{xy}$. Informally, the path access function gives the mapping between D_x and D_y when we consider only the array access functions along the path V_{xy}. We calculate H_{xy} by first setting D_x to the identity matrix and then finding all the linear decompositions along the path using algorithms `Linear_Comp_Decomp` and `Linear_Data_Decomp` until reaching D_y. Thus for paths V_{xy_1} and V_{xy_2} we have two path access functions H_{xy_1} and H_{xy_2} such that $D_y = D_x H_{xy_1}$ and $D_y = D_x H_{xy_2}$, respectively.

This gives the following equations:

$$\begin{aligned} D_y = D_x H_{xy_1} &= D_x H_{xy_2} \\ D_x(H_{xy_1} - H_{xy_2}) &= \vec{0} \end{aligned}$$

and leads to the following constraint on the nullspace of D_x:

$$\text{range}\Big((H_{xy_1} - H_{xy_2})\Big) \subseteq \mathcal{N}(D_x) \tag{4.5}$$

The data communication constraint is calculated for all arrays in each simple cycle in the interference graph. This includes the degenerate cycle formed by multiple array access functions for a single array in the same loop nest. If an array is involved in multiple cycles and multiple constraints are found, then the constraints are combined. Constraints are combined by summing the vector spaces that represent the different constraints.

For example, consider the simple cycle in the interference graph in Figure 4.2. Let v_x and v_y represent the data vertices for arrays x and y, respectively. Similarly, let v_1 and v_2 represent the computation vertices for loop nests 1 and 2, respectively. There are two paths from vertex v_x to vertex v_y, $V_{xy_1} = (v_x, v_1, v_y)$ and $V_{xy_2} = (v_x, v_2, v_y)$. This results in the following equations for path V_{xy_1}:

$$\begin{aligned} D_x H_{xy_1} &= D_y \\ D_x \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} &= D_y \end{aligned}$$

and the following equations for path V_{xy_2}:

$$\begin{aligned} D_x H_{xy_2} &= D_y \\ D_x \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} &= D_y \end{aligned}$$

Thus,

$$\begin{aligned}
D_x(H_{xy_1} - H_{xy_2}) &= \vec{0} \\
D_x\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\right) &= \vec{0} \\
\text{range}\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\right) &\subseteq \mathcal{N}(D_x) \\
\text{range}\left(\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\right) &\subseteq \mathcal{N}(D_x)
\end{aligned}$$

Simplifying the equation results in a constraint on the nullspace of D_x, span $\{(1, -1)\} \subseteq \mathcal{N}(D_x)$. Similar analysis leads to the same constraint on the nullspace of D_y, span $\{(1, -1)\} \subseteq \mathcal{N}(D_y)$. The following lemma shows that Equation 4.5 is a necessary condition for solving the communication equations.

Lemma 4.1.3 *For all arrays x, loop nests j and array accesses k, if a solution to the communication equation $D_x F^k_{xj} = C_j$ (Equation 4.2) exists, then the data communication constraints* range $\big((H_{xy_1} - H_{xy_2})\big) \subseteq \mathcal{N}(D_x)$*, where H_{xy_1} and H_{xy_2} are two distinct path access functions (Equation 4.5), is satisfied.*

Proof: We prove this lemma by contradiction. We assume that a solution to the communication equations exists and that the data communication constraint is *not* satisfied. This means that there exists linear decompositions D_x, D_y and path access functions H_{xy_1}, H_{xy_2} such that $D_x H_{xy_1} = D_y$ and $D_x H_{xy_2} = D_y$, but range $\big((H_{xy_1} - H_{xy_2})\big) \not\subseteq \mathcal{N}(D_x)$. Thus, $D_x H_{xy_1} \neq D_x H_{xy_2}$ and $D_y \neq D_y$ which is a contradiction. Thus the original assumption that the data communication constraint is not satisfied must be false, thereby proving the lemma. □

(4) Propagation Constraints. The previous three constraints determine which data elements within the *same* array are assigned to the same processor, and which iterations within the *same* loop nest are assigned to the same processor. The loops in each loop nest that are executed on the same processor force elements of the arrays referenced in that loop nest to be allocated local to the same processor. Similarly, the local array elements force iterations of the loop nests that access those elements to be executed on the same processor. This constraint specifies how the nullspaces are propagated *between* loop nests and arrays.

If two iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ in loop nest j are mapped to the same processor, then the data of array x they access must also be mapped to the same processor. We have $C_j(\vec{\imath}_1) = C_j(\vec{\imath}_2)$ and $C_j(\vec{\imath}_1 - \vec{\imath}_2) = \vec{0}$, thus for $\vec{t} = \vec{\imath}_1 - \vec{\imath}_2$, $\vec{t} \in \mathcal{N}(C_j)$. Using Equation 4.2, $D_x F^k_{xj} \vec{t} = C_j \vec{t} = \vec{0}$ and thus $F^k_{xj}\vec{t} \in \mathcal{N}(D_x)$. Formally,

$$\mathcal{N}(D_x) \supseteq \text{span}\left\{\vec{s} \mid \vec{s} = F^k_{xj}\vec{t}, \vec{t} \in \mathcal{N}(C_j)\right\} \tag{4.6}$$

Similarly, two iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ in loop nest j must be mapped to the same processor if the data of array x they access are mapped to the same processor. Again, let $\vec{t} = \vec{\imath}_1 - \vec{\imath}_2$ and using Equation 4.2, $D_x F^k_{xj}\vec{t} = C_j\vec{t} = \vec{0}$. Thus $\vec{t} \in \mathcal{N}(C_j)$ when $F^k_{xj}\vec{t} \in \mathcal{N}(D_x)$. Letting $S^k_{xj} = \text{range}\left(F^k_{xj}\right)$,

$$\mathcal{N}(C_j) \supseteq \text{span}\left\{\vec{t} \mid F^k_{xj}\vec{t} \in (\mathcal{N}(D_x) \cap S^k_{xj})\right\} \tag{4.7}$$

In the following lemma we show that it is necessary to propagate the nullspaces between the data and computation to solve for the communication equations 4.2.

Lemma 4.1.4 *For all arrays x, loop nests j and array accesses k, if a solution to the communication equation $D_x F^k_{xj} = C_j$ (Equation 4.2) exists, then the data and computation propagation constraints (Equation 4.6, Equation 4.7) are satisfied.*

Proof: This follows directly from Equations 4.6 and 4.7. □

The complete algorithm for finding the nullspaces of basic linear decomposition matrices is shown in Figures 4.4 and 4.5. Figure 4.4 contains `Propagate_Nullspaces`, the algorithm used for propagating the nullspaces. The algorithm is iterative – it calculates the effects of the loop nests on the arrays using Equation 4.6, and the effects of the arrays on the loop nests using Equation 4.7. This continues until a fixed-point for the nullspaces is found. Figure 4.5 shows `Basic_Nullspaces`, the main algorithm for calculating the nullspaces on a given interference graph.

Since algorithm `Propagate_Nullspaces` shown in Figure 4.4 is an iterative algorithm that completes only when a fixed-point is reached, we show in the following theorem that the algorithm does terminate.

```
**algorithm** Update_Arrays
        ($G_s$ : interference_graph, /* $G_s = (V_c, V_d, E)$ */
        $v_j$ : computation_vertex, /* $v_j \in V_c$ */
        $\Delta$ : **set of** vector_space)

  **foreach** array $x$ and array access $F^k_{xj}$ such that $e^k_{xj} \in E$ **do**
    $\mathcal{N}(D_x) = \mathcal{N}(D_x) + \text{span}\left\{\vec{s} \mid \vec{s} = F^k_{xj}\vec{t}, \vec{t} \in \mathcal{N}(C_j)\right\}$; /* *Equation 4.6* */
    $\Delta = \Delta + \mathcal{N}(D_x)$;
  **end foreach**;
**end algorithm**;

**algorithm** Update_Loops
        ($G_s$ : interference_graph, /* $G_s = (V_c, V_d, E)$ */
        $v_x$ : data_vertex, /* $v_j \in V_c$ */
        $\Gamma$ : **set of** vector_space)

  **foreach** loop nest $j$ and array access $F^k_{xj}$ such that $e^k_{xj} \in E$ **do**
    $\mathcal{N}(C_j) = \mathcal{N}(C_j) + \text{span}\left\{\vec{t} \mid (F^k_{xj}\vec{t} \in (\mathcal{N}(D_x) \cap S^k_{xj}))\right\}$; /* *Equation 4.7* */
    $\Gamma = \Gamma + \mathcal{N}(C_j)$;
  **end foreach**;
**end algorithm**;

**algorithm** Propagate_Nullspaces
        ($G_s$ : interference_graph, /* $G_s = (V_c, V_d, E)$ */
        $\Gamma$ : **set of** vector_space,
        $\Delta$ : **set of** vector_space)

  **while** changes **do**
    **if** changed($v_x \in V_d$) **then** Update_Loops($G_s$, $v_x$, $\Gamma$);
    **if** changed($v_j \in V_c$) **then** Update_Arrays($G_s$, $v_j$, $\Delta$);
  **end while**;
**end algorithm**;
```

Figure 4.4: Algorithm for propagating the nullspaces of linear decomposition matrices between loop nests and arrays.

```
**algorithm** Init_Nullspaces
            ($G_s$ : interference_graph, /* $G_s = (V_c, V_d, E)$ */
            *ConstrC_List* : **list of** constraint,
            *ConstrD_List* : **list of** constraint)
   **return**
            ($\Gamma$ : **set of** vector_space,
            $\Delta$ : **set of** vector_space)

   **foreach** $v_j \in V_c$ **do** $\mathcal{N}(C_j)$ = *ConstrC_List*;
   **foreach** $v_x \in V_d$ **do** $\mathcal{N}(D_x)$ = *ConstrD_List*;
   $\Gamma = \bigcup_{\forall v_j \in V_c} (\mathcal{N}(C_j))$;
   $\Delta = \bigcup_{\forall v_x \in V_d} (\mathcal{N}(D_x))$;

   **return** ($\Gamma$,$\Delta$);
**end algorithm**;

**algorithm** Basic_Nullspaces
            ($G_s$ : interference_graph) /* $G_s = (V_c, V_d, E)$ */
   **return**
            ($\Gamma$ : **set of** vector_space,
            $\Delta$ : **set of** vector_space)

   *ConstrC_List* : **list of** constraint;
   *ConstrD_List* : **list of** constraint;

   *ConstrC_List = /* Equations 4.3, 4.4 */*;
   *ConstrD_List = /* Equation 4.5 */*;
   ($\Gamma$, $\Delta$) = Init_Nullspaces($G_s$, *ConstrC_List, ConstrD_List*);

   Propagate_Nullspaces($G_s$, $\Gamma$, $\Delta$);
   **return** ($\Gamma$,$\Delta$);
**end algorithm**;
```

Figure 4.5: Algorithm for calculating the nullspaces of the basic linear decomposition matrices.

Theorem 4.1.5 *Algorithm* `Propagate_Nullspaces` *is guaranteed to terminate.*

Proof: For all $v_x \in V_d$ and for all $v_j \in V_c$, the vector spaces $\mathcal{N}(D_x)$ and $\mathcal{N}(C_j)$ increase in size monotonically as the algorithm progresses. In the worst case, the nullspaces will span the entire space and the algorithm will terminate. □

The following theorem shows that the nullspaces found by `Basic_Nullspaces` are necessary for finding the nullspaces of the matrices that satisfy the system of equations formed by the synchronization and communication equations (sufficiency is proved later, in Section 4.1.2.3 when we calculate the actual linear decomposition matrices). In other words, if a solution to the system of equations exists, then the nullspaces of the linear decomposition matrices from that solution must be at least as large as the nullspaces calculated from our constraints. As a result, the algorithm finds the smallest nullspaces that guarantee a solution to the equations. Since the nullspace represents the data and computation that are allocated to the same processor, finding the smallest nullspaces means that the algorithm is finding the *maximum parallelism* such that there is no synchronization within each loop nest and only nearest-neighbor communication.

Theorem 4.1.6 *For all arrays x, loop nests j and array accesses k, if a solution exists to the system of equations formed by the synchronization equations $C_j(\vec{e_q}) = \vec{0}$ (Equation 4.1) and the communication equations $D_x F_{xj}^k = C_j$ (Equation 4.2), then the nullspaces of the linear decomposition matrices contain the space spanned by the nullspaces found by algorithm* `Basic_Nullspaces`*, and the nullspaces are the smallest possible subspaces.*

Proof: The nullspaces found by algorithm `Basic_Nullspaces` are the nullspaces given by constraints (1)–(4) using Equations 4.3, 4.4, 4.5, 4.6 and 4.7. The necessity of these four constraints was shown in lemmata 4.1.1, 4.1.2, 4.1.3 and 4.1.4. The nullspaces found by the algorithm are the smallest possible subspaces because algorithm only ever adds vectors to the nullspaces in order to ensure that one of the constraints is satisfied. □

For an example of how `Basic_Nullspaces` works, consider the interference graph from Figure 4.2 and the corresponding code from Figure 4.1(b). The nullspaces of the linear computation decomposition matrices are initialized to $\mathcal{N}(C_1) = \mathcal{N}(C_2) = \emptyset$, that is, there are no synchronization constraints nor computation communication constraints.

The nullspaces of the linear data decomposition matrices are initialized to $\mathcal{N}(D_x) = \mathcal{N}(D_y) = \text{span}\,\{(1,-1)\}$ due to a data communication constraint, and $\mathcal{N}(D_z) = \emptyset$. First, the routine `Update_Loops` is called with data vertex v_x. Equation 4.7 is applied to computation vertices v_1 and v_2 resulting in $\mathcal{N}(C_1) = \mathcal{N}(C_2) = \text{span}\,\{(1,-1)\}$. Next routine `Update_Arrays` is called with computation vertex v_1. Equation 4.6 is applied to data vertices v_x, v_y and v_z to give $\mathcal{N}(D_x) = \mathcal{N}(D_y) = \mathcal{N}(D_z) = \text{span}\,\{(1,-1)\}$. Routine `Update_Arrays` is also called with computation vertex v_2, but all the arrays have already been updated so the call has no effect. Finally, `Update_Loops` is called for each of the data vertices; the calls have no effect and the algorithm terminates.

Calculating the Decomposition Matrices

After calculating the nullspaces of the linear decomposition matrices for each array and for each loop nest, next we calculate the actual linear decomposition matrices. The first step is to determine the number of virtual processor dimensions. The number of virtual processor dimensions n is

$$n = \max_{v_x \in V_d}(\text{dim}(S_x) - \text{dim}(\mathcal{N}(D_x))) \tag{4.8}$$

where

$$S_x = \sum_{\forall e_{xj}^k \in E} \text{range}\left(F_{xj}^k\right)$$

S_x represents the total space of array x that is accessed, typically the entire array. This equation will yield a value of n such that all the parallelism found in the `Basic_Nullspaces` algorithm from Figure 4.5 is exploited. For the example from Figure 4.2, $n = 1$.

When calculating the linear decomposition matrices, we take advantage of the fact that there is a one-to-one correspondence between edges in the interference graph and the communication equations. This means that each connected component of the interference graph corresponds to a set of arrays and loop nests that have inter-related linear decompositions. The linear decompositions of the vertices in different connected components correspond to independent systems of equations. We thus solve for the linear decompositions in each connected component of the interference graph separately. Also, since the decompositions in each connected component are all relative to one another, we can choose *one* arbitrary linear decomposition matrix and derive the rest of the linear decomposition matrices in the connected component. The algorithm chooses an $n \times m$ linear data decomposition matrix D_y for an array y of dimension m such that $\mathcal{N}(D_y)$ is the nullspace that was calculated by the `Basic_Nullspaces` algorithm.

Starting from the vertex representing array y in the interference graph, we traverse the remaining vertices in the connected component in breadth-first order. Given the linear data decomposition D_y, we find the linear computation decomposition C_j for a loop nest j that references array y using algorithm `Linear_Comp_Decomp`. Similarly, given the computation decomposition C_j, we find the data decomposition matrix D_x for another array x accessed in loop nest j using algorithm `Linear_Data_Decomp`. When there are multiple array access functions for an array in a loop nest we use the one with the maximum rank. The remaining linear decomposition matrices in the connected component are calculated in a similar fashion. After all the linear decompositions are calculated, any unspecified entries in the matrices are filled in (see Section 4.1.2.3). Note that when calculating the linear decomposition matrices, non-integer values in the matrices can result. Since the virtual processor numbers must be integers, the linear decompositions must map the data and computation into integer values. We eliminate any non-integer values in the matrices by multiplying through by the least common multiple of the denominators of the fractions. The algorithm for finding linear decomposition matrices, `Calc_Matrices`, is shown in Figure 4.6.

The following theorem shows that the `Basic_Nullspaces` algorithm is sufficient for finding the nullspaces of the matrices that satisfy the system of equations. The theorem also shows that algorithm `Calc_Matrices` constructs the matrices that have the nullspaces found by `Basic_Nullspaces`.

Theorem 4.1.7 *Algorithm* `Calc_Matrices` *finds a solution to the system of equations formed by the synchronization equations (Equation 4.1) and the communication equations (Equation 4.2) such that the linear decomposition matrices have the nullspaces found by algorithm* `Basic_Nullspaces`.

Proof: We prove this theorem by induction. The base case is the array y for which we

```
algorithm Calc_Matrices
            ($G_s$ : interference_graph,   /* $G_s = (V_c, V_d, E)$ */
            $\Gamma$ : set of vector_space,
            $\Delta$ : set of vector_space)
              /* Nullspaces calculated by Basic_Nullspaces */

   $v_y$ : data_vertex;
   $n$ : integer;
   factor : integer;

   foreach connected component $G_s' \in G_s$ do
      $(n, v_y) = \max_{y \in V_d'}(\text{dim}(S_y) - \text{dim}(\mathcal{N}(D_y)))$;   /* Equation 4.8 */
      $D_y = n \times m$ matrix with nullspace $\mathcal{N}(D_y)$;   /* $m = \text{dim}(y)$ */

      /* Find linear decomposition matrices for all vertices */
      foreach $v_x \in V_d'$, $v_j \in V_c'$ in breadth-first order starting from $v_y$ do
         for $v_j \in V_c'$, $k$ such that $F_{xj}^k$ is max rank, $C_j$ = Linear_Comp_Decomp($F_{xj}^k$, $D_x$);
         for $v_x \in V_d'$, $k$ such that $F_{xj}^k$ is max rank, $D_x$ = Linear_Data_Decomp($F_{xj}^k$, $C_j$);
      end foreach;

      /* Ensure all entries are integral */
      factor = least common multiple of denominators;
      foreach $v_x \in V_d'$, $v_j \in V_c'$ do
         $D_x$ = factor $\cdot$ $D_x$;
         $C_j$ = factor $\cdot$ $C_j$;
      end foreach;
   end foreach;
end algorithm;
```

Figure 4.6: Algorithm for calculating the linear decomposition matrices.

chose an arbitrary decomposition matrix that has the specified nullspace. For the inductive step, we show that as each linear decomposition matrix is calculated, it has the correct nullspace and the synchronization and communication equations hold. There are two cases, calculating linear data decompositions and calculating linear computation decompositions.

First we show that given a linear data decomposition D_x for array x, we calculate a linear computation decomposition C_j for loop nest j that has the correct nullspace and that the synchronization and communication equations are satisfied. The communication equations are satisfied by construction because these equations are used to calculate C_j given D_x in algorithm `Linear_Comp_Decomp` called from `Calc_Matrices`. The synchronization equations involving loop nest j are satisfied by the synchronization constraint (Equation 4.3), and the propagation constraint (Equation 4.7) ensures that $\mathcal{N}(C_j)$ is correct.

Next we show that given a linear computation decomposition C_j for a loop nest j, we calculate a linear data decomposition D_x for array x that has the correct nullspace and that the synchronization and communication equations are satisfied. Again in this case, the communication equations are satisfied by construction because these equations are used to calculate D_x given C_j in algorithm `Linear_Data_Decomp`. Note that we know a solution to the communication equations exists because of the computation communication constraint (Equation 4.4) and the data communication constraint (Equation 4.5). We do not need to consider the synchronization equations as they are functions of only the linear computation decompositions, and have no effect on the linear data decompositions. Finally, the propagation constraint (Equation 4.6) ensures that $\mathcal{N}(D_x)$ is correct. $\square$

The following theorem shows that algorithm `Calc_Matrices` is correct, and that the solution has the property that it finds basic linear decompositions with the maximum degree of parallelism.

Theorem 4.1.8 *Algorithm* `Calc_Matrices` *finds basic linear decompositions that have the maximum degree of parallelism.*

Proof: Theorem 4.1.7 showed that the linear decompositions found by the algorithm satisfy the synchronization and communication equations. The linear decompositions it finds are basic linear decompositions as the solution has no data-reorganization communication and there is no synchronization within each loop nest. Theorem 4.1.6 shows that the nullspaces of the linear decomposition matrices are as small as possible. Thus the algorithm is finding the basic linear decompositions with the maximum degree of parallelism. □

For an example of the `Calc_Matrices` algorithm, consider the on-going example from Figure 4.2. The `Basic_Nullspaces` algorithm found that the nullspaces were $\mathcal{N}(D_x) = \mathcal{N}(D_y) = \mathcal{N}(D_z) = \text{span}\,\{(1,-1)\}$, and $\mathcal{N}(C_1) = \mathcal{N}(C_2) = \text{span}\,\{(1,-1)\}$. The algorithm first sets $D_x = \left[\begin{array}{cc} 1 & 1 \end{array}\right]$. It then traverses the vertices in the order (1, 2, y, z) to calculate the remaining linear decomposition matrices. The resulting linear decompositions are:

$$C_1 = C_2 = \left[\begin{array}{cc} 1 & 1 \end{array}\right], \ D_x = D_y = D_z = \left[\begin{array}{cc} 1 & 1 \end{array}\right]$$

This is the same result we showed when we first introduced the example in Section 4.1.2.2.

4.1.3 Linear Decompositions with Replication

In this section we describe how to extend the algorithm for finding basic linear decompositions to allow a restricted form of communication, *replication*. Replication of read-only data is a commonly-used technique for improving the performance of parallel machines. The algorithm we present in this section finds the amount of read-only data replication needed to maintain the degree of parallelism inherent in the read-write data without introducing additional communication.

We consider two types of replication: *constant replication* and *dimension replication*. Constant replication creates a fixed number of copies of an entire array. Different processors may access the same elements of the array at the same time. For example, consider the code for loop nest 1 in Figure 4.7. Assume that we are given the linear computation decomposition $C_1 = \left[\begin{array}{c} 2 \end{array}\right]$, i.e. iterations of the i_1 loop are distributed across the processors with a stride of 2. Then each processor needs to access both elements `y[`i_1`]` and `y[2*`i_1`]` of array y. We can use constant replication to create two copies of y, each with a different data decomposition. Figure 4.8(a) shows the two linear data decompositions for array y in this example onto a one-dimensional virtual processor space.

Dimension replication duplicates data across all the processors along a dimension of the virtual processor space. In loop nest 2 from Figure 4.7, if we assume that the linear

```
real x[2*N], y[2*N], z[N,N]
/* Loop Nest 1 */
for $i_1$ = 1 to N do              /* doall */
    x[2*$i_1$] = y[$i_1$] + y[2*$i_1$]

/* Loop Nest 2 */
for $i_1$ = 1 to N do              /* doall */
  for $i_2$ = 2 to N do
    z[$i_1$,$i_2$] = z[$i_1$,$i_2$-1] + y[$i_2$]
```

Figure 4.7: Code used to illustrate linear decompositions with replication.

computation decomposition is $C_2 = \begin{bmatrix} 1 & 0 \end{bmatrix}$ then all processors must access the entire array y. We can apply dimension replication to y and replicate along all processors in the first (and only) dimension of the processor space. Figure 4.8(b) shows the linear data decompositions for the array in this case. Note that with dimension replication, it is not necessary to replicate the entire array – we can also apply dimension replication to subsections of an array.

The increase in space requirements to accommodate constant replication is a linear function of the array size, whereas the extra space needed for dimension replication is a function of both the number of processors and the array size. With constant replication, each copy of the array has a different data decomposition. With dimension replication, there is a single data decomposition for each array that is repeated for all processors along the replicated processor dimension.

4.1.3.1 Formulating Equations with Replication

All read-only arrays are candidates for replication. We will also replicate any arrays with read accesses that are of lower dimension than the maximum array dimension found in the program. A pre-pass examines the array accesses and marks all such arrays as *replicatable*. To find the linear decompositions with replication, we first calculate the linear decompositions ignoring all read accesses to the replicatable data. The communication

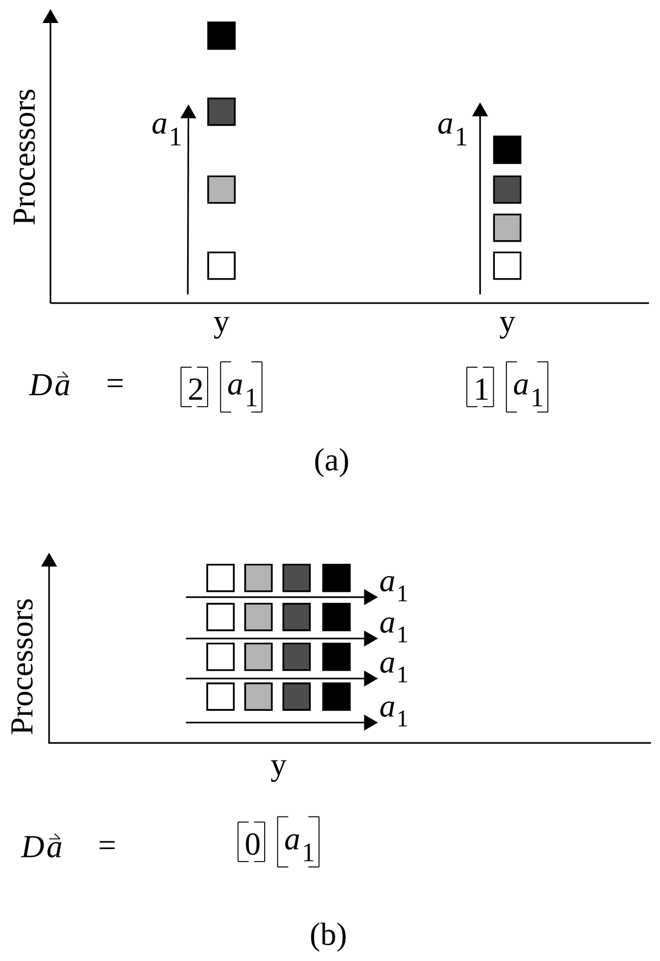


Figure 4.8: Example showing linear data decompositions onto a one-dimensional virtual processor space with different types of replication: (a) constant replication and (b) dimension replication. The elements are shaded to identify their positions.

equations and corresponding constraints that would have resulted from these read accesses are eliminated. Removing these constraints potentially allows linear decompositions that have a greater degree of parallelism. We then use the linear computation decompositions for each loop nest to derive the linear data decompositions for the replicatable arrays.

The key issue in calculating linear data decompositions for replicatable arrays from a given linear computation decomposition is how to model the relationship between the computation and data with replication. In the remainder of this section, we describe how to formulate the equations that allow constant and dimension replication, but still do not allow arbitrary data-reorganization communication.

Without replication, the system of equations we generated only allowed a single linear data decomposition D_x for each array x. To model constant replication, we allow multiple data decompositions for each replicatable array x, one for each read access F_{xj}^k of x in loop nest j. We denote the linear data decomposition for array x corresponding to the array access F_{xj}^k by D_{xj}^k.

To model dimension replication, we use a subspace of the full virtual processor space. The linear data decompositions for the replicatable arrays are calculated to map onto the subspace. Then all processors that are in the full processor space, but not in the subspace, are allocated copies of the data that are allocated to the corresponding processor in the subspace. We call this subspace of the virtual processor space the *replication subspace*. For the example from Figure 4.8(b), Figure 4.9 shows the mapping of array y in both the full processor space and the replication subspace. The full processor space is one-dimensional, and the replication subspace has zero dimensions, i.e. the entire array is mapped onto the first processor. All processors are allocated copies of array y since the entire array is allocated to a single processor in the replication subspace.

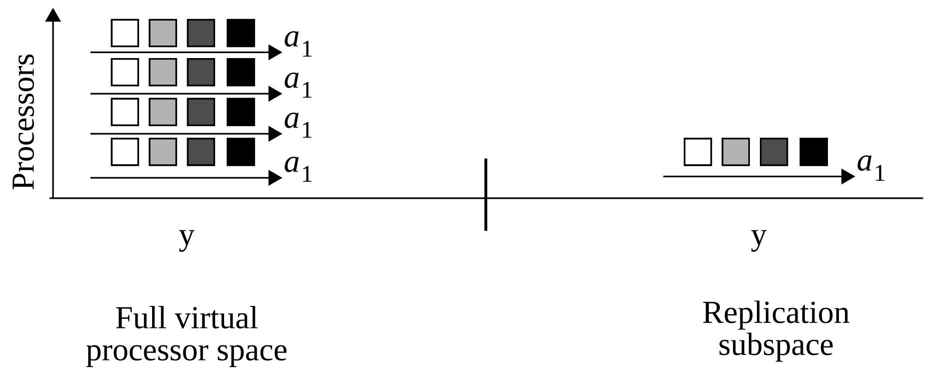


Figure 4.9: An example of dimension replication that shows the correspondence between the full virtual processor space and the replication subspace. The elements are shaded to identify their positions.

Let C_j be the computation decomposition matrix for a loop nest j that accesses a replicatable array x. C_j maps iterations onto the full processor space. To relate the full processor space to the replication subspace, for each array access k we use an $n \times n$ projection matrix R^k_{xj}. A projection matrix is a symmetric matrix that also equals its square, e.g. $R^k_{xj} = (R^k_{xj})^T$ and $R^k_{xj} = (R^k_{xj})^2$. We call R^k_{xj} the *replication matrix* for the given array access.

To generate the linear data decompositions with both dimension and constant replication, we use a modified version of the communication equation $D_x F^k_{xj} = C_j$. Given the linear computation decomposition C_j for all loop nests j and the read accesses F^k_{xj} to all replicatable arrays x, we generate the following *replication equations*:

$$D^k_{xj} F^k_{xj} \;=\; R^k_{xj} C_j \tag{4.9}$$

We solve these equations to find a linear data decomposition D^k_{xj} and a replication matrix R^k_{xj}.

The replication matrix R^k_{xj} maps the processor space, range(C_j), onto the subspace range(D_x). When the replication equations are satisfied, the data is local to the processor accessing that data in the replication subspace. Aside from any constant replication, there is no data-reorganization communication in the replication subspace. Communication due to dimension replication occurs along the dimensions that are projected away when mapping from the full space onto the replication subspace. The nullspace of R^k_{xj}, $\mathcal{N}(R^k_{xj})$, corresponds to those dimensions along which there is dimension replication. Thus when R^k_{xj} is the identity matrix, $\mathcal{N}(R^k_{xj}) = \emptyset$ and there is no dimension replication.

Definition 4.1.1 *Given an $n \times n$ replication matrix R^k_{xj} for an array access F^k_{xj}, the* **degree of replication** *is the number of processor dimensions along which the data is copied. Mathematically, the degree of replication is given by* $\dim(\mathcal{N}(R^k_{xj}))$*, or equivalently,* $n - \operatorname{rank}\left(R^k_{xj}\right)$.

The replication equations represent the necessary conditions for linear data decompositions that allow constant and dimension replication, but still do not allow arbitrary data-reorganization communication. There can be many possible solutions to these equations. For example, a trivial solution is to always map all the data onto a single processor in the replication subspace, and then replicate the entire array across all the processors. The objective, however, is to find a solution with the minimum degree of dimension replication. This corresponds to finding replication matrices R^k_{xj} such that rank $\left(R^k_{xj}\right)$ is as large as possible, or equivalently, $\mathcal{N}(R^k_{xj})$ is as small as possible.

4.1.3.2 Examples of Linear Decompositions with Replication

Consider the two loop nests from Figure 4.8. Array y is marked replicatable since it is a read-only array. We first generate the synchronization and communication equations for the two loop nests ignoring all accesses to array y:

$$\begin{array}{lll} C_2 & \begin{bmatrix} 0 \\ 1 \end{bmatrix} & = \vec{0} \\ D_x & \begin{bmatrix} 2 \end{bmatrix} & = C_1 \\ D_z & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & = C_2 \end{array}$$

A solution with the largest degree of parallelism for these equations is

$$C_1 = \begin{bmatrix} 2 \end{bmatrix}, \; C_2 = \begin{bmatrix} 1 & 0 \end{bmatrix}, \; D_x = \begin{bmatrix} 1 \end{bmatrix}, \; D_z = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

Now we generate a system of replication equations (Equation 4.9) to find the linear data decompositions for array y:

$$\begin{array}{lllll} D^1_{y1} & \begin{bmatrix} 1 \end{bmatrix} & = & R^1_{y1} & C_1 \\ D^2_{y1} & \begin{bmatrix} 2 \end{bmatrix} & = & R^2_{y1} & C_1 \\ D^1_{y2} & \begin{bmatrix} 0 & 1 \end{bmatrix} & = & R^1_{y2} & C_2 \end{array}$$

plugging in C_1 and C_2 from above,

$$\begin{array}{lllll} D^1_{y1} & \begin{bmatrix} 1 \end{bmatrix} & = & R^1_{y1} & \begin{bmatrix} 2 \end{bmatrix} \\ D^2_{y1} & \begin{bmatrix} 2 \end{bmatrix} & = & R^2_{y1} & \begin{bmatrix} 2 \end{bmatrix} \\ D^1_{y2} & \begin{bmatrix} 0 & 1 \end{bmatrix} & = & R^1_{y2} & \begin{bmatrix} 1 & 0 \end{bmatrix} \end{array}$$

A solution with the smallest degree of replication is:

$$\begin{array}{ll} D^1_{y1} = \left[\begin{array}{c} 2 \end{array}\right], & R^1_{y1} = \left[\begin{array}{c} 1 \end{array}\right] \\ D^2_{y1} = \left[\begin{array}{c} 1 \end{array}\right], & R^2_{y1} = \left[\begin{array}{c} 1 \end{array}\right] \\ D^1_{y2} = \left[\begin{array}{c} 0 \end{array}\right], & R^1_{y2} = \left[\begin{array}{c} 0 \end{array}\right] \end{array}$$

4.1.3.3 Solving Equations with Replication

This section describes how we solve the replication equations. For each replication equation $D^k_{xj} F^k_{xj} = R^k_{xj} C_j$, C_j and F^k_{xj} are given and we solve for both D^k_{xj} and R^k_{xj}. The objective is to find a solution that has the minimum degree of replication. We find replication matrices R^k_{xj} such that rank $\left(R^k_{xj}\right)$ is as large as possible, or equivalently, $\mathcal{N}(R^k_{xj})$ is as small as possible. Note that each replication equation is an independent equation; there are distinct free variables D^k_{xj} and R^k_{xj} for each equation. This is in contrast to the interdependent system of equations we generated for finding basic linear decompositions (Section 4.1.2.3); in that case we were solving for a single linear computation decomposition for each loop nest and a single linear data decomposition for each array.

We solve the replication equation $D^k_{xj} F^k_{xj} = R^k_{xj} C_j$ by first finding the nullspace of the replication matrix, $\mathcal{N}(R^k_{xj})$, that is as small as possible. We then calculate the actual matrix R^k_{xj}, and finally solve for the linear data decomposition D^k_{xj}. Under what conditions is dimension replication necessary? If two iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ of loop nest j access the same element of a replicatable array x, then that element must be replicated across the processors executing $\vec{\imath}_1$ and $\vec{\imath}_2$. For an array access F^k_{xj}, iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ access the same element of array x when $F^k_{xj}(\vec{\imath}_1) = F^k_{xj}(\vec{\imath}_2)$, that is, $F^k_{xj}(\vec{\imath}_1 - \vec{\imath}_2) = \vec{0}$. Letting $\vec{t} = \vec{\imath}_1 - \vec{\imath}_2$, $F^k_{xj}(\vec{t}) = \vec{0}$ and $\vec{t} \in \mathcal{N}(F^k_{xj})$. Using Equation 4.9, $D_x F^k_{xj} \vec{t} = R^k_{xj} C_j \vec{t} = \vec{0}$. Then $\vec{t} \in \mathcal{N}(R^k_{xj} C_j)$, and $\mathcal{N}(R_{xj} C_j) \supseteq \mathcal{N}(C_j)$. Formally,

$$\mathcal{N}(R^k_{xj}) = \text{span}\left\{ C_j \vec{t} \mid \vec{t} \in \mathcal{N}(F^k_{xj}), \vec{t} \notin \mathcal{N}(C_j) \right\} \qquad (4.10)$$

This replicates only along dimensions that use the same data and thus require dimension replication, i.e. the processors executing iterations along dimensions $\vec{t} \in \mathcal{N}(F^k_{xj})$.

After finding $\mathcal{N}(R^k_{xj})$, we set R^k_{xj} to an arbitrary projection matrix that has the given nullspace $\mathcal{N}(R^k_{xj})$. We then solve for D^k_{xj} in the replication equation $R^k_{xj}C_j = D^k_{xj}F^k_{xj}$, using algorithm `Linear_Data_Decomp` from Figure 4.3. Figure 4.10 shows the details of the `Calc_Replication` algorithm for calculating the replication matrices and linear data decompositions.

The issue remains of whether a solution always exists for D^k_{xj} once we have found R^k_{xj}. Without replication, we could always apply algorithm `Linear_Data_Decomp` to find the linear data decomposition given an array access function and a linear computation decomposition. The computation communication constraint (Equation 4.4) ensured that a solution to the communication equation $D^k_{xj}F^k_{xj} = C_j$ always existed by making $\mathcal{N}(F^k_{xj}) \subseteq \mathcal{N}(C_j)$. With replication, a solution for D^k_{xj} in $R^k_{xj}C_j = D^k_{xj}F^k_{xj}$ exists when range $\left((R^k_{xj}C_j)^T\right) \subseteq$ range $\left((F^k_{xj})^T\right)$[72]. This is equivalent to $\mathcal{N}(R^k_{xj}C_j) \supseteq \mathcal{N}(F^k_{xj})$, which is true by construction of $\mathcal{N}(R^k_{xj})$ in Equation 4.10. With dimension replication we are relaxing the previous computation communication constraint and are no longer requiring that $\mathcal{N}(F^k_{xj}) \subseteq \mathcal{N}(C_j)$; however, since $\mathcal{N}(R^k_{xj}C_j) \supseteq \mathcal{N}(F^k_{xj})$ we are guaranteed that a solution for D^k_{xj} exists in the replication subspace.

The algorithm `Calc_Replication` allows for as much replication as is necessary to maintain the same degree of parallelism in the non-replicatable data. It does not consider trading off parallelism to limit the amount of replication needed. As a result, the amount of replication called for could be much greater than is practical on the target machine. We can limit the degree of replication by projecting the virtual processor space onto a smaller physical processor space (see Section 4.2).

4.1.3.4 Broadcast and Multicast Communication

Broadcast communication occurs whenever one processor sends data to all other processors. A related concept is *multicast communication*, where one processor communicates data to a subset of the processors. Many parallel machines offer primitives that support efficient broadcast and multicast communication. Dimension replication can be viewed as a kind of multicast communication. With dimension replication, the data assigned to one processor in the replication subspace is copied to all processors along the replicated dimensions in the full processor space. In other words, each processor in the replication subspace multicasts its data to the subset of processors along the replicated dimensions.

```
algorithm Replicated_Data_Decomp
            ($F^k_{xj}$ : matrix,
             $C_j$ : matrix)
   return
            ($D^k_{xj}$ : matrix)

   $R^k_{xj}$ : matrix;

   $\mathcal{N}(R^k_{xj}) = \text{span}\left\{C_j\vec{t} \mid \vec{t} \in \mathcal{N}(F^k_{xj}), \vec{t} \notin \mathcal{N}(C_j)\right\};$  /* Equation 4.10 */
   $R^k_{xj}$ = $n \times n$ projection matrix with nullspace $\mathcal{N}(R^k_{xj})$;
        /* $n$ is the dimensionality the virtual processor space, and equals rows($C_j$) */

   $D^k_{xj}$ = Linear_Data_Decomp($F^k_{xj}$, $R^k_{xj}C_j$);
   return $D^k_{xj}$;
end algorithm;

algorithm Calc_Replication
            ($G_s$ : interference_graph)   /* $G_s$ = $(V_c, V_d, E)$ */

   ${V_d}'$ : set of data_vertex;
   $C_j$ : matrix;

   ${V_d}'$ = replicatable arrays in $V_d$;

   foreach $v_x \in {V_d}'$ and read array access $F^k_{xj}$ such that $e^k_{xj} \in E$ do
      $C_j$ = linear decomposition for computation vertex $v_j$;
      $D^k_{xj}$ = Replicated_Data_Decomp($F^k_{xj}$, $C_j$);
   end foreach;
end algorithm;
```

Figure 4.10: Algorithms for calculating linear data decompositions with replication.

We can model multicast communication along full dimensions in the processor space as dimension replication. This is a restricted form of multicast, since we are not copying the data to arbitrary subsets of the processors. The basic idea is that we allow multicast communication by applying dimension replication to only the read accesses of a non-replicatable array. The key issue is then how to modify the system of equations to only allow multicast communication. For replicatable data, we simply eliminated all the constraints that derived from the read accesses to that data. However, we can not simply ignore all read accesses to non-replicatable data; this would result in a large amount of general communication. We note that in our model of dimension replication, for an array access F_{xj}^k of a replicatable array x in loop nest j, there is some degree of dimension replication whenever $\dim(\mathcal{N}(R_{xj}^k)) > 0$ (Definition 4.1.1). By Equation 4.10, we know that only iterations in $\mathcal{N}(F_{xj}^k)$ can ever cause dimension replication – these are exactly the iterations that access the same elements of array x. The computation communication constraint (Equation 4.4) is the constraint that causes iterations in $\mathcal{N}(F_{xj}^k)$ to execute on the same processor. Thus to generate multicast communication, we eliminate the constraints caused by accesses where $\mathcal{N}(F_{xj}^k) \neq \emptyset$. Since we keep the other constraints, we are not allowing general replication of x, just multicast communication. We then use the `Basic_Nullspaces` algorithm to find the nullspaces of the linear decomposition matrices with the modified constraints. Finally, we calculate R_{xj}^k and the linear data decomposition D_x in the same way as replicatable arrays using algorithm `Replicated_Data_Decomp`.

4.1.4 Linear Decompositions with Synchronization

Up to this point we have only considered linear decompositions with no synchronization within each loop nest. We only allowed iterations of outermost *doall* loops to be distributed across the processors. However, as we saw in Section 2.3.2, only exploiting the parallelism in *doall* loops may not result in the best overall decomposition. In some cases, we can get better performance by using *doacross* parallelism in the loop nests. In this section we describe how to find linear decompositions that also allow synchronization within fully permutable loop nests by distributing iterations of *doacross* loops. The linear decompositions we find still maintain the property that there is no data-reorganization communication.

The outermost parallel loops in the entire loop nest are within the outermost fully permutable subnest. When finding linear decompositions with synchronization, we first consider exploiting *doacross* parallelism in the outermost fully permutable subnest. The linear decomposition algorithm can be re-applied to exploit parallelism within inner fully permutable subnests, starting with the outer parallel loops in that subnest. To find linear decompositions with synchronization within a fully permutable loop nest, we use essentially the same algorithm that was used for finding basic linear decompositions in Section 4.1.2. We need only update the synchronization equations and corresponding constraints on the nullspaces of the linear decomposition matrices; the rest of the algorithm is unchanged. The updated synchronization equations are presented in Section 4.1.4.1 and Section 4.1.4.2 shows an example. Finally, Section 4.1.4.3 describes how the equations are solved.

4.1.4.1 Formulating Equations with Synchronization

The purpose of the original synchronization equations was to avoid all synchronization within the loop nest, and all loops that were not outermost *doall* loops were assigned to the same processor. However, in Section 2.3.1 we saw that we can efficiently distribute iterations of *doacross* loops with finite distances across processors. Thus we now relax the synchronization equations, and do not assign iterations of such loops to the same processor. Formally, for a loop nest j of depth l, if a loop at nesting level r has a dependence that is not a finite distance, or the loop is not in the current fully permutable subnest, then all iterations $\vec{\imath}$ and $\vec{\imath} + \vec{e_r}$ in the loop nest must be allocated to the same processor, where $\vec{e_r}$ is the rth elementary vector of dimension l. This gives the equation:

$$C_j(\vec{e_r}) = \vec{0} \tag{4.11}$$

4.1.4.2 Examples of Linear Decompositions with Synchronization

Consider the following example of an ADI (Alternating Direction Implicit) integration, originally shown in Section 2.3.1:

```
/* Loop nest 1 */
for $i_1$ = 1 to N do                /* doall */
  for $i_2$ = 2 to N do
    x[$i_1$,$i_2$] = $f_1$(x[$i_1$,$i_2$], x[$i_1$,$i_2$-1])
/* Loop nest 2 */
for $i_1$ = 1 to N do                /* doall */
  for $i_2$ = 2 to N do
    x[$i_2$,$i_1$] = $f_2$(x[$i_2$,$i_1$], x[$i_2$-1,$i_1$])
```

For this example, the original set of synchronization and communication equations for a basic linear decomposition from Section 4.1.2.1 are as follows:

$$\begin{aligned} C_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} &= \vec{0} \\ C_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} &= \vec{0} \\ D_x \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} &= C_1 \\ D_x \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} &= C_2 \end{aligned}$$

The maximum rank solution for these equations is

$$C_1 = C_2 = D_x = \emptyset$$

However, the loops in both loop nest 1 and loop nest 2 are fully permutable and have dependences that can by represented by distances, i.e. $(0, 1)$. We eliminate the synchronization equations and the complete set of equations becomes:

$$\begin{aligned} D_x \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} &= C_1 \\ D_x \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} &= C_2 \end{aligned}$$

A maximum rank solution for these equations is

$$C_1 = D_x = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad C_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Note that the algorithm yields a solution that distributes the full iteration and array spaces, and does not over-constrain the decompositions unnecessarily. In both loop nests, we distribute both the outer *doall* loop and the inner *doacross* loop. This solution can have idle processors, because of the ordering constraints between the processors in the *doacross* loops. The idle processors are dealt with in the virtual-to-physical mapping, described in Section 4.2.

4.1.4.3 Solving Equations with Synchronization

The previous section presented the equations to describe linear decompositions that allow synchronization within a loop nest. To solve these equations we need only modify the synchronization constraint on the nullspace of the linear computation decomposition matrices. For a loop nest j, we only include $e_r \in \mathcal{N}(C_j)$ for those loops at nesting level r with dependences that are not finite distances, or that are not within the current fully permutable subnest. We then use `Calc_Matrices` to calculate the linear decomposition matrices.

4.1.5 Algorithm Summary: Linear Decompositions

So far in this chapter, we presented an algorithm for finding basic linear decompositions. We then extended the basic algorithm to support replication, multicast and synchronization. How do these all fit together? Our approach is to first try to find a basic linear decomposition. At this point we will also replicate any replicatable arrays. Clearly, if a non-trivial basic linear decomposition can be found then it is preferable. However, it is possible that a non-trivial basic linear decomposition does not exist, and the only solution is to put all the data and computation on a single processor. In this case, we then use multicast and synchronization to eliminate some of the constraints. Multicast and synchronization both allow more potential parallelism, but at the cost of additional overhead.

The complete algorithm for finding linear decompositions requires only a small update to the `Basic_Nullspaces` algorithm from Figure 4.5. The new version of the algorithm is shown in Figures 4.11 and 4.12. Figure 4.11 shows the new version of `Basic_Nullspaces`, called `Calc_Nullspaces`. Figure 4.12 contains the complete algorithm for calculating linear decompositions, `Calc_Linear_Decomps`.

The first step of `Calc_Nullspaces` is to calculate the nullspaces of the linear decomposition matrices starting with the complete set of constraints in Equations 4.3, 4.4 and 4.5 from Section 4.1.2.3. The nullspaces are examined to determine the degree of parallelism. If there is sufficient parallelism, then the algorithm terminates (at this point this is exactly the same as the original `Basic_Nullspaces` algorithm). Otherwise, at the next step the constraints are re-initialized to allow for multicast, and next synchronization. At each step the nullspaces are recalculated, and the algorithm terminates if at any step it determines that there is sufficient parallelism.

4.1.6 Finding Offset Decompositions

A complete affine decomposition consists of a linear decomposition and an offset decomposition. The previous sections of this chapter have described our algorithm finding the linear decompositions. In this section we describe how to calculate the offset decompositions.

We can use Equation 3.2 to find the offset computation decomposition given the offset data decomposition, and vice-versa. Given a complete affine data decomposition, $D_x(\vec{a}) + \vec{\delta}_x$, for array x referenced in a loop nest j (with the array access function $F^k_{xj}(\vec{\imath}) + \vec{\zeta^k_{xj}}$) the computation offset $\vec{\gamma}_j$ is:

$$\vec{\gamma}_j = D_x \vec{\zeta^k_{xj}} + \vec{\delta}_x \tag{4.12}$$

The offset data decomposition, $\vec{\delta}_y$, for another array y accessed in the same loop is:

$$\vec{\delta}_y = \vec{\gamma}_j - D_y \vec{\zeta^k_{yj}} \tag{4.13}$$

As we expect communication at the offset level to be relatively inexpensive nearest-neighbor communication, we will not force a loop to execute on a single processor to avoid communication due just to offsets. However, we try to minimize any communication

```
algorithm Calc_Nullspaces
            ($G_s$ : interference_graph,   /* $G_s = (V_c, V_d, E)$ */
            Γ : set of vector_space,
            Δ : set of vector_space,
            ConstrC_Table : array of (list of constraint),
            ConstrD_Table : array of (list of constraint))
   return
            (Γ : set of vector_space,
            Δ : set of vector_space)

   enum Decomp_Kind { Basic, Multicast, Synchronization };
   curr_kind : integer = current Decomp_Kind for $G_s$;

   Propagate_Nullspaces($G_s$, Γ, Δ);
   if sufficient parallelism return (Γ, Δ);

   for $k$ = curr_kind +1 to max(Decomp_Kind) do
      set Decomp_Kind for $G_s$ to $k$;
      (Γ, Δ) = Init_Nullspaces($G_s$, ConstrC_Table[$k$], ConstrD_Table[$k$]);
      Propagate_Nullspaces($G_s$, Γ, Δ);
      if sufficient parallelism return (Γ, Δ);
   end for;

   return (Γ, Δ);
end algorithm;
```

Figure 4.11: Algorithm for calculating the nullspaces of the linear decomposition matrices with multicast and synchronization.

```
algorithm Init_Constraint_Tables
    return
            (ConstrC_Table : array of (list of constraint),
             ConstrD_Table : array of (list of constraint))

    enum Decomp_Kind { Basic, Multicast, Synchronization };

    ConstrC_Table[Basic] = /* Equations 4.3, 4.4 */;
    ConstrD_Table[Basic] = /* Equation 4.5 */;

    ConstrC_Table[Multicast] = /* Equation 4.3 */;
    ConstrD_Table[Multicast] = /* Equation 4.5 */;

    ConstrC_Table[Synchronization] = /* Equation 4.11 */;
    ConstrD_Table[Synchronization] = /* Equation 4.5 */;

    return (ConstrC_Table, ConstrD_Table);
end algorithm;

algorithm Calc_Linear_Decomps
            ($G_s$ : interference_graph)   /* $G_s = (V_c, V_d, E)$ */

    ConstrC_Table : array of (list of constraint);
    ConstrD_Table : array of (list of constraint);
    $\Gamma$ : set of vector_space;
    $\Delta$ : set of vector_space;

    mark replicatable arrays;
    (ConstrC_Table,ConstrD_Table) = Init_Constraint_Tables;

    set Decomp_Kind for $G_s$ to Basic;
    ($\Gamma$, $\Delta$) = Init_Nullspaces($G_s$, ConstrC_Table[Basic], ConstrD_Table[Basic]);
    ($\Gamma$, $\Delta$) = Calc_Nullspaces($G_s$, $\Gamma$, $\Delta$, ContrC_Table, ConstrD_Table);
    Calc_Matrices($G_s$, $\Gamma$, $\Delta$);
    if replication then Calc_Replication($G_s$);
end algorithm;
```

Figure 4.12: Algorithm for calculating linear decompositions with replication, multicast and synchronization.

caused by conflicting offsets whenever possible. The offsets are calculated after the linear decomposition matrices have already been determined. We use a simple greedy strategy to find the offset decompositions. We start with an array vertex in the interference graph and set its offset decomposition to $\vec{0}$. We then traverse the vertices in the interference graph in breadth-first order and set the computation and data offsets using Equations 4.12 and 4.13, respectively. Whenever there is a choice of edges, we select the edge with the largest offset. Note that when calculating the offset decompositions in this manner, negative values can result. Since we use the convention that the virtual processor numbers are all non-negative, offset decompositions must map the data and computation into non-negative integer values. We eliminate any negative numbers in the offsets by shifting the offsets decompositions by the largest negative number in each processor dimension.

4.2 Finding Physical Processor Mappings

In this section we describe how we map the virtual processor space onto the physical processor space. The goal of this step is to effectively utilize the limited physical resources and to further optimize communication for the target architecture.

At this point, the virtual processor space has n dimensions where n is the maximum degree of parallelism (given by Equation 4.8). Since distributing as many dimensions as possible tends to increase the computation to communication ratio[71, 81], by default we partition the virtual processor space into n-dimensional units. We thus treat physical processors of the target machine as also having n dimensions.

We consider three possible folding functions for each dimension that is distributed: BLOCK, CYCLIC and BLOCK-CYCLIC(b). Let P_v be the number of virtual processors in a given dimension and P_p be the number of physical processors. A BLOCK folding function means that $\left\lceil \frac{P_v}{P_p} \right\rceil$ contiguous virtual processors are assigned to each physical processor. With a CYCLIC folding function, each virtual processor is mapped to a physical processors using a round-robin assignment. Similarly, with BLOCK-CYCLIC(b), b contiguous virtual processors are assigned round-robin across the physical processors. The number of contiguous virtual processors mapped to each physical processor, or block size, for the folding functions is thus $\left\lceil \frac{P_v}{P_p} \right\rceil$, 1 and b for BLOCK, CYCLIC and BLOCK-CYCLIC(b), respectively.

To calculate the folding function, each loop nest is examined to gather any constraints that loop nest may have on the folding function. If the execution time of each iteration in a distributed loop is highly variable, a CYCLIC folding function is needed to improve the load balance. An example of such code is if a distributed loop with loop index variable i contains an inner loop whose bounds are a function of i. If a loop nest has a large amount of nearest neighbor communication, then a BLOCK folding function is needed to reduce the amount of communication. Assigning contiguous blocks of virtual processors to each physical processor eliminates any nearest-neighbor communication between processors in the same block. This is particularly important if we are using synchronization within a distributed *doacross* loop, as the BLOCK folding function not only reduces the communication volume but also helps reduce the frequency of synchronization. If we find both CYCLIC and BLOCK constraints for a virtual processor dimension, then we use a BLOCK-CYCLIC folding function. If there are no constraints on the folding function imposed by any of the loop nests, then by default we use a BLOCK folding function.

The affine functions that map the data and computation onto the virtual processor space are composed with the folding function that maps the virtual processor space onto the physical processor space. The full decompositions are represented by linear inequalities. Given affine computation decompositions $c(\vec{\imath})$ and data decompositions $d(\vec{a})$ in each dimension, and the folding function with block size b, the corresponding inequalities are:

$$\begin{array}{ccccc} bp & \leq & c(\vec{\imath}) & < & bp+1 \\ bp & \leq & d(\vec{a}) & < & bp+1 \end{array}$$

where p represents the processor number.

We perform one additional optimization in the virtual-to-physical processor mapping. It is possible that the number of virtual processor dimensions is larger than the nesting depth of a loop nest. As a result, there will be *idle processors* as only a fraction of the processors will ever be busy. For example, this can happen when loop nests access only subsections of an array. We use the affine computation decompositions to find any idle processor dimensions, that is, virtual processor dimensions where the computation is local to a single processor. Let n' be the number of non-idle virtual processor dimensions. Then we only map the n' non-idle virtual processor dimensions onto an n'-dimensional physical processor space. An overview of the complete static decomposition algorithm is shown in Figure 4.13.

```
**algorithm** Static_Decomps

    $G_s$ : interference_graph;

    $G_s$ = build interference graph;

    Calc_Linear_Decomps($G_s$);
    calculate the offset decompositions for $G_s$;
    calculate the virtual-to-physical mapping for $G_s$;
**end algorithm**;
```

Figure 4.13: Algorithm for finding static decompositions.

4.3 Summary

In this chapter, we presented an algorithm for calculating static decompositions. Starting from the array access functions for a set of arrays accessed within a set of loop nests, the algorithm calculates a data decomposition for each array and a computation decomposition for each loop nest.

The algorithm is based on the two-step mathematical model of decompositions presented in the previous chapter. First, it finds an affine decomposition onto the virtual processor space and then it maps the virtual processor space onto the physical processor space. The algorithm for calculating affine decompositions onto the virtual processors is again split into two phases, based on the observation that the communication due to mismatches in the linear transformation part of the affine decomposition are more expensive than mismatches in the offset part. Thus the algorithm first solves for the linear decomposition, and then solves for the offset decomposition to form the complete affine decomposition.

To calculate linear decompositions, we first considered the restricted case of basic linear decompositions. The algorithm works by setting up a system of equations that represents the necessary conditions that a valid basic linear decomposition must satisfy. The algorithm solves the equations by first calculating the nullspaces of the matrices that represent the linear decompositions, and then calculating the specific matrices in a separate step. The nullspaces represent the data and computation that are assigned to the same processor, and thus determine the degree of parallelism in the computation. The algorithm is optimal in that it is guaranteed to find basic linear decompositions with smallest possible nullspaces, and thus the maximum degree of parallelism.

Next we extended the basic linear decomposition algorithm to allow restricted forms of communication, replication and multicast. Finally, we added the ability to have regular synchronization within fully permutable loop nests.

Note that the algorithm for finding static decompositions finds the maximum degree of parallelism given that there is only one decomposition for each (non-replicatable) array. It does not consider any tradeoffs between data-reorganization communication and parallelism. In the worse case, the algorithm can still return with all the data and computation on a single processor. This indicates that data reorganization is needed, and that the decompositions must be allowed to change dynamically.

Chapter 5

A Dynamic-Decomposition Algorithm

In this chapter we describe our algorithm for finding dynamic decompositions. A decomposition is dynamic if the data decompositions are allowed to change across different loop nests. The static-decomposition algorithm from the previous chapter is used as a building block in the dynamic-decomposition algorithm. We begin in Section 5.1 by formally specifying the dynamic decomposition problem. Section 5.2 discusses the complexity of the problem and shows that finding optimal dynamic decompositions is NP-complete. Our algorithm for finding dynamic decompositions is described in Section 5.3 and a detailed example is presented in Section 5.4. Finally, in Section 5.5 we discuss related work on decomposition algorithms.

5.1 Formulation of the Dynamic Decomposition Problem

This section describes our formulation of the dynamic decomposition problem. First, Section 5.1.1 explains the program representation and cost model that serve as inputs to the dynamic-decomposition algorithm. Then, in Section 5.1.2 we present a formal statement of the problem.

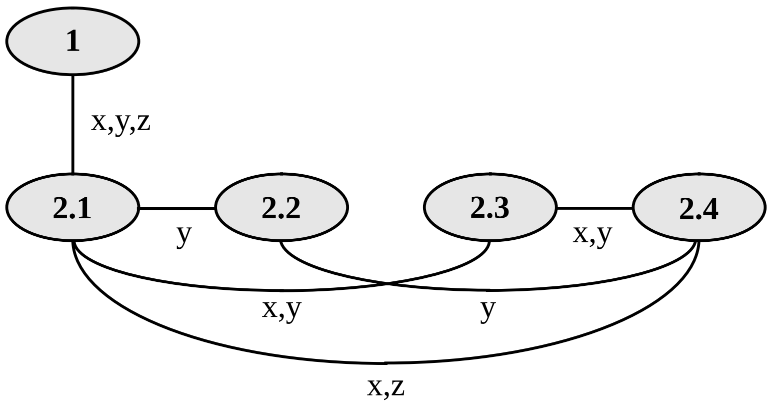
5.1.1 Program Representation

We represent each procedure in the program using a *communication graph.* The vertices in the graph correspond to the loop nests in the procedure with one or more degrees of parallelism. Each of the loop nests is in the form of nests of fully permutable loop nests (see Section 2.3.1). Each vertex has a table with $l + 1$ associated weights, where l is the depth of the loop nest represented by that vertex. For each loop $i \in \{1 \ldots l\}$ in the loop nest, table entry i is an estimate of the execution time if that loop is distributed across the processors. We also add a final entry to the table with an estimate of the sequential execution time if the entire loop nest is run locally on a single processor. We only need a single weight per loop because of the assumption that all distributed loops are sufficiently large to keep all the processors busy (Section 4.1.2). Our current implementation uses static estimates for the loop execution times. The compiler examines the code and estimates the time to execute the computation in the loop nest. The execution time for the parallel loops is the computation time divided by the number of processors, plus the cost of any necessary synchronization.

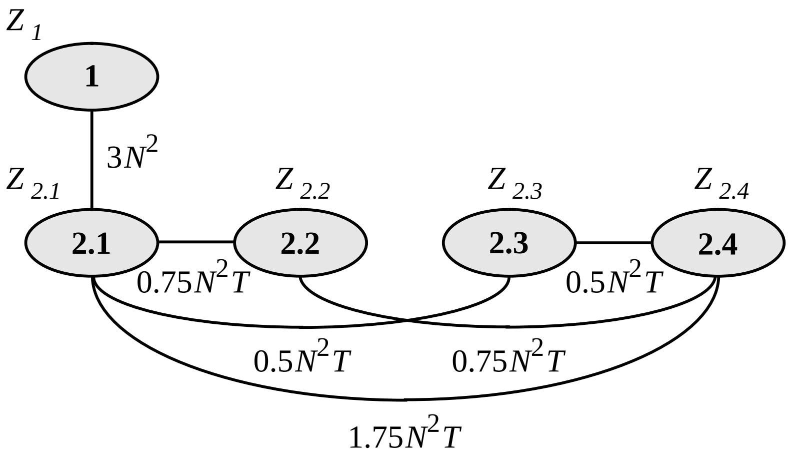
The edges in the communication graph represent potential communication in the procedure. The edges are undirected, and an edge (u, v) with weight $w(u, v)$ indicates that if the data decompositions for the arrays accessed in u are not equal to the data decompositions for those same arrays in v, then the communication cost is at most $w(u, v)$. There is an edge between two vertices u and v if the data decomposition for any array in vertex u can ever reach vertex v. The edges are analogous to standard def-use chains and are found in a manner similar to the standard reaching definitions data flow problem[3]. The key differences are that we calculate the edges only for array data, and we treat all array accesses within a loop nest as both a *use* and a *definition* of that array. There is at most one edge between any two vertices u and v; a single edge is used even if there are multiple arrays in u with decompositions that reach v. Figure 5.2(a) shows the communication graph for the code from Figure 5.1. In the figure, all vertices are labeled with the numbers of the loop nests they represent. The edges are annotated with the list of arrays that produced the edge. For example, there is an edge between vertices 2.1 and 2.4 because the data decompositions of arrays x and z reach from loop nest 2.1 to loop nest 2.4. Array x reaches along the path (2.1, 2.2, 2.4) since it is not accessed at vertex 2.2, and array z reaches along both paths (2.1, 2.2, 2.4) and (2.1, 2.3, 2.4).

```
real x[N,N], y[N,N], z[N,N]
/* Loop nest 1 */
for $i_1$ = 1 to N do            /* doall */
  for $i_2$ = 1 to N do          /* doall */
    x[$i_1$,$i_2$] = $f_1$($i_1$,$i_2$)
    y[$i_1$,$i_2$] = $f_2$($i_1$,$i_2$)
    z[$i_1$,$i_2$] = $f_3$($i_1$,$i_2$)
  end for
end for

/* Loop 2 */
for time = 1 to T do
  /* Loop nest 2.1 */
  for $i_1$ = 1 to N do          /* doall */
    for $i_2$ = 1 to N do        /* doall */
      x[$i_1$,$i_2$] = x[$i_1$,$i_2$] + z[$i_1$,$i_2$]
      y[$i_1$,$i_2$] = y[$i_1$,$i_2$] + z[$i_1$,$i_2$]
    end for
  end for
  if (expr) then
    /* Loop nest 2.2 */
    for $i_1$ = 1 to N do        /* doall */
      for $i_2$ = 1 to N do
        y[$i_1$,$i_2$] = y[$i_1$,$i_2$] + y[$i_1$,N-$i_2$+1]
  else
    /* Loop nest 2.3 */
    for $i_1$ = 1 to N do        /* doall */
      for $i_2$ = 1 to N do      /* doall */
        y[$i_1$,$i_2$] = x[$i_1$,$i_2$] + x[N-$i_1$+1,$i_2$]
  end if

  /* Loop nest 2.4 */
  for $i_1$ = 1 to N do          /* doall */
    for $i_2$ = 1 to N do        /* doall */
      z[$i_1$,$i_2$] = x[$i_1$,$i_2$] + y[$i_1$,$i_2$] + z[$i_1$,$i_2$]
end for
```

Figure 5.1: Code example used to illustrate dynamic decompositions.



(a)



KEY:

$Z_1 = (100N, 100N, 100N^2)$ $\quad Z_{2.2} = (75NT, 75N^2T)$

$Z_{2.1} = Z_{2.4} = (100NT, 100NT, 100N^2T)$ $\quad Z_{2.3} = (25NT, 25NT, 25N^2T)$

(b)

Figure 5.2: Example communication graphs for the code in Figure 5.1: (a) the communication graph annotated with the lists of arrays that produced each edge, and (b) the communication graph annotated with vertex and edge weights. All vertices are labeled with the numbers of the loops they represent.

The weight of edge (u, v), $w(u, v)$, is computed from the frequency with which data decompositions in vertex u will reach the other vertex v, and the amount of data in the arrays. If we let $A_{(u,v)}$ represent the set of all arrays that reach from vertex u to v, then the weight of edge (u, v) is:

$$w(u, v) = \sum_{x \in A_{(u,v)}} \psi(u, v, x) \cdot \omega(\mid x \mid) \tag{5.1}$$

where $\psi(u, v, x)$ is the frequency of execution along the path from u to v for array x, $\omega(k)$ is the time it takes to move k data elements on the target machine, and $\mid x \mid$ is the size of array x. The frequency of execution $\psi(u, v, x)$ can differ for different arrays on the same edge (u, v), depending on the path along which the decomposition for each array reaches from u to v. This model of edges and associated weights treats each array as a unit – it represents communicating entire arrays between the two vertices. Once an array is communicated along (u, v), then it is reorganized and can have a different data decomposition in u than in v.

The complete communication graph with vertex weights and edge weights for the sample code in Figure 5.1 is shown in Figure 5.2(b). The figure assumes that the `then` branch of the `if` statement is taken 75% of the time. For the vertex weights, we assume that the time to execute each loop body takes 100 cycles to execute sequentially, and that running any loop in parallel speeds up the execution by a factor of N. In this example, all vertices represent loop nests of depth 2. The vertices are labeled with values representing the sequential execution time, and the parallel execution time for each of the loops in the nest (represented in the figure by the labels Z_1 through $Z_{2.4}$). For example, vertex 2.1 has weights (100NT, 100NT, 100N^2T), which represents an execution time of 100N^2T cycles if all loops in loop nest 2.1 run sequentially, and 100NT cycles if either loop i_1 or loop i_2 is parallelized. Vertex 2.2 has weights (75NT, 75N^2T); the coefficient of the weights is 75 since the loop nest is only executed 75% of the time, and there are only two values since only the i_1 loop in loop nest 2.2 is parallelizable. For the edge weights, we assume a unit communication cost per data element, i.e. $\omega(k) = k$. The edge weights are computed using Equation 5.1:

$$\begin{array}{lllll}
w(1,2.1) & = & \psi(1,2.1,x)\cdot N^2+\psi(1,2.1,y)\cdot N^2+\psi(1,2.1,z)\cdot N^2 & & \\
 & = & N^2+N^2+N^2 & = & 3N^2 \\
w(2.1,2.2) & = & \psi(2.1,2.2,y)\cdot N^2 & = & 0.75N^2T \\
w(2.1,2.3) & = & \psi(2.1,2.3,x)\cdot N^2+\psi(2.1,2.3,y)\cdot N^2 & & \\
 & = & 0.25N^2T+0.25N^2T & = & 0.5N^2T \\
w(2.1,2.4) & = & \psi(2.1,2.4,x)\cdot N^2+\psi(2.1,2.4,z)\cdot N^2 & & \\
 & = & 0.75N^2T+N^2T & = & 1.75N^2T \\
w(2.2,2.4) & = & \psi(2.2,2.4,y)\cdot N^2 & = & 0.75TN^2 \\
w(2.3,2.4) & = & \psi(2.3,2.4,x)\cdot N^2+\psi(2.3,2.4,y)\cdot N^2 & & \\
 & = & 0.25N^2T+0.25N^2T & = & 0.5N^2T
\end{array}$$

5.1.2 Problem Statement

How do we express dynamic decompositions using the communication graph? Given a communication graph, a dynamic decomposition is represented by a partitioning of the vertices into disjoint sets. Each set of vertices has that property that there is a single, static decomposition in the region of the program represented by those vertices. Across different sets, the data decompositions can change dynamically. Thus, there is no data-reorganization communication within a set, but data-reorganization communication may occur across different sets. We refer to these sets of vertices as *static decomposition regions*.

Partitioning the vertices into static decomposition regions uniquely determines the computation and data decompositions at every loop nest in the procedure. For all loop nests and all arrays in a given static decomposition region, we can directly calculate their computation and data decompositions using the static-decomposition algorithm from the previous chapter. The algorithm is guaranteed to return static decompositions with the largest degree of parallelism. Given a dynamic decomposition for a communication graph, each vertex is labeled with the computation decomposition for the loop nest represented by that vertex, and the data decompositions for all arrays accessed within the loop nest.

The *cost* of a dynamic decomposition for a communication graph represents an estimate of the total execution time for the procedure. The cost of the dynamic decomposition is computed by summing the edge costs and the vertex costs from the communication graph. The cost of an edge is the edge weight if the endpoints of the edge are in different static decomposition regions; otherwise the edge cost is zero. The edge costs represent the time spent to communicate data as the decompositions change across the different static decomposition regions.

The cost of the vertices are calculated as follows. Let k be the number of distributed loops in the computation decomposition for a loop nest of depth l, with $0 \leq k \leq l$. If $k > 0$, then the k distributed loops are used to index into the weight table at the vertex for that loop nest; this returns a list of k weights, representing the execution time of the loop nest when each of the k loops is distributed. We set the cost of the vertex to the maximum over the list of k weights, which gives a worst-case estimate of the time to execute the loop nest with k distributed loops. If $k = 0$, then the cost of the vertex is the sequential execution time of the loop nest. Note that the nullspaces of the linear computation decomposition matrices specify which loops in each loop nest are executed locally on the same processor; all other loops are distributed. Thus we only need the nullspaces to compute the cost of a vertex – the complete decompositions are not necessary (we make use of this fact in our dynamic decomposition algorithm in Section 5.3). Taken together, the vertex costs represent an estimate of the computation time for the procedure. The total cost of a dynamic decomposition for a communication graph cost is thus the sum of the communication time and the computation time, and is an estimate of the total execution time for the procedure.

For a given communication graph, the objective function for the dynamic decomposition problem is to partition the graph into static decomposition regions such that the cost is minimized. We define the dynamic decomposition problem formally as follows. Given a communication graph $G_c = (V, E)$ with weighted vertices and weighted edges, find a function $g : V \rightarrow \{1, 2, \ldots, \mid V \mid\}$ such that the cost of the resulting dynamic decomposition for G_c is minimized. The value of the function g for a vertex $v \in V$ is the number of the static decomposition region that contains v. The maximum number of static decomposition regions is $\mid V \mid$, the total number of vertices, as it is possible for each static decomposition region to contain a single vertex.

5.2 Complexity of the Dynamic Decomposition Problem

To analyze the complexity of the dynamic decomposition problem, we first turn it into a decision problem by asking: for a given communication graph $G_c = (V, E)$, does there exist a function $g : V \rightarrow \{1, 2, \ldots, \mid V \mid\}$ such that the total cost of G_c is less than some positive integer B?

Theorem 5.2.1 *The dynamic decomposition problem is NP-complete.*

Proof: The dynamic decomposition problem is in NP since a nondeterministic algorithm can guess the function g and check in polynomial time that the cost of the dynamic decomposition is less than B.

We transform the known NP-complete problem, Colored Multiway Cut[26], into a subproblem of the dynamic decomposition problem. The Colored Multiway Cut (CMC) problem is: given a graph $G = (V, E)$ with weighted edges, and a partial k-coloring of the vertices, i.e., a subset $V' \subseteq V$ and a function $g : V' \rightarrow 1, 2, \ldots, k$, can g be extended to a total function such that the total weight of edges that have different colored endpoints is less than some positive integer B?

Consider the dynamic decomposition subproblem (DDS) in which the program accesses only a single array x. Let an arbitrary instance of CMC be given by a graph $G = (V, E)$ and positive integers k and B. We can reduce an instance of CMC into an instance of DDS in polynomial time by writing the input program outlined below. The strategy is to construct a program such that there is a one-to-one mapping between the colors in CMC and the static decompositions regions in DDS. The program will have at most k possible static decomposition regions, where each region leads to a decomposition that distributes one dimension of array x. The program is constructed so that the vertices in CMC become vertices in the communication graph, and the edges in CMC become the edges in the communication graph.

- The single array x is k-dimensional, where k is the number of colors in the Colored Multiway Cut problem. All k dimensions are of equal size N.
- Each vertex in the original CMC problem becomes a loop nest of depth k in the DDS input program, and thus a vertex in the communication graph.

- For each edge (u, v) in G with weight $w(u, v)$, we add a conditional branch statement after the loop nest representing vertex u whose target is the loop nest representing vertex v. We write the branch such that its frequency of execution is $\psi(u, v, x) = w(u, v)/\omega(N^k)$. From Equation 5.1, this results in an edge in the communication graph for DDS with weight $(w(u, v)/\omega(N^k)) \cdot \omega(N^k) = w(u, v)$. As a result, the weighted edges in the communication graph for DDS correspond directly to the weighted edges in CMC.

- For each vertex $v \in V'$ of color r in CMC, we generate the following array accesses in the loop nest for v:

```
for $i_1$ = 1 to N/2 do
...
  for $i_r$ = 1 to N do            /* doall */
  ...
    for $i_k$ = 1 to N/2 do
      x[$i_1$,...,$i_r$,...,$i_k$] = $f$(x[$2*i_1$,...,$i_r$,...,$2*i_k$])
```

The rth loop is a *doall* loop, and the remaining $k - 1$ loops in the loop nest are sequential. Also, the parallel loop can legally be moved to the outermost position in the loop nest.

For each vertex $v \in V - V'$ (the vertices that do not have a preassigned color) we generate the following array accesses in the loop nest for v:

```
for $i_1$ = 1 to N do              /* doall */
...
  for $i_k$ = 1 to N do            /* doall */
    x[$i_1$,...,$i_k$] = $f$(x[$i_1$,...,$i_k$])
```

All k loops in the loop nest are *doall* loops.

In both cases above, the index expression for the ith dimension of array x is always an affine function of only the ith loop index variable. This ensures that the resulting data and computation decompositions will have a one-to-one correspondence between the distributed dimensions of the array and the distributed loops in the loop nest, i.e. if the data decomposition for the array distributes dimension i, then the computation decomposition for the loop nest distributes loop i. Also, all array accesses are perfectly nested, and the entire loop nest is fully permutable.

- The weight table for each of the vertices in DDS is set as follows. The entry for sequential execution is set to a large value, larger than B. The entries for distributing each of the loops is set to 0. These weights guarantee that a solution to DDS can never run a loop nest sequentially, as the cost is always lower to distribute a loop.

The constructed dynamic decomposition problem has a solution with cost less than B, if and only if the original Colored Multiway Cut problem has a solution such that the total weight of edges that have different colored endpoints is less than B. The construction of DDS is such that the solution will have at most k static decomposition regions, each corresponding to distributing a single dimension of array x and distributing one loop in each loop nest. There is a one-to-one correspondence between vertices in CMC and DDS. The number of each vertex's static decomposition region in the solution to DDS is the number of the color for the corresponding vertex in CMC. Clearly the transformation from Colored Multiway Cut into the dynamic decomposition problem is polynomial. Thus, since Colored Multiway Cut is NP-complete the dynamic decomposition problem is NP-complete. □

5.3 Finding Dynamic Decompositions

The number of possible dynamic decompositions for a given communication graph $G_c = (V, E)$ is exponential in the number of vertices, i.e. $2^{|V|}$. The number of loop nests in a procedure, and thus the number of vertices, is a large number in practice. In designing an algorithm for solving the dynamic decomposition problem, we must decide whether to solve exactly using an exponential algorithm, or whether to use heuristics and find an approximate solution. There are a number of compiler problems that successfully use algorithms that are exponential in the worst case. For example, Fourier-Motzkin elimination is used to compute data dependences[59], to calculate new loop bounds after applying loop transformations[8] and to map array accesses across procedure boundaries[25, 37, 38]. Even though the worst-case behavior is exponential, for these problems the algorithm has good behavior in the common case. In fact, as discussed in the next chapter, the interprocedural version of our decomposition algorithm makes use of Fourier-Motzkin elimination to map data decompositions and array accesses across procedure boundaries. Unfortunately, the dynamic decomposition problem itself does not have any common-case behavior that can be easily exploited to create an optimal algorithm that is efficient in practice. Thus, our strategy is to use a simple and efficient heuristic algorithm. The emphasis of the algorithm is on finding static decomposition regions that are as large as possible. The priority is to eliminate expensive data-reorganization communication completely, rather than concentrate on small differences in communication cost.

We use a greedy algorithm that eliminates the largest amounts of potential communication first from the most frequently executed paths in the program. To represent the most frequent paths in the program, we impose a hierarchical structure on the communication graph. We augment the graph by adding *hierarchy vertices* representing outer, sequential loops and directed *hierarchy edges* that form the vertices into a forest of trees. There is a hierarchy edge from vertex u to vertex v if the loop nest represented by v is directly nested within u. The original vertices in the communication graph are now the leaf vertices in the tree formed by the hierarchy edges. For example, Figure 5.3 shows the communication graph from Figure 5.2 augmented with hierarchy edges and vertices. In the figure, the vertex labeled **top** corresponds to the outermost nesting level of the procedure, and all other vertices are labeled with the numbers of the loops they represent.

The basic design of the algorithm is as follows. Each vertex in the communication graph starts out in its own static decomposition region. The algorithm then tries to merge the vertices that have the greatest edge weights into the same static decomposition region, thereby eliminating the possibility of data reorganization between the two loop nests represented by the vertices. The analysis is performed in order from the innermost to the outermost levels in communication graph hierarchy. This has the effect of pushing communication into the outermost loops as much as possible. An overview of the dynamic-decomposition

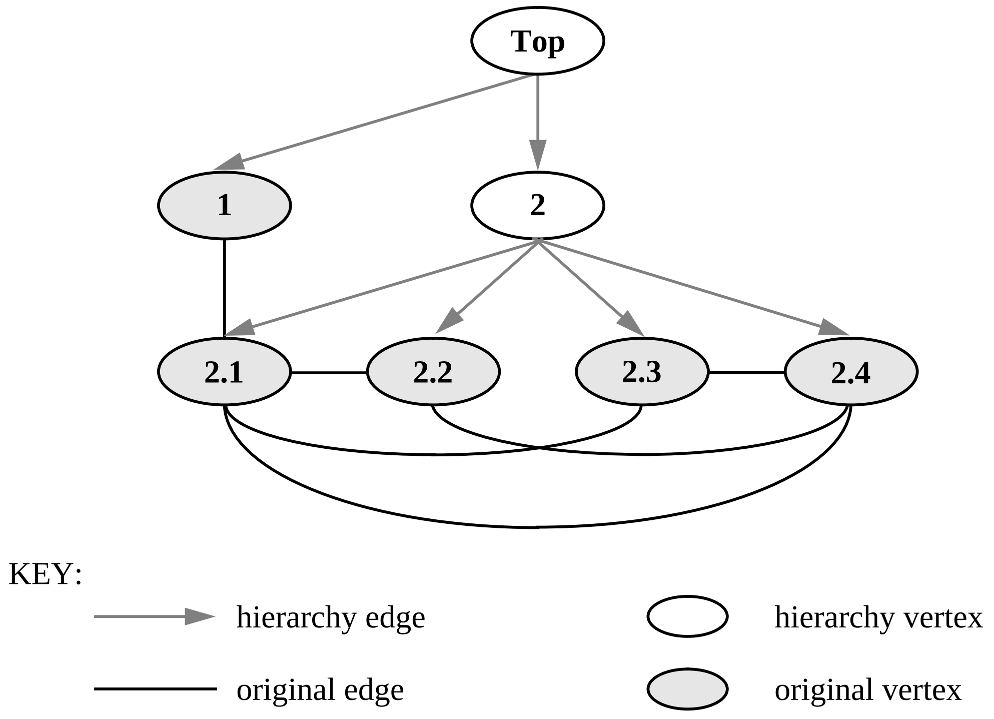


Figure 5.3: An example communication graph with hierarchical structure for the code in Figure 5.1. The vertex labeled **top** corresponds to the outermost nesting level of the procedure, and all vertices are labeled with the numbers of the loops they represent.

algorithm is shown in Figures 5.4 through 5.6.

The driver for the dynamic-decomposition algorithm, `Dynamic_Decomps_Driver` in Figure 5.4, starts by building the hierarchical communication graph for the current procedure. Our current implementation uses static estimates for the path frequencies when computing the communication graph edge weights. More accurate frequencies can be obtained by instrumenting the program and collecting path profile information[12]. The `Dynamic_Decomps` algorithm in Figure 5.5 then places each loop nest in the communication graph in its own static decomposition region. Each static decomposition region is represented by its bipartite interference graph (see Section 4.1.2.3). There is a one-to-one correspondence between the vertices in the static decomposition region of the communication graph and the computation vertices in the interference graph. The initial interference graphs for each static decomposition region thus contain a single computation vertex, and

```
**algorithm** Dynamic_Decomps_Driver
              (*Curr_Proc* : procedure)

   $G_c$ : communication_graph;

   $G_c$ = build communication graph for *Curr_Proc*;
   Dynamic_Decomps($G_c$);

   calculate the offset decompositions for $G_c$;
   calculate the virtual-to-physical mapping for $G_c$;
**end algorithm**;
```

Figure 5.4: Driver algorithm for finding dynamic decompositions.

one data vertex for each array accessed in the corresponding loop nest. The algorithm then calls the `Single_Level_Decomps` algorithm in Figure 5.6 to examine each nesting level of the communication graph in a bottom-up order, from innermost nesting level to outermost.

Within each level of the communication graph hierarchy, the edges are sorted by decreasing edge weight. For example, in the communication graph in Figure 5.2, the order of the edges nested within loop 2 is $(2.1, 2.4)$, $(2.1, 2.2)$, $(2.2, 2.4)$, $(2.1, 2.3)$, $(2.3, 2.4)$.

For each edge (u, v), the algorithm tries to merge u and v into the same static decomposition region. The interference graphs for u and v are merged into a single interference graph, and then the nullspaces for the merged graph are calculated by calling the `Calc_Nullspaces` algorithm from Figure 4.11. This has the effect of putting the two loop nests in the same static decomposition region, and eliminates the data reorganization cost of the edge. When merging two interference graphs, any data vertices common to both interference graphs are combined into a single data vertex, and their constraints are combined. The computation vertices are always distinct in each interference graph, so they are copied directly into the merged graph. If any new cycles are formed in the merged graph, then the data communication constraints must also be updated (Section 4.1.2.3).

The merge may cause some (or all) of the loop nests to execute sequentially, or it may

```
algorithm Dynamic_Decomps
            ($G_c$ : communication_graph)  /* $G_c = (V, E)$ with hierarchy information */

   $G_s$ : interference_graph;
   ConstrC_Table : array of (list of constraint);
   ConstrD_Table : array of (list of constraint);
   $\Gamma$, $\Delta$ : set of vector_space;

   mark replicatable arrays;
   (ConstrC_Table,ConstrD_Table) = Init_Constraint_Tables;

   /* Initialize each vertex into its own static decomposition region */
   foreach $v \in V$ do
      $G_s$ = build interference graph for $v$;
      set Decomp_Kind for $G_s$ to Basic;
      ($\Gamma$, $\Delta$) = Init_Nullspaces($G_s$, ConstrC_Table[Basic], ConstrD_Table[Basic]);
      ($\Gamma$, $\Delta$) = Calc_Nullspaces($G_s$, $\Gamma$, $\Delta$, ContrC_Table, ConstrD_Table);
   end foreach;

   /* Create static decompositions regions at each level in $G_c$ */
   foreach level $\pi$ in $G_c$ in bottom-up order do
       Single_Level_Decomps($G_c$, $\pi$, ContrC_Table, ConstrD_Table);
   end foreach;

   foreach $G_s \in$ set of static decomposition regions in $G_c$ do
      ($\Gamma$, $\Delta$) = set of nullspaces for vertices in $G_s$;
      Calc_Matrices($G_s$, $\Gamma$, $\Delta$);
      if replication then Calc_Replication($G_s$);
   end foreach;
end algorithm;
```

Figure 5.5: Core algorithm for finding dynamic decompositions.

```
algorithm Single_Level_Decomps
            ($G_c$ : communication_graph,  /* $G_c = (V, E)$ with hierarchy information */
            $\pi$ : level,  /* current level in $G_c$ */
            ConstrC_Table : array of (list of constraint),
            ConstrD_Table : array of (list of constraint))

  $G_s$, $G_s'$, $G_s''$ : interference_graph;
  $\Gamma$, $\Delta$ : set of vector_space;
  $\Gamma'$, $\Delta'$ : set of vector_space;
  curr_cost : integer;

  ($\Gamma$, $\Delta$) = set of nullspaces for computation and data in $G_c$;
  curr_cost = cost($G_c$, $\Gamma$, $\Delta$);

  foreach $(u, v) \in E$ at level $\pi$, in order of decreasing weights do
    $G_s'$ = get interference graph for $u$;
    $G_s''$ = get interference graph for $v$;
    $G_s$ = merge interference graphs $G_s'$ and $G_s''$;
    ($\Gamma'$, $\Delta'$) = set of nullspaces for vertices in $G_s$;
    ($\Gamma$, $\Delta$) = ($\Gamma$, $\Delta$) $-$ ($\Gamma'$, $\Delta'$) $+$
        Calc_Nullspaces($G_s$, $\Gamma'$, $\Delta'$, ContrC_Table, ConstrD_Table);

    if cost($G_c$, $\Gamma$, $\Delta$) $<$ curr_cost then
      curr_cost = cost($G_c$, $\Gamma$, $\Delta$);
      record ($\Gamma$, $\Delta$) in $G_c$;
      commit the merge;
    else
      discard the merge;
    end if;

  end foreach;
end algorithm;
```

Figure 5.6: Algorithm for finding dynamic decompositions at a single level of the communication graph.

generate replication or synchronization within loops. The algorithm calculates the total cost of the dynamic decompositions for the communication graph before and after the new nullspaces have been calculated. If the cost of the dynamic decomposition is less after the merge, then the new interference graph is saved and both u and v are set to use the new interference graph. The algorithm then records the new nullspaces of all loops nests and arrays within the new static decomposition region. Otherwise, u and v are in different static decomposition regions and there is data-reorganization communication along the edge.

After the nullspaces for all the loop nests (and corresponding arrays) have been found, then the algorithm calculates the linear decomposition matrices within each static decomposition region. It then calculates the offset decompositions to give complete affine decompositions onto the virtual processor space. Finally, the algorithm calculates the virtual-to-physical processor mappings to yield the final decompositions.

There a few key points to note about this algorithm. As the algorithm progresses, it uses the nullspaces of the linear decomposition matrices to gather constraints on the decompositions. Since we use the nullspaces directly to calculate the cost of the current communication graph, this allows us to solve incrementally as we merge loop nests into larger and larger static decomposition regions. The algorithm relies on the fact that the nullspaces calculated by the static-decomposition algorithm are the minimum nullspaces that meet the constraints. This means that there are no extraneous constraints, and allows the static decomposition regions to grow as large as possible.

Our greedy approach is a simple heuristic, and clearly other heuristics are possible. For the experiments we ran, however, we found that the algorithm works well in practice (see Chapter 7). Our experiments, in addition to other work with hand-parallelized applications[70, 81], show that efficient parallel codes reorganize data infrequently. Our approach of merging loop nests into static decomposition regions in order of the most frequently executed paths in the program, is thus a reasonable strategy for these kinds of programs.

5.4 A Dynamic Decomposition Example

Figure 5.7 shows how interference graphs representing static decomposition regions are used to merge loop nests for the example code from Figure 5.1 and the corresponding communication graph from Figure 5.2 and Figure 5.3. For the purposes of this example, we assume that both the array dimension size N and the bound of the `time` loop T are large values. The array access functions for this example are as follows:

$$\begin{array}{llll}
F^1_{x1} = F^1_{y1} = F^1_{z1} = & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & & \\
F^1_{x2.1} = F^1_{y2.1} = F^1_{z2.1} = & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & & \\
F^1_{y2.2} = & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, & F^2_{y2.2} = & \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \\
F^1_{x2.3} = F^1_{y2.3} = & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, & F^2_{x2.3} = & \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \\
F^1_{x2.4} = F^1_{y2.4} = F^1_{z2.4} = & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & &
\end{array}$$

The algorithm first initializes each of the loop nests into its own static decomposition region. At this point the nullspaces of the linear computation decomposition matrices are:

$$\begin{array}{rcl}
\mathcal{N}(C_1) & = & \emptyset \\
\mathcal{N}(C_{2.1}) & = & \emptyset \\
\mathcal{N}(C_{2.2}) & = & \text{span}\,\{(0,1)\} \\
\mathcal{N}(C_{2.3}) & = & \text{span}\,\{(1,0)\} \\
\mathcal{N}(C_{2.4}) & = & \emptyset
\end{array}$$

The nullspaces of the linear data decomposition matrices for each of the arrays accessed within the loop nests have the same nullspaces as the corresponding linear computation decomposition matrices, e.g. $\mathcal{N}(D_x) = \mathcal{N}(D_y) = \mathcal{N}(C_{2.3})$ for arrays x and y in loop nest 2.3. The `Calc_Nullspaces` algorithm finds that $\mathcal{N}(C_{2.2}) = \text{span}\,\{(0,1)\}$ because

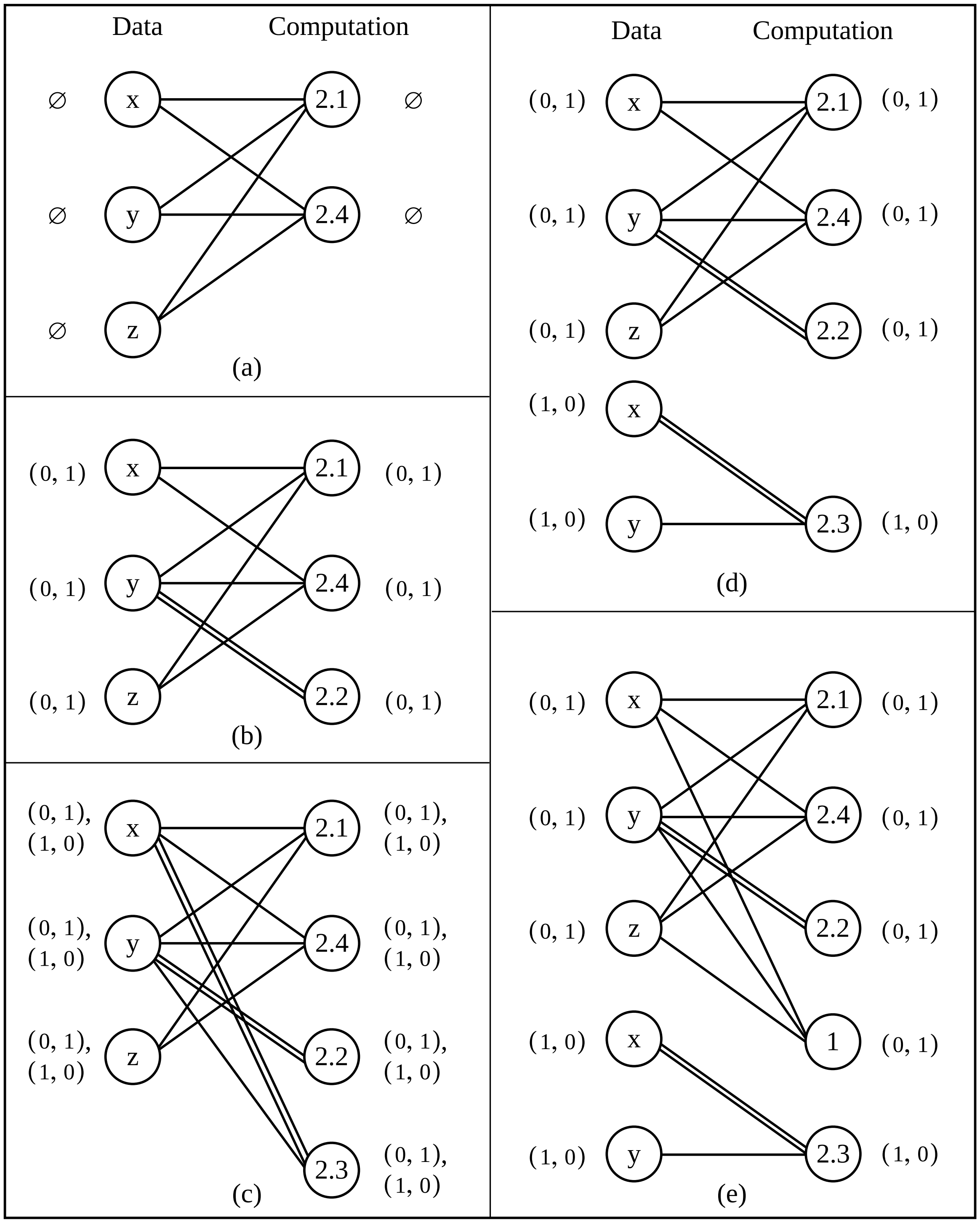


Figure 5.7: Merging loop nests using interference graphs. The computation vertex numbers correspond to the loop nests from Figure 5.1. The basis vectors of the nullspaces for each array and each loop nest are shown next to each vertex.

of a synchronization constraint from the inner sequential loop in loop nest 2.2 (see Section 4.1.2.3). Calling `Propagate_Nullspaces` makes $\mathcal{N}(D_y) = \mathcal{N}(C_{2.2})$ in loop nest 2.2. Also, $\mathcal{N}(C_{2.3}) = \text{span}\,\{(1,0)\}$ because of a data communication constraint arising from the cycle in the interference graph from array accesses $F^1_{x2.3}$ and $F^2_{x2.3}$ (see Section 4.1.2.3). This constraint requires that $D_x F^1_{x2.3} = D_x F^2_{x2.3} = C_{2.3}$, thus $(1,0) \in \mathcal{N}(D_x)$. After calling `Propagate_Nullspaces`, we have $\mathcal{N}(D_x) = \mathcal{N}(D_y) = \mathcal{N}(C_{2.3})$ in loop nest 2.3. We can now calculate the total cost of the initial dynamic decomposition for the communication graph $G_c = (V, E)$. Let W be the sum of the edge weights, $W = \sum_{(u,v)\in E} w(u,v) = 4.25N^2T + 3N^2$ and let Z be the sum of the vertex weights if the loop nests are parallelized, $Z = 300NT + 100N$. Thus the total cost of the initial dynamic decomposition is $W + Z$.

The algorithm first analyzes the vertices at the innermost nesting level, containing the vertices nested within loop 2. The first edge we examine is $(2.1, 2.4)$, and we merge vertices 2.1 and 2.4 into the same static decomposition region. This step is shown in Figure 5.7(a). The nullspaces at both 2.1 and 2.4 were initialized to $\emptyset$, and merging the two vertices creates no additional constraints. We have eliminated the communication along edge (2.1,2.4) and the total cost of the communication graph is $Z + (W - w(2.1, 2.4))$. The cost of the new communication graph is less than the cost of the initial graph, and the merge is committed. Figure 5.7(b) shows the results of processing the next edge, $(2.1, 2.2)$, and merging vertex 2.2 into the same static decomposition region with 2.1 and 2.4. In this case `Calc_Nullspaces` returns with $\mathcal{N}(D_x) = \mathcal{N}(D_y) = \mathcal{N}(D_z) = \mathcal{N}(C_{2.1}) = \mathcal{N}(C_{2.2}) = \mathcal{N}(C_{2.4}) = \text{span}\,\{(0,1)\}$. The next communication graph edge is $(2.2, 2.4)$ but since vertices 2.2 and 2.4 are already in the interference graph, the interference graph is unchanged. At this point, the cost of the current dynamic decomposition is $Z + (W - w(2.1, 2.4) - w(2.1, 2.2) - w(2.2, 2.4)) = Z + (w(2.1, 2.3) + w(2.3, 2.4) + w(1, 2.1)) = (300NT + 100N) + (N^2T + 3N^2)$.

In Figure 5.7(c), we visit edge $(2.1, 2.3)$ and add vertex 2.3 to the interference graph. Here, when `Calc_Nullspaces` returns, all the nullspaces are set to span $\{(0,1), (1,0)\}$ and thus span the entire space. This means that all the data and computation are allocated onto a single processor. The vertex costs now use the sequential execution time for loop nests 2.1 . . . 2.4, and the total graph cost is $(300N^2T + 100N) + (w(2.3, 2.4) + w(1, 2.1))$

$= (300N^2T + 100N) + (0.5N^2T + 3N^2)$. Clearly, the cost is now much larger than before we merged vertex 2.3, i.e. we eliminated the cost of an edge weighted $0.5N^2T$, but incurred a large increase of nearly $300N^2T$ in computation time. Thus the merge is discarded and loop nest 2.3 is split off into its own static decomposition with a separate interference graph as shown in Figure 5.7(d). The cost of the communication graph is now back to $Z + (w(2.1, 2.3) + w(2.3, 2.4) + w(1, 2.1))$. The next communication graph edge is $(2.3, 2.4)$ which has no effect. The algorithm is now finished with the subgraph for the loop nests that are nested within loop nest 2.

Next the algorithm proceeds to the outermost nesting level. It merges loop nest 1 into the static decomposition region containing loop nest 2.1. The final nullspaces are shown in Figure 5.7(e). At this point we have two static decomposition regions in the communication graph, one containing the set of vertices $\{1, 2.1, 2.2, 2.4\}$ and the other containing the set of vertices $\{2.3\}$. The final graph cost is $Z + w(2.1, 2.3) + w(2.3, 2.4)$, meaning that we have data-reorganization communication along edges $(2.1, 2.3)$ and $(2.3, 2.4)$. Figure 5.8 shows the hierarchical communication graph with the static decomposition regions for each vertex. In the figure, the different static decomposition regions are shown with dark grey backgrounds.

We then calculate the linear decomposition matrices and the offsets to find the final affine decompositions. For the static decomposition region $\{1, 2.1, 2.2, 2.4\}$ we have:

$$\begin{array}{lllllllll} C_1 & = & C_{2.1} & = & C_{2.2} & = & C_{2.4} & = & \begin{bmatrix} 1 & 0 \end{bmatrix} \\ D_x & = & D_y & = & D_z & & & = & \begin{bmatrix} 1 & 0 \end{bmatrix} \end{array}$$

For the static decomposition region $\{2.3\}$ we have:

$$\begin{array}{lllll} C_{2.3} & = & & & \begin{bmatrix} 0 & 1 \end{bmatrix} \\ D_x & = & D_y & = & \begin{bmatrix} 0 & 1 \end{bmatrix} \end{array}$$

We use the default BLOCK virtual-to-physical mapping for both static decomposition

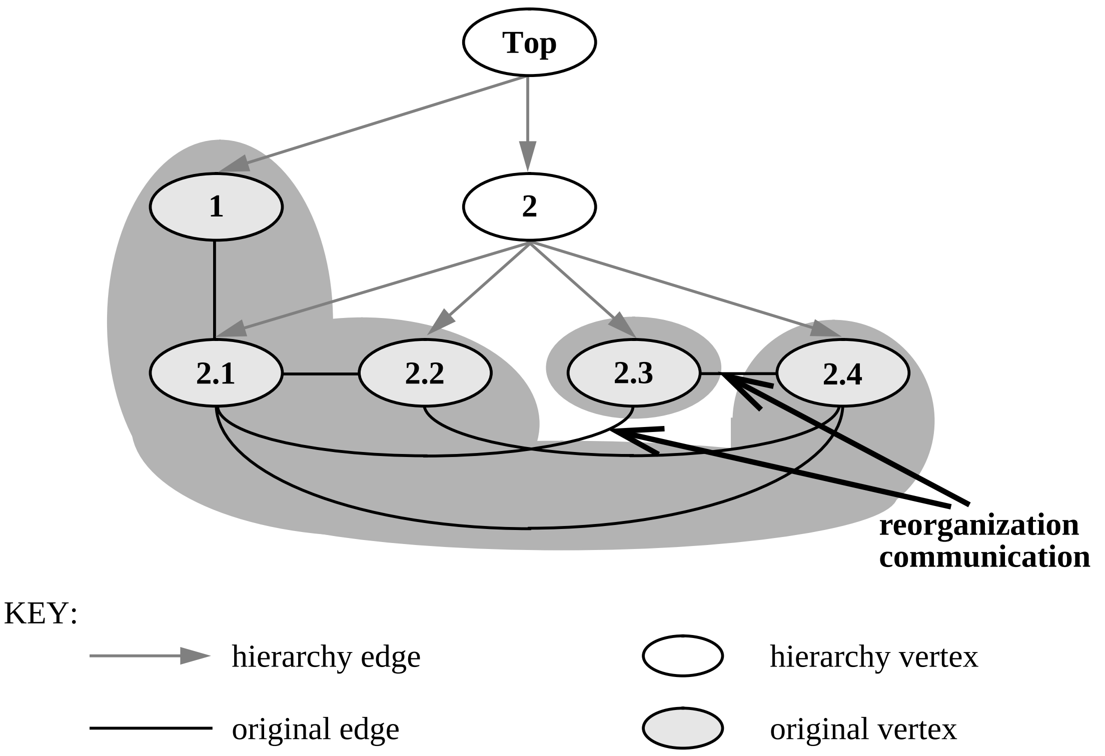


Figure 5.8: Communication graph from Figure 5.3 with static decompositions regions. The different static decomposition regions are shown with dark grey backgrounds. The vertex labeled **top** corresponds to the outermost nesting level of the procedure, and all other vertices are labeled with the numbers of the loops they represent.

regions. The final decompositions for the static decomposition region $\{1, 2.1, 2.2, 2.4\}$ are:

$$\begin{array}{rcccl} bp & \leq & \begin{bmatrix} 1 & 0 \end{bmatrix} \vec{\imath} & < & bp+1 \\ bp & \leq & \begin{bmatrix} 1 & 0 \end{bmatrix} \vec{a} & < & bp+1 \end{array}$$

where b is the block size and p is the processor number. Similarly, for the static decomposition region $\{2.3\}$ we have:

$$\begin{array}{rcccl} bp & \leq & \begin{bmatrix} 0 & 1 \end{bmatrix} \vec{\imath} & < & bp+1 \\ bp & \leq & \begin{bmatrix} 0 & 1 \end{bmatrix} \vec{a} & < & bp+1 \end{array}$$

5.5 Related Work

This section explores previous work on automatic decomposition algorithms. We focus the discussion on compiler algorithms that calculate decompositions for dense matrix scientific codes. In all cases, the statements in a loop nest are the basic unit of computation and the data structures are arrays. Techniques for mapping the computation and/or data of a single loop nest have been presented in the literature[42, 52, 53]. Here we focus on approaches that consider multiple loop nests. All previously proposed algorithms are intraprocedural, so we discuss them in the context of the intraprocedural subset of our algorithm (our interprocedural algorithm is described in the following chapter, Chapter 6).

The proposed algorithms vary widely in terms of the target machines, the domain of input programs they handle, the range of decompositions they generate (for example, whether the decompositions are static or dynamic) and the type of framework they use. We begin in Section 5.5.1 by discussing algorithms that calculate strictly communication-free decompositions. In Section 5.5.2 we describe algorithms that operate only on *data parallel* computations. A computation is considered data parallel if the parallelism comes from performing simultaneous operations across all elements of the arrays, rather than from multiple threads of control. In these approaches communication can only occur between two consecutive loop nests or array operations. In Section 5.5.3 we discuss algorithms that handle more general forms of loop-level parallelism. In these algorithms the parallelism is not strictly element-wise and communication can occur within loop nests. Our algorithm falls into this last category.

For each project we describe the input language and the range of decompositions they generate. Many of the compilers we present generate only data decompositions and rely on the owner-computes rule to generate the computation decomposition, while others generate the computation and data decompositions simultaneously.

We also describe how the target architecture impacts the decomposition algorithms. Some algorithms target SIMD (single instruction, multiple data) machines exclusively, while others target MIMD (multiple instruction, multiple data) machines, or both. SIMD machines execute the same instruction on multiple processors, but each processor operates on a different data stream. In MIMD machines each processor has its own instruction stream and its own data, and they are thus more flexible than SIMD machines. MIMD architectures include both shared and distributed address space machines.

A number of SIMD machines were available in the 1980's and early 1990's, for example, the Thinking Machines CM-1[40] and CM-2 and the MasPar MP-1[17] and MP-2. Many of the early compiler techniques developed for finding decompositions targeted these SIMD machines. Today, MIMD machines are the primary architecture for general-purpose parallel computing. Most of the earlier large-scale MIMD machines were distributed address space machines such as the nCUBE series, the Intel iPSC series, Delta and Paragon, and the IBM SP-1 and SP-2. Thinking Machines moved from a SIMD architecture to a MIMD architecture with the introduction of the CM-5[61]. Large-scale shared address space machines such as Stanford DASH multiprocessor[55], MIT Alewife[2], Kendall Square Research KSR-1[32], SGI Origin and Convex Exemplar started being developed in the early 1990's. Because the remote latencies are so high on distributed address space machines, much of the more recent work on decomposition algorithms has focused on these architectures.

5.5.1 Communication-Free Parallelism

Ramanujam, Huang and Sadayappan at Ohio State University have presented algorithms to calculate communication-free decompositions[41, 63]. The input is a sequential program and the algorithms generate *hyperplane partitions.* A hyperplane partition of a loop nest (or array) is a set of iterations (or array elements) $i^1, i^2, \ldots, i^k$ such that $h^1 i^1 + h^2 i^2 + \ldots + h^k i^k = \alpha$, where $h^1, h^2, \ldots, h^k$ and α are rational numbers. The algorithms are aimed at finding decompositions for MIMD distributed address space machines.

The entire program is modeled by a system of equations. The equations specify the conditions that must be met in order to have a communication-free hyperplane partition. Given the array access functions in the program, the algorithms then solve for the computation and/or data hyperplanes. They present the necessary and sufficient conditions for finding both communication-free single-hyperplane and multiple-hyperplane partitions.

Bau et al.[15] have presented a method for calculating communication-free affine decompositions using the mathematical framework we developed (see Section 3.2). They generate a system of equations that specifies the conditions under which the affine decompositions are communication-free, and then solve for decompositions that have the maximum degree of parallelism (i.e. maximum rank).

5.5.2 Data Parallelism

In this section we describe compilers that perform decomposition analysis on data parallel computations. All the algorithms in this category divide the problem into two phases: *alignment* followed by *distribution*. The alignment phase positions the arrays in the program with respect to each other so that the amount of communication is minimized. In operations with two or more operands, the operands must be aligned, i.e. the corresponding elements of operands must be stored at the same processor. Whenever operands are not aligned, communication is needed to move the data to the necessary processor. The distribution phase then partitions the arrays onto the processors of the parallel machine. The rationale for this two-step approach is that it separates the machine-independent part (alignment) from the machine-dependent part (distribution). Most languages with data decomposition extensions as HPF[48], FORTRAN-D[31] and Vienna FORTRAN[21] are also based on this two-step model.

For data parallel computations, alignment is often the critical issue in determining performance. The algorithms in this category thus place a heavy emphasis on the alignment problem. The alignment is typically split into three separate components: *axis*, *stride* and *offset*. The axis alignment determines the correspondence between different array axes. The stride gives the spacing between array dimensions and the offset gives the displacement of the start of the array dimension.

Compass, Inc. Albert, Knobe, Lukas, Natarajan, Steele and Weiss at Compass, Inc. developed a compiler that calculates decompositions for SIMD machines[4, 45, 46, 47]. The input to the compiler is FORTRAN-77 extended by FORTRAN-90 style array operations, and it targets the Connection Machine CM-2 and MasPar MP-1. The Compass compiler was one of the first compilers that calculated complete alignments. The compiler first finds an alignment of the data onto a virtual processor space. This was sufficient for the CM-2 since it had direct support for virtual processors in firmware[4]. On the MP-1, an additional distribution step was needed to map the virtual processors onto the physical processors of the machine[46].

In the alignment phase, the program is modeled using a *preference graph*. The vertices in the graph represent occurrences of arrays or array sections in the program text, and the edges represent allocation requests (called preferences) that specify the optimal relative alignment between the array occurrences. There are several different types of preferences. An *identity preference* connects the definition of an array with the use of that array. This preference indicates that if the array does not have the same alignment at the two occurrences, communication is required. A *conformance preference* connects array occurrences that are operated on together in an expression or by an assignment. Conformance preferences can connect different arrays, and indicate that communication is required to align the arrays before the operation can execute. Finally an *independence anti-preference* indicates that array dimensions should be allocated across the processors so as to maintain the degree of parallelism in the original input program. Whereas preferences specify that array dimensions should be allocated on the same processor, anti-preferences specify that array dimensions should be distributed across the processors. Preference edges are weighted with the communication cost if the preference is not satisfied, and anti-preference edges are weighted with the execution time penalty for not executing in parallel.

Conflicts between preferences can only occur when there is a cycle in the preference graph. To locate the cycles, the Compass compiler builds a spanning tree using a greedy algorithm that adds edges in order of cost. If adding an edge creates a cycle and the cycle causes a conflict, then the preference associated with the edge is not honored. Axis, stride and offset alignments are calculated from the array accesses in each conflict-free region of the preference graph.

For the MP-1, the Compass compiler's distribution phase finds either a block, cyclic or block-cyclic folding function for each aligned dimension of the virtual processor space[46]. A search-based technique based on the estimated cost of various distributions is used to select the distribution.

Crystal. Li and Chen developed an automatic decomposition algorithm as part of the Crystal project at Yale University[56, 57, 58]. The compiler they developed for Crystal (a functional language) was one of the first compilers to perform decomposition analysis. It targets distributed address space machines, including both SIMD machines as well as MIMD machines (i.e. the Intel iPSC/2 and nCUBE). The Crystal compiler first finds an alignment of the data onto a virtual processor space (referred to as the *index domain*) and then distributes the virtual processor space onto the physical processor space. It finds a single static decomposition for the data across the program region being analyzed.

The Crystal compiler's alignment phase focuses on the problem of axis alignment. They model the problem using a *component affinity graph*(CAG). The vertices in the CAG represent dimensions of arrays (e.g. the graph for a program with three two-dimensional arrays would have six vertices). The vertices are arranged in columns, where each column contains a vertex for each dimension of a single array. For each distinct array access in the program, an edge is generated between two vertices if the two corresponding dimensions are functions of the same index variable. For example, for the statement `x[`i_1`,`i_2`] = y[`i_2`,`i_1`]` there would be one edge connecting the vertex for dimension 1 of x with dimension 2 of y, and another edge connecting dimension 2 of x with dimension 1 of y. Edges generated by the same statement that are incident on the same vertex indicate more than one equally good alignment and are weighted with ϵ (a small number), and all other edges have weights of 1.

Let n be the maximum number of vertices in any column, that is, the maximum dimensionality over all the arrays. The axis alignment problem is to partition the vertices in the CAG into n disjoint subsets, such that no two vertices in the same column are in the same subset. The objective function is to minimize the weight of the edges that have endpoints in different subsets. All vertices in the same subset correspond to array dimensions that are aligned, and edges between subsets represent communication. Li and Chen show their formulation of the axis-alignment problem to be NP-complete[56], and present a practical greedy heuristic.

The Crystal compiler's distribution phase finds either a block or cyclic folding function for each aligned dimension of the virtual processor space. Given a virtual processor space with a fixed size, the compiler enumerates all possible distributions onto the physical processor space. For each distribution, it generates an estimate of the communication cost. The cost estimates are derived by pattern matching array access patterns into communication primitives (e.g. All-to-All Broadcast, Uniform-Shift).

Excalibur. Chatterjee, Gilbert, Schreiber, Sheffler and Teng have presented a set of decomposition algorithms within the context of the Excalibur project[22, 23, 34, 67, 68]. Their algorithms operate on array-based languages such as FORTRAN-90 and target distributed address space machines. Excalibur first calculates the axis and stride alignment of the data onto a virtual processor space (called a template, using HPF terminology), and then calculates the offset alignment in a later pass. After the alignment has been calculated, a separate pass distributes the virtual processor space onto the physical processors.

Excalibur represents the program using an *alignment-distribution graph* (ADG). Vertices in the graph represent computation and edges represent the flow of data. Edges are directed, and an endpoint of an edge (called a *port*) represents an array object with a specified decomposition. Edges thus move an array object from one decomposition to another, and data is re-aligned when the ports of an edge differ. Each edge is weighted with the amount of data it moves, and represents an approximate model of the communication cost.

We first describe Excalibur's axis and stride alignment algorithm presented in [67] (an earlier algorithm was also presented in [23]). The ADG is first transformed into a simpler, alignment-specific graph called the *constraint graph* (CG). A constraint is a mapping from the alignment of one array object to another. The constraint graph is used to represent the communication costs if the constraints imposed by the input program are not satisfied. Each port in the ADG becomes a vertex in the CG, and represents the array objects in the program. An edge with weight $w(u, v)$ in the ADG represents a constraint on the alignment that can be violated with communication cost $w(u, v)$. It is directly translated into an edge with weight $w(u, v)$ in the CG. Vertices in the ADG represent constraints that cannot be violated and become edges with weight ∞ in the CG. For example, the matrix addition statement `x = y + z`, is represented by a vertex in the ADG with two incoming ports for y and z and one outgoing port for x. The corresponding CG has three vertices, one for each port in the ADG. The edges (x, y) and (x, z) are weighted ∞ to represent that after the addition statement executes, x must have the same alignment as y and z. Finally, each edge is labeled with the constraint that specifies the alignment relationship between the two vertices connected by that edge.

Given the CG, the axis/stride alignment problem is to label the vertices with alignments such that the cost of the unsatisfied constraints is minimized. This problem is NP-complete[67] and the authors present heuristic techniques to generate an approximate solution. The algorithm is based on finding a maximal satisfiable subgraph, that is, within the subgraph all constraints are satisfied and there is no communication. Initially the subgraph contains all the vertices in the CG, but none of the edges. At each step, the algorithm adds an edge to the subgraph and then checks if the new subgraph is satisfiable. A number of techniques for contracting the constraint graph into a smaller, equivalent graph are also presented. Since the contraction is inexpensive compared to the alignment itself, contraction can significantly reduce the running time of the compiler. After calculating the axis and stride alignment, Excalibur calculates the offset alignment by reducing the problem to integer programming[23].

Excalibur's distribution algorithm operates on the original ADG for the program. The algorithm first calculates a set of candidate distributions. Each vertex is weighted with the estimated execution time for the computation represented by that vertex, under each of the candidate distributions. The goal is to label each vertex with a distribution. The cost of a given candidate distribution is the sum of the vertex weights for that candidate, plus the sum of edge weights whose endpoints have different distributions. The distribution problem is then to label the vertices with distributions, such that estimated execution time is minimized. Their algorithm uses a divide-and-conquer approach to find dynamic distributions[22]. The program is recursively divided into regions, where each region has a static distribution. The conquer stage merges regions when the cost of the dynamic re-distribution is worse than the static distribution. In a later paper, techniques for contracting the ADG to reduce the size of the distribution problem are described[68].

The algorithms presented in this section model only data parallel computations. While data parallel programs map naturally to SIMD architectures, MIMD machines allow multiple threads of control. MIMD machines have the opportunity to exploit coarser-grain parallelism in addition to data parallelism. For a decomposition algorithm to go beyond data parallelism and take advantage of different choices of parallelism, it must model the trade-offs between the different choices of parallelism and the cost of communication.

5.5.3 Loop Nest Parallelism

In this section, we discuss decomposition algorithms that handle more general forms of loop-level parallelism. The algorithms are still geared towards a data-parallel style of computation, however, the parallelism is not restricted to element-wise computations on arrays. These algorithms weigh the benefit of parallelizing different loops within a loop nest against the communication cost, and optimize to find the best overall execution time.

The D System. Kremer, Kennedy and Bixby at Rice University have developed an automatic decomposition tool[16, 44, 50] as part of the D system[1]. Their tool takes a sequential FORTRAN program as input, and generates data layout annotations for a language such as HPF. The tool first divides the program up into phases, and for each phase generates a list of candidate decompositions. It then estimates the costs of the candidates for each phase and the cost of reorganizing the data between phases, and selects among them.

The candidate selection process first calculates possible alignments, and then finds possible distributions in a separate step. The alignment analysis step only performs axis alignment and uses Li and Chen's component affinity graph representation[56]. Rather than use a greedy heuristic to find an approximate solution as was done in the Crystal compiler, the D System formulates the problem as a 0-1 integer programming problem. Within each phase, they solve for the optimal axis alignment. The axis alignments across different phases are merged if no additional communication results. All the distinct axis alignments found across the different phases become the candidate alignments within each phase. The candidate distributions are generated using either exhaustive search or heuristics. The set of candidate decompositions for each phase is then the cross product of the candidate alignments and the candidate distributions.

The final step is to select a single decomposition for each program phase from among the candidates. The decomposition selection problem is modeled using a *data layout graph* (DLG). The graph has one vertex for each candidate decomposition and the edges represent possible data reorganizations between decompositions. The vertices are weighted with an estimate of the execution time for the code given the corresponding decomposition. The edges are weighted with an estimate of the time to reorganize the data between the two decompositions connected by that edge. Given the DLG, the problem is to select a decomposition for each phase such that the sum of weights of the corresponding vertices and edges is minimized. This problem is NP-complete[50]. Rather than use heuristics to calculate an approximate solution, the D System formulates the problem as a 0-1 integer programming problem and finds an exact solution. In the D System tool, both the axis alignment problem and the decomposition selection problem are solved exactly using 0-1 integer programming. Even though 0-1 integer programming is exponential in the worse case, the rationale is that because the data layout tool is outside of the compiler, it can afford longer running times.

PARADIGM. The PARADIGM compiler, developed at the University of Illinois, performs automatic data distribution starting from sequential FORTRAN-77 programs and generates code for distributed address space machines[13]. Gupta and Banerjee developed the compiler's static decomposition algorithm[35, 36]. Later, Palermo and Banerjee extended the algorithm to also handle dynamic decompositions[60].

PARADIGM's static data decomposition algorithm uses a *constraint-based* approach. Constraints represent desirable properties of the decomposition, and are weighted with a quality measure of the benefit to the overall execution time if the constraint is satisfied. The algorithm performs axis alignment followed by distribution. The distribution analysis is itself subdivided into three steps: choosing a block or cyclic folding function for each dimension, determining the block size for each dimension and selecting the number of processors in each processor dimension (the algorithm considers at most two processor dimensions). Within each of the four steps, any relevant constraints are recorded, then an estimate of the quality is calculated for each constraint and finally a solution is found for that step. For example, to compute the quality measure of an alignment constraint, two communication time estimates are calculated – one is the communication time if the array dimensions are aligned, the other is the communication time if the array dimensions are not aligned. The quality of the constraint is the difference between the two times.

Axis alignment is calculated using Li and Chen's component affinity graph[56]. The edge weights are equal to the quality measure of the alignment constraint generated for the two vertices connected by the edge. PARADIGM's algorithm for finding the axis alignment is based on Li and Chen's greedy algorithm. Next, PARADIGM chooses either a block or cyclic folding function for each aligned dimension. It gathers the constraints that each statement places on the folding function, and calculates the quality measure for each constraint. The folding function that results in the highest total quality is selected. In the next step, the block sizes are calculated using a similar formulation to the axis alignment problem. Finally, the number of processors in each dimension is computed by searching through a fixed set of possible choices.

PARADIGM's dynamic decomposition algorithm builds on top of the static decomposition algorithm. First, the program is divided into a hierarchy of candidate phases. Initially, the entire program is viewed as a single phase and a static distribution is calculated for that phase. Each phase is recursively split into two subphases if the best static decomposition for the single phase has a higher execution time estimate than the two subphases combined. At this point, the cost of reorganizing the data between the phases is not considered. After the program has been divided into phases, the phases are represented in a *phase transition graph*. The vertices in the graph are the program phases, and an edge between two phases is weighted with the data reorganization cost. The final phases, and thus the corresponding decompositions, are calculated by computing the shortest path through the phase transition graph.

The algorithms in this section take the same basic approach as the HPF language and the data-parallel decomposition algorithms from Section 5.5.2. They all divide the decomposition problem into two steps: alignment followed by distribution. A difficulty with this formulation of the problem is that once different choices of parallelism are considered, then alignment alone does not capture the machine-independent aspects of the program. There is now a trade-off between potential communication and the parallelism in the program, independent of the target machine. The parallelism is represented by the distribution, i.e. which array dimensions are allocated across the processors and which are sequential. However, the parallelism in the program impacts the alignment. For example, communication due to mismatches in alignment can be eliminated by modifying the distribution to make those dimensions sequential. The optimal overall decomposition may be one that does not have optimal alignment. Thus, in this case the two-step approach does not accurately model the problem. To avoid this difficulty, some researchers have started calculating alignments and distributions at the same time[33].

5.5.4 Discussion

Our algorithm is similar in scope to the algorithms in Section 5.5.3 that consider the parallelism in general loop nests. However, our approach is most closely related to the communication-free formulations. There are several advantages to this formulation of the decomposition problem. First, we can calculate communication-free regions (or in our case, static decomposition regions that are free of data-reorganization communication) systematically using our mathematical framework. We do not need to rely on selecting a list of candidate decompositions, as do a number of the other approaches. As a result, we avoid the inaccuracies and scalability problems involved in generating a reasonable set of candidates. Second, by representing decompositions directly as affine functions, we avoid the circularity in finding alignments and distributions in two separate steps. Also, using affine functions allows us to generate a wider range of possible decompositions than algorithms that calculate separate axis, stride and offset alignments, e.g. skew decompositions where each processor accesses data along a diagonal of the array.

Unlike the communication-free approaches, we also consider the trade-offs between parallelism and communication. We try to minimize the overall execution time by balancing the communication costs across static decomposition regions with the parallelism within each region.

5.6 Summary

In this chapter, we presented an algorithm for calculating dynamic decompositions. First, we described how each procedure in the program is represented by a communication graph. The loop nests in the procedure translate into vertices in the graph, and the vertices are weighted with estimates of the computation time for the loop nest. There is a weighted edge between two vertices which represents the communication cost if the data decompositions between the vertices do not match.

We then expressed a dynamic decomposition as a partitioning of the vertices of a communication graph into disjoint sets, called static decomposition regions. Static decomposition regions have the property that there is a single, static decomposition in the region of program represented by the vertices in the set. For all loop nests and all arrays in a given static decomposition region, their computation and data decompositions are uniquely determined using the static-decomposition algorithm from the previous chapter. The cost of a given dynamic decomposition on a communication graph is the sum of the edge costs and vertex costs. The cost of an edge is the edge weight if the endpoints of the edge are in different static decomposition regions; otherwise the edge cost is zero. The cost of a vertex is the vertex weight for the decomposition given by the vertex's static decomposition region. The objective of the dynamic decomposition problem is then to partition the communication graph into static decomposition regions such that the cost of the resulting dynamic decomposition for the graph is minimized.

We proved that the dynamic decomposition problem is NP-complete. We then presented a heuristic algorithm for finding dynamic decompositions. We use a greedy approach that tries to eliminate the largest amounts of potential communication from the most frequently executed parts of the program first. The algorithm starts by placing each communication graph vertex in its own static decomposition region. It then tries to merge the vertices into larger and larger static decomposition regions. The analysis is performed on each nesting level in the procedure in bottom-up order, from the innermost level to the outermost. This has the effect of pushing any necessary communication into the outermost loops as much as possible. Within a level, the algorithm tries to merge the vertices connected by edges with the largest edge weights first, in order to eliminate the most expensive communication.

Chapter 6

An Interprocedural Decomposition Algorithm

If the data decompositions of arrays do not match across procedure boundaries, then the program could potentially incur large amounts of communication at every procedure call entry and call return. Any decomposition algorithm that handles realistic programs must be able operate across procedure boundaries. In this chapter we describe the interprocedural version of our decomposition algorithm. The interprocedural algorithm is built on top of the decomposition algorithms from the previous chapters. The problem we solve in this chapter is how to propagate and represent the necessary information across different procedures. We begin in Section 6.1 by describing the cases when decomposition analysis must be performed across procedure boundaries. Section 6.2 then presents our interprocedural algorithm for finding affine decompositions onto the virtual processor space and Section 6.3 describes how to map the virtual processor space onto the physical processor space. Section 6.4 presents a detailed example. In Section 6.5 we describe common programming paradigms that make interprocedural decomposition analysis difficult. Finally in Section 6.6 we discuss how our algorithm interfaces with libraries and user-defined decompositions.

6.1 When is Interprocedural Analysis Needed?

There are two cases when decomposition analysis must be performed across procedure boundaries: parallel loops that contain procedure calls and multiple procedures that access the same array. For example, in the code in Figure 6.1(a), the *doall* loop in procedure `main1` contains a call to `sub1`. The array access in `sub1` is a function of loop index variable i_1, but the access is in a different procedure than the loop itself. To analyze the code, we must translate the access to the formal parameter y in `sub1` into an access of the actual parameter x in `main1`. We find that every access `y[`i_2`]` corresponds to the same memory location as `x[2*`i_1`,42]`, and can then calculate the computation decomposition for the loop and the data decomposition for the array.

The code in Figure 6.1(b) shows an example where the same array is accessed in distinct loop nests in two different procedures. If we were to calculate the decompositions for each procedure separately, the linear data decomposition for formal parameter y in `sub2` would be $D_y = \begin{bmatrix} 1 & 0 \end{bmatrix}$ with $\mathcal{N}(D_y) = \text{span}\,\{(0,1)\}$. In `main2`, for actual parameter x, $D_x = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ with $\mathcal{N}(D_x) = \emptyset$. For each call to `sub2`, the array would have to be reorganized since the data decompositions of the actual and formal parameter do not match. To find a consistent decomposition across both procedures, the constraint on array y in `sub2`, $\mathcal{N}(D_y) = \text{span}\,\{(0,1)\}$, must be translated into a constraint on x in `main2`, $\mathcal{N}(D_x) = \text{span}\,\{(0,1)\}$.

The cases that require decomposition analysis across procedures occur commonly in practice. We thus need a mechanism to efficiently propagate information across procedure boundaries. For decomposition analysis, this involves translating both array accesses and data decompositions between corresponding variables in the different procedures.

A simple solution would be to eliminate the procedure boundaries altogether by performing *inline substitution*. Inline substitution replaces each procedure call by a copy of the callee procedure. We could then run the intraprocedural decomposition algorithm from the previous chapter on the single resulting procedure. Unfortunately, this approach is not practical for large programs. Inline substitution can cause the code to grow to an unmanageable size. Also, since each procedure is re-analyzed every time it is called, the analysis

```
program main1
  real x[N,N]
  /* Loop Nest 1 */
  for $i_1$ = 1 to N/2 do        /* doall */
    call sub1(x[1,42], 2*$i_1$)    /* pass column 42 of x */
end

subroutine sub1(y, $i_2$)
  real y[N]
  integer $i_2$
  y[$i_2$] = $f_1$($i_2$)
end
```

(a)

```
program main2
  real x[N,N]
  /* Loop Nest 2 */
  for $i_1$ = 1 to N do          /* doall */
    for $i_2$ = 1 to N do        /* doall */
      x[$i_1$,$i_2$] = $f_2$($i_1$,$i_2$)
  call sub2(x)
end

subroutine sub2(y)
  real y[N,N]
  /* Loop Nest 3 */
  for $i_1$ = 1 to N do          /* doall */
    for $i_2$ = 2 to N-1 do
      y[$i_1$,$i_2$] = y[$i_1$,$i_2$] + y[$i_1$,$i_2$+1] + y[$i_1$,$i_2$-1]
end
```

(b)

Figure 6.1: Code examples used to illustrate interprocedural decomposition analysis: (a) parallel loop that contains a procedure call, and (b) multiple procedures that accesses the same data.

times can quickly become unacceptable. We thus use interprocedural analysis to calculate decompositions across procedure boundaries. We summarize the necessary information for every procedure, and then map that information across the procedure calls. During the analysis, there is a single copy of each procedure, and the procedures do not need to be re-analyzed for each call.

6.2 Finding Virtual Processor Mappings

This section describes the interprocedural algorithm for finding affine decompositions onto the virtual processor space. We start with the *call graph* for the program. The call graph $G = (V, E)$ has a vertex $v \in V$ for each procedure, and a directed edge $(u, v) \in E$ for each call in procedure u to procedure v. Given the call graph, the interprocedural decomposition algorithm visits each procedure twice, once in a bottom-up traversal of the call graph and once in a top-down traversal of the call graph.

First we run the intraprocedural decomposition algorithm from the previous chapter on the leaf procedures in the call graph. The bottom-up pass propagates a summary of the array accesses, and any constraints found on the arrays, up from the callee procedure into the caller procedure. Then we run the intraprocedural decomposition analysis on the caller procedures, and continue up the call graph. By the time the algorithm reaches the main procedure, all the constraints have been collected and we calculate the final linear decompositions. The top-down pass then pushes the final linear decompositions down from the caller procedures into the callee procedures, and calculates the offsets to form the complete affine decompositions.

In this algorithm, there are two types of information that flow between procedures: linear decompositions and array access functions. The bottom-up pass must map both linear decompositions and array access functions from callee procedures into the calling context of the caller procedure. The top-down pass must map the final linear decompositions from the caller back down into the callee. In our framework, both affine decompositions and array access functions are represented as affine functions (linear decompositions are just affine decompositions with zero offsets). Thus a key component of our interprocedural decomposition algorithm is translating affine functions in a callee procedure into equivalent affine functions in the caller procedure, and vice-versa. First, we describe our method for mapping affine functions across calls in Section 6.2.1. Then, we present the details of the bottom-up traversal in Section 6.2.2 and the top-down traversal in Section 6.2.3.

6.2.1 Mapping Affine Functions Across Calls

Let $\vec{g}$ be an affine function (either an array access or an affine decomposition) that involves variables that are visible in procedure q. Also, let there be an edge in the call graph between q and procedure r representing a call site s. To map $\vec{g}$ from q into r, we must translate the variables in q into equivalent expressions of variables visible in r at the call site s. The translation is not always straightforward because the array accesses and affine decompositions always involve an array variable. The relationship between memory locations for arrays in different procedures can be complex if the array is *reshaped* across the call. Array reshapes occur when the number or size of array dimensions is altered at a call site, e.g. passing a slice of an array into a procedure.

We generate a system of linear inequalities that describes the correspondence between variables visible in q and variables visible in r at the call site s. Let y refer to the array variable in q and let x refer to the corresponding array variable in r (e.g. y is a formal parameter and x is the actual parameter). Then, in addition to the original affine function, the system of linear inequalities includes the conditions that describe the relationship between memory locations of x in r and memory locations of y in q. Also included in the system of inequalities are the bounds of x, the bounds of y and any additional known facts about variables from the call site s. We project away the variables visible in q and replace them with variables visible in r. The projection step is based on Fourier-Motzkin elimination that has been enhanced for the integer domain[6, 37, 38].

For example, the affine array access function for the write to array y in `sub1` of Figure 6.1(a) is $F^1_{y1}(i_2) = \left[\begin{array}{c} 1 \end{array}\right] i_2$. If we let $\vec{a_y} = (a_{y_1})$ represent an index into array y, the corresponding equation is

$$i_2 \;=\; a_{y_1}$$

Letting $\vec{a_x} = (a_{x_1}, a_{x_2})$ represent an index into array x in `main1`, the complete set of inequalities is:

$$\begin{array}{rcll} i_2 & = & a_{y_1} & \\ i_2 & = & 2i_1 & \\ 1 & \leq & a_{y_1} & \leq N \\ 1 & \leq & a_{x_1} & \leq N \\ 1 & \leq & a_{x_2} & \leq N \end{array}$$

$$a_{y_1} - 1 = N * (a_{x_2} - 42) + (a_{x_1} - 1)$$

The last equation above is the one that describes the correspondence between memory locations of x and y. After eliminating a_{y_1} and i_2, we have the following linear inequality in `main1`:

$$\begin{array}{rcll} 42 & = & a_{x_2} & \\ 2i_1 & = & a_{x_1} & \\ 1 & \leq & a_{x_1} & \leq N \\ 1 & \leq & a_{x_2} & \leq N \end{array}$$

and the resulting affine array access function for x is $F^1_{x1}(i_1) = \begin{bmatrix} 2 \\ 0 \end{bmatrix} i_1 + \begin{bmatrix} 0 \\ 42 \end{bmatrix}$.

For an example of mapping an affine data decomposition across a call, let the affine data decomposition for array y in `sub2` of Figure 6.1(b) be $D_y(\vec{a_y}) = \begin{bmatrix} 1 & 0 \end{bmatrix} \vec{a_y}$. Letting $\vec{a_y} = (a_{y_1}, a_{y_2})$ represent an index into array y, the corresponding equation is

$$p_1 = a_{y_1}$$

where $\vec{p} = (p_1)$ represents an index into a one-dimensional virtual processor space. If we let $\vec{a_x} = (a_{x_1}, a_{x_2})$ represent an index into array x in `main2`, the complete set of inequalities is:

$$\begin{array}{rcll} p_1 & = & a_{y_1} & \\ 1 & \leq & a_{x_1} & \leq N \\ 1 & \leq & a_{x_2} & \leq N \\ 1 & \leq & a_{y_1} & \leq N \\ 1 & \leq & a_{y_2} & \leq N \end{array}$$

$$N * (a_{x_1} - 1) + (a_{x_2} - 1) \quad = \quad N * (a_{y_1} - 1) + (a_{y_2} - 1)$$

After eliminating the subscripts of y, (a_{y_1}, a_{y_2}), the linear inequality for x in `main2` is:

$$\begin{array}{rcccl} p_1 & = & a_{x_1} & & \\ 1 & \leq & a_{x_1} & \leq & N \\ 1 & \leq & a_{x_2} & \leq & N \end{array}$$

and the resulting affine data decomposition is $D_x(\vec{a_x}) = \left[\begin{array}{cc} 1 & 0 \end{array} \right] \vec{a_x}$.

In order for the mapping of an affine function from procedure q into procedure r to succeed, the resulting function in r must also be affine. Some array reshapes can cause the mapping to fail (see Section 6.5.2 for more detail). We also cannot handle unknown array bounds, or symbolic array bounds except in the outermost dimension; otherwise the resulting function is not affine (see Section 6.5.3).

If the mapping of any dimension of an affine function across a call fails, then the decomposition algorithm ignores that dimension. If the mapping of an array access fails, the consequence of ignoring the access is that communication may be needed when that access is executed. If the mapping of an affine decomposition fails, then the decomposition analysis runs as if the array is not accessed within the procedure, and the program may incur communication upon procedure entry and exit. In the remainder of this discussion, we assume that all affine functions are successfully mapped across the calls, and that each formal parameter or global variable in the callee q maps to a single variable in the caller r. When all procedures access the variables in a common block consistently, then we split up the variables into distinct global variables. Common blocks that cannot be split, as well as unions and equivalences, are represented in terms of offsets from the base memory address.

Other compiler techniques have been developed that can more accurately map certain types of information across procedures in the presence of array reshapes. Amarasinghe has developed an algorithm for propagating summaries of array accesses across procedure boundaries when the arrays are reshaped[6].

6.2.2 Bottom-up Traversal

The bottom-up decomposition analysis is built on top of the intraprocedural decomposition algorithm, `Dynamic_Decomps`, from the previous chapter. The interprocedural analysis starts by running `Dynamic_Decomps` to find linear decompositions at the leaf procedures of the call graph. The bottom-up traversal performs three main steps for each caller procedure r. First, information about the data decompositions for each callee procedure q is propagated into r. Next, array accesses from procedure q are collected into r. Finally, the `Dynamic_Decomps` algorithm from the previous chapter is run on the procedure r. These three steps are described in more detail in the following subsections.

6.2.2.1 Propagating Decompositions into Calling Procedures

In this section we describe how to propagate information about a callee procedure's decompositions into the caller procedures. The intraprocedural decomposition algorithm, `Dynamic_Decomps`, has already been run on the callee procedures. The loop nests in the callee have been partitioned into static decomposition regions and the linear decompositions onto the virtual processor space have been calculated.

Within a caller procedure r, we represent information about a call to procedure q by treating the call as if it were just another loop nest in r. We add vertices to r's communication graph for each call to q outside of a parallel loop nest (calls inside parallel loops are already represented by the communication graph vertex for that loop nest). Any constraints on the decompositions from the body of q become constraints on the call vertex in r. This representation lets us find dynamic decompositions involving the call vertex that minimize data-reorganization communication between the call and the surrounding loop nests in r. We refer to a communication graph vertex representing a call to procedure q by v_q.

To create a vertex v_q in r's communication graph, we must determine which arrays are accessed within the code represented by that vertex, along with the corresponding array access functions. We set the vertex v_q to access all actual parameters at the call site and all global arrays accessed in q. We then translate the linear data decompositions for the arrays in v_q across the call from q into the context of r. Let D_x be the linear data decomposition of array x mapped into r. Then the constraints for array x in v_q are initialized to $\mathcal{N}(D_x)$.

In this way, all constraints found in the callee procedures are propagated up in the caller procedure.

Next we calculate the array access functions for the arrays in the call vertex. Let $\mathcal{P}$ be the virtual processor space for the callee's static decomposition region. We treat $\mathcal{P}$ as the iteration space in the call vertex, i.e. $\mathcal{I} = \mathcal{P}$. The array access function for the array variable x in the call vertex is given by solving for F_{xq} in the equation $D_x F_{xq} = I$, where I is the identity matrix. The linear data decomposition functions $D : \mathcal{A} \rightarrow \mathcal{P}$ thus become the array access functions $F : \mathcal{I} \rightarrow \mathcal{A}$ in the call vertex, where $\mathcal{A}$ is the array space for each array. Using the linear data decompositions as access functions preserves the relationship between the linear data decompositions of different arrays. This information was provided by the original array access functions in the intraprocedural version of the algorithm. After calculating the array access functions, the vertex and edge weights involving the call vertex are computed in the same way as all other vertices.

The complete algorithm for creating call vertices, `Create_Call_Vertices`, is shown in Figure 6.2. The algorithm starts by looking at the static decomposition regions for the callee q. For simplicity of presentation we assume that there is a single static decomposition region for entire procedure q; below we describe what happens if this is not the case. The algorithm creates a separate vertex in the caller's communication graph for each connected component of the callee's interference graph. The decompositions for the data and computation within each connected component are relative to one another, and the decompositions for data and computation in separate connected components are completely independent. Thus by creating distinct vertices in the communication graph, we ensure that the decompositions we find for each connected component remain independent.

A difficulty arises, however, whenever aliased arrays are in separate connected components. For example, consider the following code:

```
program main1
  real x[N]
  call sub1(x, x)
end
```

```
algorithm Create_Call_Vertices
            ($G_c$ : communication_graph,   /* caller's communication graph */
            $q$ : procedure,   /* callee */
            $r$ : procedure,   /* caller */
            call : call_site)
  return ($G_c$ : communication_graph)

  $G_s$, $G_s'$ : interference_graph;   /* $G_s$ = $(V_c, V_d, E)$ */
  $D_x$, $D_y$ : matrix;
  mapped : array of boolean;
  $v_q$, $v_q'$ : communication_graph_vertex;

  if call within parallel loop nest then return $G_c$;

  $G_s$ = static decomposition region for $q$ in $G_c$;
  foreach connected component $G_s'$ of $G_s$ do
    $v_q$ = new communication_graph_vertex;
    Add $v_q$ to $G_c$;

    foreach $v_y \in V_d'$, where $y$ is an actual parameter or global variable do
      $D_y$ = linear data decomposition for $v_y$;
      $D_x$ = map $D_y$ up from call from $y$ in $q$ to $x$ in $r$;
      add $x$ to list of arrays in $v_q$;
      initialize constraints for $x$ to $\mathcal{N}(D_x)$;
      solve for $F_{xq}$ in $D_x F_{xq} = I$ and add to list of array access functions for $x$ in $v_q$;

      if mapped[$x$] in call then
        $v_q'$ = communication graph vertex containing $x$ in $G_c$;
        $v_q$ = merge $v_q$ and $v_q'$;
      end if;
      mapped[$x$] = true;
    end foreach;
  end foreach;
  return $G_c$;
end algorithm;
```

Figure 6.2: Algorithm for creating call vertices in the caller's communication graph.

```
subroutine sub1(y, z)
  real y[N], z[N]
  ...
end
```

In FORTRAN, this code is legal only if arrays y and z in `sub1` are read-only. It is possible that the arrays y and z are in separate connected components of the interference graph representing the static decomposition region for the callee `sub1`. However, both y and z map to the same array x in the caller `main1`. The data decompositions in the connected components for y and z are not really independent as the variables are aliased. In this case, we simply merge the communication graph vertices corresponding to the two connected components for y and z. The constraints on x from both y and z are summed and the array access functions are combined.

After creating the call vertices, the algorithm initializes the constraints for the arrays and generates the list of array accesses. To create the array access function for array x, we solve for F_{xq} in the equation $D_x F_{xq} = I$, where D_x is an $n \times m$ matrix and I is the $n \times n$ identity matrix. To guarantee that a solution exists, there are three cases to consider:

1. The system $D_x F_{xq} = I$ has a single solution for F_{xq}.

2. The system $D_x F_{xq} = I$ is under-constrained and has infinitely many solutions for F_{xq}. This occurs when the rank of the $n \times m$ matrix D_x is less than the number of columns, $\text{rank}(D_x) < m$. In this case there are free variables in the solution which are completely arbitrary. For example, given the linear data decomposition $D_x = \begin{bmatrix} 0 & 1 \end{bmatrix}$, we have $\text{rank}(D_x) = 1$ and $m = 2$. This gives the equation

$$\begin{aligned} D_x F_{xq} &= I \\ \begin{bmatrix} 0 & 1 \end{bmatrix} F_{xq} &= \begin{bmatrix} 1 \end{bmatrix} \\ F_{xq} &= \begin{bmatrix} f_{11} \\ 1 \end{bmatrix} \end{aligned}$$

where f_{11} can be any arbitrary value. Informally, the first dimension of array x is already local to a single processor in the linear data decomposition D_x. Thus, it does not matter what the array access function for the first dimension of x is, since all accesses to that dimension go to the same processor (the fact that the first dimension of x is local is already represented in the constraints, $\mathcal{N}(D_x) = \text{span}\,\{(1,0)\}$). The array access function we derive in this example, F_{xq}, is equivalent to the array access in the following loop nest:

```
for $i_1$ = 1 to N do
  x[$f_{11}$*$i_1$,$i_1$] = ...
```

3. The system $D_x F_{xq} = I$ is over-constrained and has no solution for F_{xq}. This could happen if the rank of the $n \times m$ matrix D_x is less than the number of rows, rank $(D_x) < n$. Specifically, from linear algebra we know that the system only has a solution when range $(I) \subseteq$ range (D_x)[72]. Since I is the $n \times n$ identity matrix, range $(I) = \mathcal{R}^n$, the full n-dimensional space. Since we know that rank $(D_x) < n$, then range $(I) \supset$ range (D_x). In this case, we add the $n -$ rank (D_x) basis vectors to the columns of D_x to give $D_x{}'$ with range $(D_x{}') = \mathcal{R}^n$. At this point, since rank $(D_x{}') = n$ (i.e. $D_x{}'$ is full row-rank) the matrix $D_x{}'$ has a right-inverse[72], thus we can solve for F_{xq}.

   For example, consider the linear data decomposition $D_x = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$. We have rank $(D_x) = 2$ and $n = 3$. This gives the equation

$$\begin{aligned} D_x F_{xq} &= I \\ \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} F_{xq} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

We then add the column vector $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$, to D_x, giving $D_x{}' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$, and

$$\begin{aligned} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} F_{xq} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ F_{xq} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \end{aligned}$$

Informally, what happens in this case is that the array is not mapped into all dimensions of the virtual processor space. In the above example, the original linear data decomposition D_x maps the two-dimensional array x into the first and third dimensions of a three-dimensional virtual processor space. Adding the additional column to create $D_x{}'$ has the effect of expanding x into three dimensions and then mapping that third dimension into the second dimension of the three-dimensional virtual processor space. The resulting array access function F_{xq} in this example is equivalent to the array access function for the following code:

```
for $i_1$ = 1 to N do
  for $i_2$ = 1 to N do
    for $i_3$ = 1 to N do
      x[$i_1$,$i_2$,$i_3$] = ...
```

For an example of how call vertices are created, consider the code from Figure 6.1(b). We add a vertex to `main2`'s communication graph for the call to `sub2`. Since the actual parameter x in `main2` corresponds to the formal parameter y in `sub2`, the new vertex is set to access array x. If the linear data decomposition for array y in `sub2` is $D_y = \begin{bmatrix} 1 & 0 \end{bmatrix}$, then mapping the decomposition into `main2` gives $D_x = \begin{bmatrix} 1 & 0 \end{bmatrix}$. The array access function for x at the call vertex is then $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, and the constraints for array x are initialized to $\mathcal{N}(D_x) = \text{span}\,\{(0,1)\}$.

The above discussion assumed that there was a single static decomposition region for each callee procedure. When a callee procedure has multiple static decomposition regions, the arrays have a different linear data decomposition for each region. Which of these different linear data decompositions do we use to create the call vertices? Only the decompositions at procedure entry and exit in q are relevant to the caller r. We want the decompositions in the code executed before the call to q to match the decompositions upon entry to q, and the decompositions for code executed after the call to q to match the decompositions at exit from q. To simplify finding the data decompositions for each array at procedure entry and exit in the callee, we modify the procedures so that they have a single entry and a single exit point. Then we add two extra vertices to the communication graph, one at the procedure entry point and one at the exit point. These two vertices are set to access all arrays that are accessed within the procedure. The data decompositions found at the entry and exit vertices represent the data decompositions at procedure entry and exit, respectively.

In the case where there are multiple static decomposition regions in a callee q, we create two sets of communication graph vertices for the call to q in the caller r. One set of vertices corresponds to the region at entry to procedure q and one set corresponds to the region at exit from procedure q. We use the static decomposition regions for q's entry and exit vertices, respectively. In r's communication graph, we connect the entry vertices for the call to q to the exit vertices by edges with weight 0. This ensures that the algorithm will not try to merge these vertices into the same static decomposition region when analyzing r.

6.2.2.2 Propagating Array Accesses into Calling Procedures

For any calls within a parallel loop nest, the array access functions within the callee are mapped across the call and are added to the list of array access functions for that loop nest. The algorithm for summarizing array accesses, `Gather_Call_Accesses`, is shown in Figure 6.3. The algorithm takes all the array accesses in procedure q and maps them into the caller r. If the array access is only a function of variables local to q, then it is discarded.

Otherwise, the algorithm adds the access to the list of accesses in the caller. If the call is within a parallel loop nest, then the algorithm adds the accesses to the list of accesses for the parallel loop nest.

For the example code from Figure 6.1(a), the algorithm maps the array access `y[`i_2`]` in `sub1` into an access of `x[2*`i_1`,42]` in `main1`. It then adds this access to list of array accesses for loop nest 1.

```
algorithm Gather_Call_Accesses
            ($G_c$ : communication_graph,   /* caller's communication graph */
            $q$ : procedure,   /* callee */
            $r$ : procedure,   /* caller */
            call : call_site)
   return
            ($G_c$ : communication_graph)

   $F$, $F'$ : matrix;
   $v_q$ : communication_graph_vertex;

   foreach array access $F$ in $q$ do
      $F'$ = map $F$ across call;
      add $F'$ to list of accesses in $r$;
      if call within parallel loop nest then
         $v_q$ = communication graph vertex for loop containing call in $G_c$;
         add $F'$ to list of array access functions in $v_q$;
      end if;
   end foreach;

   return $G_c$;
end algorithm;
```

Figure 6.3: Algorithm for gathering the array accesses in a callee procedure.

```
algorithm Bottom_Up_Traversal
            ($G$ : call_graph)  /* $G$ = $(V, E)$ */

   $G_c$ : communication_graph;
   $u$, $v$ : procedure;

   foreach $v \in V$ in bottom-up order do
      $G_c$ = build communication graph for $v$;

      foreach call-site call $\in$ succ($v$) do
         $u$ = callee procedure at call-site call;
         $G_c$ = Create_Call_Vertices($G_c$, $u$, $v$, call);
         $G_c$ = Gather_Call_Accesses($G_c$, $u$, $v$, call);
      end foreach;

      Dynamic_Decomps($G_c$);
   end foreach;
end algorithm;
```

Figure 6.4: Algorithm for propagating decomposition constraints up the call graph representing the program.

6.2.2.3 Moving up the Call Graph

The driver algorithm for the bottom-up traversal is shown in Figure 6.4. As the algorithm moves up the call graph, constraints on decompositions from callee procedures are represented in the call vertices of the caller's communication graph. Array accesses in the callee procedures are also passed up into the caller procedure, in case there are any parallel loops that contain calls. The algorithm then runs `Dynamic_Decomps` (Figure 5.5) on the caller's communication graph. The vertices, including the vertices representing procedure calls, are partitioned into static decomposition regions and the linear decompositions are calculated. The linear decompositions and array accesses in the caller procedure are then passed up the call graph to its callers, and so on. By the time the algorithm reaches the main program, all the constraints for the entire program have been collected and the final linear decompositions are calculated. Since the information for each callee procedure is summarized in its caller's communication graphs, each procedure is only analyzed once during the bottom-up pass through the call graph.

For the example from Figure 6.1(b), the communication graph for procedure `main2` will contain one vertex for the call to `sub2` and one vertex for loop nest 2. The constraints for array x at the call vertex are $\mathcal{N}(D_x) = \text{span}\{(0,1)\}$ and at the loop nest vertex $\mathcal{N}(D_x) = \emptyset$. Running `Dynamic_Decomps` merges these two vertices into the same static decomposition region, resulting in $\mathcal{N}(D_x) = \text{span}\{(0,1)\}$ at both vertices. The final linear data decomposition for x is then $D_x = \begin{bmatrix} 1 & 0 \end{bmatrix}$, and the linear computation decomposition for both the loop nest and the call is also $\begin{bmatrix} 1 & 0 \end{bmatrix}$.

6.2.3 Top-down Traversal

After the linear data decompositions have been found at the main procedure, the algorithm traverses the call graph once more in top-down order. The linear data decompositions for the global variables and actual parameters found in each caller procedure are mapped down into all the callee procedures. Figure 6.5 shows the algorithm `Record_Call_Vertices` which takes the data decompositions in caller procedures and applies them to callee procedures.

The `Record_Call_Vertices` algorithm starts with a call vertex v_q in the caller's communication graph. It takes the linear data decompositions for each array at v_q in the caller r and translates them across the call into data decompositions in the callee q. This gives us the linear data decompositions for the formal parameters and global arrays in q.

The next step is to calculate the linear data decompositions for any local arrays in q, and the linear computation decompositions for the loop nests in q. The algorithm runs the `Propagate_Nullspaces` algorithm from Figure 4.4 to update the nullspaces of the linear decompositions for the local arrays and loop nests, and uses `Calc_Matrices` from Figure 4.6 to calculate the linear decompositions matrices. Finally, the algorithm calculates the offsets for all arrays and loop nests in the procedure to give the complete affine decompositions.

For the on-going example from Figure 6.1(b), The linear data decomposition for x in `main2` $D_x = \begin{bmatrix} 1 & 0 \end{bmatrix}$, is mapped back down into `sub2` to give $D_y = \begin{bmatrix} 1 & 0 \end{bmatrix}$. This

```
algorithm Record_Call_Vertices
            ($G_{cr}$ : communication_graph,  /* caller's communication graph */
            $G_{cq}$ : communication_graph,  /* callee's communication graph */
            $r$ : procedure,  /* caller */
            $q$ : procedure,  /* callee */
            call : call_site)
   return
            ($G_{cq}$ : communication_graph)

   $G_s$, $G_s{}'$ : interference_graph;  /* $G_s = (V_c, V_d, E)$ */
   Γ, Δ : set of vector_space;
   $D_x$, $D_y$ : matrix;
   $v_q$ : communication_graph_vertex;
        /* $v_q$ contains a list of arrays and corresponding array access functions */

   if call within parallel loop nest then return $G_{cq}$;

   $G_s$ = static decomposition region for $q$ in $G_{cq}$;
   foreach connected component $G_s{}'$ of $G_s$ do
      $v_q$ = communication graph vertex for $G_s{}'$ in $G_{cr}$;

      foreach array $x$ in $v_q$, where $x$ is an actual parameter or global variable do
         $D_x$ = linear data decomposition for $x$;
         $D_y$ = map $D_x$ down through call from $x$ in $r$ to $y$ in $q$;
         set linear data decomposition for $v_y \in V_d$;
      end foreach;
   end foreach;

   (Γ, Δ) = current nullspaces for $G'_s$;
   Propagate_Nullspaces($G_s$, Γ, Δ);  /* See Figure 4.4 */
   Calc_Matrices($G'_s$, Γ, Δ);  /* See Figure 4.6 */
   calculate the offset decompositions for $G_{cq}$;

   return $G_{cq}$;
end algorithm;
```

Figure 6.5: Algorithm for recording linear decompositions in the callee's communication graph from call vertices in the caller's communication graph.

results in a linear computation decomposition for loop nest 3 of $C_3 = \begin{bmatrix} 1 & 0 \end{bmatrix}$.

The complete algorithm for the top-down traversal is shown in Figure 6.6. If there are multiple paths in the call graph to a callee procedure from different callers, then it is possible for there to be multiple conflicting decompositions required in the callee procedure. In this case, the compiler can *clone* the callee procedure to create a new copy of the procedure for each different decomposition.

6.3 Finding Physical Processor Mappings

We find the physical processor mappings in the interprocedural case using the same technique as for the intraprocedural case described in Section 4.2. The physical processor mapping phase examines each loop nest (in all procedures) to gather any constraints that loop nest may have on the folding function. The constraints are then combined to find the final folding function. The complete algorithm for finding interprocedural decompositions is shown in Figure 6.7.

6.4 An Interprocedural Decomposition Example

In this section we illustrate how the interprocedural decomposition algorithm works for the code in Figure 6.8. Figure 6.9 shows the call graph and the corresponding communication graphs for each procedure in the sample code. For simplicity of presentation, only the vertices that correspond to either loop nests or procedures calls are shown in the communication graphs; the entry and exit vertices, and the hierarchy structure are not shown.

In the bottom-up traversal, the algorithm first finds the linear decompositions for the leaf procedure `sub3`. The array access functions for the loop nests inside this procedure are:

$$F^1_{x5} = F^1_{y5} = F^1_{z5} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```
algorithm Top_Down_Traversal
            ($G$ : call_graph)  /* $G = (V, E)$ */

   $G_{cu}$, $G_{cv}$, $G_{cv}'$: communication_graph;
   $u$, $v$, $v'$ : procedure;
   recorded : boolean;

   foreach $v \in V$ in top-down order do
      recorded = false;
      $G_{cv}$ = communication graph for $v$;

      foreach call-site call $\in$ pred($v$) do
         $u$ = caller procedure at call-site call;
         $G_{cu}$ = communication graph for $u$;
         $G_{cv}'$ = Record_Call_Vertices($G_{cu}$, $G_{cv}$, $u$, $v$, call);

         if recorded and not compatible($G_{cv}$, $G_{cv}'$)
            $v'$ = clone $v$;
            set communication graph for $v'$ to $G_{cv}'$;
         else
            $G_{cv}$ = $G_{cv}'$;
         end if;
         recorded = true;
      end foreach;
   end foreach;
end algorithm;
```

Figure 6.6: Algorithm for propagating linear data decompositions down the call graph representing the program.

```
**algorithm** IPA_Decomps_Driver
            ($G$ : call_graph)  /* $G$ = $(V, E)$ */

   Bottom_Up_Traversal($G$);
   Top_Down_Traversal($G$);
   calculate the virtual-to-physical mapping;
**end algorithm**;
```

Figure 6.7: Interprocedural algorithm for finding decompositions.

`Dynamic_Decomps` find the following linear decompositions for `sub3`:

$$C_5 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad D_x = D_y = D_z = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Next the algorithm examines procedure `sub2`, which has a single call to `sub3`. Vertex 6 is created in `sub2`'s communication graph to represent the call to `sub3`, and the linear data decompositions for arrays x, y and z are then mapped into `sub2`. This results in the following array access functions for procedure `sub2`:

$$F^1_{x4} = F^1_{y4} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad F^2_{x4} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$F^1_{x6} = F^1_{y6} = F^1_{z6} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

which gives the following decompositions for `sub2`:

$$C_4 = C_6 = \begin{bmatrix} 0 & 1 \end{bmatrix}, \quad D_x = D_y = D_z = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

Similarly, the algorithm creates the vertex 7 in `sub1`'s communication graph to represent

```
program main
  real x[N,N], y[N,N], z[N,N]
  for $i_1$ = 1 to N do          /* doall */          /* Loop nest 1 */
    for $i_2$ = 1 to N do        /* doall */
      x[$i_1$,$i_2$] = y[$i_1$,$i_2$] = z[$i_1$,$i_2$] = $f_1$($i_1$,$i_2$)
    end for

  for $i_1$ = 1 to N do          /* doall */          /* Loop nest 2 */
    for $i_2$ = 1 to N do        /* doall */
      x[$i_1$,$i_2$] = x[$i_1$,$i_2$] + z[$i_1$,$i_2$]
      y[$i_1$,$i_2$] = y[$i_1$,$i_2$] + z[$i_1$,$i_2$]
    end for
  call sub1(x, y, z)
  call sub2(x, y, z)
end

subroutine sub1(x, y, z)
  real x[N, N], y[N,N], z[N,N]
  for $i_1$ = 1 to N do          /* doall */          /* Loop nest 3 */
    for $i_2$ = 1 to N do        /* doall */
      x[$i_1$,$i_2$] = y[$i_1$,$i_2$] + y[$i_1$,N-$i_2$+1]
  call sub3(x, y, z)
end

subroutine sub2(x, y, z)
  real x[N,N], y[N,N], z[N,N]
  for $i_1$ = 1 to N do          /* doall */          /* Loop nest 4 */
    for $i_2$ = 1 to N do        /* doall */
      y[$i_1$,$i_2$] = x[$i_1$,$i_2$] + x[N-$i_1$+1,$i_2$]
  call sub3(x, y, z)
end

subroutine sub3(x, y, z)
  real x[N,N], y[N,N], z[N,N]
  for $i_1$ = 1 to N do          /* doall */          /* Loop nest 5 */
    for $i_2$ = 1 to N do        /* doall */
      z[$i_1$,$i_2$] = z[$i_1$,$i_2$] + x[$i_1$,$i_2$] + y[$i_1$,$i_2$]
end
```

Figure 6.8: Sample code with multiple procedures.

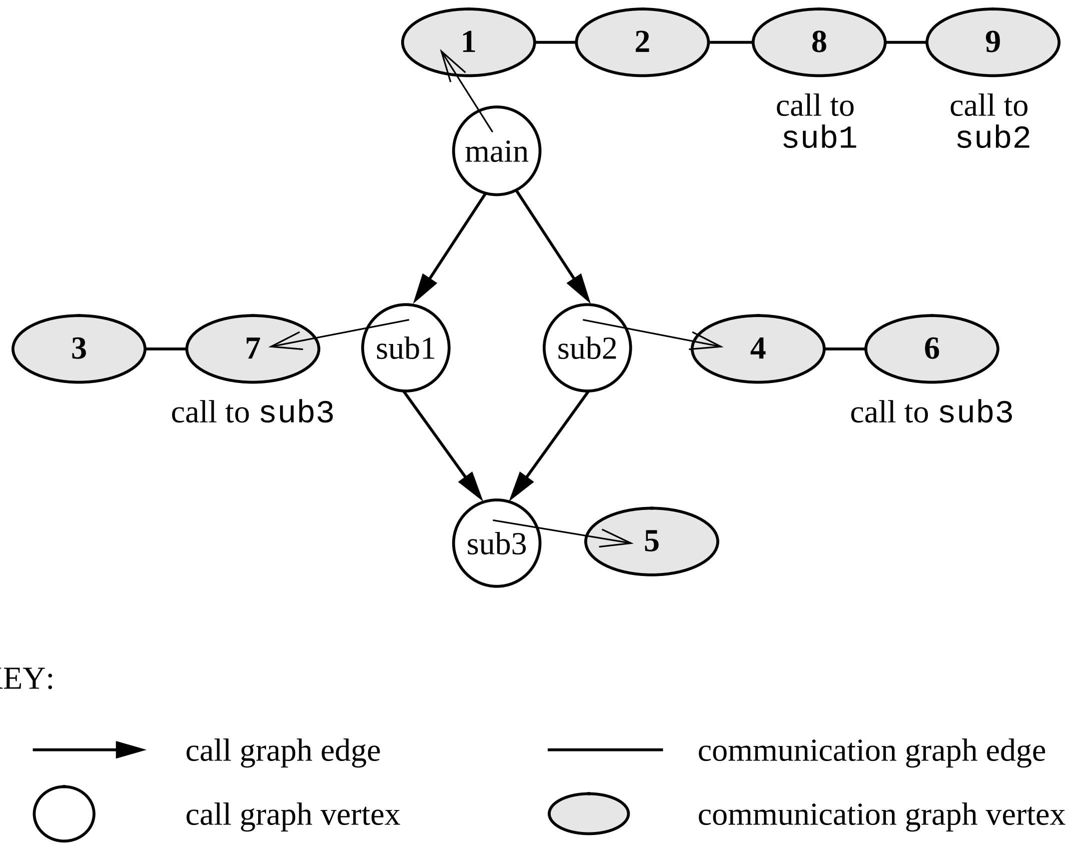


Figure 6.9: Call graph and the corresponding communication graphs for the code in Figure 6.8 during the bottom-up traversal.

the call to `sub3`. The array access functions in `sub1` are:

$$F^1_{x3} = F^1_{y3} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad F^2_{y3} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$F^1_{x7} = F^1_{y7} = F^1_{z7} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and the resulting decompositions for procedure `sub1` are as follows:

$$C_3 = C_7 = \begin{bmatrix} 1 & 0 \end{bmatrix}, \; D_x = D_y = D_z = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

Next, the linear data decompositions for arrays x, y and z from `sub1` and `sub2` are mapped into `main`. The call to `sub1` is represented by vertex 8 in `main`'s communication graph and the call to `sub2` is represented by vertex 9. This leads to the following array access functions for procedure `main`:

$$\begin{aligned}
F^1_{x1} = F^1_{y1} = F^1_{z1} &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\
F^1_{x2} = F^1_{y2} = F^1_{z2} &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\
F^1_{x8} = F^1_{y8} = F^1_{z8} &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\
F^1_{x9} = F^1_{y9} = F^1_{z9} &= \begin{bmatrix} 0 \\ 1 \end{bmatrix}
\end{aligned}$$

In the resulting linear decompositions, vertices 1, 2 and 8 form one static decomposition region and vertex 9 forms its own static decomposition region:

$$\begin{array}{lll}
C_1 = C_2 = \begin{bmatrix} 1 & 0 \end{bmatrix}, & C_8 = \begin{bmatrix} 1 \end{bmatrix}, & D_x = D_y = D_z = \begin{bmatrix} 1 & 0 \end{bmatrix} \\
 & C_9 = \begin{bmatrix} 1 \end{bmatrix}, & D_x = D_y = D_z = \begin{bmatrix} 0 & 1 \end{bmatrix}
\end{array}$$

The bottom-up traversal is now complete, and the algorithm begins the top-down traversal. Figure 6.10 shows the call graph and the corresponding communication graphs for each procedure in the sample code during the top-down pass. First the algorithms maps the linear data decompositions found at vertex 9 in `main` back down into procedure `sub2`, and the linear data decompositions for vertex 8 in `main` down into the subroutine `sub1`. The algorithm then calculates the final linear decompositions for `sub1`:

$$C_3 = \begin{bmatrix} 1 & 0 \end{bmatrix}, \; D_x = D_y = D_z = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

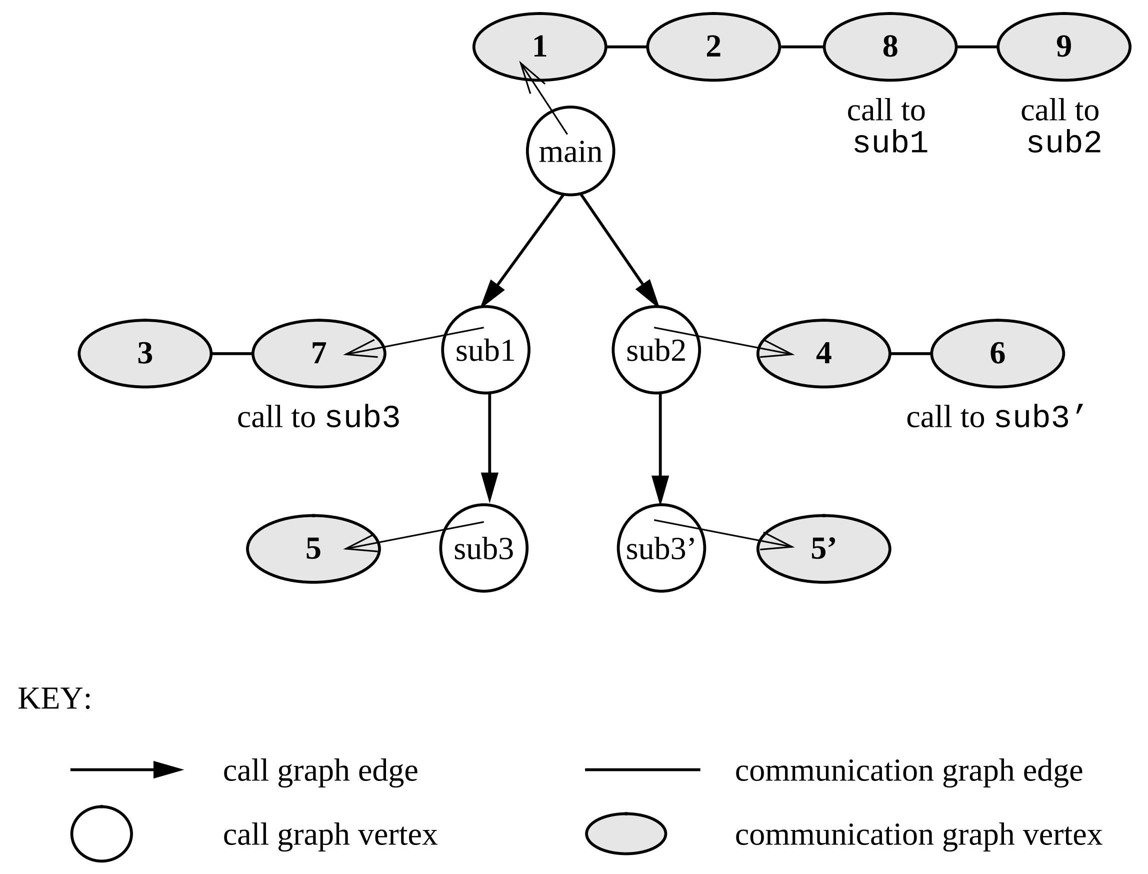


Figure 6.10: Call graph and the corresponding communication graphs for the code in Figure 6.8 during the top-down traversal.

Similarly, the final linear decompositions for `sub2` are:

$$C_4 = \begin{bmatrix} 0 & 1 \end{bmatrix}, \ D_x = D_y = D_z = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

The linear data decompositions in `sub1` and `sub2` are then mapped into the subroutine `sub3`. The linear data decompositions D_x, D_y and D_z in `sub3` are $\begin{bmatrix} 1 & 0 \end{bmatrix}$ along the path from `sub1` and $\begin{bmatrix} 0 & 1 \end{bmatrix}$ along the path from `sub2`. Since these decompositions differ, the compiler clones `sub3` to create two copies. The routine `sub3` is now called from only `sub1` and the routine `sub3′` is now called from `sub2`. The final linear decompositions for `sub3` are:

$$C_5 = \begin{bmatrix} 1 & 0 \end{bmatrix}, \; D_x = D_y = D_z = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

and the final linear decompositions for `sub3'` are:

$$C_{5'} = \begin{bmatrix} 0 & 1 \end{bmatrix}, \; D_x = D_y = D_z = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

In all cases, the algorithm sets the offsets to zero. We use the default BLOCK virtual-to-physical mapping for both static decomposition regions. The final decompositions for the loop nests and arrays in static decomposition region $\{1, 2, 3, 5, 7, 8\}$ are:

$$\begin{array}{rcccl} bp & \leq & \begin{bmatrix} 1 & 0 \end{bmatrix} \vec{\imath} & < & bp + 1 \\ bp & \leq & \begin{bmatrix} 1 & 0 \end{bmatrix} \vec{a} & < & bp + 1 \end{array}$$

where b is the block size and p is the processor number. Similarly, for the loop nests and arrays in static decomposition region $\{4, 5', 6, 9\}$, we have:

$$\begin{array}{rcccl} bp & \leq & \begin{bmatrix} 0 & 1 \end{bmatrix} \vec{\imath} & < & bp + 1 \\ bp & \leq & \begin{bmatrix} 0 & 1 \end{bmatrix} \vec{a} & < & bp + 1 \end{array}$$

6.5 Issues in Interprocedural Decomposition Analysis

Many programs rely on programming language characteristics that make interprocedural decomposition analysis difficult. It is possible that we are able to analyze individual procedures, but lose precision when extending the analysis across procedure boundaries. In this section, we discuss three problems common to scientific codes that impact interprocedural decomposition analysis: unnecessary storage re-use, array reshapes and insufficient type information. Unnecessary storage re-use occurs when unrelated arrays share the same memory locations, and cause unnecessary constraints on the decompositions. Array reshapes and insufficient type information can cause the mapping of array access functions and affine decompositions across calls to fail.

These problems are not unique to decomposition analysis and can hinder other types of interprocedural analyses as well. Here, we discuss specifically how they affect the quality of the decompositions that our algorithm is able to find. We describe these three issues in more detail in the following subsections, and outline additional analyses that can be used to address the problems. These additional analyses have not yet been implemented in the SUIF compiler, unless noted otherwise.

6.5.1 Unnecessary Storage Reuse

Programmers commonly re-use storage for unrelated data objects to save space. Since decomposition analysis finds decompositions for each array based on its memory location, all accesses to arrays that share the same memory must be treated as accesses to the same array. Thus, if there are conflicting constraints on these data objects, the quality of the resulting decompositions may suffer. For example, consider the following code:

```
real x[N,N], y[N,N]
program main
  call sub
  call init_array(x)
  /* Loop Nest 1 */
  for $i_1$ = 1 to N do          /* doall */
    for $i_2$ = 1 to N do
      x[$i_1$,$i_2$] = y[$i_1$,$i_2$] + x[$i_1$,N-$i_2$+1]
  call sub
end

subroutine sub
  call init_array(x)
  /* Loop Nest 2 */
  for $i_1$ = 1 to N do
    for $i_2$ = 1 to N do        /* doall */
      x[$i_1$,$i_2$] = 2 * x[N-$i_1$+1,$i_2$]
end
```

```
subroutine init_array(z)
  real z[N,N]
  /* Loop Nest 3 */
  for $i_1$ = 1 to N do          /* doall */
    for $i_2$ = 1 to N do        /* doall */
      z[$i_1$,$i_2$] = $f_1$($i_1$,$i_2$)
end
```

In both `sub` and `main`, the global variable x is completely over-written by the call to `init_array` before it is used in the procedure. No values of x calculated in `main` are ever used in `sub`, and similarly, no values of x calculated in `sub` are ever used in `main`. However, the decomposition analysis must honor all constraints on x in all procedures. The linear data decomposition for array x in `sub` is $D_x = \begin{bmatrix} 0 & 1 \end{bmatrix}$ with $\mathcal{N}(D_x) = \text{span}\,\{(1,0)\}$. The constraint on x is then propagated into procedure `main` for the calls to `sub` (there are no constraints on x from `init_array`). In `main`, the initial constraint for x at loop nest 1 is $\mathcal{N}(D_x) = \text{span}\,\{(0,1)\}$. When the constraints on x are merged in `main`, we have $\mathcal{N}(D_x) = \text{span}\,\{(0,1),(1,0)\}$ which spans the entire array space. Since the values in x are not actually shared between `main` and `sub`, we would like to avoid merging the constraints.

The compiler could mitigate the problem of unnecessary storage re-use by performing *interprocedural array renaming* analysis. This analysis would determine if there are any true dependences between the data objects that share storage. If there are no true dependences, then different memory could be allocated for the logically different variables. This analysis requires the compiler to determine whether the different variables are entirely written before being read and is analogous to array privatization analysis[37, 75].

6.5.2 Array Reshapes

Many languages (including FORTRAN-77, FORTRAN-90 and C) allow programs that rely on the model that memory is one dimensional and linearly addressed. A consequence of this model is that a legal program can access the same memory in very different ways. In particular, arrays in scientific codes are often reshaped across procedure calls such that the number or size of the dimensions differs across procedures. As we saw in Section 6.2.1, array reshapes cause the mapping of an affine function across a call to fail if the resulting function is not affine. In order for the mapping to succeed, the dimensions of the array in the callee must correspond to complete dimensions of the array in the caller. This means that we can handle array sections, where one or more complete dimensions of an array are passed as a parameter (for example, the code in Figure 6.1(a)). However, we cannot handle subsections of array dimensions passed as parameters, or linearized arrays, for example:

```
program main
  real data[200]
  call sub1(data)
  call sub2(data)
end

subroutine sub1(x)
  real x[20,10]
  for $i_1$ = 1 to 20 do
    for $i_2$ = 1 to 10 do
      x[$i_1$,$i_2$] = ...
end

subroutine sub2(x)
  real x[5,10,4]
  for $i_1$ = 1 to 5 do
    for $i_2$ = 1 to 10 do
      for $i_3$ = 1 to 4 do
        x[$i_i$,$i_2$,$i_3$] = ...
end
```

The data decomposition for x in `sub1` treats it as a two-dimensional array. When the data decomposition is mapped into `main`, the array is now one-dimensional and the resulting data decomposition function is non-affine. The compiler can solve this problem if it can find a consistent size and shape for the array everywhere in the program. If the memory allocated for `data` were always accessed as a 20 $\times$ 10 array, then the compiler could simply replace the declaration of `data[200]` by `data[20,10]`. However, if the array is accessed with different shapes in different parts of the program, as in `sub2` above, then we cannot map affine functions successfully for this array.

6.5.3 Insufficient Type Information

Many languages allow formal parameters to have incomplete or parameterized types. In these cases, the semantics of the program again rely on the assumption of a linear memory model. Since we often need array bounds information to map affine functions across calls, missing type information can cause the mapping to fail. In many cases, the compiler could address this problem by propagating type information from the actual parameters down into the formal parameters. For example, in the following code, the array z in procedure `sub1` has a parameterized type (legal in FORTRAN):

```
program main
  real x[20,10], y[100,50]
  call sub1(x, 20, 10)
  call sub1(y, 100, 50)
end

subroutine sub1(z, m, n)
  real z[m,n]
  integer m, n
  ...
end
```

If the variables m and n are known constants, the compiler can perform interprocedural constant propagation to complete the type information (this analysis is implemented in

SUIF and was used for our experimental study). If m and n have different values along different paths to the subroutine `sub1` then the compiler can clone the subroutine to make different copies for each of the different values.

Another case where additional analysis can be used to extract type information is when the dimension size is unspecified, e.g. in FORTRAN, the dimension size is marked with `*`. For example, in the code below the array z in procedure `sub1` has a variable size:

```
program main
  real x[10,20,3], y[80,20]
  call sub1(x, 10*20*3)
  call sub1(y, 80*20)
end

subroutine sub1(z, size)
  real z[*]
  integer size
  ...
end
```

In this case, the compiler could use a simple interprocedural type propagation analysis to fill in the missing types. Again, cloning may be necessary if there is different type information along different paths to the same procedure in the program.

6.6 Libraries and User-Defined Decompositions

In order to perform full interprocedural analysis, the compiler must have all the sources of the input program. If the sources are not available then communication of global arrays and parameters may be incurred upon entry and exit to the unanalyzed procedures. For library routines where the sources are not available, different versions of the routines can be provided with different data decompositions and the compiler can call the routine with matching decompositions after they are found.

Another option is to keep a summary of the expected decompositions for the variables in the library routines. The requested decompositions at the call are then used as input in the decomposition algorithm. The decompositions at the call are marked as fixed, and the algorithm will then try to match the decompositions in the other loop nests. The same technique is used to deal with any user-specified data decompositions already in the program.

6.7 Summary

In this chapter, we presented an algorithm for calculating decompositions interprocedurally. The algorithm is built on top of the decomposition algorithms from the previous chapters. The focus of this chapter was on how to propagate and represent the necessary information across the different procedures.

The interprocedural algorithm for finding affine decompositions onto the virtual processor space visits each procedure twice, once in a bottom-up traversal of the call graph and once in a top-down traversal of the call graph. The bottom-up traversal begins by running the intraprocedural decomposition algorithm from the previous chapter on the leaf procedures in the call graph. It then propagates the array accesses and linear data decompositions up from the callee procedure into the caller procedure. Next, the algorithm runs the intraprocedural decomposition analysis on the caller procedures, and continues up the call graph. When the algorithm reaches the main procedure, it calculates the final linear decompositions. The top-down pass then pushes the final linear decompositions down from the caller procedures into the callee procedures, and calculates the offsets to form the complete affine decompositions.

We also discussed three problems common to scientific codes that can hinder interprocedural decomposition analysis: unnecessary storage re-use, array reshapes and insufficient type information. Unnecessary storage re-use generates unnecessary constraints on the decompositions, and can cause the quality of the final decompositions to suffer. Array reshapes and insufficient type information can cause the mapping of array access functions and affine decompositions across calls to fail. It is possible, however, for the compiler to perform additional analysis to mitigate the effects of some of these problems.

Chapter 7

Experimental Results

All the algorithms described in this paper have been implemented in the SUIF compiler system[76]. To evaluate the effectiveness of our proposed algorithm, we applied the compiler to a suite of benchmark programs. We ran the compiler-generated code on the Stanford DASH multiprocessor[55] and a Digital AlphaServer 8400, and compared our results to those obtained without using our techniques.

7.1 Experimental Setup

7.1.1 Target Architectures

We ran our experiments on two different architectures, a 32-processor Stanford DASH multiprocessor and an 8-processor Digital AlphaServer 8400.

Stanford DASH Multiprocessor. DASH is a distributed shared address space multiprocessor. The machine we used for our experiments consists of 32 processors, organized into 8 clusters of 4 processors each. Each cluster is based on a Silicon Graphics POWER Station 4D/340, a bus-based centralized memory machine. The processors are 33 MHz MIPS R3000s, each with a 64 KB first-level cache and a 256 KB second-level cache. Both the first- and second-level caches are direct-mapped and have 16-byte lines. Each cluster has 28 MB of main memory. A directory-based protocol is used to maintain cache coherence across clusters. It takes a processor 1 cycle to retrieve data from its first-level cache, about 10 cycles from its second-level cache, 30 cycles from its local memory and 100-130 cycles from a remote memory. The DASH operating system allocates memory to clusters at the page level. The page size is 4 KB and pages are allocated to the first cluster that touches the page. Within a cluster, the operating system uses a standard page-coloring page placement policy where consecutive virtual pages are mapped round-robin to consecutive colors (physical pages with the same color map to the same location in a physically-indexed cache).

On DASH, communication between processors in different clusters results in a 100-130 cycle latency. This long latency (compared to 1 cycle for a first-level cache hit or even 30 cycles for access to local memory) means that minimizing communication is essential to performance. False-sharing is not likely to be a problem due to the small cache-lines (each line holds only two double-words) on DASH. However, it is important for applications to have good spatial locality since the directed-mapped caches can lead to conflict misses.

Digital AlphaServer 8400. The Digital AlphaServer 8400 is a bus-based centralized shared address space multiprocessor. The machine we used consisted of 8 300 MHz 21164 processors. Each 21164 has on-chip 8 KB split instruction and data first-level caches, and a 96 KB combined second-level cache. The first-level caches are direct-mapped and the second-level cache is three-way set associative. The cache line size for the second-level cache is 64 bytes. Each processor also has a 4 MB direct-mapped external cache and the machine was configured with 4 GB of main memory. It takes a processor 2 cycles to retrieve data from the first-level cache, 6 cycles from the second-level cache, 12 cycles from the external cache and a minimum of 90 cycles from main memory[27, 30]. The page size is 8 KB, and the operating system uses a bin-hopping page placement policy where virtual pages are assigned colors in the order that the page faults occur.

The AlphaServer has a single centralized memory and no remote memory. It uses a write-invalidate cache-coherence protocol – when a processor does a write, all other cached copies are invalidated. The next time another processor accesses the data, it misses in the cache. Thus, if a processor has data cached, then communication between processors incurs at least a 90 cycle latency. Minimizing communication is important to performance, only if the data would have been in the cache. If the data are not likely to be in the cache, then the processor would have had to go to main memory to access it anyway. The long cache lines (8 double-words) mean that false-sharing is a potential performance problem. The direct-mapped first- and third-level caches can lead to conflict misses, making good spatial locality also important for performance.

7.1.2 The SUIF Compiler

The inputs to the SUIF compiler are sequential FORTRAN or C programs. The output is parallelized source code that is a combination of C and/or FORTRAN with calls to a portable run-time library. The SUIF output is then compiled on the parallel machine using the native C and FORTRAN compilers.

The applications parallelized by SUIF for shared address space machines follow a master/slave model of parallelism. The master process executes the sequential portions of the program while the slaves wait at a barrier. When the master reaches the start of a parallel region, it notifies the slaves. The slaves and the master then operate in parallel until they reach a barrier at the end of the parallel region.

7.1.3 Methodology

We compiled each program under SUIF both with and without the decomposition analysis. We then compiled the SUIF output using the native C and FORTRAN compilers on the target machine. When compiling with the native compilers, we always used the highest optimization level available.

All of our results are expressed in terms of speedup in execution time over the best sequential version of the programs. In all cases, we timed the execution of the complete application, including any time spent doing initialization or post-processing. All timings use wall-clock time, and the runs were done on an unloaded system. To obtain the best sequential version of a program, we compared the execution time of the program compiled with SUIF against the execution time of the program compiled with only the native compiler.

We compiled several different versions of each program, described below. An overview of the passes of the SUIF compiler for shared address space machines is shown in Figure 7.1. A description of the context of the decomposition analysis within the complete compiler system was described in Chapter 2. Here we describe the specific passes of the SUIF compiler that were used in each of our experiments:

Base Parallelization (*base*): The programs are compiled using only basic parallelization techniques. No decomposition analysis is performed. The parallelization pass has a loop nest optimizer that analyzes one loop at a time to expose outermost loop parallelism. Iterations of the outermost *doall* loop in each loop nest are distributed across the processors, and each processor executes equal-sized blocks of consecutive iterations. In Figure 7.1 this option corresponds to executing the compiler passes along the path marked 1.

The following two variations perform decomposition analysis in addition to basic parallelization. The decomposition pass finds a mapping of both the computation and the data across the processors. Since we are performing our experiments on shared address space machines, using the data decompositions to perform data transformations is an optimization – it is not needed for correctness. This allows us to investigate the impact of using just the computation decompositions to generate code, as well as the impact of using both computation and data decompositions (it does not make sense to use just the data decompositions without the computation decompositions because that would map the data to processors that are not necessarily going to use that data).

Computation Scheduling Only (*comp sched*): The programs are compiled with decomposition analysis to find computation decompositions (and the corresponding data decompositions). The computation decompositions are passed to a scheduler that generates code to partition the parallel loops across the processors and inserts calls to the run-time library. Partitioning iterations of the loops using the computation decompositions means that the processors execute computation that re-uses the same data, thereby improving temporal locality. The scheduler also takes advantage of the data decomposition information to optimize the synchronization in the program[73]. The data layouts are left unchanged and are stored according to the default convention

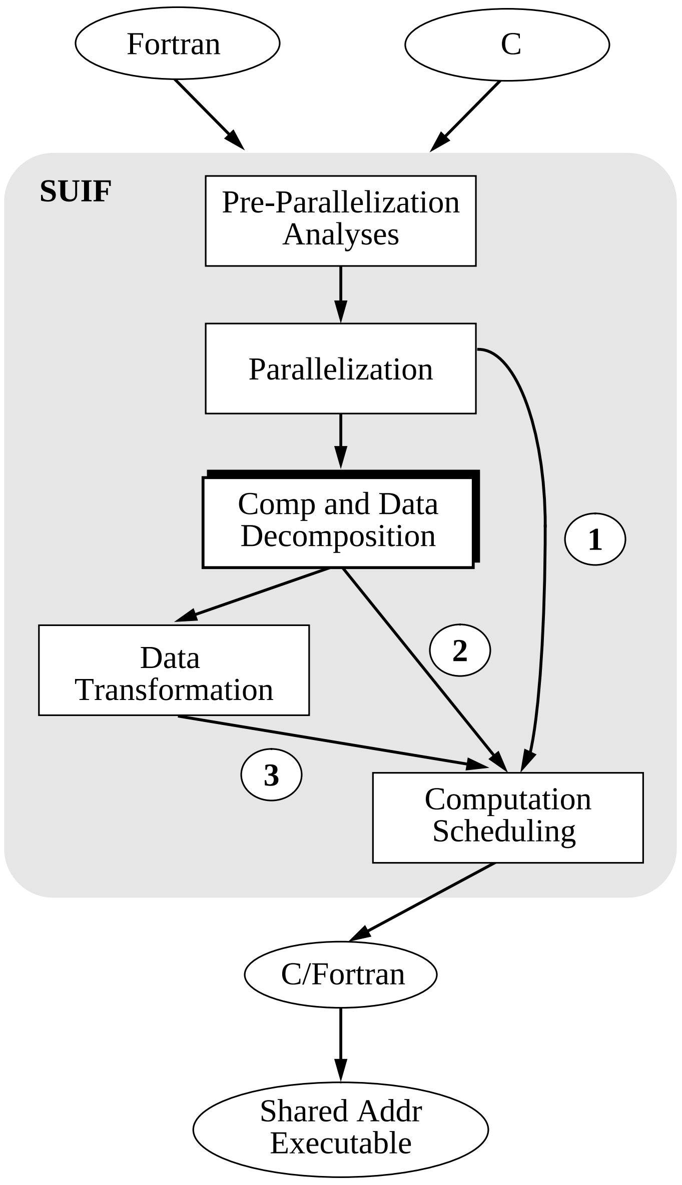


Figure 7.1: An overview of the SUIF parallelizing compiler for shared address space machines. The numbered arrows show different possible paths through the compiler.

of the input language. In Figure 7.1 this option corresponds to executing the compiler passes along the path marked 2.

Computation Scheduling and Data Transformations (*comp sched + data transform*): The programs are compiled with decomposition analysis to find the computation and data decompositions. The computation is scheduled in the same manner as with the *comp sched* option above. In addition, the compiler uses the data decompositions to transform the data layout in the parallelized code to improve spatial locality. In Figure 7.1 this option corresponds to executing the compiler passes along the path marked 3.

The decomposition analysis also provides opportunities for additional optimizations (see Section 2.5). We investigated the impact of one such optimization, compiler-directed page coloring (CDPC)[18] on the AlphaServer. CDPC is a technique for improving cache utilization and eliminating cache conflicts of parallelized code on shared address space machines. With CDPC, the compiler uses its knowledge of the data decompositions to direct the operating system's page allocation policy into making each processor's data contiguous in the physical address space. The operating system uses these hints to determine the virtual-to-physical page mapping when the pages are allocated.

The following two variations use compiler-directed page coloring with the base compiler and with the decomposition analysis. In the latter case, we only run coloring with the version that uses both the computation and data decompositions to generate code. This is because coloring relies on each processor's data for each *individual* array already being contiguous in the shared address space, while it tries to make the data *across* different arrays contiguous. However, only using the computation decomposition tends to scatter each processor's data across the shared address space and thus it does not make sense to combine it with CDPC.

Base Parallelization with Coloring (*base + coloring*): The programs are compiled with the base parallelizer and compiler-directed page coloring is applied. CDPC relies on the compiler to supply information about the data access patterns of each processor to a run-time library. The run-time library then uses the information to customize the application's page mapping policy. The base parallelizer does not know exactly which processor accesses which data. Thus in this case the compiler assumes a default partitioning of the data across the processors, where each processor accesses equal-sized blocks of the outermost dimension of each array.

Computation Scheduling and Data Transformations with Coloring (*comp sched + data transform + coloring*): The programs are compiled in the same manner as the *comp sched + data transform* version above. The data decompositions specify exactly which processor is going to access which data, and this information is passed to the coloring run-time library.

7.1.4 Application Suite

The list of benchmarks used in this study are shown in Table 7.1. The first four benchmarks in the table are program kernels, and the remaining benchmarks are complete programs. The kernels are used to help explain the behavior observed in the programs. In addition to benchmarks from the SPEC92 and SPEC95 benchmark suites, we also have programs from Lawrence Livermore National Lab (LLNL) and the Institute for Computer Applications in Science and Engineering (ICASE). Table 7.2 shows the data set sizes of the benchmarks for the problem sizes that we used in this study.

Benchmark	Description	Source
ADI integration	alternating direction implicit integration	hand-written
LU decomposition	LU decomposition without pivoting	hand-written
stencil	five-point stencil	hand-written
vpenta	invert pentadiagonal matrices	SPEC92 (nasa7)
applu	partial differential equation solves	SPEC95
erlebacher	3D tridiagonal solves	ICASE
simple	2D Lagrangian hydrodynamics	LLNL
swim	shallow water simulation	SPEC92,SPEC95
tomcatv	mesh generation	SPEC92,SPEC95

Table 7.1: Descriptions of the benchmarks.

These benchmarks were chosen because the SUIF compiler is able to find a significant amount of parallelism in them, yet these programs still show poor speedups when using only basic parallelization techniques (i.e. without any decomposition analysis). Figure 7.3

Benchmark	Problem Size	Data Set Size (MB)
ADI integration	256×256	1.5
	512×512	6.0
	1024×1024	24.0
	2048×2048	96.0
LU decomposition	256×256	0.5
	512×512	2.0
	1024×1024	8.0
	2048×2048	32.0
stencil	512×512	4.0
	1024×1024	16.0
	2048×2048	64.0
vpenta	128×128	1.6
applu	$33 \times 33 \times 33$	31.6
erlebacher	$64 \times 64 \times 64$	4.6
simple	202×182	3.1
swim	256×256	3.6
	512×512	14.2
tomcatv	256×256	3.6
	512×512	14.2

Table 7.2: Data set sizes of the benchmarks.

shows the *parallel coverage* and the 32-processor speedups obtained on the Stanford DASH multiprocessor with the base parallelizer. Parallel coverage is defined as the percentage of the original sequential program that can be executed in parallel. For all the benchmarks studied on DASH, the parallel coverage is 99% or greater. A program with a coverage of 99% with perfect utilization on 32 processors results in a speedup of 24.4. However, the speedups we observed often fell far below the ideal, ranging anywhere from as low as 4.2 for vpenta to a high of 19.5 for the 1024×1024 size of LU decomposition.

Figure 7.4 shows the parallel coverage and the best speedups obtained on the AlphaServer with the basic parallelizer. The parallel coverage for the benchmarks on the AlphaServer is 96% or more. The coverages are lower than on DASH as the Alpha 21164 processors are significantly faster than DASH's MIPS R3000 processors. This effect is due

Benchmark	Problem Size	Parallel Coverage %	Speedup (32 processors)
ADI integration	256 × 256	100	6.0
	1024 × 1024	100	8.0
LU decomposition	256 × 256	100	8.1
	1024 × 1024	100	19.5
stencil	512 × 512	100	15.6
vpenta	128 × 128	100	4.2
erlebacher	64 × 64 × 64	100	11.6
swim	256 × 256	99	15.6
tomcatv	256 × 256	99	4.9

Table 7.3: Parallel coverage and 32-processor speedups for the benchmarks on the DASH multiprocessor. Parallel coverage is defined as the percentage of the sequential execution time that can be executed in parallel.

to the fact that the parallelizable loop nests also tend to be more amenable to other compiler optimizations, such as software pipelining, loop unrolling, prefetching, etc. The impact of these optimizations is significant on the Alpha 21164, a statically scheduled quad-issue processor with prefetch instructions. In contrast, the R3000 is a simple single-issue processor with no prefetch instruction, and its native compiler does not need to perform these aggressive optimizations. The result is that the small amounts of non-parallelizable code are not as well optimized and take up a larger percentage of the total execution time on the Alpha.

Several of the programs showed high speedups, notably the 1024 × 1024 problem size of stencil at 9.4 and the 512 × 512 problem size of swim at 11.3. These programs have super-linear speedups because once the data is partitioned across the 8 processors, it fits into the individual processor's caches. However, a number of the programs showed very poor speedups, for example, 256 × 256 tomcatv at 1.5, 256 × 256 ADI at 1.7, and both simple and vpenta at 1.8. For comparison, perfect utilization of 8 processors with 96% coverage should give a speedup of 6.25.

Benchmark	Problem Size	Parallel Coverage %	Speedup (8 processors)
ADI integration	256×256	100	1.7
	512×512	100	2.8
	1024×1024	100	3.6
	2048×2048	100	4.3
LU decomposition	512×512	100	5.9
	1024×1024	100	8.9
	2048×2048	100	9.0
stencil	512×512	100	5.5
	1024×1024	100	9.4
	2048×2048	100	4.4
vpenta	128×128	96	1.8
applu	$33 \times 33 \times 33$	100	4.9
erlebacher	$64 \times 64 \times 64$	99	4.5
simple	202×182	99	1.8
swim	256×256	99	6.4
	512×512	100	11.3
tomcatv	256×256	97	1.5
	512×512	97	2.9

Table 7.4: Parallel coverage and 8-processor speedups for the benchmarks on the Digital AlphaServer. Parallel coverage is defined as the percentage of the sequential execution time that can be executed in parallel.

7.2 Experimental Results

In this section we present experimental results for each of the benchmark programs, compiled under the different schemes described above.

7.2.1 Distributed Shared Address Space Machine

The speedup graphs for the Stanford DASH Multiprocessor are shown in Figure 7.2 through Figure 7.8. The problem size is shown in the upper left corner of the graphs. Each figure plots the speedups for the *base* version of the application, in addition to the *comp sched* and *comp sched + data transform* versions.

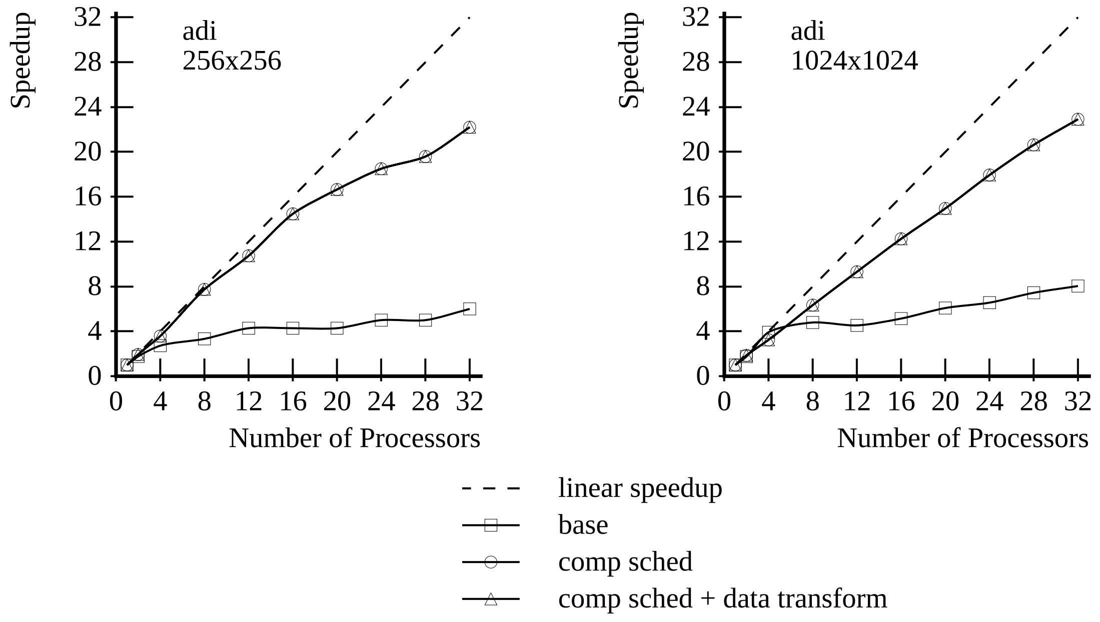


Figure 7.2: Speedups for ADI integration on the DASH Multiprocessor.

ADI Integration. Figure 7.2 shows the results for ADI integration. ADI is an iterative computation and each iteration has two phases — the first phase sweeps along the columns of the arrays and the second phase sweeps along the rows (two representative loops of the code were shown in Section 4.1.4.1).

The *base* compiler analyzes each loop nest separately and parallelizes the column sweeps in the first phase, and the row sweeps in the second phase. This means each processor accesses very different data in different parts of the program, and causes poor temporal locality. Furthermore, while data accessed by a processor in the row sweeps are contiguous (the code is written in C and the arrays are thus allocated in row-major order), the data each processor accesses in the column sweeps are distributed across the shared address space. This leads to poor spatial locality. As a result, the base version performs poorly on this program, with maximum speedups of only 6 and 8 on the 256×256 and 1024×1024 problem sizes, respectively.

In the *comp sched* version, the compiler finds decompositions that use *doall* parallelism in the row-sweep phase and switch to *doacross* parallelism in the column-sweep phase. Now each processor accesses the same block of rows during both the row sweeps and the column sweeps. This computation decomposition improves the spatial locality in addition to improving the temporal locality and eliminating most of the inter-processor communication. This version of ADI gets a speedup of 23 on 32 processors. Since each processor access blocks of rows of the array, each processor's data is already contiguous in the shared address space and the data transformations have no effect.

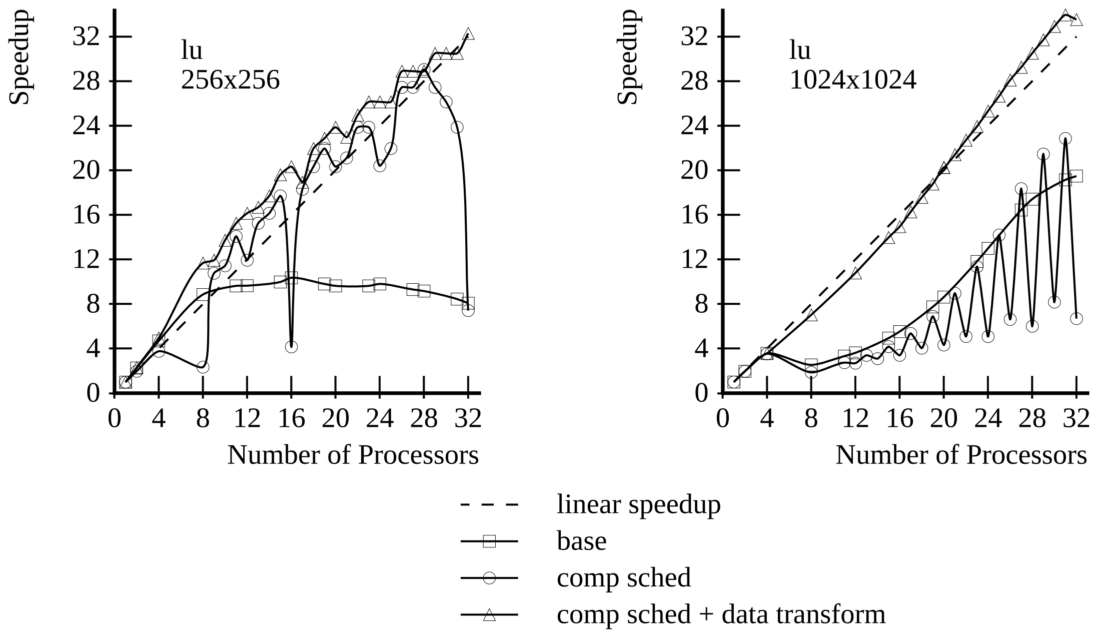


Figure 7.3: Speedups for LU decomposition on the DASH Multiprocessor.

LU Decomposition. Figure 7.3 shows the results for LU decomposition. In the *base* version, the number of iterations in the parallel loop varies with the index of an outer sequential loop. As a result, each processor accesses different data each time through the outer loop.

The decomposition analysis assigns all operations on the same column of data to the same processor. For load balance, the columns and operations on the columns are distributed across the processors in a cyclic manner. By fixing the assignment of computation to processors, the compiler replaces the barriers that followed each execution of the parallel loop by locks. The *comp sched* version has good load balance, good data re-use and inexpensive synchronization; however, the local data accessed by each processor are scattered in the shared address space, increasing chances of interference in the cache between columns of the array. The interference is highly sensitive to the array size and the number of processors. This interference effect is especially pronounced when the array size and the number of processors are both powers of 2. For example, with a 1024×1024 matrix, every 8th column maps to the same location in DASH's direct-mapped 64 KB cache. The speedup for 31 processors is 5 times better than for 32 processors.

The data transformation pass restructures the columns of the array so that each processor's cyclic columns are made into a contiguous region. After restructuring, the performance of the *comp sched + data transform* version stabilizes and is consistently high. In this case the compiler is able to take advantage of inexpensive synchronization and data re-use without incurring the cost of poor cache behavior. Speedups become super-linear in some cases due to the fact that once the data are partitioned among enough processors, each processor's working set starts to fit into the cache.

Five-point Stencil and Swim. Figure 7.4 shows the speedups for five-point stencil. The application swim also performs a stencil computation and has the same behavior as the five-point stencil kernel. Figure 7.7 shows the results on DASH for swim.

In the *base* version, the compiler distributes the outermost parallel loop across the processors, and each processor updates a block of array columns (the code is written in FORTRAN and thus the arrays are allocated in column-major order).

The decomposition analysis assigns two-dimensional blocks to each processor, since this mapping has a better computation to communication ratio than a one-dimensional mapping used by the base version (in Figure 7.4, the number of processors in each of the two dimensions is also shown under the total number of processors). However, without also changing the data layout, the performance of the *comp sched* version is worse than the base version because now each processor's portion of the data is not contiguous in the

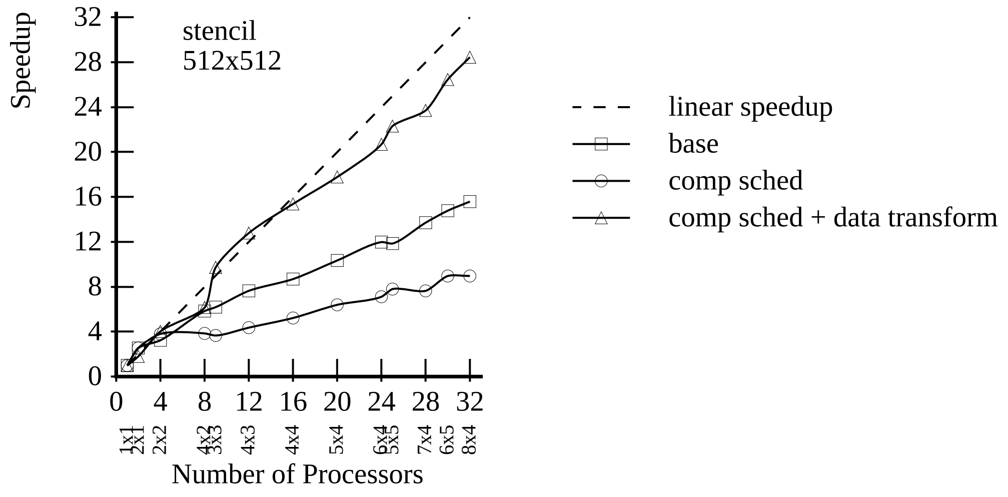


Figure 7.4: Speedups for five-point stencil on the DASH Multiprocessor.

shared address space. As a result the program's poor spatial locality outweighs the benefits of the better computation to communication ratio.

After the data transformation is applied, the *comp sched + data transform* version of the program has good spatial locality as well as less communication, and thus we achieve a speedup of 29 on 32 processors. Note that the performance is very sensitive to the number of processors. This is due to the fact that each DASH cluster has 4 processors and the amount of communication across clusters differs for different two-dimensional mappings.

Vpenta. The performance results for vpenta are shown in Figure 7.5. In the *base* version, the compiler interchanges the loops in the original code so that the outer loop is parallelizable and the inner loop carries spatial locality. Without such optimizations, the program would not even get the slight speedup obtained with the base compiler.

For this particular program, the base compiler parallelizes the same loops as the decomposition analysis. However, since the compiler can determine that each processor accesses exactly the same partition of the arrays across the loops, the code generator can eliminate barriers between some of the loops. This accounts for the slight increase in performance of the *comp sched* version over the base compiler.

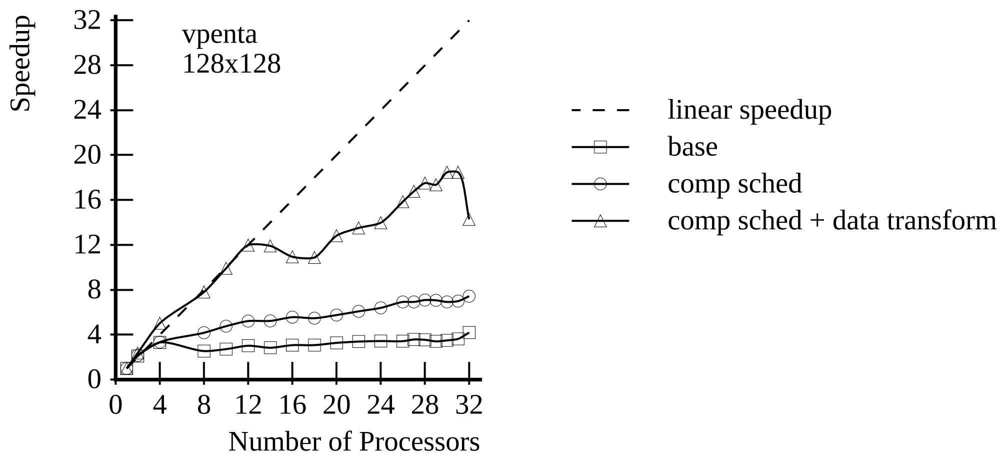


Figure 7.5: Speedups for vpenta on the DASH Multiprocessor.

This program operates on a set of two-dimensional and three-dimensional arrays. Each processor accesses a block of columns for the two-dimensional arrays, thus no data reorganization is necessary for these arrays. However, each plane of the three-dimensional arrays is partitioned into blocks of rows, each of which is accessed by a different processor. Thus after applying the data transformations the data accessed by each processor is contiguous. With the improved data layout, the *comp sched + data transform* version of the program finally runs with a decent speedup. We observe that the performance dips slightly when there are about 16 processors, and drops when there are 32 processors. This performance degradation is likely due to increased cache conflicts between different arrays on the same processor. Further data and computation optimizations that focus on operations on the same processor would be useful.

Erlebacher. The erlebacher application performs three-dimensional tridiagonal solves. It has a number of fully parallel computations that are interleaved with multi-dimensional reductions and computational wavefronts in all three dimensions caused by forward and backward substitutions. Partial derivatives are computed in all three dimensions with three-dimensional arrays. Figure 7.6 shows the resulting speedups.

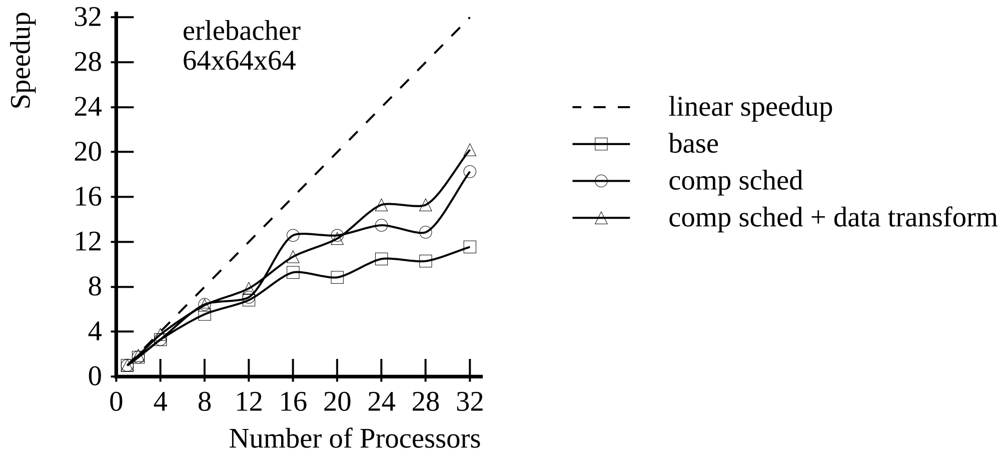


Figure 7.6: Speedups for erlebacher on the DASH Multiprocessor.

The *base* version always parallelizes the outermost parallel loop. This strategy yields local accesses in the first two phases of erlebacher when computing partial derivatives in the X and Y dimensions, but ends up causing non-local accesses in the Z dimension.

The decomposition analysis finds a computation decomposition so that no non-local accesses are needed in the Z dimension. Each processor accesses a block of columns for the two arrays that hold the partial derivatives in the X and Y directions, and a block of rows for the array in the Z direction. Thus in this version of the program, the third array has poor spatial locality. As a result, the *comp sched* version only improves the performance of erlebacher slightly over the base-line version. The data transformation phase of the compiler restructures the Z direction array so that local references are contiguous in memory. Because two-thirds of the program is perfectly parallel with all local accesses, the optimizations only realize a modest performance improvement.

Tomcatv. Figure 7.8 shows the speedups for tomcatv. This program contains several loop nests that have dependences across the rows of the arrays and other loop nests that have no dependences. Since the *base* version always parallelizes the outermost parallel loop, each processor accesses a block of array columns in the loop nests with no dependences.

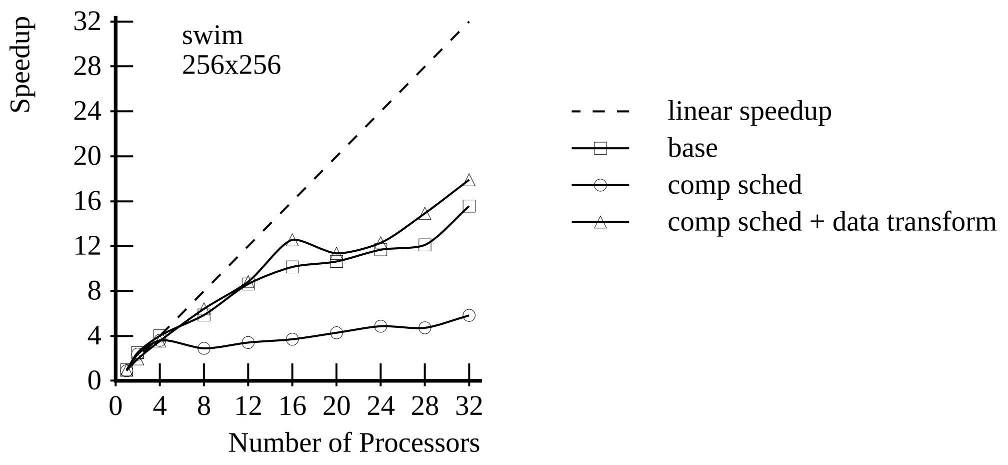


Figure 7.7: Speedups for swim on the DASH Multiprocessor.

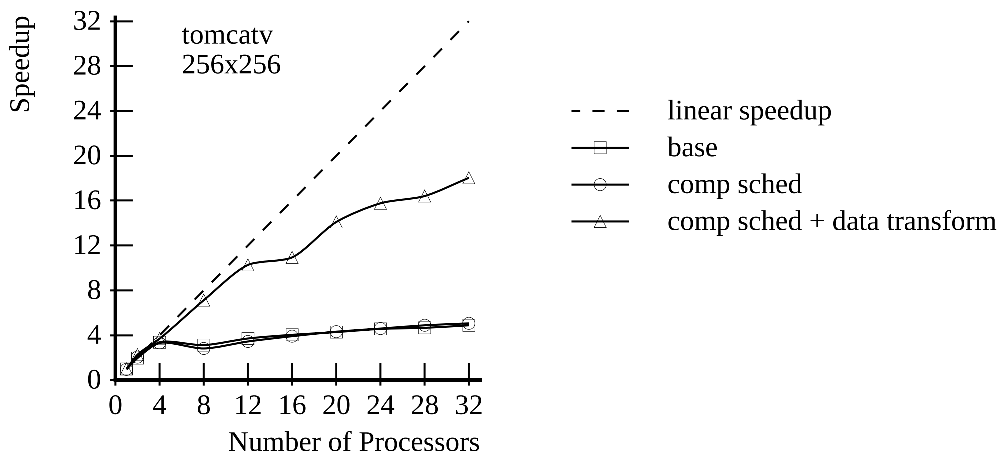


Figure 7.8: Speedups for tomcatv on the DASH Multiprocessor.

However, in the loop nests with row dependences, each processor accesses a block of array rows. As a result, there is little opportunity for data re-use across loop nests. Also, there is poor cache performance in the row-dependent loop nests because the data accessed by each processor is not contiguous in the shared address space.

The decomposition analysis selects a decomposition so that each processor always accesses a block of rows. The row-dependent loop nests still execute completely in parallel. Thus the *comp decomp* version of tomcatv exhibits good temporal locality; however, the speedups are still poor due to poor spatial locality. After transforming the data to make each processor's rows contiguous, the cache performance improves. Whereas the maximum speedup achieved by the *base* version is 5, the *comp sched + data transform* version of tomcatv achieves a speedup of 18.

A summary of the experimental results for DASH are shown in Table 7.5. For each program we compare the speedups on 32 processors obtained with the base compiler against the speedups obtained with decomposition analysis together with computation scheduling and data transformations. The table shows that decomposition analysis can have a significant impact on the performance of applications on DASH. Among the kernels, the optimized versions ran from 1.7 times faster than the base version (1024 $\times$ 1024 LU decomposition) to 4.0 times faster (256 $\times$ 256 LU decomposition) on 32 processors. For the full programs, the improvements for the optimized version over the base version ranged from 1.1 times faster for swim to as much as 3.7 times faster for tomcatv on 32 processors.

7.2.2 Centralized Shared Address Space Machine

The speedup graphs for the Digital AlphaServer 8400 are shown in Figure 7.9 through Figure 7.17. The problem size is shown in the upper left corner of the graphs. Each figure plots the speedups for the *base*, *comp sched* and *comp sched + data transform* versions. In addition, we also show results for *base + color* and *comp sched + data transform + color* versions.

ADI Integration. Figure 7.9 shows the results for ADI integration on the AlphaServer. As was the case on DASH, the versions with decomposition analysis outperform the *base*

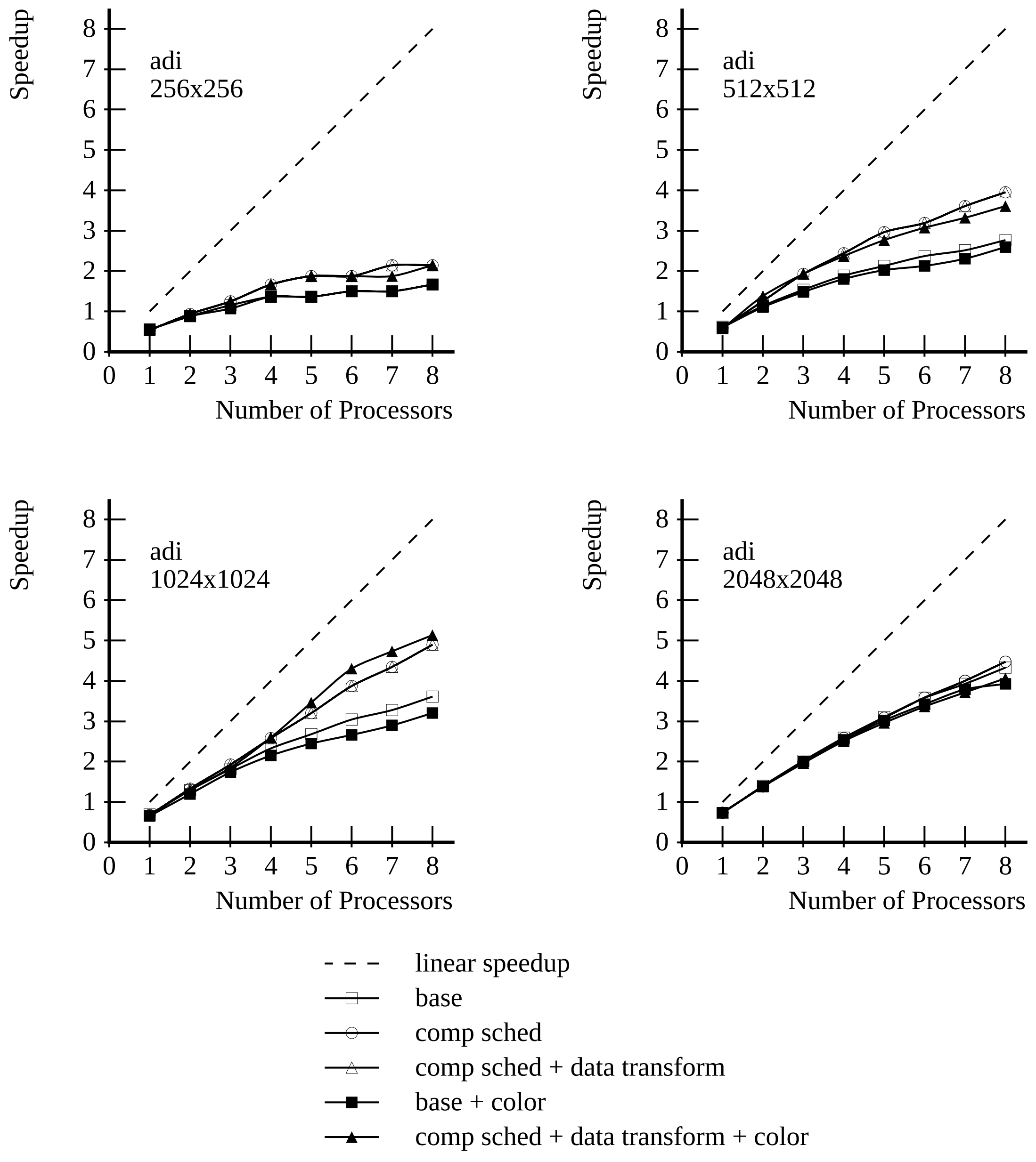


Figure 7.9: Speedups for ADI integration on the AlphaServer 8400.

Benchmark	Problem Size	Base	Optimized	Ratio
		Speedups (32 processors)		Optimized/Base
ADI integration	256 × 256	6.0	22.2	3.7
	1024 × 1024	8.0	22.9	2.9
LU decomposition	256 × 256	8.1	32.3	4.0
	1024 × 1024	19.5	33.5	1.7
stencil	512 × 512	15.6	28.5	1.8
vpenta	128 × 128	4.2	14.3	3.4
erlebacher	64 × 64 × 64	11.6	20.2	1.7
swim	256 × 256	15.6	17.9	1.1
tomcatv	256 × 256	4.9	18.0	3.7

Table 7.5: Summary of results on the Stanford DASH Multiprocessor. The table compares the 32-processor speedups obtained with the base compiler against the speedups obtained with decomposition analysis.

version due to better temporal locality for the 256 × 256, 512 × 512 and 1024 × 1024 problem sizes. The *comp sched* version uses *doacross* parallelism for some of the loops in ADI integration. The improvements due to the decomposition analysis are greater in the larger problem sizes because the overhead of the *doacross* parallelism is better amortized. For the 2048 × 2048 problem size the decomposition analysis has little effect. This is due to the fact that each array has $2048 \cdot 2048 \cdot 8 = 32$ MB of data (the array elements are doubles). Even when partitioned across 8 processors, each processor's portion of a single array uses the entire 4 MB external cache and there is no opportunity for re-use. Compiler-directed page coloring had little impact on the performance.

LU Decomposition. Figure 7.10 shows the results for LU decomposition. The 512×512 and 1024 × 1024 problem sizes of the program showed similar performance. The entire working set of the 512 × 512 size (2 MB) completely fits in the 4 MB external cache of one processor, so on the AlphaServer we do not see same erratic behavior that we saw on DASH. The 1024 × 1024 size also fits entirely in the external caches at two processors, and again the performance for all versions is very similar. Even at 2048 × 2048 enough of the working set fits into the cache that all the versions without coloring have the same behavior. Since there is a single array, coloring defaults to a standard policy of mapping

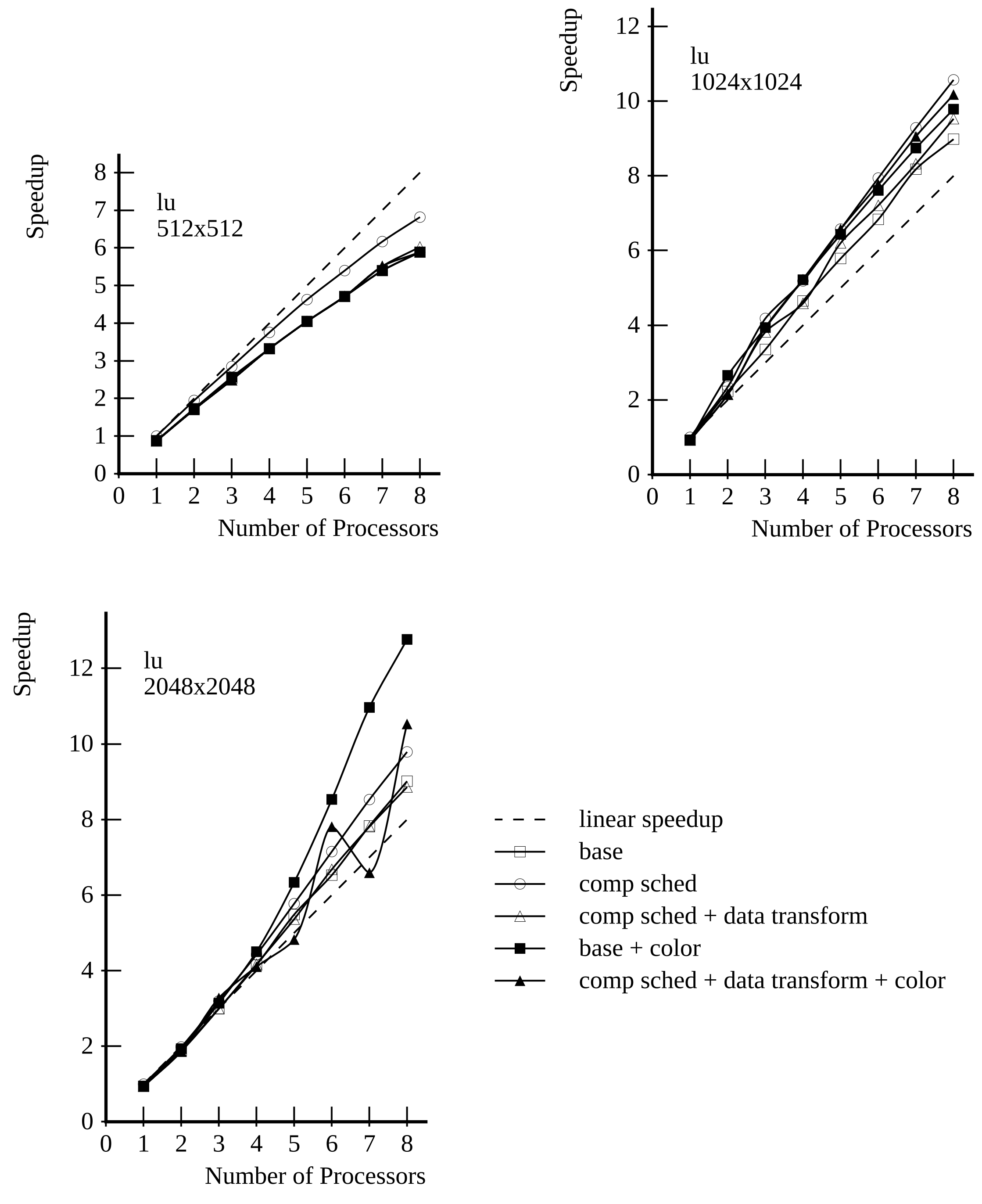


Figure 7.10: Speedups for LU decomposition on the AlphaServer 8400.

pages in a round-robin fashion based on their virtual addresses.

Five-point Stencil. Figure 7.11 shows the results for five-point stencil. The speedups for the versions with decomposition analysis are step-functions due to the way the 8 processors are partitioned for two dimensions of parallelism (in Figure 7.11, the number of processors in each of the two dimensions is also shown under the total number of processors). For example, even when there are 7 processors we still have a 3×2 partitioning and thus only use 6 processors. For the 512×512 problem size, *base* outperforms the versions with decomposition analysis. As we saw with DASH, the poor spatial locality of the *comp sched* version causes its performance to degrade. In the *base* version, an access to a non-local element often prefetches the next element needed since each processor is updating a block of array columns (with the array allocated column-major). This effect does not occur as often in the versions with decomposition analysis because each processor is updating a two-dimensional $N/\sqrt{P} \times N/\sqrt{P}$ block, where N is the size of each array dimension and P is the number of processors. In the *base* version, each processor must communicate $2N/8$ cache lines, since 8 array elements (doubles) fit in one 64-byte cache line. In the versions with decomposition analysis, each processor must communicate $(2N/\sqrt{P} + 2N/(8\sqrt{P}))$ cache lines. For machines like the AlphaServer with small values of P and long cache lines, the computation to communication ratio is actually worse with two-dimensional blocks than with one-dimensional strips.

For the 1024×1024 problem size, the versions of the program without CDPC show similar behavior to the 512×512 problem size. The speedups are super-linear in the *base* version because the data set size is 16 MB (two arrays of doubles) which starts to fit in the 4 MB external cache at 4 processors (in the 512×512 problem size the data size is 4 MB and it fits into the external cache even on 1 processor). The speedups with CDPC are extreme because the coloring optimization lays out the data so that it fits exactly into each processor's cache at 4 processors. For the 2048×2048 problem size, however, the 32 MB working set just starts to fit into the caches at 8 processors, and coloring has little effect.

Vpenta. Figure 7.12 shows the results for vpenta. The performance of *comp sched* and *base* versions are the essentially the same on the AlphaServer, whereas on DASH

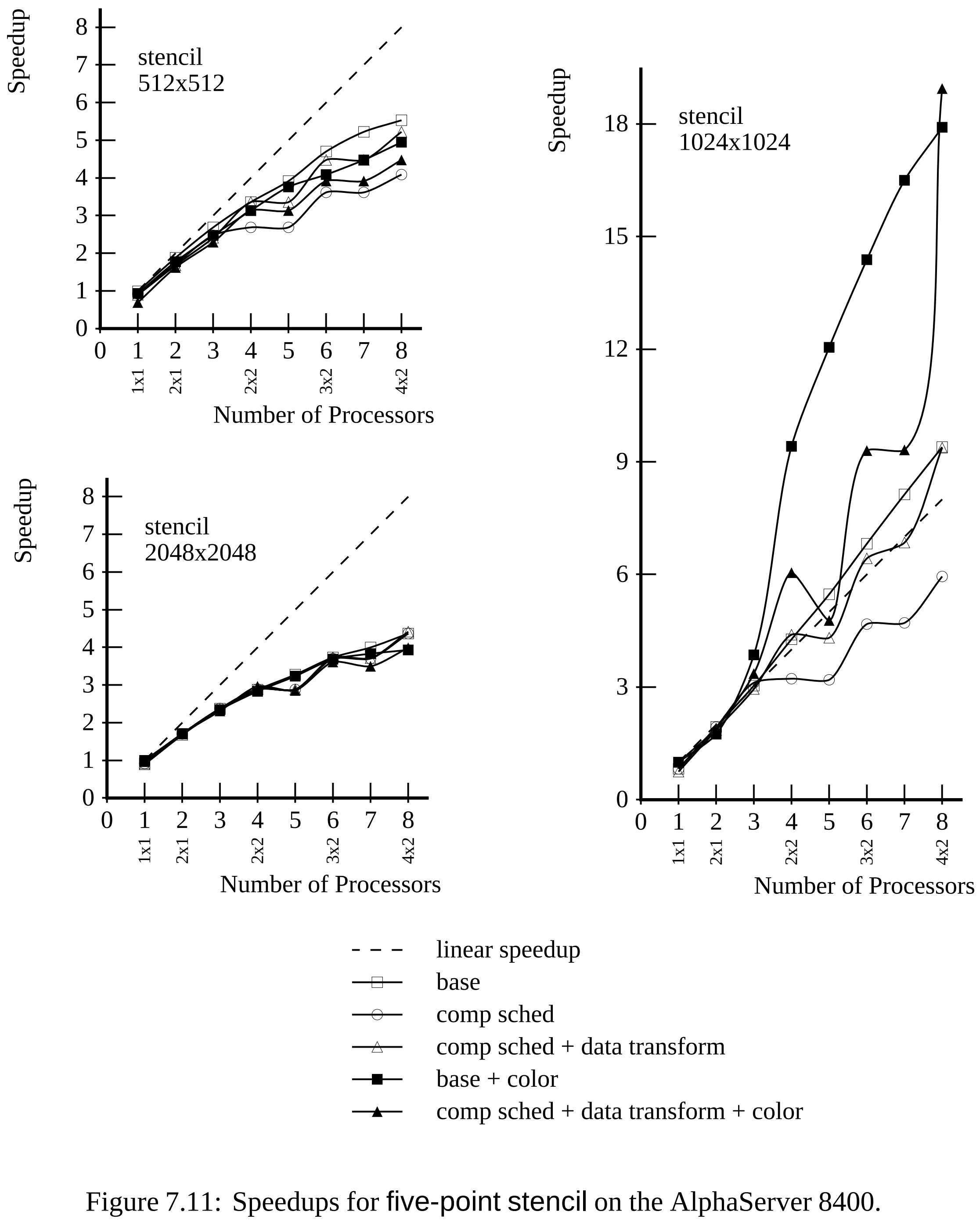


Figure 7.11: Speedups for five-point stencil on the AlphaServer 8400.

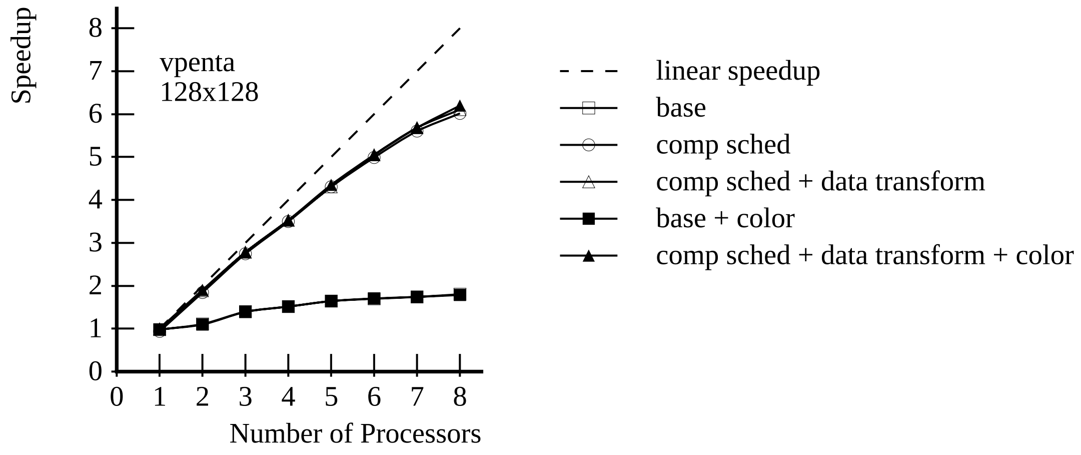


Figure 7.12: Speedups for vpenta on the AlphaServer 8400.

the *comp sched* version showed a slight performance increase due to the synchronization optimizations.

Applu. Figure 7.13 shows the results for applu. There are two key routines in applu that account for 54% of the sequential execution time on the AlphaServer. One routine iterates across planes of the arrays in a forward direction, and the other iterates across the planes in a reverse direction. Since the *base* version always partitions consecutive loop iterations across the processors, different processors access different data across these two routines. For example, if the block size is $b = \left\lceil \frac{N}{P} \right\rceil$ where N is the size of the array dimension and P is the number of physical processors, then in the forward routine processor 0 accesses planes $1 \ldots b$ whereas in the reverse routine it accesses planes $N \ldots (N - b + 1)$. In the *comp sched* version, however, the computation is scheduled so that each processor accesses the same data across the two routines. The resulting increase in temporal locality accounts for the performance gain of *comp sched* over *base*. The data transformations have no effect as the data accessed by each processor are already contiguous in the shared address space. CDPC combined with decomposition analysis, gives the best overall performance.

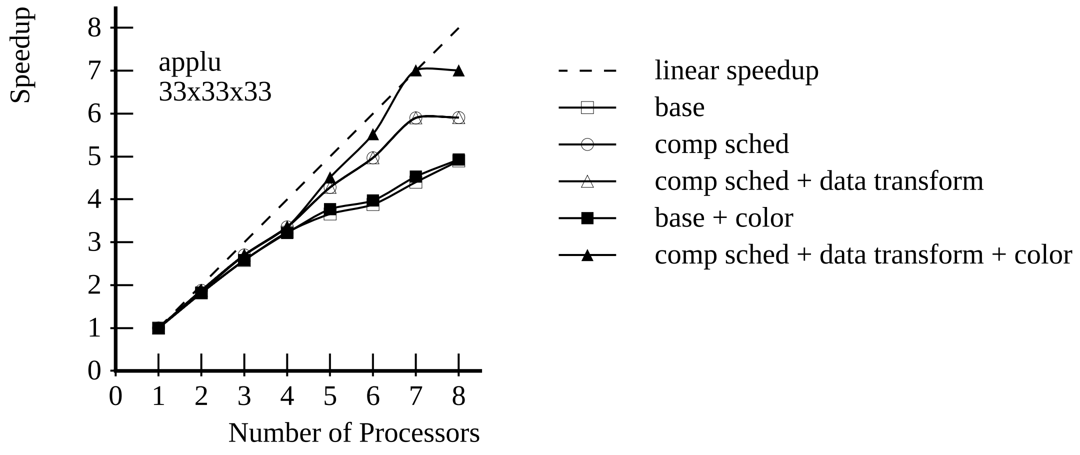


Figure 7.13: Speedups for applu on the AlphaServer 8400.

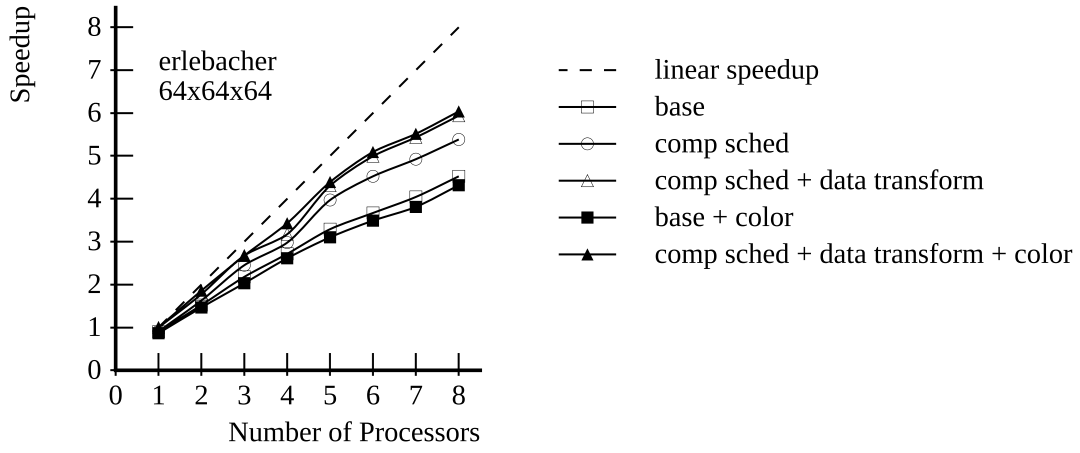


Figure 7.14: Speedups for erlebacher on the AlphaServer 8400.

Erlebacher. Figure 7.14 shows the results for erlebacher. The speedup curves are similar to those on DASH and coloring has little impact on the performance.

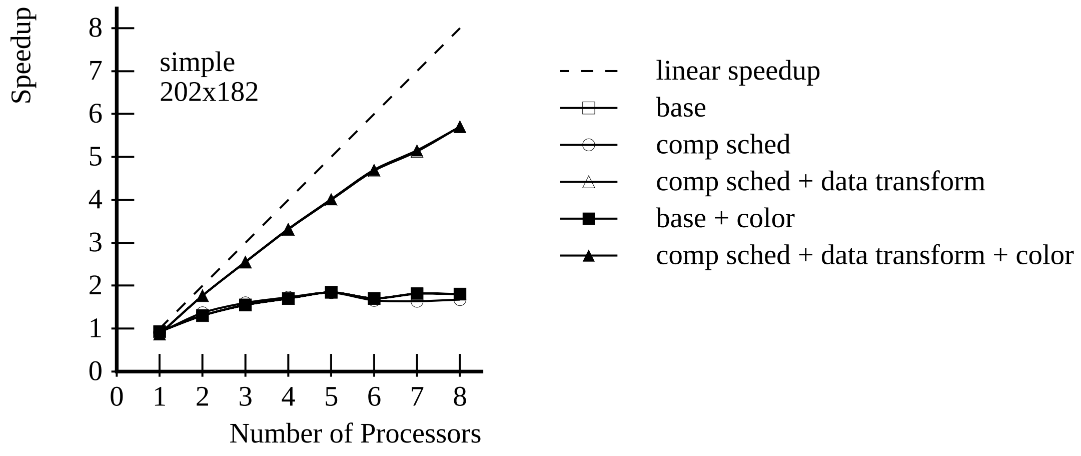


Figure 7.15: Speedups for simple on the AlphaServer 8400.

Simple. Figure 7.15 shows the results for simple. For this program, we modified the original version of the application (distributed as part of the RiCEPs benchmark suite) slightly. We performed *array renaming* on two arrays to eliminate unnecessary storage re-use across procedures (see Section 6.5.1). We also fused two loops that were nested inside an outer loop to create a perfectly nested loop. These modifications are automatable, though they are currently not implemented in the SUIF compiler (simple is the only application in the suite that we modified manually). The benchmark contains an ADI integration which does row sweeps and column sweeps across two-dimensional arrays. As was the case with the ADI integration kernel, the decomposition analysis uses *doall* parallelism in the row-sweep phase and switches to *doacross* parallelism in the column-sweep phase. Since the decomposition analysis partitions rows of the arrays across the processors (the program is written in FORTRAN and the arrays are allocated column-major), the data transformations are needed to realize any performance gains.

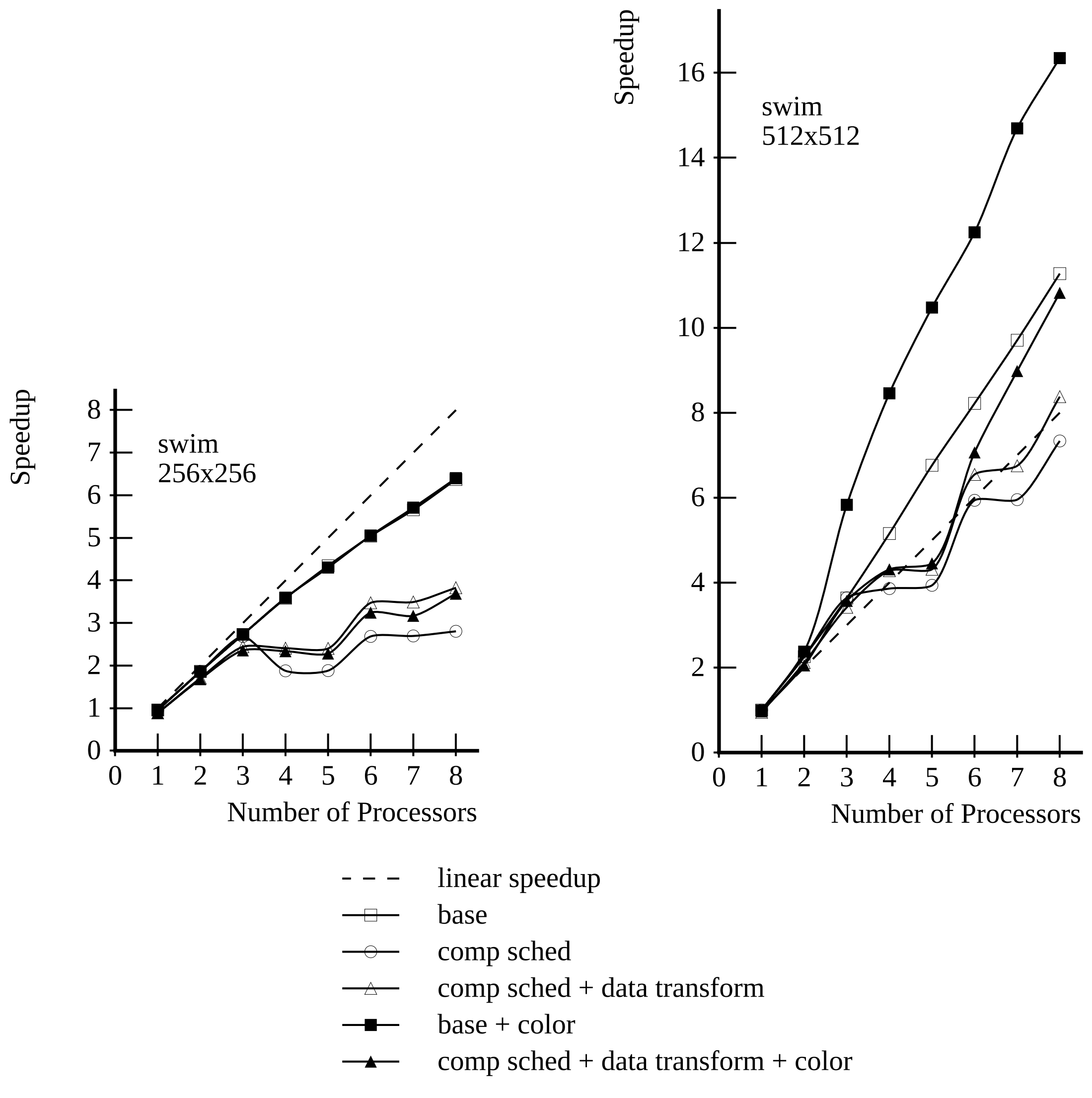


Figure 7.16: Speedups for swim on the AlphaServer 8400. The graph on the left is the SPEC92 version of the benchmark and the graph on the right is the SPEC95 version.

Swim. Figure 7.16 shows the speedups for the application swim on the AlphaServer. The behavior is similar to the behavior of the five-point stencil shown in Figure 7.11. Again, the *base* versions outperform the versions with decomposition analysis. The one-processor times for all versions are very close in performance, which indicates that the computation scheduling and data transformations are not introducing additional overhead. In this case, the computation to communication ratio problem is worse than for five-point stencil since the array elements are single-precision (4 bytes) rather than doubles. This means that in the *base* version, each processor must communicate only $2N/16$ cache lines for a dimension of size N, since 16 elements fit in one 64-byte cache line. In the versions with decomposition analysis, each processor must communicate $(2N/\sqrt{P} + 2N/(16\sqrt{P}))$ cache lines, where P is the number of processors.

Tomcatv. Figure 7.17 shows the results for tomcatv. The SPEC92 version (256 $\times$ 256 problem size) has the same behavior on the AlphaServer as on DASH. The SPEC95 version (512 $\times$ 512 problem size) shows super-linear speedup with coloring because the 14 MB working set starts fitting in the caches at 4 processors.

A summary of the experimental results for the AlphaServer are shown in Table 7.6. For each program we compare the speedups on 8 processors obtained with the base compiler against the speedups obtained with decomposition analysis together with computation scheduling and data transformations. For both the base and optimized numbers, we use the maximum speedup obtained either with or without compiler-directed page coloring. The table shows that decomposition analysis can lead to large improvements in application performance, even on a centralized shared address space machine such as the AlphaServer. Among the kernels, the optimized versions ran as much as 3.4 times faster than the base version for vpenta on 8 processors, however the 512 $\times$ 512 five-point stencil and the 2048 $\times$ 2048 LU decomposition slow down slightly. Among the applications, the optimized versions of 256 $\times$ 256 tomcatv and simple performed well, with 3.9 and 3.2 times improvement over the base version, respectively. The optimizations caused swim to degrade in performance for both problem sizes; the speedup for the optimized version is only 59% of the speedup obtained for the base version on the 256 $\times$ 256 problem size.

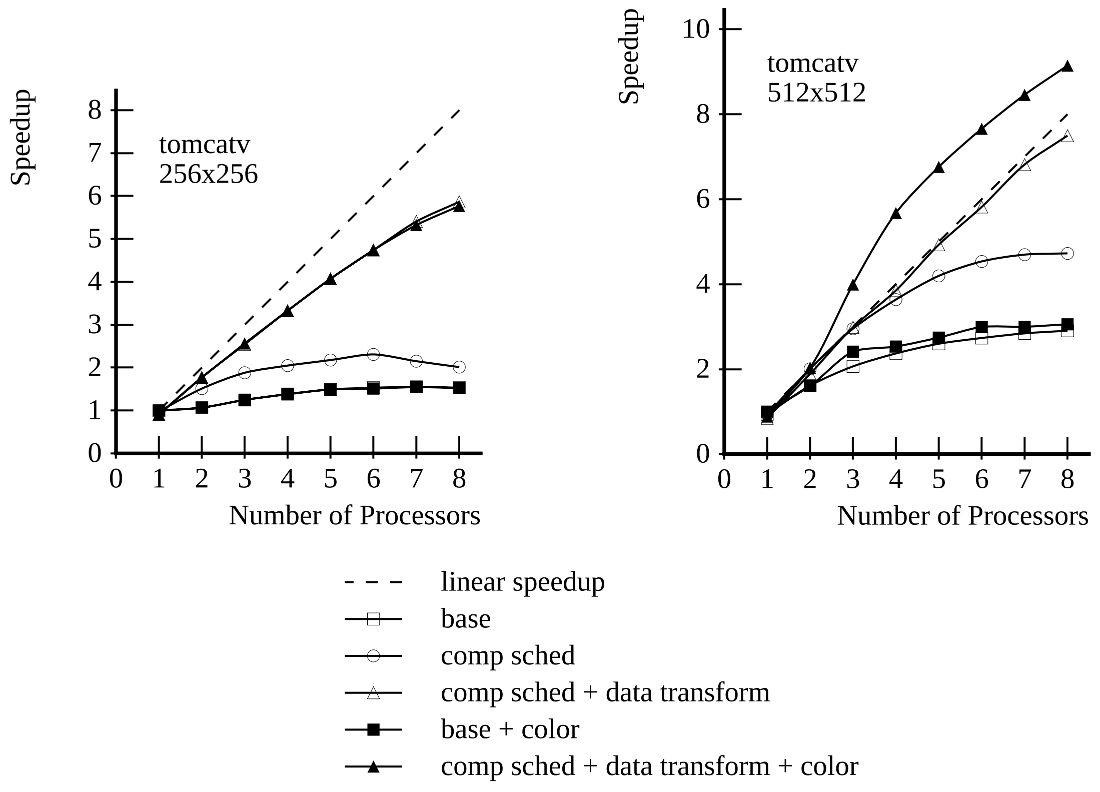


Figure 7.17: Speedups for tomcatv on the AlphaServer 8400. The graph on the left is the SPEC92 version of the benchmark and the graph on the right is the SPEC95 version.

7.2.3 Summary of Results

Our experimental results on DASH and the AlphaServer demonstrate that decomposition analysis can significantly improve application performance. The programs in our application suite are all highly parallelizable, but their speedups on both DASH and the AlphaServer were disappointing using only basic parallelization techniques.

On DASH, a distributed shared address space machines with non-uniform memory access times, good placement of the data and computation is often a pre-requisite to scalable performance. We also found that decomposition analysis can improve performance on centralized address space machines such as the AlphaServer. Even though the memory access times are uniform, the improved spatial locality and reduction in coherence traffic

Benchmark	Problem Size	Base	Optimized	Ratio
		Speedups (8 processors)		Optimized/Base
ADI integration	256 × 256	1.7	2.1	1.2
	512 × 512	2.8	4.0	1.4
	1024 × 1024	3.6	5.1	1.4
	2048 × 2048	4.3	4.7	1.1
LU decomposition	512 × 512	5.9	6.0	1.0
	1024 × 1024	9.8	10.2	1.0
	2048 × 2048	12.9	9.4	.73
stencil	512 × 512	5.5	5.2	.95
	1024 × 1024	17.9	18.9	1.1
	2048 × 2048	4.4	4.4	1.0
vpenta	128 × 128	1.8	6.2	3.4
applu	33 × 33 × 33	4.9	7.0	1.4
erlebacher	64 × 64 × 64	4.5	6.0	1.3
simple	202 × 182	1.8	5.7	3.2
swim	256 × 256	6.4	3.8	.59
	512 × 512	16.3	10.8	.66
tomcatv	256 × 256	1.5	5.9	3.9
	512 × 512	3.1	9.1	2.9

Table 7.6: Summary of results on the AlphaServer 8400. The table compares the 8-processor speedups obtained with the base compiler against the speedups obtained with decomposition analysis.

can lead to considerable improvements in performance.

We found that to realize all of the gains of the decomposition analysis, we need to perform both computation scheduling according to the specified computation decompositions and data transformations according to the data decompositions. In some cases, just using the computation decompositions without the data decompositions resulted in performance that was worse than the base case. We also found that decomposition analysis is useful for an additional optimization, compiler-directed page coloring.

Chapter 8

Conclusions

To achieve good performance on parallel systems, programs must make effective use of the computer's memory hierarchy as well as its ability to perform computation in parallel. A key performance issue is finding a good decomposition of the data and computation across the processors of the machine.

A popular approach to decomposition problem has been to use languages with data decomposition extensions, such as HPF. However, it is often difficult for a programmer writing sequential code to determine a good data decomposition for a program. Because the mapping of data and computation are so tightly coupled, the programmer must fully analyze the parallelism of the program to find the data decompositions. This is a challenging task for programmers that have little experience with parallel applications. Since the programmer does not relay the computation decomposition to the HPF compiler, the HPF compiler must re-derive the programmer's intended computation decomposition. This means that the user must have also have a good understanding of how the compiler calculates the computation decompositions to understand the resulting performance of the program.

The best solution is for the compiler to calculate both the computation and data decompositions automatically. Performing the decomposition analysis automatically not only frees programmers from doing the complex analysis themselves, but can also can lead to more efficient code. By calculating the data and computation decompositions at the same time, the compiler is able to model both the parallelization and communication inherent in the program. The compiler no longer has to infer the computation decomposition indirectly from the data decomposition. In the cases where the compiler is unable to fully analyze the code, it would be more useful for the programmer to supply the compiler with facts about the program (e.g. no arrays in this procedure are aliased), rather than specifying the data decompositions directly.

8.1 Contributions

In this thesis we have presented a new compiler algorithm that calculates decompositions for dense-matrix scientific codes. The contributions of this thesis are as follows:

Decomposition Framework. We have developed a linear algebra framework for expressing and calculating decompositions. This framework has several important properties. First, we can generate a system of equations that specifies the conditions the computation and data decompositions must satisfy, and then solve for the decompositions systematically. Second, we are not limited to an arbitrary set of possible decompositions. Finally, using the mathematical model allows us to succinctly represent the data and computation that are assigned to the same processor by the nullspaces of the decomposition matrices.

Decomposition Algorithm. Based on our mathematical decomposition framework, we developed a novel compiler algorithm that calculates data and computation decompositions. The algorithm is based on partitioning the program into static decomposition regions, regions of the program that have no data reorganization communication. Within the regions, the decompositions we find are optimal in that they are guaranteed to have the largest degree of parallelism with no data reorganization.

Within each region, we use the mathematical model to generate a system of equations that describes the decompositions. As the algorithm progresses, it gathers constraints on the nullspaces that must be satisfied in order for a solution to exist to the set of equations. Since we use the nullspaces directly to calculate the cost of a particular decomposition, we can solve incrementally as we merge loop nests into larger and larger static decomposition regions.

Ours is the first algorithm that calculates decompositions directly while simultaneously modeling the benefits of parallelization and the cost of communication. Decomposition algorithms that are modeled after languages such as HPF and have separate alignment and distribution phases, face the problem that alignment and distribution are inter-related. We avoid this problem by solving directly for the affine decompositions. Our algorithm also handles dynamic data reorganization, and incorporates replication and synchronization.

Interprocedural Decompositions. We developed the first decomposition algorithm that performs interprocedural analysis. Any decomposition algorithm that handles realistic programs must be able to analyze across procedures. Otherwise, if the data decompositions of arrays do not match across procedure boundaries, then the program could potentially incur large amounts of communication at every procedure call entry and call return. Also, interprocedural analysis is needed to find decompositions for parallel loops that contain procedure calls.

Our interprocedural decomposition algorithm succinctly summarizes all the necessary information on decompositions within a procedure. It does not need to re-solve for the decompositions in a procedure each time that procedure is called. We also identified several program characteristics that cause a loss of precision when performing interprocedural decomposition analysis.

Implementation and Evaluation. We implemented our decomposition algorithm as part of the SUIF compiler system, and showed how decomposition analysis fits into the design of a complete parallelizing compiler. We evaluated the effectiveness of our algorithm by applying it to a suite of benchmark programs. We ran the compiler-generated code on two different architectures, a distributed shared address space machine and a centralized shared address space machine.

Our experimental results show that our decomposition algorithm improves program performance by as much as a factor of four on both of these machines. In many cases, decomposition analysis is a pre-requisite for achieving scalable performance. Decomposition analysis can also enable additional optimizations on shared address space machines, including synchronization optimizations and compiler-directed page coloring.

8.2 Future Work

Our compiler currently splits the parallelization analysis, decomposition analysis, data transformations and uniprocessor locality analysis into separate phases (see Chapter 2). One drawback to this organization is that the decomposition analysis is restricted by the loop nest structure of the program. We operate on code within loop nests and treat the statements within a single iteration as an indivisible unit. This limits the range of decompositions that our analysis can find; for example, we cannot assign different computation decompositions to two statements in the same loop nest. Another drawback to the current compiler organization is that uniprocessor optimizations happen only after all the multiprocessor transformations are completed. However, there may be cases where between two decompositions that are equivalent in terms of parallelism and communication, one is preferable in terms of uniprocessor locality. To address these issues, tighter integration is needed between the different passes. The compiler also needs to model the effects of data and computation transformations on both uniprocessor and multiprocessor performance.

Bibliography

[1] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, C.-W. Tseng, and S. Warren. Requirements for data-parallel programming environments. *IEEE Parallel and Distributed Technology*, 2(3):48–58, Fall 1994.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, June 1995.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[4] E. Albert, K. Knobe, J. Lukas, and G. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[5] J. R. Allen and K. Kennedy. A parallel programming environment. *IEEE Software*, 2(4):22–29, July 1985.

[6] S. P. Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities*. PhD thesis, Dept. of Electrical Engineering, Stanford University, January 1997.

[7] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on*

Programming Language Design and Implementation, pages 126–138, Albuquerque, NM, June 1993.

[8] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, Williamsburg, VA, April 1991.

[9] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 166–178, Santa Barbara, CA, July 1995.

[10] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993.

[11] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *Computing Surveys*, 26(4), December 1994.

[12] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, Paris, France, December 1996.

[13] P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10), October 1995.

[14] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.

[15] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill. Solving alignment using elementary linear algebra. In *Languages and Compilers for Parallel Computing, Seventh International Workshop*, volume 892. Springer-Verlag, 1995.

[16] B. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, Montreal, Canada, August 1994.

[17] T. Blank. The MasPar MP-1 architecture. In *Proceedings of the 1990 Spring COMPCON*, San Francisco, CA, February 1990.

[18] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 244–257, Cambridge, MA, October 1996.

[19] D. Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Dept. of Computer Science, Rice University, March 1987.

[20] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[21] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[22] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Array distribution in data-parallel programs. In *Languages and Compilers for Parallel Computing, Seventh International Workshop*, volume 892, pages 76–91. Springer-Verlag, 1995.

[23] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, pages 16–27, Charleston, SC, January 1993.

[24] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Pappworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 180–192, October 1987.

[25] B. Creusillet and F. Irigoin. Interprocedural array region analyses. In *Languages and Compilers for Parallel Computing, Eighth International Workshop,* volume 1033. Springer-Verlag, 1996.

[26] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiway cuts. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, May 1992.

[27] J. H. Edmondson et al. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal,* 7(1), 1995. Special Edition.

[28] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Parallel Processing,* pages 377–381, St. Charles, IL, August 1991.

[29] S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 257–270, Boston, MA, April 1989.

[30] D. M. Fenwick, D. J. Foley, W. B. Gist, S. R. VanDoren, and D. Wissell. The Alphaserver 8000 series: High-end server platform development. *Digital Technical Journal*, 7(1), 1995. Special Edition.

[31] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[32] S. Frank, H. Burkhard III, and J. Rothnie. The KSR-1: Bridging the gap between shared memory and MPPs. In *Proceedings of the 1993 Spring COMPCON*, pages 285–294, San Francisco, CA, February 1993.

[33] J. Garcia, E. Ayguadé, and J. Labarta. A novel approach towards automatic data distribution. In *Proceedings of Supercomputing '95,* San Diego, CA, December 1995.

[34] J. R. Gilbert, S. Chatterjee, and R. Schreiber. Mobile and replicated alignment of arrays in data-parallel programs. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[35] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, College of Engineering, University of Illinois at Urbana-Champaign, September 1992. UILU-ENG-92-2237, CRHC 92-19.

[36] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[37] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[38] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S.-W. Liao, and M. S. Lam. Interprocedural analysis for parallelization. In *Languages and Compilers for Parallel Computing, Eighth International Workshop*, volume 1033. Springer-Verlag, August 1996.

[39] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[40] W. Hillis. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985.

[41] C.-H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. *Journal of Parallel and Distributed Computing*, 19(2):90–102, October 1993.

[42] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[43] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[44] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[45] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[46] K. Knobe, J. Lukas, and M. Weiss. Optimization techniques for SIMD Fortran compilers. *Concurrency: Practice and Experience*, 5(7):527–552, October 1993.

[47] K. Knobe and V. Natarajan. Automatic data allocation to minimize data motion on SIMD machines. *The Journal of Supercomputing*, 7:387–415, 1993.

[48] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[49] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[50] U. Kremer. *Automatic Data Layout for Distributed Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, October 1995. CRPC-TR95-559-S.

[51] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[52] D. Kulkarni, K. Kumar, A. Basu, and A. Paulraj. Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[53] K. G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hierarchical parallel machines in polynomial time. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 82–91, Washington, DC, July 1992.

[54] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21th International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

[55] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–105, Gold Coast, Australia, May 1992.

[56] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers '90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[57] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[58] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(2):213–221, October 1991.

[59] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[60] D. J. Palermo and P. Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *Languages and Compilers for Parallel Computing, Eighth International Workshop*, volume 1033. Springer-Verlag, 1995.

[61] J. Palmer and G. Steele, Jr. Connection Machine model CM-5 system overview. In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[62] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[63] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[64] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.

[65] M. Schlansker and M. McNamara. The Cydra 5 computer system architecture. In *Proceedings of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '88)*, October 1988.

[66] Steven L. Scott. Synchronization and communication in the T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 26–36, Cambridge, MA, October 1996.

[67] T. J. Sheffler, R. Schreiber, J. R. Gilbert, and S. Chatterjee. Aligning parallel arrays to reduce communication. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 324–331, McLean, VA, February 1995.

[68] T. J. Sheffler, R. Schreiber, W. Pugh, J. R. Gilbert, and S. Chatterjee. Efficient distribution analysis via graph contraction. In *Languages and Compilers for Parallel Computing, Eighth International Workshop*, volume 1033. Springer-Verlag, 1996.

[69] `http://www.sgi.com/hardware/servers/technology.html`. Silicon Graphics web site.

[70] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[71] J.P. Singh, T. Joe, A. Gupta, and J. L. Hennessy. An empirical comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors. In *Proceedings of Supercomputing '93*, pages 214–225, Portland, OR, November 1993.

[72] G. Strang. *Linear Algebra and Its Applications*. Harcourt Brace Jovanovich, Orlando, FL, Third edition, 1988.

[73] C-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–155, Santa Barbara, CA, July 1995.

[74] P.-S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1989.

[75] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[76] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[77] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, August 1992. CSL-TR-92-538.

[78] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[79] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[80] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

[81] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[82] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[83] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, New York, NY, 1991.