DYNAMIC MEMORY MANAGEMENT FOR THE LOCI FRAMEWORK

By

Yang Zhang

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2004

DYNAMIC MEMORY MANAGEMENT FOR THE LOCI FRAMEWORK

By

Yang Zhang

Approved:

Edward A. Luke Assistant Professor of Computer Science and Engineering (Major Professor)	Eric Hansen Associate Professor of Computer Science and Engineering (Committee Member)
Yoginder Dandass Assistant Professor of Computer Science and Engineering (Committee Member)	Susan M. Bridges Professor of Computer Science and Engineering, and Graduate Coordinator, Department of Computer Science and Engineering

A. Wayne Bennett
Dean of the College of Engineering

Name: Yang Zhang

Date of Degree: May 8, 2004

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Edward A. Luke

Title of Study: DYNAMIC MEMORY MANAGEMENT FOR THE LOCI FRAMEWORK

Pages in Study: 94

Candidate for Degree of Master of Science

Resource management is a critical part in high-performance computing software. While management of processing resources to increase performance is the most critical, efficient management of memory resources plays an important role in solving large problems. This thesis research seeks to create an effective dynamic memory management scheme for a declarative data-parallel programming system. In such systems, some sort of automatic resource management is a requirement. Using the Loci framework, this thesis research focuses on exploring such opportunities. We believe there exists an automatic memory management scheme for such declarative data-parallel systems that provides good compromise between memory utilization and performance. In addition to basic memory management, this thesis research also seeks to develop methods that take advantages of the cache memory subsystem and explore balances between memory utilization and parallel communication costs in such declarative data-parallel frameworks.

DEDICATION

in memory of my grandma

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Ed Luke, for his time, suggestions, and guidance. He was very patient in explaining various difficult concepts and helped point me in the right direction when I was in hard times. It has been pleasure to work with him. Thanks to Eric Hansen and Yogi Dandass for serving on my thesis committee and the comments and suggestions they made on my work. Thanks to JunXiao Wu for kindly providing the FUELCELL program. Thanks to Ed Allen for his thesis template, which made my life much easier.

I would also like to thank the Department of Computer Science and Engineering and the Engineering Research Center for providing financial support and facilities to my thesis research. This thesis was created using LaTeX 2_ε, GNU EMACS, Tgif, and Xmgrace. I would like to express my appreciation to their authors for making these nice tools freely available.

Finally special thanks to my parents, for their love and continuous support.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Background

Numerical simulation is becoming increasingly important. We use simulations to advance our understandings of scientific theories or to improve our capabilities in modern engineering design. Most of these numerical modeling problems are inherently complex. Due to the vast computing requirements of these simulations, supercomputers are often needed to gain better and more realistic results. With the recent advances in microprocessor power and high speed interconnection networks, distributed parallel systems based on groups of networked workstations have become available. They are often called *clusters*. Some of these clusters (e.g., the Beowulf project [15]) are starting to compete with the largest traditional supercomputing machines while offering a far better performance-cost ratio. This has made numerical modeling much more economically feasible than ever before.

Unfortunately, while the hardware cost is dropping radically, the software cost required to utilize these resources is still substantial. Often, programming these clusters requires message-passing-based paradigms, which are usually tedious and error prone compared to traditional sequential programming. MPI is now the *de facto* standard for distributed mem-

ory programming. Significant research has been conducted in developing portable and reusable parallel code. Object-orientation has been a trend in recent numerical software. The development of high-performance value classes [14, 17, 10] tends to provide better abstractions for common mathematical constructs, while the development of application toolkits [3] tends to reduce the complexities of coordinating loosely coupled components in an application.

1.1.1 The Loci Framework

Loci [12] is a programming framework for high-performance computational field simulations. It seeks to reduce the complexity and cost associated with developing large-scale scientific applications, such as computational fluid dynamics software. The design of Loci realizes that in developing large applications, a significant portion of complexity and the bugs are from the errors in incorrect looping structures, improper calling sequences, or incorrect data movements. Loci eliminates such internal inconsistencies by using a run-time deduction engine that generates the application control structure and data movement operations automatically from component specifications.

Loci is a declarative programming framework. It changes the way that numerical software is specified. In the framework, users do not need to explicitly construct a program. They only provide descriptions of attributes (data) and the transformations between attributes in terms of “rules,” as in logic programming. Then they query a particular result

similar to a database query. Loci will automatically derive a machine-executable program that satisfies the users' request.

1.2 The Problems

In addition to the ease of application construction and guaranteed internal consistency, automatic parallelization is another great strength of Loci. The underlying numerical model does not have to refer to any explicit parallel execution. This is a natural "side effect" of the Loci approach. It essentially demonstrates Loci's capability to do intra-application resource management.

Loci is targeted at numerical software. The type of problems modeled by these software are usually complicated and large (in terms of computational resource). Parallel processing is used not only to speed up computing, but also to gain enough of the memory required by the simulation. Thus, besides computational power utilization, memory utilization poses another challenge in numerical simulation.

As mentioned before, the software costs to utilize these computational resources are substantial. Loci is a novel framework that reduces the software cost dramatically. However, Loci does not presently address the memory problem, despite of its ease to perform resource management. Loci currently uses a naive preallocation scheme. In this preallocation scheme, all variables are allocated in advance before the program starts and are deleted at the end of the program. Therefore the lifetime of every variable is equal to the lifetime of the program. Part of the reason for using preallocation is that it tends to min-

imize the computational cost associated with allocation, allowing for high performance. But from the memory utilization point of view, preallocation yields maximum memory bound, which is not efficient. So the first question we have is: Is it possible to do memory management that provides reasonable compromises between memory and performance? In other words, can we utilize the memory resources in a declarative framework on the same system to solve larger problems than before? Can we achieve this goal without unduly impacting execution time?

As the gap of processor and memory speed grows larger and larger, cache is becoming increasingly important in performance critical applications. Many theories and methods [5, 11, 13] have been devised to increase cache performance. Loci is a declarative framework, meaning users do not have direct control over resources. Thus, we have several questions concerning the cache: Can we identify the cache aspects for scientific applications in a declarative framework? Do we have a good management policy in a declarative framework to increase the cache performance for scientific applications? Are there any relationships between memory management and cache utilization? In other words, do they support each other, or do they conflict with each other?

We anticipate that memory management will introduce extra costs into the Loci framework. But parallel computing also has inherent overheads, such as communications. We would also like to explore the relations between memory management and parallel overheads. Does optimizing memory management require changing the parallel computation schedule generated by Loci? What impact do these changes have on performance? Mem-

ory management in a declarative framework certainly introduces overheads, but can we gain other rewards by having a good memory management scheme?

Finally, can we characterize the common patterns of memory utilization in scientific applications so that we can further our understanding of memory management in general and have insights into possible future research projects?

1.3 The Hypothesis and Goal

The hypotheses of this thesis research are:

- In a declarative data-parallel programming system such as Loci, some sort of automatic resource management is a requirement. We believe that there exists an automatic memory management scheme for such declarative systems that provides a good compromise between memory utilization and performance. Specifically, we claim that such an allocation scheme will have reasonable run-time overhead compared to the preallocation strategy while also providing relatively significant improvements in memory utilization.
- We also claim that through careful resource management and utilization, the execution speed of applications using dynamic memory management scheme will outperform applications using preallocation strategy. We make such claim with the anticipation that the allocation scheme will take advantage of cache memory subsystems in a manner that is not possible with preallocation.
- Additionally, we claim that there exists performance trade-offs between memory utilization and communication costs in data-parallel programming systems. Due to these trade-offs, a balanced approach will require interactions between the memory and communication scheduling strategies.

The goals of this thesis research are:

- We aim to design an efficient and effective memory management scheme for the Loci framework. Particularly, we want to reduce the peak memory requirement of an application built using the Loci framework so that larger problems can be solved on the same system.

- We want to extend Loci's intra-application resources management ability to include the cache. And we want to evaluate the possible effects of cache optimizations in the Loci framework.
- We want to study the possibility of incorporating some static and run-time policies into the Loci framework. We seek to improve Loci's adaptability so that users or Loci itself can choose or switch to more appropriate actions under different circumstances.

Loci provides an ideal platform for testing and evaluating some of the ideas for scientific application memory management. We also hope that through this research we can achieve a better understanding of designing declarative frameworks for numerical software.

CHAPTER II

RELATED WORK

The memory system is a central part of modern computer architectures. It has been studied extensively in the past decades. Memory is used to store program instructions and data. Any system has a limited amount of memory available, thus the efficient utilization and management of the memory is important. Memory management can be traced to the hardware and operating system level, where the actual physical devices are more concerned. From the application program's point of view, memory management supplies the amount of space needed by the application and recycles memory that is no longer needed. Thus memory management at the application level involves allocation and recycle. This section discusses some of the techniques that have been developed and how they relate to Loci memory management.

2.1 Allocation Technique

Generally, allocation is implemented as a library, such as `malloc` in C. It is often referred to as an *allocator*. Usually at run time, the allocator receives large blocks of memory from the operating system directly. Then the allocator partitions the memory, supplies the

partitions for program requests, and later recycles them. Usually these storages are allocated at the “heap.”

In conventional allocators, once a block of memory has been allocated, the allocator cannot move it or copy the contents to other places (compact memory). Also application programs can request an arbitrary size memory at an arbitrary time and return the memory at any time later. Thus, fragmentation poses a serious problem for allocators. Often, there are external fragmentation and internal fragmentation. External fragmentation refers to allocator’s inability to grant requests from application programs, although there is enough total free memory. Because all the free blocks are small and are scattered, no one free block is large enough to satisfy the request. If the allocator supplies too large of a block to a request, then the rest of the memory in that block cannot be used by others, causing internal fragmentation.

Therefore, the typical techniques used to design allocators are to choose a good place for allocation and to have a good recycling management. It would be ideal not to waste space and time, but, in general, this is not possible to achieve because the application program behaviors are hard to predict and the allocator has to deal with general programs. Therefore, heuristics are often used in the allocator to guide the placement policy. There are many techniques and placement strategies, each with its own strength and weakness. Recycling is also similar to placement. Often coalescing, merging fragmented memory segments, is used to combine small free blocks into larger ones. But one has to make some

trade-offs in determining when to coalesce. These algorithms are surveyed in Wilson et al. [19].

Locality is another problem to be considered in designing an allocator. Cache and page misses can sometimes greatly affect the program performance. Some dynamic storage allocation algorithms are designed to be aware of the locality problem. The cache performance of various allocation algorithms are studied in Grunwald et al. [9].

A conventional general purpose allocator cannot always perform well for all applications. As a consequence, special purpose, or custom, allocators are often built for a particular type of program. Frameworks [2, 4] are sometimes used to build custom allocators. Custom allocators often take advantage of domain-specific knowledge or certain patterns in the allocation and can be designed to have an extremely low cost.

A widely used technique for optimizing dynamic memory allocation is to use regions. In a region based allocator, objects are often grouped into a specified region. The region is allocated once, and inside it, object allocation is managed through simple pointer manipulations. Objects in a region cannot be freed individually; instead the whole region is destroyed. Region-based memory management often results in good localities and flexible policies. It is possible to use different allocators on different regions, and even garbage collection may be used on some regions. Gay and Aiken [7, 8] discussed adding region support into languages directly.

2.2 Manual Memory Management

The allocator is a low level design concern — rarely do programmers care about it. From the programmers' point of view, memory management is more of a strategy. Either they explicitly manage memory, or the system automatically handles the management.

With manual memory management, the programmer has direct control over memory recycling. This is done explicitly by using `free` or `dispose` statements in the program. The benefits of this approach are clear: Programmers gain direct control over the memory recycling, they have clear ideas of the whole picture, and sometimes this is more efficient.

But in general, it is hard to manage memory explicitly. As the program grows larger and larger, managing memory becomes complex. It is hard to keep track of it. Thus, bugs could be easily introduced into the program and are then hard to find. The most common errors are known as *dangling pointer* and *memory leak*. A dangling pointer occurs when memory is recycled too early, while the memory leak occurs when memory is not recycled. Moreover, this manual approach does not scale well and does not encourage modular programming in general. Because a large part of the code is used to handle memory management, component interfaces are often polluted by irrelevant and complex memory management constraints.

2.3 Automatic Memory Management

Automatic memory management is the opposite of manual memory management. It is often a system service and is a general technique that automatically recycles useless

memory. Thus programmers are freed from bookkeeping details and can concentrate on the fundamental programming requirements.

Garbage collection is the most prevalent automatic memory management technique. In this technique, useless memory is considered *garbage* and is automatically recycled by the run-time system. Although in general it is undecidable whether an object is garbage or not, in practice, approximations are often effective. Garbage collection is often incorporated into programming languages since the object's layout and roots are needed by the run-time system. Many modern programming languages support garbage collection, such as *Java*, *ML*, *Smalltalk*, etc. Some languages like *C* and *C++* use manual management, but they also have conservative garbage collection extensions.

In garbage collection, the *garbage collector* runs periodically to reclaim useless memory. Tracing or reference counting are often used to distinguish live and dead objects. More advanced techniques like incremental collection and generational collection [18] are also being used and studied.

Region inference is another form of automatic memory management. It is a relatively new technique compared to garbage collection. It was proposed by Tofte et al. [16] and was implemented and studied in the *ML kit* compiler. Instead of relying on run-time garbage collection, region inference relies on static program analysis. It is a compile-time method and uses the region concept. The compiler analyzes the source program and infers how many regions are needed, where they should be allocated and deleted, and to which region each allocation should be bound. The region lifetime obeys stack discipline,

thus eliminating the need for garbage collection. In addition to being fully automatic and safe, this approach also eliminates the run-time overhead of memory management as in the garbage collection. In Aiken et al. [1], the stack restriction of region lifetime has been removed by solving a constraints problem. But in general, this static method is sensitive to the program style: A small change in the source program may result in significantly different inference of memory management. To date, this technique is only available for typed, high-order, call-by-value language ML because ML's clean semantics and the strong typing system made the inference possible.

2.4 Summary

This thesis focuses on the memory management strategies for the Loci framework, rather than on low level designs such as customize allocators. Loci is a coordination framework, choosing and designing a custom strategy is more important than focusing on custom allocators. They are issues on different levels: Custom allocators are subsidiaries of the management strategy that fine tune memory layout and fragmentation issues. Whereas Loci requires larger scale assembly management more similar to region inference and garbage collection.

The goal and programming style of Loci makes manual memory management either impossible or inadequate. It complicates the design of the program, thus contradicting Loci's primary goal. Also, the declarative programming style means memory cannot be directly controlled.

Loci targets scientific software. Scientific programs tend to use all of the available memory. Thus the primary goal of the strategy is to reduce the peak memory required so that larger problems can be solved on the same system, enabling more realistic simulations. Reducing the peak memory requires reducing the memory usage bound. Techniques like garbage collection tend to have poor predictability because the allocation and deallocation are decoupled. The recycling process completely relies on run-time decisions, and the garbage collector may also consume additional space and time. Therefore, the memory bounds are hard to guarantee. Loci manages and assembles the application components. It can easily perform global analysis to determine relationships between computations and variable lifetimes. Thus, coupling the allocation and deallocation is possible. This is important to obtain more stable memory bound and to reduce run-time management overhead.

The candidate strategy is more like the region inference method. It is automatic at the application level and the framework “infers” appropriate allocations and recycles. Regions are not used in Loci because Loci is mainly a coordination framework. It operates on collections of entities and assembles different components together. It does not require fine-grained allocation.

CHAPTER III

THE LOCI FRAMEWORK

This chapter details some design principles, internal structures, usages, and characteristics of the Loci framework.

3.1 Elements of Loci

Loci is a framework to build high-performance scientific applications. Loci coordinates and assembles applications from component specifications. The fundamental programming paradigm of Loci is declarative.

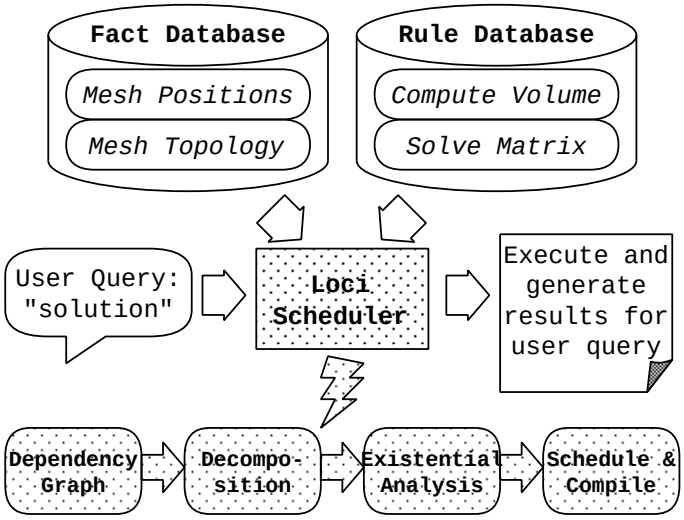

Figure 3.1 Loci Architecture

The general system architecture of the Loci framework can be characterized in Figure 3.1. Loci uses two databases, the *Fact* and *Rule* database, to store and manage the descriptions and specifications from user. The fact database maintains the data for the application, and the rule database maintains the transformations between facts. The central part of the system is the *Loci scheduler*, which is a deductive engine that automatically derives application control flow, data movement, and aggregates computations. The shaded parts in Figure 3.1 depict the major internal steps of the scheduler. From the user's point of view, developing an application using Loci is to build and maintain the fact and rule databases; executing the application is to supply a query to Loci.

The following sections discuss each part in detail.

3.2 The Loci Data Model

The fundamental concept in Loci is *entity*. Entities represent sites where computations may occur. For example, in a triangular mesh, a single entity may represent a triangle in the mesh, or an edge, or a node. Loci automatically aggregates computations on entity collections, therefore abstractions of data are also on the level of entity collections. There are basically four types of data models in Loci. The *store* construct provides an injective mapping from entities to values. The *parameter* construct maps a collections of entities to a single value. The *map* construct provides a way to model the relations between entities. The *constraint* construct maps an attribute onto a subset of entities, which is then used to

constraint computations on that subset of entities. Figure 3.2 illustrates the concept of the four basic constructs.

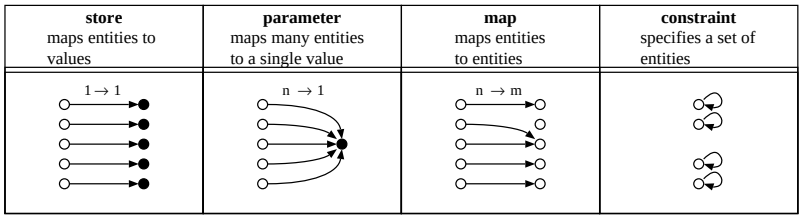

Figure 3.2 Four Basic Database Constructs

These constructs are used to formulate the fact database that describes the problem. Each fact provides some information of a subset of entities (i.e., collection of entities), such as the positions of nodes, or maps relating triangles to edges. Each of these facts is given an identifier in the fact database, thus facts are also referred to as *variables*. An identifier consists of a name and an optional pair of curly braces with iteration[1] information inside it. The general form of an identifier is $\alpha\{\tau + \theta\}$ where α is the name and represents the attribute of the fact, τ is the iteration level and θ is the iteration offset. For example, `energy{n+1}` represents the attribute `energy` at the next iteration of iterator `n`.

The store and map constructs can be think of as array-like containers, where store holds values for entities and map holds the related entities for entities. Accessing the associated value for an entity in stores and maps may be coded using C++ array access as `energy[e]`, where `e` represents an entity. The map construct can also be com-

[1] See section 3.3.4 for information on iteration specification.

posed with the store construct to provide an abstraction of indirection. For example, `energy[left[e]]`, in which, `energy` is a store and `left` is a map contains the left neighbor of an entity (e.g., a cell) in a mesh. This means we are indirectly accessing the store `energy`. In Loci, this access of `energy` through the mapping `left` is denoted as `left`→`energy`. The "→" operator is used to represent the composition of maps and stores.

3.3 Rule Specifications

In addition to the fact database, the rule database describes transformations between facts that are used by Loci to deduce new facts and to infer program control flows that lead to the user query. These rules of transformation are functions or algorithms that operate on facts or attributes and generate new facts or attributes, such as rules for evaluating the areas of faces, or for solving equations. These rules correspond to the fundamental steps in computing the final result. In Loci, rules are denoted using character strings called "rule signature." Rule signature has the general form `head`←`body`, where `head` and `body` are lists of facts. This means the facts in `head` are generated by the application of this rule, and the facts in `body` are accessed during the evaluation of the rule. For example, the rule signature `area`←`face2node`→`position` represents that the value for `area` can be inferred provided when facts `face2node` and `position` are present. Note, here `face2node` is a map that connects faces to their defining nodes. Therefore in the body of the rule signature, we are accessing the position of nodes that defining a face. An

important implication of a rule is that it can be applied to any given entity if the conditions (i.e., attributes) in the body are met on the entity.

Loci categorizes different types of computations in scientific computing into rule classes. Analogous to the constructs in the fact database, each rule class has its own semantic meaning and provides a template to formulate the rule database. In essence, rule class defines the composition and application of rules over a collection of entities.

3.3.1 Point-wise Rule Class

Point-wise rule class is the most common computation in Loci applications. The point-wise rule class represents an entity by entity computation of attributes. The rule is applied on each entity. Point-wise computation produces new facts (or attributes). The new facts are usually store constructs associated with the collection of entities that the computation was applied on. The semantics of point-wise computation requires that an output fact can only define one value per entity. It is treated as an error if more that one rules compute the same attribute for the same entity. Recursion is allowed in point-wise computation, provided that the semantics are not violated. Point-wise computation can be described as:

$$f = r(e_1) \wedge r(e_2) \wedge \cdots \wedge r(e_i) \wedge \cdots \wedge r(e_n) \tag{3.1}$$

In which, r is the rule that applies on entity; $\{e_i \mid i \in [1, n]\}$ is the collection of entities that the computation operates on; $\wedge$ means a single evaluation is independent of others; f is the resulting store fact that has domain $\{e_i \mid i \in [1, n]\}$.

Note that parallel point-wise computation can be performed, provided that each process has a subset of entities $\{e_p \mid p \in [j,k]\}, 1 \leq j \leq k \leq n$. Each process performs a subset of point-wise computation:

$$f_p = r(e_{p_1}) \wedge r(e_{p_2}) \wedge \cdots \wedge r(e_{p_n}) \tag{3.2}$$

Then the final fact can be obtained by:

$$f = f_1 \cup f_2 \cup \cdots \cup f_i \cup \cdots \cup f_p \tag{3.3}$$

3.3.2 Singleton and Parameter Rule Class

The singleton rule class is a special case of the point-wise computation. Since the collection of entities share the same attribute value, rules only need to be applied once. The parameter construct is used to map entities to the attribute value. Singleton computation can be described as:

$$f = r(e_i) \tag{3.4}$$

The definitions are the same as those in point-wise computation, except only exactly one computation is performed. The resulting fact f is a parameter that has domain $\{e_i \mid i \in [1,n]\}$. The singleton computation can also be parallelized. Since there is only one computation, each process just duplicates the computation and the resulting parameter fact.

3.3.3 Reduction Rule Class

Reduction defines another computation abstraction. In reduction computation, in addition to the point-wise computation, all resulting attribute values are "joined" together to produce the final value. Reduction computation can be described as:

$$f = \epsilon \oplus r(e_1) \oplus r(e_2) \oplus \cdots \oplus r(e_i) \oplus \cdots \oplus r(e_n) \tag{3.5}$$

In addition to the previous definition, $\oplus$ is an associative and commutative operator that is defined on the type of attribute returned by r; ϵ is an identity element of the operator $\oplus$; f[2] is the resulting fact that has domain $\{e_i \mid i \in [1, n]\}$.

Reduction can also be evaluated in parallel other than left to right sequential evaluation because of the associative property of $\oplus$. Parallelization can be obtained by partition the computation as:

$$f = \{\epsilon \oplus r(e_1) \oplus r(e_2) \oplus \cdots \oplus r(e_i)\} \oplus \{\epsilon \oplus r(e_{i+1}) \oplus r(e_{i+2}) \oplus \cdots \oplus r(e_n)\} \tag{3.6}$$

The identity ϵ is required in each partition to indicate the initialization.

3.3.4 Iteration Rule Class

Iteration in Loci is defined by a collection of rule classes: the build rule classes that initiate the iteration; the advance rule classes that advance the iteration; the collapse rule classes that terminate the iteration. The specification is inductive. The build, advance,

[2]There are actually two types of reductions. The global reduction produces parameter facts, and the local reduction produces store facts.

and collapse specifications are usually point-wise computation and the collapse condition specifications are usually singleton computation.

Iterations are specified by adding an iteration label to variable identifiers. Iteration labels are organized into a hierarchy that rooted at stationary time (facts that do not iterate). For example, v{n} represents variable v in iteration n. The {n} is the iteration label for variable v. The relationship between Loci iteration label hierarchy and imperative loop is shown in Figure 3.3.

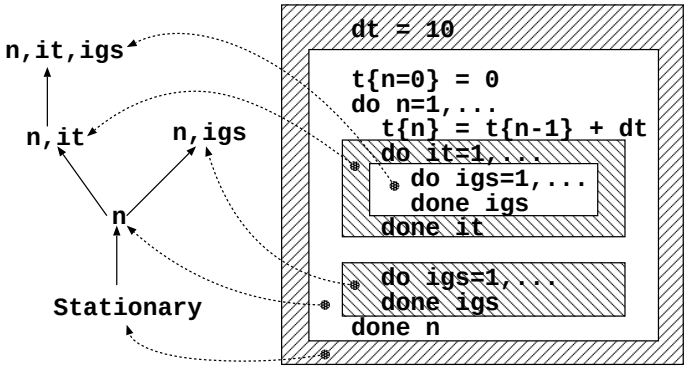

Figure 3.3 Iterations and Iteration Label

For example, using rule signatures, we can specify an iteration as following: a build rule q{n=0}←init; an advance rule q{n+1}←q{n},deltaq{n}; a collapse rule solution←q{n},CONDITION(converged). In the advance specification, the q{n+1} means the value of q in the next iteration (hence it is inductive definition). The CONDITION(converged) in the collapse specification determines the termination of the iteration. Upon exit, the computation of solution is executed.

Computation in an iteration can access values computed at either its own iteration level or at parent levels. During the Loci scheduling phase, variables in lower iteration level are automatically promoted up the iteration hierarchy. For example, variables computed at level {n} are communicated to level {n,it} automatically. In addition, specifications independent of iteration (i.e., specifications do not have iteration labels involved) can be promoted to any level in the iteration hierarchy.

3.4 The Scheduler

Given the specifications and descriptions in the databases, an application is formed by searching for an effective computation that leads to the user goal. As in Figure 3.1, the Loci scheduler is responsible for application synthesize. The current Loci scheduler is organized into four phases.

3.4.1 Dependency Graph Generation

Given the fact and rule databases and the goal, the program synthesis discovers all relevant rules that contribute to the solution, invoke them in a proper order and infer the domain for each rule and fact. A directed graph is used in Loci to model the program control structure and data movement. The first step in the scheduler is to search through the databases and set up the dependences for all the rules that can be applied and all the variables that need to be generated.

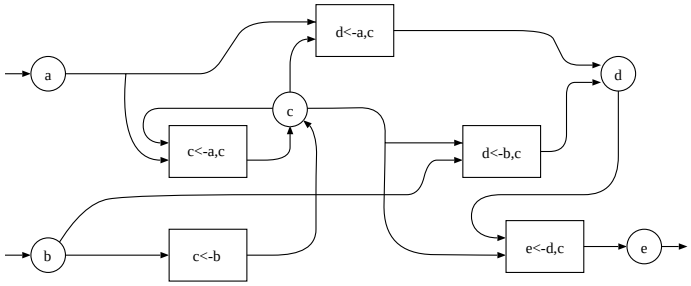

Figure 3.4 The Dependency Graph

A dependency graph is illustrated in Figure 3.4. In the graph, both rules (computations) and variables (facts or attributes) are represented as vertices. Edges in the graph connect rules to variables or variables to rules, whereas rule to rule connection and variable to variable connection do not exist. Usually there are edges from the rule vertex to its output variable vertices and there are edges from the variable vertices that are in the rule body to the rule vertex. In addition, Loci may add "internal" rules other than rules from the database. These rules will be used in managing variable promotion and renaming.

3.4.2 Decomposition

Decomposition is the refinement of the first step. The dependency graph is further reduced to a multilevel graph where each level is a directed acyclic graph (DAG). In this step, certain computations such as iterations, conditions or recursions are grouped into subgraphs respectively. Analogous to structured programming, the resulting multilevel graph (recursively) represents the structure and the natural order of computations that will lead to the solution.

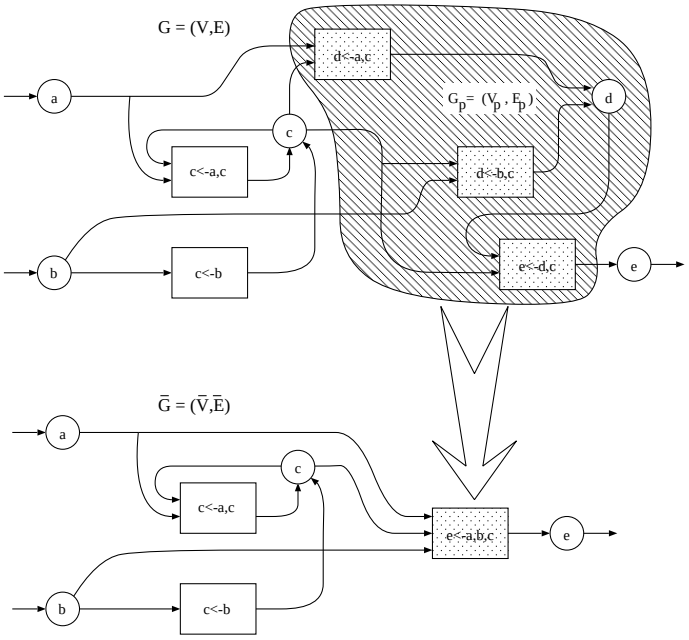

Figure 3.5 Decomposition

Decomposition is illustrated in Figure 3.5. The shaded region in the above graph has been identified as a certain structure. It is then grouped into a subgraph. The subgraph appears as a vertex in the original graph, resulting in the new graph below.

3.4.3 Existential Analysis

Once all applicable rules and their orders are formed, the correct domains for each rule and fact are inferred in the existential analysis phase. For example, the rule `p←rho,R,T` implies the fact that has attribute `p` will have domain:

$$domain(p) = domain(rho) \cap domain(R) \cap domain(T)$$

The existential deduction begins with the given facts, and follows the multilevel graph until the goal is reached. Then a pruning phase starts from the goal, and works backward through the graph. All the attribute domains that do not contribute to the final solution are pruned in the pruning phase, resulting in an optimized computation schedule.

3.4.4 Schedule and Compile

At this point, all information for program synthesis are deduced and set by the Loci scheduler. The multilevel graph is then recursively scheduled and together with the rule specifications in the database, a machine executable program is then produced and put in execution.

3.5 Summary

The Loci framework has numerous advantages. Using the declarative programming approach, the internal consistency of an application is guaranteed. This feature greatly facilitates the construction of large-scale and multidisciplinary scientific applications.

Another unique feature of Loci is the automatic aggregation of computations. Users of Loci can easily create abstractions using composeable objects at the fine-grain level with simple semantics, yet avoid the run-time cost associated with dynamic dispatch.

The declarative approach also makes automatic intra-application resource management possible. In the current implementation, automatic parallelization is supported. The semantics of aggregations are simple and clear for parallelization. The scheduler can nat-

urally produce a parallel schedule, the underlying specification does not have to refer to any explicit parallelism. This resource management ability also facilitates the automatic memory and cache management, which is the central theme of this thesis research.

This chapter presents an overview of the Loci framework and also provides necessary background and terminologies for this thesis research.

CHAPTER IV

BASIC DYNAMIC MEMORY MANAGEMENT

This chapter gives a comprehensive description of the design and implementation of an automatic memory management scheme for the declarative data-parallel programming framework Loci. The design of the automatic memory management provides the foundation of this thesis research. All the following research attempts are built on top of the work described in this chapter. The general guideline followed in the design of the automatic memory management scheme is the first hypothesis described in chapter I: The memory management scheme should be fully automatic, without any user intervention; it should also provide good compromise between memory utilization and application performance.

4.1 Memory Management as Graph Decoration

As discussed in chapter II, explicit memory management and garbage collection are not adequate for Loci. A new specialized strategy must be developed. For Loci memory management, being fully automatic means the framework itself should handle proper memory allocation and recycling. As shown in chapter III, Loci uses directed graph to model the application control flow and data movement. Thus the first decision in designing the Loci

memory management scheme is to incorporate the memory management process into the application control flow graph.

In the dependency graph, computations and variables are vertices and are connected together to form a partial order. The graph is finally scheduled and compiled into a program. Thus, a natural extension for including memory management in the dependency graph is to represent the memory allocation and recycling as vertices and insert them into the existing dependency graph. Then when the graphs get compiled, proper memory management instructions are included into the application and will be invoked in execution. This process of including memory management instructions into the dependency graph is referred to as *graph decoration*. The automatic dynamic memory management for Loci then becomes the graph decoration problem.

As shown in chapter III, Loci performs a decomposition after generating the dependency graph, resulting in a multilevel graph. Both dependency graph and the multilevel graph represent the application control flow and data movement, but the multilevel graph is more structured than the dependency graph. Thus, there are two possible graph decorations: either decorate the dependency graph or decorate the multilevel graph. Decorating the multilevel graph turned out to be easier then decorating the dependency graph. The dependency graph is not acyclic, cycles are possible. While in the multilevel graph, each level is a DAG. Moreover, if the dependency graph is decorated, the decomposition algorithm also requires adjustments for an optimized decoration. Thus, multilevel graph decoration is chosen.

The central problem in multilevel graph decoration is to find correct and optimized positions for proper memory allocation and recycling vertices. Since the graph has multiple levels, information is possible to cross the boundary of each level, a global analysis is needed. The purpose of the global analysis is to traverse the subgraph hierarchy to collect information and perform analysis for correct and optimized decoration of each subgraph.

The multilevel graph represents structured application control flow. In Loci, a separate C++ class hierarchy is dedicated for scheduling and compiling the multilevel graph. In the global analysis, different types of traversal actions are needed. These actions are implemented as a parallel visitor [6] class hierarchy. Therefore, new actions can be conveniently added as concrete visitors.

4.2 The Multilevel Graph

This section describes the structure and contents of the multilevel graph. In Loci, the multilevel graph is a collection of subgraphs. These subgraphs are organized in a graph hierarchy as levels. The multilevel graph has several levels, each level is a DAG and has vertices possibly represent another level of graph. Figure 4.1 is the top level of the multilevel graph of a simple Loci application. This graph is a simple DAG, but it contains other graphs. All the circles in the graph are variables (i.e., facts); the rectangular shaped vertices are user supplied rules; the two octagonal shaped vertices represent other subgraphs. Any vertex that represents a subgraph is referred to as a “super node,” hence

a prefix “SN” is added to the signature of each super node, the number after “SN” is an identifier for that node.

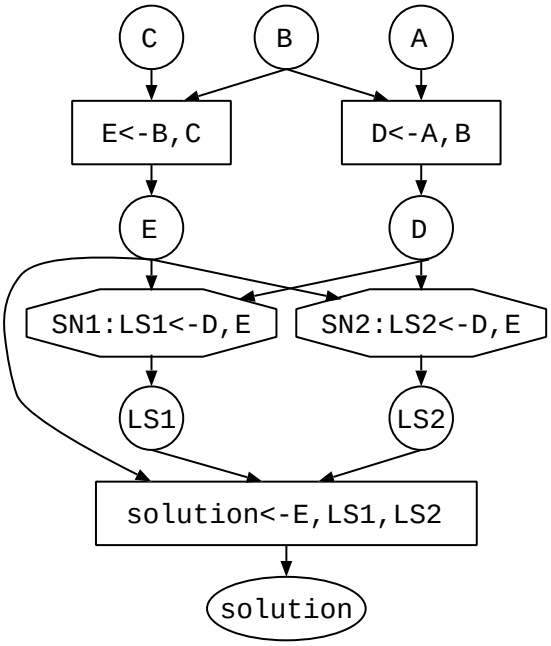

Figure 4.1 The Multilevel Graph: The Top Level

Figure 4.2 shows the contents of the left super node in Figure 4.1 (the vertex with signature “SN1:LS1<-D,E”). Figure 4.2 is also a simple DAG, but it contains yet another subgraph: the “SN4” super node. Only bottom levels in the multilevel graph are conventional DAG, they do not contain other subgraphs.

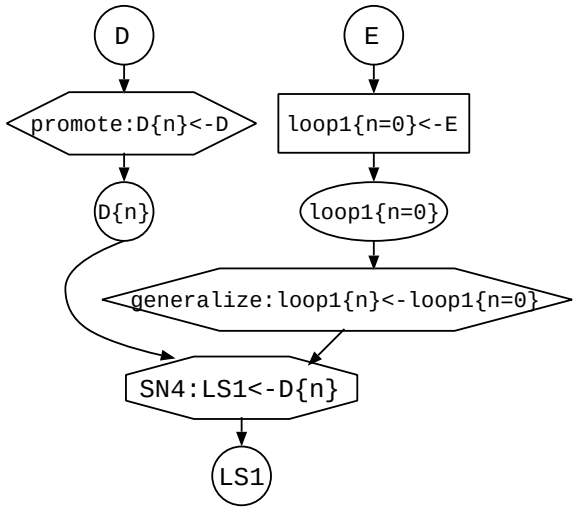

Figure 4.2 The Multilevel Graph: Another Level

4.2.1 The Loop Structure

Loop (i.e., iteration) is a major structure in Loci applications. A loop contains conditional structure and simple DAG structure. It could also contain other loops, hence nested loops are allowed. Since loops are specified inductively,[1] therefore, they have more complex structures. In Figure 4.2, the "SN4" super node represents a loop subgraph. The overall structure of this loop is shown in Figure 4.3. The hexagonal shaped vertex with a "looping" qualify is a Loci generated rule that ties each part of the loop together.

Recall from section 3.3.4, iteration is specified inductively by three steps: The building step, the advance step and the collapse step. In Loci, loops are therefore decomposed into two parts: the collapse part and the advance part. Thus, loops are represented by two DAGs in Loci. For the loop in Figure 4.3, the structures of these DAGs are shown in Fig-

[1] See section 3.3.4 for iteration specification.

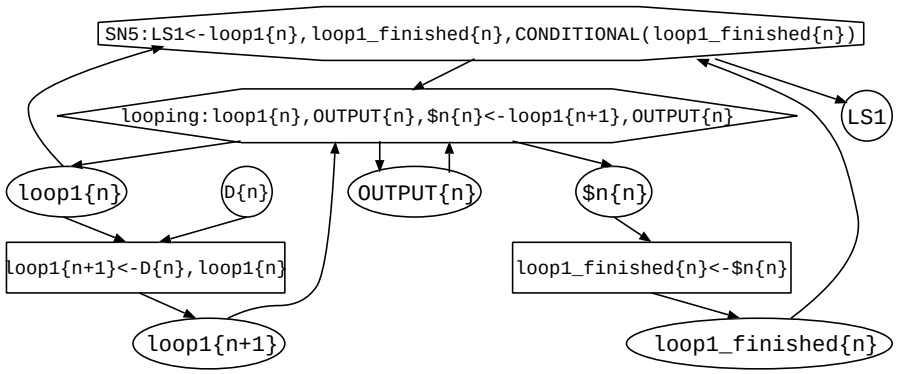

Figure 4.3 Loop Structure

ure 4.4 (the collapse DAG) and Figure 4.5 (the advance DAG). An important property of the collapse DAG and the advance DAG is they may share variables, but they will never share any rules. As in Figure 4.4 and Figure 4.5, no rules are shared in two graphs, but the variable `loop1{n}` exists in both graphs. In addition to the loop decomposition, a *rotation list* is built for each loop. Since loop is specified inductively, variable `loop1{n+1}` represents the value of `loop1{n}` in the next iteration. Therefore, at the end of each iteration, the contents of `loop1{n}` and `loop1{n+1}` are swapped. The rotation list contains variables `loop1{n}` and `loop1{n+1}`, they maintain the history of the loop.

The collapse part of the loop also contains a conditional subgraph. Figure 4.6 shows the conditional node for the collapse DAG in Figure 4.4. The conditional subgraph represents the computations for the final results of the loop, it is only scheduled and executed once, upon the exit of the loop. When scheduling the loop, the collapse DAG is always scheduled first, if the condition fails, then the advance DAG is scheduled. In the last iteration of the

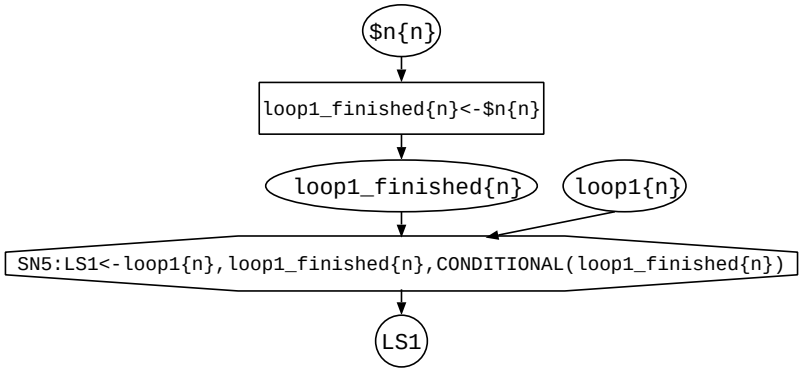

Figure 4.4 Loop Structure: The Collapse Part

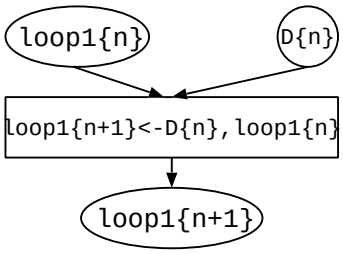

Figure 4.5 Loop Structure: The Advance Part

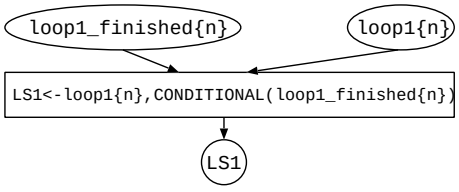

Figure 4.6 Loop Structure: The Conditional Part

loop, the conditions are met, then the conditional node in the collapse part is scheduled. The advance part is not scheduled in the last iteration.

4.2.2 Recurrence Internal Rules

Internal rules refer to rules that do not come from the rule database. They are Loci generated rules, such as the looping rule in Figure 4.3. They act as glue that hooks the graph together. Internal rules are represented as hexagons in all previous figures. Loci has three types of important internal rules that will affect the memory management. They are *generalize rules*, *promote rules* and *rename rules*. They together are referred to as recurrence internal rules.

Figure 4.2 shows the generalize and promote rules. They are related to iterations. In iteration specification, the first step is the building specification. The generalize rule is used to generalize the iteration label of variables. As in Figure 4.2, the initial iteration label $\{\texttt{n=0}\}$ is generalized to $\{\texttt{n}\}$. Recall from section 3.3.4, computation in an iteration can access values computed at either its own iteration level or at parent levels. The purpose of promote rule is to promote variables up in the iteration hierarchy so that they can be accessed in child iteration levels. There is also rename rules, but they do not have special signatures as generalize and promote rules. They can only been determined through internal data structures. The purpose of rename rules is for efficiency. For example, given a rename rule: $A \leftarrow B$. It instructs Loci to do in-place update: Variable A occupies the

same memory location as variable B, the computation would erase the contents of B and fill in the contents of A.

Given a rule `head`←`body`, the variables in `body` are referred to as *sources* for this rule; the variables in `head` are referred to as *targets* for this rule. From the memory management point of view, the generalize rules, promote rules, and the rename rules specify a recurrence relationship between the sources and the targets of a rule. In generalize and promote rules, the sources and targets are actually the same variables. They share the same memory location and the same contents in that memory location, the only difference is the iteration labels. In rename rules, the sources and targets share the same memory location, but they do not share the contents, the existence of sources and targets is mutually exclusive.

4.3 Graph Decoration Algorithm

4.3.1 Single Rule Decoration

Two internal rules are created to represent memory allocation and recycling respectively. The signature of the rule for allocation is: `ALLOCATE:V`←`CREATE`; the signature of the rule for recycling is: `DELETE:V`←`DESTROY`. `ALLOCATE` and `DELETE` are qualifies for the rules. The symbol `V` represents a variable list, i.e., all the variables to be allocated or deleted. `CREATE` and `DELETE` are virtual variables, they do not serve any purpose. Their existence are to satisfy the rule signature format only.

The smallest unit for decoration in the graph is a rule. Normally, a rule computes its targets from its sources. From the memory management point of view, memory for the targets need to be allocated before the rule proceeds and the memory for the sources is no longer useful when the rule finishes, they can be recycled. Therefore, we can modify the rule to include the memory management rules. Given a rule `targets←sources`, Figure 4.7 shows the decoration. The memory management rules now join the sources and targets of this rule. All this specifies is a partial order. When this rule is scheduled, memory management and computation are then interleaved. Note, the allocation rule has no incoming edges, it only points to other rules, while the delete rule has no outgoing edges.

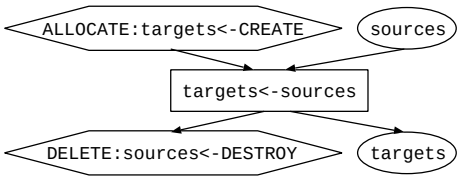

Figure 4.7 Single Rule Decoration

Recurrence internal rules need to be handled specially because they are not real computations. No memory management actions are required for the recurrence internal rules. Therefore, they do not require decoration.

4.3.2 Single Graph Decoration

Given a DAG, a variable may be produced and consumed by multiple rules. To decorate a DAG, the dependency relations must be considered. To allocate a variable, the allocate rule must have edges that point to all the rules that produce the variable. To delete a variable, all the rules that consume this variable must have edges point to the delete rule for the variable. In this way, the dependence of memory management operations and computations can be set. Figure 4.8 shows an example of DAG decoration. In Figure 4.8, the right graph is the decoration of the left DAG. In the decoration, all the hexagons are memory management rules. Unshaded hexagons represent allocation; shaded hexagons represent recycling.

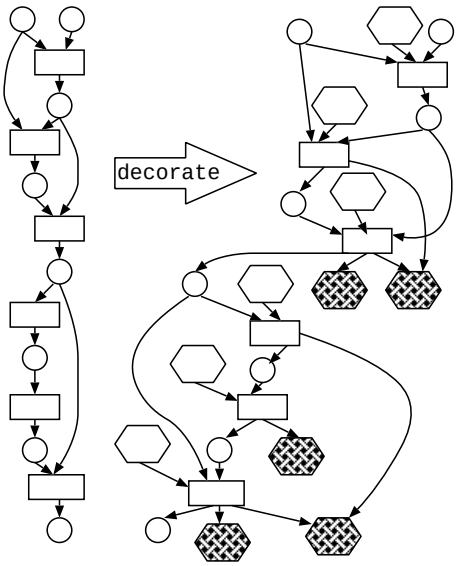

Figure 4.8 Single DAG Decoration

However, there are two problems with single DAG decoration in a multilevel graph. The first one relates to the recurrence internal rules. Normally, we need to allocate every target variable and delete every source variable in a DAG. A DAG may include recurrence internal rules. As discussed in the previous section, no memory operations are required for a recurrence internal rule. Therefore, the targets of recurrence internal rules are excluded from the allocation list and the sources of recurrence internal rules are excluded from the deletion list. They do not participate in memory management.

In a multilevel graph, a DAG may also contain vertices that represent a subgraph (i.e., super nodes). For efficiency concern, allocation should happen as late as possible and recycle should be performed as earlier as possible. Therefore, if a variable is only produced by one super node, the allocation is deferred or transferred to the subgraph represented by the super node. For the same consideration, if a variable is last consumed by one super node, then the recycling is deferred to the subgraph represented by the super node.

```
DAG-ALLOC-DECO(gr)
 1  vars ← GETWORKINGVARS(gr)
 2  grt ← TRANSPOSE(gr) ▷ Transpose the graph
 3  for vi ∈ vars
 4       do next ← SUCCESSOR(grt, vi)
 5          ▷ Get all the rules that produce this variable
 6          rules ← EXTRACTRULES(next)
 7          ▷ Get the number of super nodes and the total number of rules
 8          snum ← GETSUPERNODENUM(rules)
 9          rnum ← GETNUM(rules)
10          if not (snum = 1 and rnum = 1)
11            then ▷ Create an allocation rule for vi
12                 alloc ← CREATEALLOC(vi)
13                 ADDEDGES(gr, alloc, rules)
```

The procedure DAG-ALLOC-DECO performs the decoration for a single DAG. It includes all the discussions in this section. The procedure GETWORKINGVARS returns all the targets in the graph but excludes the targets of recurrence internal rules. The deletion decoration algorithm has a similar structure than DAG-ALLOC-DECO.

4.3.3 Multiple Level Decoration

If we traverse the multilevel graph in a top-down order, we get the order of computations. Ideally, we could traverse the multilevel graph top-down and apply algorithms discussed in the previous section for each level. But real applications are complex, so are their corresponding multilevel graphs. Subgraphs in the multilevel hierarchy are usually tightly related to each other. Information usually crosses the boundary of a single level. Simple DAG decorations as in the previous section are not adequate. Decoration of the multilevel graph should also consider global information.

```
MLG-DECO(mlg)
1  levels ← TRAVERSE(mlg)
2  for level ← TOP(levels) to BOTTOM(levels)
3       do vars ← GETWORKINGVARS(level)
4          for vi ∈ vars
5               do if RESPOND(level, vi)
6                     then place ← COMPUTEPLACE(mlg, level, vi)
7                          PLACEDECO(place, vi)
```

The general strategy to decorate the multilevel graph is shown in procedure MLG-DECO. We traverse the multilevel graph top-down. For each level, all the variables that could be allocated or deleted are gathered. Then for each variable, we determine whether

this particular variable is handled in the current level. If so, a place for decoration is then computed. Note, the place for decoration is not necessary in the same level.

There are three key steps in MLG-DECO: the procedure GETWORKINGVARS, procedure RESPOND, and the procedure COMPUTEPLACE. The procedure GETWORKING-VARS gathers candidate variables for allocation and deletion for each level. It works the same as the one in section 4.3.2 but with some augmentations. The one in section 4.3.2 only considers variable recurrence relations in a single DAG. In a multilevel graph, the variable recurrence relations could themselves become a DAG. Therefore, a preprocessing step is performed for each recurrence relation DAG. A variable for allocation and a variable for deletion are picked out from each such DAG, other variables in the DAG do not participate in memory management.

The procedure RESPOND determines whether a variable should be processed in a particular level. It uses the algorithm in section 4.3.2: if a variable is only produced or consumed by a single super node, then the allocation and deletion is skipped in the current level. In addition, an allocated variable set and a deleted variable set are maintained. Once an allocation or deletion decoration finishes, the corresponding variables are added into these sets.

If an application has no iterations, the algorithm for decoration in section 4.3.2 should suffice, only the dependency of a single level need to be considered. Iterations added complications to decoration. As a result, the level where the decoration of a variable to be put is not necessary the same as the level where this variable is processed.

First, as shown in section 4.2.1, the iteration has its own internal structure representations. Loop has a collapse part and an advance part, the two parts may share the same variables. In addition, loop has rotation list. The variables in the rotation list maintain the loop history, they need to be allocated before the loop starts. Therefore, the placement of the allocation of rotation list is not in the loop graph, it is in the parent graph that contains the loop as a super node. For example, in Figure 4.2, this level contains a super node "SN4", which is a loop. Therefore, allocation of the rotation list of "SN4" should be placed at this level. When scheduling the loop, the collapse part is always scheduled first, followed by the advanced part. Thus, the allocation of the shared variables between the two parts is placed in the collapse part, and the deletion of these shared variables is placed in the advance part. Each collapse part has a conditional node that computes the final results of this loop, it is only scheduled and executed once, upon exit of the loop. The recycling of the rotation list is therefore placed in the conditional node. In the last iteration of the loop, the advance part is not scheduled, hence, the deletion of the shared variables is not scheduled. An additional recycling of the shared variables is therefore placed in the conditional node in the collapse part.

Loops also put constraints on recycling of conventional variables (i.e., variables that do not belong to loop rotation list and loop shared variables). Figure 4.9 shows a possible application control flow. There are two loops, which are nested, in the application. The first loop is contained in a DAG as a subgraph and the inner loop contains a DAG as a subgraph. Variables $v1, v2, v3, v4$ are allocated in $dag1, loop1, loop2$ and $dag2$ respectively.

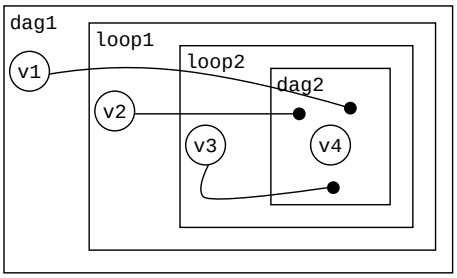

Figure 4.9 Placement of Recycling

All the variables are consumed by $dag2$ only (possibly through promotion). Therefore, $dag2$ is responsible for recycling of all the variables. The placement of deletion of $v3$ and $v4$ could be within $dag2$, which means we could delete $v3$ and $v4$ immediately after $dag2$ consumes them. But the placement of deletion of variables $v1$ and $v2$ cannot be inside $dag2$. Although $v2$ is only consumed by $dag2$, it is allocated by $loop1$. When the application runs, $loop2$ will be executed for several iterations, so is $dag2$. If $v2$ is deleted within $dag2$, then in the next iteration of $loop2$ and $dag2$, $v2$ is no longer available. Thus, the deletion of $v2$ should be placed after all execution (i.e., iteration) of $dag2$. We choose to delete $v2$ upon exit of $loop2$. The placement is therefore in the conditional node within the collapse part of $loop2$. For the same reason, the placement of deletion of $v1$ is in the conditional node within the collapse part of $loop1$.

```
MLG-DEL-PLACE(v, vLevel, mlg)
 1  loops ← GETPARENTLOOPS(vLevel, mlg)
 2  ▷ Count the number of all parent loops
 3  loopsNum ← GETNUM(loops)
 4  if loopsNum = 0
 5     then return vLevel
 6     else curLoop ← BOTTOM(loops)
 7          ▷ Get the allocation of curLoop and all its sub-nodes
 8          alloc ← GETALLOC(curLoop)
 9          if v ∈ alloc
10            then return vLevel
11          curLoop ← PARENT(curLoop, loops)
12          alloc ← GETALLOC(curLoop)
13          while v ∉ alloc and curLoop ≠ NIL
14               do curLoop ← PARENT(curLoop, loops)
15                  alloc ← GETALLOC(curLoop)
16          if curLoop = NIL
17            then pLoop ← TOP(loops)
18            else pLoop ← CHILD(curLoop, loops)
19          pLevel ← CONDNODE(pLoop)
20          return pLevel
```

The algorithm for recycling placement of conventional variable in the multilevel graph is given in procedure MLG-DEL-PLACE. It takes three parameters: v is the variable to be deleted; $vLevel$ is the level that handles deletion request; mlg is the multilevel graph. MLG-DEL-PLACE returns the level that the deletion of v should be put. For example, for $v2$ in Figure 4.9, call to MLG-DEL-PLACE$(v2, dag2, mlg)$ would return the *conditional node* of $loop2$. Procedure MLG-DEL-PLACE works by traversing the parent loop hierarchy starting from $vLevel$ and examines the allocation of the loop hierarchy. The procedure GETPARENTLOOPS returns all the parent loops starts from $vLevel$. If $vLevel$ is itself a loop, it is also included in the results. For example, for Figure 4.9, GETPARENTLOOPS$(dag2, mlg)$ would re-

turn $[loop1, loop2]$; GETPARENTLOOPS$(loop2, mlg)$ would return $[loop1, loop2]$ too; GETPARENTLOOPS$(dag1, mlg)$ would return NIL. Procedure GETALLOC returns the allocation of the given loop and all its sub-nodes. The result is used to test where the variable v is allocated. Procedure CONDNODE returns the conditional node of the given loop.

4.4 Summary

This chapter presents the design and implementation of an automatic memory management scheme for the Loci framework. The equivalence of memory management and the multilevel graph decoration is first established. This is the fundamental idea for memory management in Loci, it enables seamless integration of the memory management and the Loci framework. The basis and main problems in decorating the multilevel graph are then outlined. The variable recurrence relations and various implications of the loop structures are discussed in detail.

CHAPTER V

CHOMPING TECHNIQUE

This chapter describes a cache optimization scheme for the Loci framework based on the work of dynamic memory management in chapter IV. This cache optimization scheme is automatic, it does not require user intervention. The objectives of this cache optimization scheme are to further reduce the memory requirement of an application and also to increase the performance of Loci applications.

5.1 The Chomping Idea

The cache optimization for Loci is based on the idea of *chomping*, or also known as *strip mining*. A general strategy for cache optimization is to partition the data into small chunks that can fit into the data cache and arrange the access pattern to these chunks so that they stay in the cache as long as possible. Thus, data partitioning and accessing form the central theme in the chomping technique for cache optimization in Loci.

In Loci, rules are elements of computation in an application. Rules produce and consume variables. In an application, only the user queried variables are useful result, others are all intermediate variables, their existence are to contribute to the final solution only. Normally, a rule is computed once over its domain and the target variables are produced

entirely. Since Loci variables are often containers that hold large amount of data, all the intermediate variables are thus ideal candidate for data partitioning. In the cache optimization scheme, we chomp the domain of a rule. Therefore, a rule is no longer computed once, instead, the computation is broken into small sub-computations. In each of these sub-computation, only part of the target variables are produced. This implies only partial allocation is required for all the intermediate variables. Because these partial allocations could be potentially small, they enhance cache utilization and further reduce the memory requirement.

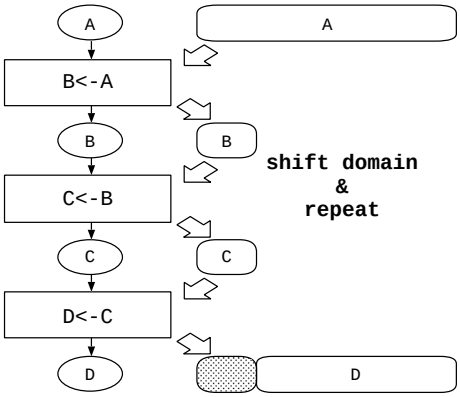

Figure 5.1 The Chomping Idea

The basic chomping idea is illustrated in Figure 5.1. In the rule chain, A is the given variable, D is the final result of the chain, both B and C are intermediate variables. Before the chain starts, the entire memory required for D is allocated, for B and C only small chunks are allocated. Then the rule chain is executed iteratively. In each execution iter-

ation, all the rules in the chain only execute over a sub-domain. The size of the domain computed at each iteration equals the size of the allocated domain for B and C. D is therefore only partially computed. Then the domains of all the rules in the chain are shifted for the computation of next iteration. The iteration of execution terminates until the entire D is produced.

Normally, the chain of rules is executed only once over the entire domain of the rules. With dynamic memory management, at least two entire containers need to be kept in the memory at one time. With chomping, the required memory to be kept is the entire space for D and part of the space for B and C. We can choose the size of the chunks for B and C in memory, they can be less than one entire container. Therefore, the entire memory required by chomping is even less than the space used in a normal run with dynamic memory management. If the topological structure of the chain of rules are "flat"[1], then the memory savings are even more. On the other side, since the size of chunks for B and C are small, they can fit into the data cache and stay inside it during an iteration execution of the chain. Thus, the cache utilization is better than single run of the rule chain. If the cache benefit is greater than the overheads of additional management in chomping, the performance of the application will be improved.

The implementation of chomping borrows the idea of decomposition of the dependency graph in the Loci scheduler. In each level of the multilevel graph, all the chain of rules suitable for chomping are identified first. Then each chain is substituted by a

[1]A flat chain means a chain that has rules that consume most of the chomped variables as input. When topologically laid out, the chain looks "flat."

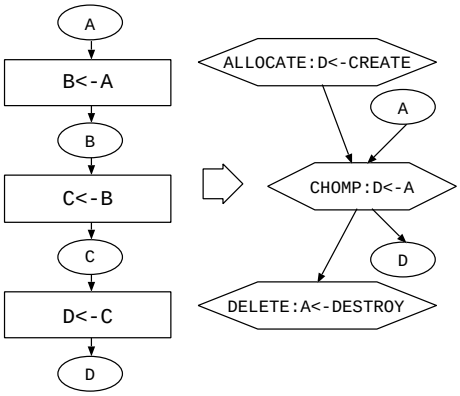

Figure 5.2 Implementation of Chomping

special vertex. This allows smooth integration of chomping and the dynamic memory management scheme designed in chapter IV. As shown in Figure 5.2, the graph decoration required by dynamic memory management does not have to be aware of chomping. Only in the execution phase, a special C++ class handles the management and execution of the chomp rule. The chomp rule is regarded as Loci internal rule.

5.2 Searching for Chompable Subgraph

Identifying rule chains that are suitable for chomping is the central problem in the implementation. Chompable rule chains do not cross the boundary of a single level in the multilevel graph. Hence, the identification and replacement of rule chains occur in each level of the multilevel graph.

In searching for chompable rule chains in each level, maps present a major complication. Recall from section 3.2, map models the relations between entities; composition of

map and store models indirect access of store containers. Therefore, any store container that involves maps cannot be chomped directly. Because store with maps means random access of the store container. While in chomping, we only produce or consume part of a container at each time. If a container needs to be random accessed, then we cannot anticipate directly which segment of domain needs to be allocated in chomping at each time. Therefore, as a result, stores that involve any map need to be allocated entirely.

The searching algorithm consists of three steps: a preprocessing step, a merging step and an optimization step. In the first preprocessing step, all variables in a DAG are categorized into two classes: variables that are suitable for chomping (i.e., chomping candidates) and variables not suitable for chomping. As discussed previously, because of the presence of map, not all stores can be chomped directly. Those chomping candidates found in the preprocessing step are the theoretical upper bound of the total number of variables in a DAG that can be actually chomped.

The following merging step forms all chomping rule chains in a DAG. It works by merging those chomping candidates found in the first step. The merging is based on several properties in chomping.

- If a variable is chomped, then all rules connect to this variable must all been chomped. Therefore, any two chomping candidate variables that share any rules can be merged together to form a larger chain. This also implies that once the chomping chain is formed, all intermediate variables inside the chain are invisible from outside (i.e., no references to those variables from rules outside of the chain).
- The edited DAG that includes chomping rule chains must still be acyclic. This restricts the merging algorithm. At some step, we may be forced to discard some chomping candidate variables in order not to create cycles.

- Any non-chompable variable cannot be an intermediate variable in the chomping chain.

These merging guidelines are illustrated in following examples:

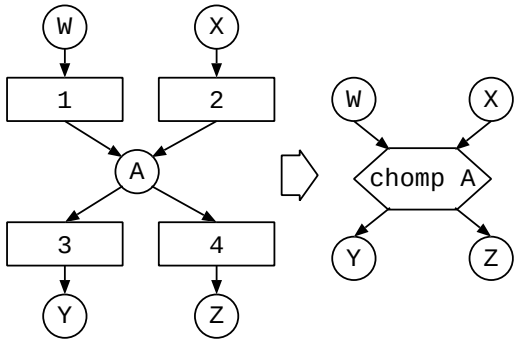

Figure 5.3 Chomping Property One

Figure 5.3 shows the invisibility of chomped variables and rules. In Figure 5.3, variable A is chomped, $1, 2, 3$ and 4 are rules that produce and consume variable A. The chomp rule hides all of them, only variable W, X, Y and Z serve as sources and targets of this chomp rule.

Real Loci applications often have complex relations between variables and rules. If graph editing is not properly handled, cycles can be easily created. Figure 5.4 is such an example. Variable B and C are chompable, while D and E are non-chompable variables, D is an ancestor of E. If we chomp both B and C, the resulting graph will look as the right one in Figure 5.4. A cycle is therefore formed. In this case, we will have to discard B, albeit it is a chomping candidate variable.

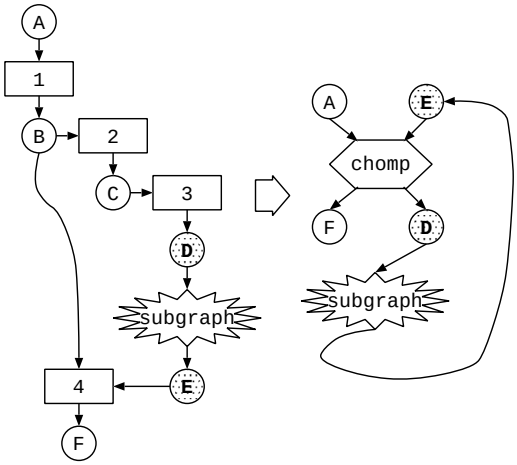

Figure 5.4 Chomping Property Two

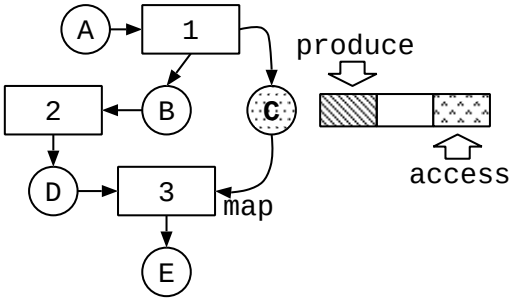

Figure 5.5 Chomping Property Three

Figure 5.5 shows another complication. In the graph, B and D are both stores and are chomping candidates. Rule 3 access its source variable C through a map, thus C cannot be chomped. If we chomp both B and D, C will become an intermediate variable of the chomping chain. This will have problem even if we allocate C entirely. Because rule 3 access C through a map, it is possible that the accessed region has not been computed yet. In this case, we will have to discard either B or D from the chomping chain.

```
MERGE-CANDIDATES(gr, candVars)
 1  chains ← NIL
 2  validCandVars ← candVars
 3  while candVars ≠ NIL
 4        do begin ← POP(candVars)
 5           rules ← GETCONNECTEDRULES(gr, begin)
 6           merge ← TRUE
 7           while merge
 8                 do merge ← FALSE
 9                    mergedVars ← NIL
10                    for vi ∈ candVars
11                          do rs ← GETCONNECTEDRULES(gr, vi)
12                             if rules ∩ rs ≠ NIL
13                               then g ← SUBGRAPH(gr, rules + rs)
14                                    if not HASPROBLEM(g)
15                                      then rules ← rules + rs
16                                           mergedVars ← mergedVars + vi
17                                           merge ← TRUE
18                    candVars ← candVars − mergedVars
19           c ← SUBGRAPH(gr, rules)
20           POSTPROC(c)
21  return chains
```

The algorithm for merging is illustrated in MERGE-CANDIDATES. Given a DAG and all chomping candidate variables, the algorithm first selects a variable from the candidates and builds a smallest chain. Using this chain as the basis, the algorithm iterates through rest of the candidates and merges them according to the guidelines discussed previously.

The procedure HASPROBLEM tests for cycles and non-chompable internal variables. Procedure POSTPROC performs post processing on a formed chain before we finally accept it. They are described in the following algorithms:

```
HASPROBLEM(g)
1  if CYCLE(g)
2     then return TRUE
3  ivs ← GETINTERNALVARS(g)
4  pvs ← ivs − validCandVars
5  if pvs ≠ NIL
6     then return TRUE
7  return FALSE
```

```
POSTPROC(c)
 1  cvs ← GETCHOMPEDVARS(c)
 2  if SIZE(cvs) = 1
 3     then if HASPROBLEM(c)
 4             then v ← FIRST(cvs)
 5                  candVars ← candVars − v
 6                  validCandVars ← validCandVars − v
 7                  DISPATCHNOTIFY(v)
 8                  return
 9  stvs ← GETSOURCETARGETVARS(c)
10  rmvs ← stvs ∩ candVars
11  candVars ← candVars − rmvs
12  validCandVars ← validCandVars − rmvs
13  validCandVars ← validCandVars − cvs
14  chains ← chains + c
```

```
DISPATCHNOTIFY(v)
1  rcs ← NIL
2  for ci ∈ chains
3       do invs ← GETINTERNALVARS(ci)
4          if v ∈ invs
5            then rcs ← rcs + ci
6                 cvs ← GETCHOMPEDVARS(ci)
7                 candVars ← candVars + cvs
8                 validCandVars ← validCandVars + cvs
9  chains ← chains − rcs
```

The purpose of POSTPROC is to prune the $candVars$ set according to the formed chain. If the merged chain only has one chompable variable, then no other variables are merged aside from the initial beginning variable. The chain is therefore not tested yet. It is then checked by HASPROBLEM. Because MERGE-CANDIDATES does not perform other orders of merging, if the chain failed in the testing, then the single chomped variable inside it is treated non-chompable from that time. The procedure DISPATCHNOTIFY is used to signal already formed chains of this change. It is needed here because this single variable may already silently included into other chains. It is possible because a rule could have multiple targets. When we form a subgraph from one target, other targets will also been included into the subgraph. In DISPATCHNOTIFY, all previously formed chains are searched. If any of them contains this particular variable as an internal variable, then the chain is canceled and all chomped variables are pushed to the chomping candidates set again.

If the formed chain has more than one chomped variables, it is then already a valid chain. Because at least one merge happened and therefore the the chain was tested. But since during the merging, some chomping candidates may not be merged into this chain due to fail to pass the HASPROBLEM test, these variables will become either source or target variables of this chain. Then part of the rules connect to these variables will be hidden by this particular chomping chain and no other references to these rules are allowed from outside of this chain. Therefore, these variables can not be included in any future formed chains and are taken off from the chomping merging candidates set. Note the dis-

tinction of $candVars$ and $validCandVars$ in MERGE-CANDIDATES. Variable $candVars$ is the chomping candidates set for merging, while the variable $validCandVars$ is used inside HASPROBLEM for testing invalid internal variables in a chomping chain. The difference between two variables is after the **for** loop at line 10 in MERGE-CANDIDATES, all merged chomping candidates in the **for** loop are removed from $candVars$, while these variables are removed from the $validCandVars$ all together in POSTPROC (line 13). The reason for this distinction is that during the merging process, when a chain is checked through HASPROBLEM, the already merged variables in this chain are still valid. In a sense that $validCandVars$ records the still valid variables that can be chomped when we start a new merge at line 4 in MERGE-CANDIDATES.

In the merging step, we may discard some of the chomping candidates due to cycles and non-chompable variables being internal variables of a chain. This is a combinatorial searching problem, the algorithm described here does not perform an exhaustive searching. Under certain circumstances, it may discard more variables than it should. Thus a final optimization pass is added with the hope of getting back some of those lost variables. The optimization works similar to the merging step. But we iterate through all chains, if any two chains share any chomping candidates, we may test and merge them together. Because these "boundary" chomping candidate variables may be overlooked in the merging. Theoretically, the optimization algorithm may suffer the same problem as in merging. They may still miss valid chompings. But since we are testing and merging between chains, the

problem space is reduced significantly, therefore the chances for missing valid chompings are greatly reduced.

The run-time complexity for the merging algorithm (includes the optimization) is $O(n^2(V+E))$, where n is the number of chomping candidates in a given DAG and V, E are the number of vertices and the number of edges of the given DAG respectively. This algorithm runs well in practical, both the running time and the searching results are of satisfactory for large Loci applications.[2]

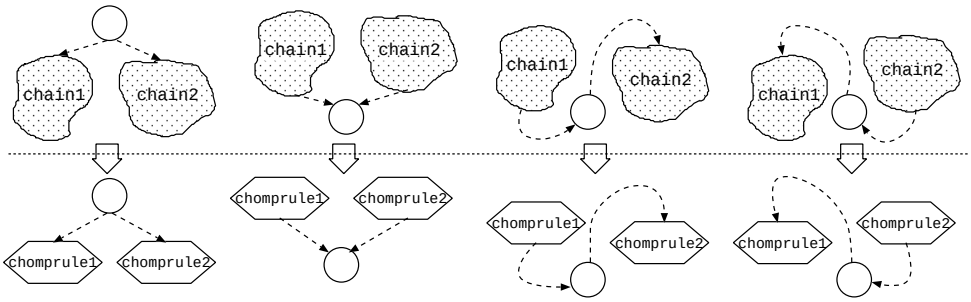

Figure 5.6 Positions of Chomp Chains in A DAG

After the search finishes, each chomping rule chain will be substituted by a special chomp rule in the DAG. The merging step guarantees a single chomping rule chain substitution will not create cycle in the DAG (all chains have passed the cycle testing). The merging step also guarantees two chains will not have intersections except for source or target vertices. Therefore, the relations of two chains in the DAG can be categorized into four possible cases as shown in the top part in Figure 5.6. The bottom part in Figure 5.6

[2]See chapter VII for results analysis.

shows the graph editing results. We can conclude no cycles will be introduced by graph editing.

5.3 The Chomping Size

The chomping size is the total allocation size for all of the chomped variables in a chain. In the implementation, it is chosen to be approximately half the size of the data cache. Because chomping is not exactly the same as traditional matrix blocking. In chomping, the source and target variables are not chomped, they could be large and they may also have access of data through maps. In the implementation, the user can also specify a particular chomping size before the program starts.

5.4 Summary

In this chapter, a cache optimization scheme is proposed for the Loci framework. It is based on the idea of chomping, or strip-mining. The computation of a rule in Loci application is broken into multiple small sub-computations. These sub-computations may help to improve cache utilization and reduce memory requirement. The algorithm that discovers chomping chains is discussed in detail.

CHAPTER VI

SCHEDULING POLICIES

This chapter presents a new scheduling policy for directed acyclic graphs in Loci. This new policy will improve memory performance of an application with the cost of increased frequency of communication points in the scheduled program.

6.1 Relations Between Memory Utilization and Communication Costs

The fundamental strategy for memory management is outlined in chapter IV. The relations of managing memory and directed graph decoration is established. The multilevel graph is decorated with memory management instructions; but it only specifies a dependency relation for the vertices. It is up to the scheduler to generate a particular execution order that satisfies the dependency relation. Different scheduling results in different interleaves of allocation, computation, and recycling. From the memory utilization point of view, the order to schedule allocation and recycle affects the peak memory requirement of the application. An optimized schedule for memory usage will reduce the application peak memory requirement to an absolute minimum. Therefore in exploring the relations between memory management and parallel overheads, the basic question is: What impact does optimizing memory management have on the parallel computation schedule?

Loci is a data-parallel programming system. In a data-parallel program, each process executes the same instructions on its own local data set. After certain amount of work, each process participates in a global synchronization, where all processes synchronize their work and exchange information. For a Loci application, synchronization is needed for targets of pointwise rule. This means for every step in the schedule, there will be a barrier on target variables of all pointwise rules. From this communication point of view, different schedule may create different numbers of synchronization points. With respect to parallel overhead, less synchronization is preferred. For an application, the number of synchronization points does not change the total volume of data communicated. But in a schedule with less synchronization, more computations happen at each step and more variables are generated at each step. Thus we can group more data together in a synchronization point and this helps to save the start-up cost in communication.

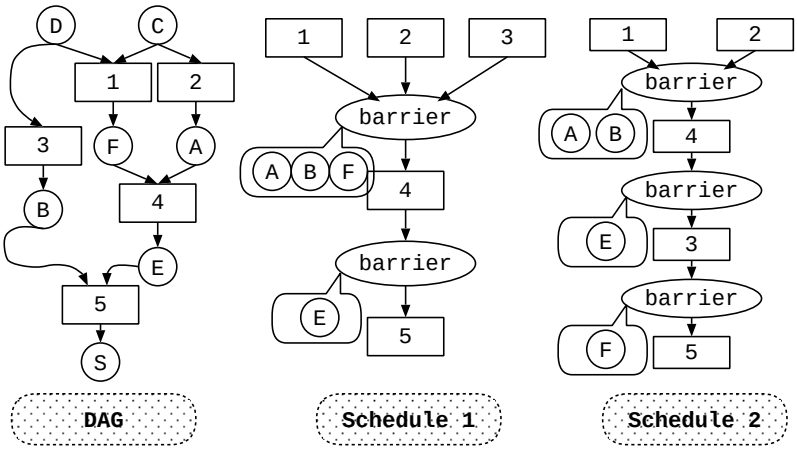

Figure 6.1 Different Scheduling for A DAG

With dynamic memory management, if we want to minimize the peak memory requirement for an application, then the resulting schedule will increase the total number of synchronization points. On the other side, if we want to minimize the total number of synchronization points for an application, then the resulting schedule will use more memory.

Figure 6.1 illustrates the effect of different scheduling of a DAG. Schedule one is greedy on computation in the application, while schedule two is greedy on the memory usage of the application. In schedule one, it schedules all possible rules to execute in a step. Therefore, more variables are generated, potentially increase the peak memory usage. But in the first step, more variables can be grouped together, saving synchronization points. In schedule two, only necessary rules are scheduled at each step, then less variables are generated, potentially reduce the peak memory usage. But in such a schedule, variables are spread over more scheduling steps, hence more synchronization points are needed. From Figure 6.1, we can see the total volume of data communicated is the same in two schedules. Both are variables A, B, E and F, but the grouping of communication for these variables is different.

Therefore, optimizing memory management will create more synchronization points in the application, hence more communication start-up costs and result in a slow program. On the contrary, attempting to minimize the synchronization points in an application will result in a fast program with more memory usage. Thus, trade-offs exist in such system and can be customized under different circumstances. For example, if memory is the

limiting factor, then a memory optimization schedule is preferred. In this case, speed is sacrificed for getting the program run. On the other hand, if time is the major issue, then a computation greedy schedule is preferred. Users have to supply more memory for the speed.

6.2 Memory Greedy Scheduling

The current scheduler in Loci is greedy on computation. It schedules every vertex in the DAG that can be scheduled in a step and therefore minimizes the synchronization points. It will produce schedules similar to schedule one in Figure 6.1. In order to examine and verify the trade-offs discussed previously, an alternative scheduler is added to Loci. It tries to minimize the memory usage of an application.

An optimal schedule for memory management is a combinatorial problem and requires exhaustive search in the DAG. The Loci scheduler is part of the run-time system, thus a fast algorithm is needed. We use a greedy algorithm in the alternative scheduler and rely on heuristics to choose vertices to schedule in the DAG. The algorithm is described in the following procedures:

```
LOCI-GRAPH-SCHEDULER(gr)
 1  grt ← TRANSPOSE(gr) ▷ Transpose the graph
 2  ▷ Initialize priority, V is all vertices in gr
 3  ▷ The smaller the weight, the higher the priority
 4  for vi ∈ V
 5       do p[vi] ← 0
 6  PRIOGRAPH(gr)
 7  sched ← NIL      ▷ The Schedule
 8  visited ← NIL    ▷ Visited vertices set
 9  ▷ We start off from all source vertices in the graph
10  wait ← GETSOURCE(gr)
11  while wait ≠ NIL
12       do q ← ENQUEUE(wait)
13          vs ← NIL
14          while q ≠ NIL and vs = NIL
15               do vs ← POP(q)
16                  vs ← GETVALIDSCHED(vs)
17          if q = NIL and vs = NIL
18            then error "graph has cycles"
19            else wait ← wait − vs
20                 new ← GETNEW(vs)
21                 wait ← wait + new
22                 visited ← visited + vs
23                 sched ← APPEND(sched, vs)
24  return sched
```

```
GETVALIDSCHED(vs)
1  valid ← NIL
2  for vi ∈ vs
3       do pre ← SUCCESSOR(grt, vi)
4          if pre ∈ visited
5            then valid ← valid + v
6  return valid
```

```
GETNEW(vs)
1  new ← NIL
2  for vi ∈ vs
3       do next ← SUCCESSOR(gr, vi)
4          new ← new + next
5  return new
```

```
PRIOGRAPH(gr)
 1  ▷ Memory greedy prioritize
 2  ▷ Initialization
 3  l ← NIL
 4  for vi ∈ V
 5       do a ← ALLOCNUM(vi)
 6          d ← DELNUM(vi)
 7          o ← TARGETOUTEDGENUM(vi)
 8          l ← APPEND(l, (vi, a, d, o))
 9  prio ← 0
10  for i ← 1 to LENGTH(l)
11       do s ← l[i]
12          if s.a = 0
13            then p[s.vi] ← prio
14                 ERASE(l, l[i])
15  prio ← 1
16  ▷ Sort l according to ascending order of a
17  SORT(l, ASCEND(a))
18  ▷ Stable sort l according to descending order of d
19  STABLESORT(l, DESCEND(d))
20  for i ← 1 to LENGTH(l)
21       do s ← l[i]
22          if s.d ≠ 0
23            then p[s.vi] ← prio
24                 ERASE(l, l[i])
25                 prio ← prio +1
26  ▷ Sort l according to ascending order of o
27  SORT(l, ASCEND(o))
28  for i ← 1 to LENGTH(l)
29       do s ← l[i]
30          p[s.vi] ← prio
31          prio ← prio +1
```

The procedure LOCI-GRAPH-SCHEDULER is a generic scheduling infrastructure for Loci. It schedules the graph according to the weight of each vertex. It only knows the topological structure and the weight of vertices. LOCI-GRAPH-SCHEDULER starts off by building the waiting set to schedule from all of the source vertices in the graph. Then each time, a priority queue is built for the $wait$ vertex set according to the priority of each vertex

inside $wait$. The procedure ENQUEUE forms the priority queue for $wait$. The scheduler tries to schedule vertices with highest priority each time. It uses a POP function and checks the dependency constraints until it finds a set of vertices to schedule. Note, different than usual priority queue, the POP function here pops all vertices with the highest priority from the queue, not just one at each time. If the input graph has cycles, then eventually nothing can be scheduled while we still have a set of $wait$ vertices. The scheduler reports error in that case. Otherwise, it appends the schedule with the selected vertex set in the previous step and modify the $wait$ set accordingly. Scheduled vertices are removed from $wait$, and new vertices reachable from the scheduled one are added to $wait$. The scheduler will always terminate. If the input graph has cycles, eventually the scheduler will discover the error and stop. If the input graph is a DAG, the scheduler always schedules something at each step. The graph has finite number of vertices. Therefore at certain point, there are no new vertices introduced into the $wait$ set, and the scheduler will stop when it consumes all vertices inside $wait$. The run-time complexity of LOCI-GRAPH-SCHEDULER largely depends on the PRIOGRAPH function at line 6.

With this scheduling infrastructure, the computation greedy schedule and memory greedy schedule only differ from how to provide the vertex priority. The current computation greedy scheduling can be viewed as having a PRIOGRAPH function that sets all vertices with the same weight.

The algorithm for the memory greedy scheduling relies on the use of heuristic. The heuristic is designed to try to minimize the memory usage at each scheduling step. It

should also be simple enough that does not introduce excessive overhead to Loci. The basic idea of the heuristics are illustrated as following: In a given graph, variables and rules that do not cause memory allocation have the highest priority and are scheduled first. They are packed into a single set in the schedule. If no such vertices can be scheduled, then we must schedule rules that cause allocation. The remaining rules are categorized. For any rule that causes allocation, it is possible that it also causes memory deletion. We schedule one such rule that causes most deletions. If multiple rules have the same number of deletion rules attached, we schedule one that causes fewest allocations. Finally, we schedule all rules that do not meet the previous tests, one at a time with the fewest outgoing edges from all of its target variables. This is based on the assumption that the more outgoing edges a variable has, the more places will it be consumed, hence the longer lifetime will this variable have.

The algorithm is shown in PRIOGRAPH. We start off from building a list that contains statistical information for every vertex. For each rule, the number of allocation rules attached, the number of deletion rules attached, and the number of outgoing edges for all of its target variables are collected respectively as shown between line 5 and line 7. For variables, all these numbers are just 0. Then we looping over the list, for any vertex with no allocation number, we set the highest priority for it. They all get assigned with priority 0 because we want them to be scheduled together. We also remove these finished vertices from the list. We then sort the list. The first sorting is based on the ascending order of allocation number, and the second sorting is based on descending order of deletion num-

ber. This is the same meaning as in the previous description of the heuristic. Because at this stage, all vertices in the list cause allocation, we want to schedule the one that lead to most deletions and has fewest allocations if there are multiple vertices with same number of deletions. Stable sort is required in the second sorting to keep the relative order from the first sort. After sorting, we looping over the list again, for any one with non-zero deletions, we assign priority to it. Note, this time the priority is increased one at a time because we only want one vertex to be scheduled at each time from now on. We again remove finished vertices from the list. Finally we sort the list according to the outgoing edge numbers and set corresponding weight for each remaining vertex. The algorithm terminates with all vertices processed. It has a worst case run-time complexity of $O(V + E + VlgV)$ and best case run-time complexity of $O(V + E)$. V and E are the number of vertices and edges for the input graph respectively. In worst case, the sorting dominates the runtime.

When counting numbers of allocation rules and deletion rules, only allocate and delete of store variables are counted. Stores are the only non-trivial variables in present Loci. This scheduling tends to minimize memory usage, but it also increases the synchronization points. Because for rules that cause allocation, only one of them are chosen at a scheduling step. The target variables are distributed more sparsely in the schedule, and therefore more synchronization points are needed.

6.3 Summary

This section briefly presents the relations between memory management and parallel communication costs. An optimized schedule for memory usage will likely increase the parallel communication costs. On the contrary, an optimized schedule for communication will likely increase the memory bound for an application. These trade-offs can be customized to different application requirements. An alternative memory greedy scheduler for Loci is also presented in detail in this chapter.

CHAPTER VII

RESULTS

This chapter presents the experimental results of the dynamic memory management scheme proposed in this thesis. Some analyses are also given as the results are presented.

7.1 The Evaluation Methods

The evaluation of the dynamic memory management scheme consists of four parts: performance evaluation of algorithms, space profiling, performance profiling, and characterization of the trade-offs in memory utilization and communication costs, if any. First, we conduct a measurement on the performance and behavior of all the algorithms developed in this thesis. Then, in space profiling, the benefits of using dynamic memory management are measured in detail. This includes how much space saving can be achieved by using dynamic memory management, chomping, and memory greedy scheduling respectively when comparing with the preallocation scheme. In performance profiling, the run-time performance of Loci applications are measured under different memory management configurations. This examines the run-time overhead associated with dynamic memory management, the performance improvements due to cache benefits by using chomping. In

the last part, the parallel performance versus memory usage are measured under different schedulers.

The evaluation is mainly based on two applications: the CHEM program and the FUELCELL program. Both of them are chemistry solvers but for different problem domains. Both applications are developed using the Loci framework and are being used to solve real world engineering problems.

The space and performance profiling are conducted on both sequential and parallel architectures. The trade-offs between memory utilization and communication costs are measured on parallel machines. For sequential testing, an SGI Challenge 10000 XL (8 195MHz R10000 processors with 2 Gigabytes of RAM), an Intel Pentium 4 PC (2GHz with 512 Megabytes of RAM, running on Linux), a single node on an IBM Linux Cluster (see below), and an Intel Pentium III PC (1.2GHz dual processors with 1.2 Gigabytes of RAM, running on linux) are used. For parallel testing, an SGI Origin 2000 (64 195MHz R10000 processors with 32 Gigabytes of RAM) and an IBM Linux Cluster (total 1038 1GHz and 1.266GHz Pentium III processors on 519 nodes, 607.5 Gigabytes of RAM) are used. On the SGI machines, the applications are compiled using the SGI CC compiler. On Intel PC, the GNU g++ compiler is used.

7.2 Issues in Evaluation

Loci uses the system allocator `malloc` at the lowest level. Usually the system call `brk` is used inside `malloc` to ask for heap space from the operating system. `malloc`

usually asks for a large block of memory from `brk` call, and then it partitions the block and supplies space for application requests. The same strategy is used when freeing memory. `malloc` aggregates large enough amount of blocks and then calls `brk` to shrink the heap space. This caching strategy is helpful for performance improvement because `brk` is an expensive system call. In the GNU C library, by default, `malloc` requests large allocation through the `mmap` call to find addressable memory space. The difference from `brk` call is that the memory allocated through `mmap`, once freed, is immediately returned to the operating system. The advantage of this approach is that it helps to reduce memory fragmentation and makes large memory available to system faster. But from the measurement point of view, this introduces unpredictable operating system overhead into the program. We try to avoid such random overheads in the measurement. Therefore on all the Linux testing platforms, we disabled the `mmap` mechanism in `malloc` so that it always uses `brk` call to ask for memory. We also set the `brk` return threshold to be large enough that `malloc` never calls `brk` to return memory to the operating system.[1]

In space profiling, factors such as additional message buffer in the program, memory fragmentation, and the quality of memory allocator, etc. all affect the measured memory bound. In order to know the exact benefit from dynamic managing memory for variables in applications, we also perform bean-counting[2] on memory usage. A memory profiler is implemented for Loci. When activated, the memory profiler collects heap size information

[1]This is done by setting the environmental variables `MALLOC_TRIM_THRESHOLD_` and `MALLOC_MMAP_MAX_`.

[2]By bean-counting we mean tabulating the exact amount of memory requested from the allocator.

and also computes bean-counting memory bounds. Both real measurement and bean-counting measurement results are presented in the following sections.

7.3 Measurement Results

In the following tables and figures, if not otherwise specified, applications running with preallocation is abbreviated as *pre*, applications running with dynamic memory management is abbreviated as *dmm*, and applications running with dynamic memory management and chomping is referred to as *chomp* (note the chomping is used together with the dynamic memory management). Computation greedy schedule is abbreviated as *comp greedy*, and memory greedy schedule is abbreviated as *mem greedy*. Real measurement number is referred to as *real*, and bean-counting number is referred to as *bc*. The CHEM program can be configured to run under four different modes: implicit time method, implicit time method with chemistry model, explicit time method, and explicit time method with chemistry model. They are abbreviated as CHEM-I, CHEM-IC, CHEM-E, and CHEM-EC respectively. Measurements are conducted for all four modes. Because the implementation of the FUELCELL program is different than the CHEM program, we are only able to chomp one variable for the testing problem. Therefore, we do not measure the chomping option for the FUELCELL program. In the space and performance profiling sections, we mainly present the sequential measurement results. Because Loci is data-parallel, in the parallel case, each process executes the same program with a smaller dataset. The parallel results are essentially the same.

7.3.1 Loci Scheduler Statistics

Because Loci does the assembly and scheduling all at run-time, anything added into Loci should not severely degrade the performance of Loci scheduler. Therefore we first measure the run-time performance and behavior of the algorithms developed in this thesis in addition to application performance measurements.

Table 7.1 Statistics of Loci Scheduler

unit: second	CHEM-I	CHEM-IC	CHEM-E	CHEM-EC
dmm decoration	0.5188	0.5402	0.3540	0.3656
searching and forming of chomping chains	0.3595	0.3760	0.2620	0.2749
comp greedy schedule	0.0101	0.0103	0.0070	0.0071
mem greedy schedule	0.1270	0.1350	0.0682	0.0693
total Loci schedule time[a]	10.8339	10.9706	7.4180	7.5380

[a]This is the total time when using comp greedy schedule, using mem greedy schedule will have a similar result. This total time is the sum of Loci graph processing time and the existential analysis time.

Table 7.1 shows the run-time performance for various stages of the Loci scheduler under four different configurations for the CHEM program. This measurement is conducted on a 2GHz Intel Pentium 4 PC with Linux . The problem chosen is the same as the one used in the following space profiling and timing. Although this is a modest size problem that could be run on a single processor, it is of typical complexity in applications built using

Loci. The graph processing part of Loci only depends on the complexity of applications (i.e., the number of rules and variables involved), not the problem size (i.e., the size of variables). The algorithms discussed in this thesis belong to the graph processing part of Loci. Hence, they are independent of problem size.

From Table 7.1, we can conclude all algorithms developed in this thesis only add small amount of overhead to the Loci scheduler. Typical runtime of Loci applications range from several hours on a single processor to several days on large parallel machines. Therefore this amount of overhead is negligible. We also noticed the memory greedy schedule algorithm usually runs an order of magnitude slower than the computation greedy schedule. This is because more complex heuristics are used in the memory greedy scheduling. While the computation greedy scheduling only performs pure graph processing, or it could be regarded that each vertex has the same weight.

We next perform a measurement on the outcomes of the chomping searching and forming algorithm discussed in section 5.2 chapter V. Because if too many chompable variables are missed by the algorithm, we cannot get sufficient benefit from chomping. Thus the quality of this algorithm plays an important role in the chomping technique.

Table 7.2 shows the results of the algorithm for the same problem used in Table 7.1. In the table, “total variables” refers to all the variables that are allocated in the program, this does not include any input variables. The “upper bound of chompable variables” is the number of chomping candidates in the program. This represents an upper bound on the number of variables that can be chomped in the program. However, relationships

Table 7.2 Statistics of chomped variables

	CHEM-I	CHEM-IC	CHEM-E	CHEM-EC
total variables	192	196	162	166
upper bound of chompable variables	47	49	49	51
number of chomped variables	40	42	44	47
% of the size of chomped variables in total variables	32.25	32.39	44.74	51.03

between rules and variables may force us to discard some chompable variables. This is the major task performed in the searching and forming algorithm. We do not know whether the results of our algorithm for this problem are optimal, but they are good enough for practical use. Considering the size of the variables discovered by the algorithm, from the memory management point of view, doing chomping alone would save considerable amount of memory.

7.3.2 Space Profiling Results

The main objective of having memory management is to save memory. In comparing the memory bound, the measurements of application running with preallocation serve as the baseline. Preallocation could be regarded as the upper bound of memory usage for a program. For all the measurements in space profiling, we used a default 128KB chomping

size for all applications running with chomping. Because from the space profiling point of view, the chomping size does not affect the memory bound in a noticeable way.

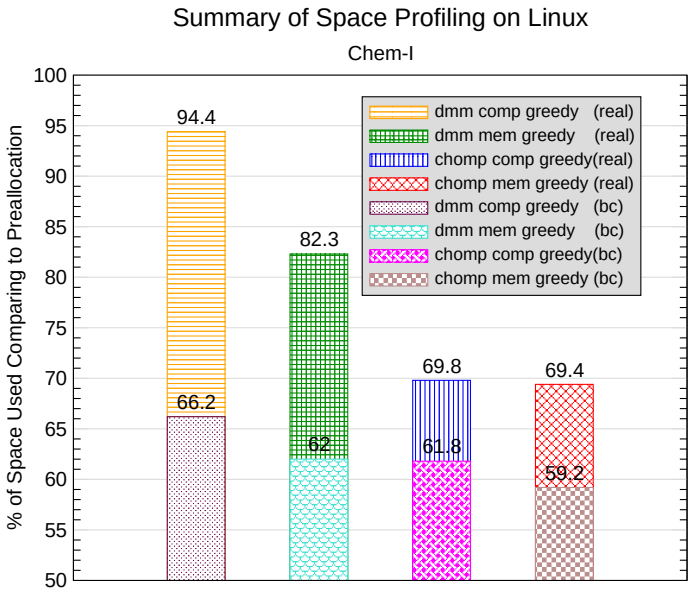

Figure 7.1 Summary of Space Profiling on Linux (Chem-I)

Figure 7.1 to Figure 7.4 summarize the space profiling results for CHEM on Linux. They are performed on a single node on the IBM Linux Cluster.

Figure 7.5 and Figure 7.6 summarizes the space profiling results for CHEM on SGI. The reason that we do not include the results for CHEM-I and CHEM-IC is that we noticed on SGI, when CHEM running with dynamic memory management and dynamic memory management with chomping under these two modes, the real measured memory usages exceed the bound of preallocation significantly. During the measurement, we also noticed when running under these two modes with “dmm” and “chomp”, the memory bound will

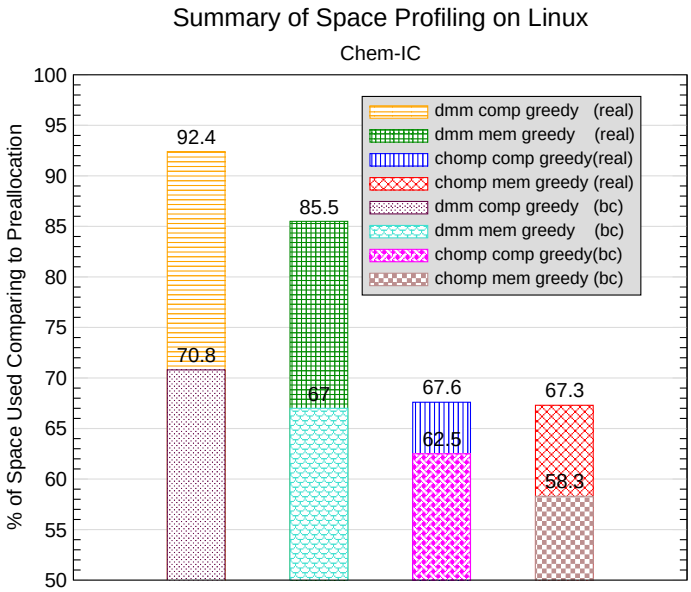

Figure 7.2 Summary of Space Profiling on Linux (Chem-IC)

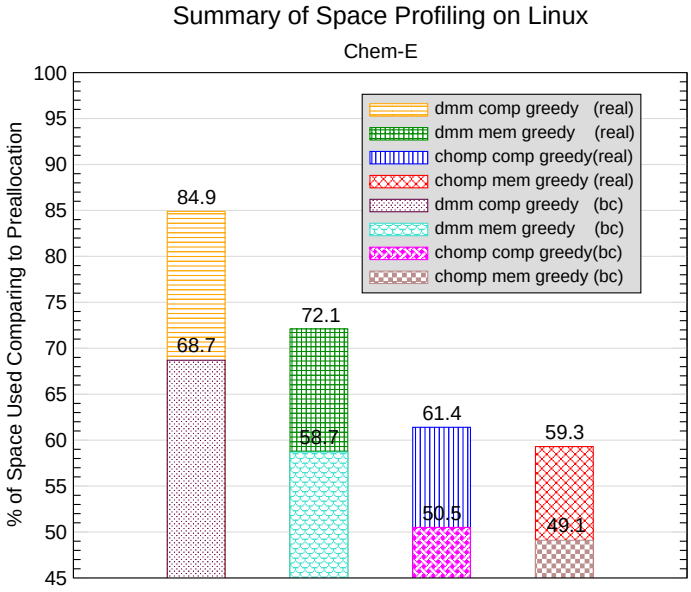

Figure 7.3 Summary of Space Profiling on Linux (Chem-E)

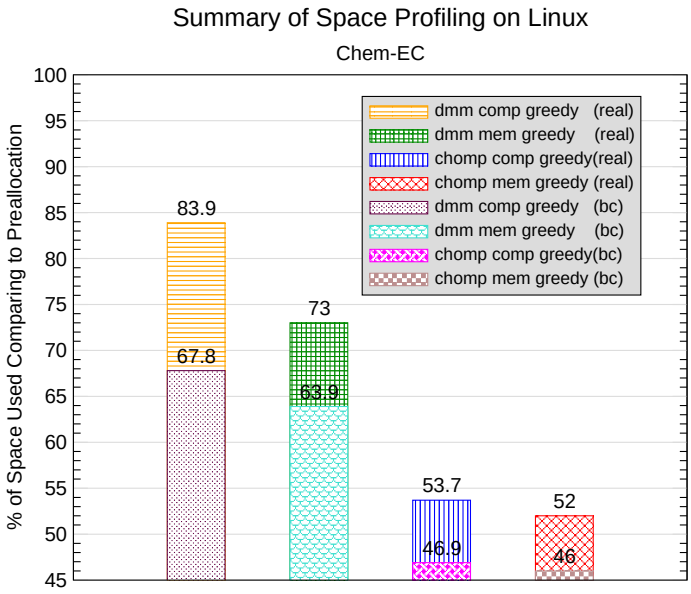

Figure 7.4 Summary of Space Profiling on Linux (Chem-EC)

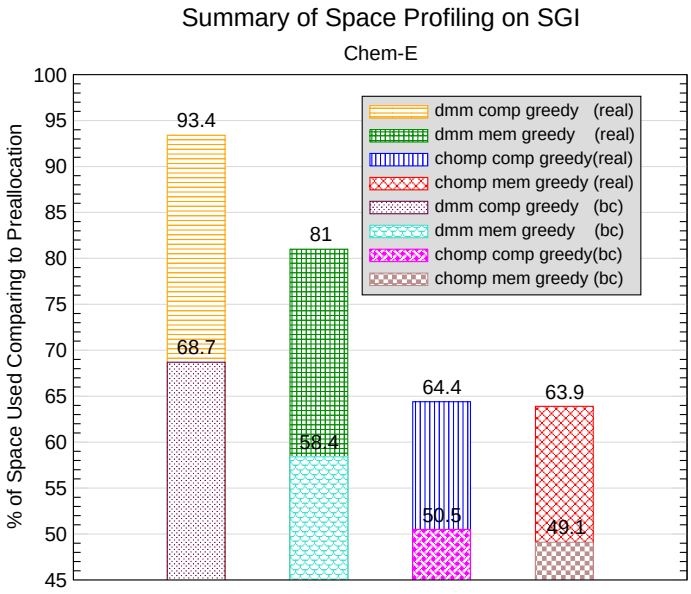

Figure 7.5 Summary of Space Profiling on SGI (Chem-E)

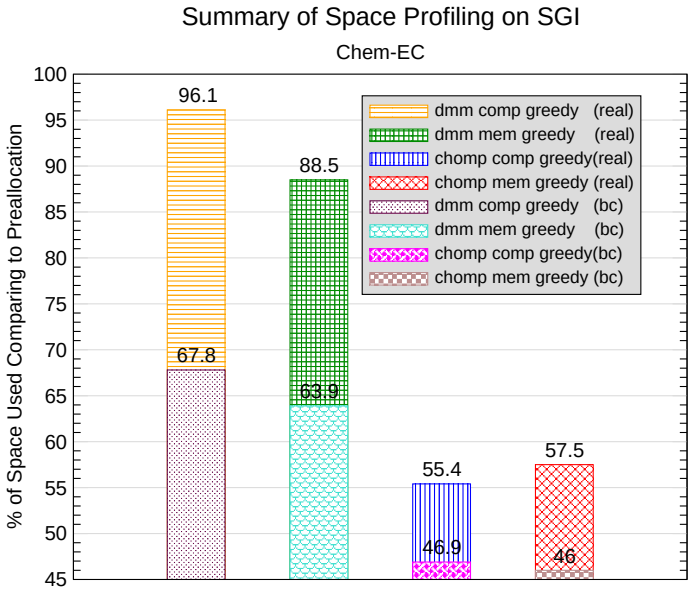

Figure 7.6 Summary of Space Profiling on SGI (Chem-EC)

gradually increase as the program runs. The reason for this abnormal behavior is still under investigation, and we leave it for future work. Our current explanation is that the allocator from the SGI compiler has serious fragmentation problems under these circumstances.[3] Fragmented memory cannot be reused and is not returned to the system, thus it has to repeatedly ask for memory from the operating system which in turn causes the memory usage increase as the program runs.

Table 7.3 shows the space profiling results of the FUELCELL program on the Intel Pentium III PC. We do not have a compilation of the FUELCELL program for SGI, therefore we only give measurements on Linux.

[3]Our code has passed a memory debugger under Linux.

Table 7.3 Space Profiling results for FUELCELL on Linux

	comp greedy		mem greedy	
unit: MB	real	bc	real	bc
pre	268.247	179.163	268.876	179.163
dmm	212.446	98.448	208.110	98.448

From the results, we can conclude using dynamic memory management helps to save memory. For CHEM, the saving ranges from roughly 4%–50%, depending on program configurations and the memory management options.

We also noticed when using "dmm" alone, the memory savings are typically not significant, especially when using computation greedy scheduling. However the theoretical numbers (the bean-counting results) indicate a more aggressive outcome. Our explanation to this is that the quality of allocator matters. For using "dmm" alone, the allocations tend to be large, the fragmented memory cannot be reused and causes a large peak memory. The results of "dmm" with memory greedy scheduling and "chomp" support this argument. We noticed a significant reduction of peak memory when using "chomp" compared to the corresponding case under "dmm" alone. While their bean-counting numbers show much smaller gaps. In chomping, not only does the program consume less memory, but also the allocation for chomped variables tend to be small. They therefore are less affected by memory fragmentation, since small blocks could possibly fit into fragmented memory. From the profiling results, we can see the bean-counting numbers for "dmm" with computation greedy scheduling and "dmm" with memory greedy scheduling typically differ

within 5%, while the differences between real measured numbers are usually more than 10%. Because the memory greedy scheduling is more aware of the memory usage, the allocation and deletion patterns are different than those in the computation greedy schedule. This difference also contributed to alleviate the memory fragmentation problem.

Table 7.3 suggests a large amount of memory is wasted in the FUELCELL program, even under preallocation. In "dmm," the bean-counting peak memory is only around 46% of the real measured peak. Part of this is due to the use of external solver in the FUELCELL program. The external solver may use and manage its own memory, and we did not count that part. We also noticed, for the FUELCELL program, memory greedy scheduling yields no obvious difference than computation greedy scheduling. This also indicates that the design of a program affects the memory bound. In order to maximize the advantage of dynamic memory management, it is suggested that one should try to use more variables in a program with approximately identical size and relatively short lifetime.

From the space profiling results, we conclude a good allocator is necessary to take advantage of the dynamic memory management. The fragmentation problem significantly affects the memory bound and the practicability of dynamic memory management. But we also discovered that chomping can greatly reduce the memory fragmentation problem under most cases. The reason is that chomping reduces the number of large allocations and the chomped variables can fit into fragmented memory more easily. From this point of view, it is also suggested that in the program design, using small size variables helps to alleviate the memory fragmentation problem. As an example, from the space profiling

results for CHEM on Linux, the "dmm" results are less suffered from memory fragmentation under CHEM-E and CHEM-EC modes compared to CHEM-I and CHEM-IC modes. Because under CHEM-E and CHEM-EC modes, the size of variables are much smaller than those in CHEM-I and CHEM-IC modes. Fragmentation is a major problem in allocator design and largely affects the quality of the allocator. Thus our space profiling results suggest if one has no choice of the allocator, then using memory greedy scheduling and chomping can alleviate the fragmentation problem.[4]

7.3.3 Performance Profiling Results

In this section, the timing results of dynamic memory management and chomping are presented and discussed. These results reflect the amount of run-time overhead associated with dynamic memory management and the benefit from chomping.

Figure 7.7 and Figure 7.8 summarize the timing results for CHEM on Linux (measured on a single node on the IBM Linux Cluster) and SGI respectively. The timing results for dynamic memory management and chomping are shown as relative speed to preallocation. Various chomping sizes are selected for testing with chomping. Table 7.4 exhibits the results for FUELCELL program on Linux.

During our measurement of timing on Linux, we found the results usually have large variations, depending on system and program configurations. We do not yet fully under-

[4]Enabling the "mmap" mechanism in GNU C library can also partially alleviate the fragmentation problem, since large memory blocks are allocated through "mmap" and are immediately returned to the system when freed.

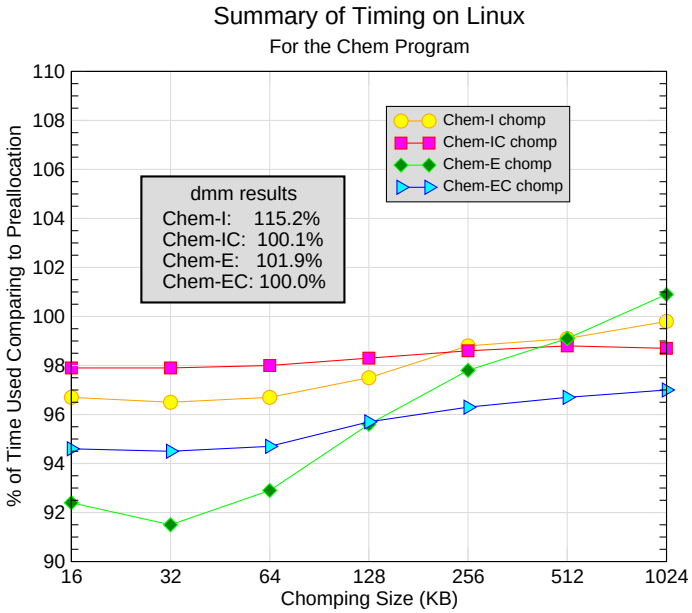

Figure 7.7 Summary of Performance Measurement for CHEM on Linux

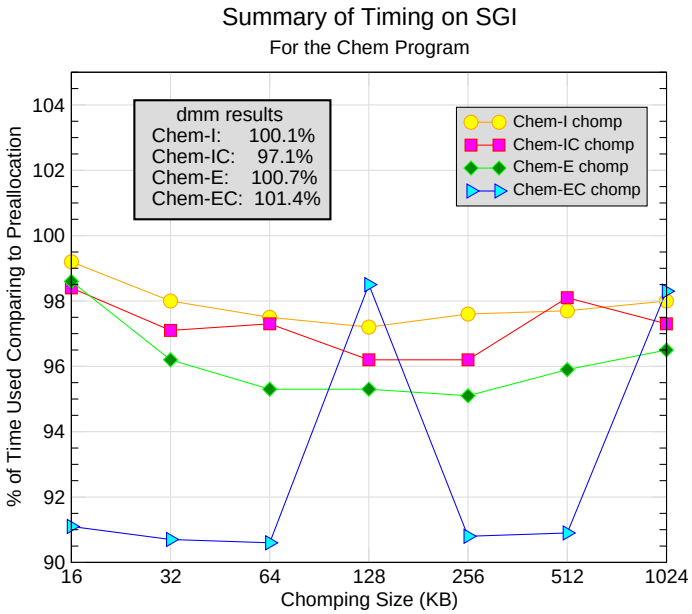

Figure 7.8 Summary of Performance Measurement for CHEM on SGI

Table 7.4 Timing results for FUELCELL on Linux

unit: second	time
pre	107.204
dmm	106.404

stand the reason for this result. We found the Linux operating system seems to be more actively involved when a program is running. There may exist other interactions between the operating system and the application program that we were not aware of. We leave the investigation as one of the future work. On the SGI architecture, the timing results are more consistent. On the SGI, applications with dynamic memory management is generally slightly slower than applications with preallocation, except for in one case (CHEM-IC), "dmm" is faster. From the measured data, we conclude for application with dynamic memory management, the run-time overhead is reasonable. On the SGI architecture, the largest overhead observed is about 1.4%. On Linux, only in CHEM-I, there is considerable amount of overhead. Other results are typically close to the preallocation measurement. The reason for large overhead in CHEM-I is unknown and in fact it is possibly due to random system interactions.[5] As stated, these issues are left for future study.

The timing results for chomping show that typically speedup is achieved, the performance of applications with chomping outperform the performance of applications with preallocation. But overall the speedup of chomping is below our expectation. There might

[5]Apparently it is not caused by short periodical system interruptions. For each data point in the figures, we took 10 sample measurements and there are no obvious deviations between all of the samples.

be several possible reasons. First, complex system interactions may destroy the chomping benefit, especially on the Linux operating system. Second, the study of chomping presented in this thesis provides an infrastructure that supports the chomping idea, there are other issues that are not considered. For example, the Linux PC and SGI server used in these testing cases have a 512KB level 2 data cache and a 2MB unified secondary cache. But we noticed the optimal chomping size is usually 32KB on Linux and 64KB to 256KB on the SGI, which is far below the cache size. This indicates our chomping strategy has other overheads. For example, as discussed in section 5.3 chapter V, the source and target variables in the chomp chain may present a considerable amount of overhead in chomping. These issues may adversely affect the performance of chomping, and they are candidates of our future study.

Table 7.5 Timing under Swapping for CHEM on Linux

	memory (MB)	swap (%)	time (s)
pre (10 iter)	700.331	50	2937.32
dmm (10 iter)	645.135	50	2445.59
chomp (128KB) (10 iter)	470.651	16	709.025
chomp (128KB) (120 iter)	470.651	48.4	7918.03

As an additional study to the potential benefit of chomping, we present at here an extreme case: the performance comparison of application with preallocation and chomping under swapping condition. This is measured on an Intel Pentium 4 PC with 512MB of RAM. CHEM is configured to run under implicit time method with chemistry model with

a larger grid for the same problem used in all previous measurements. We used the default 128KB chomping size in this case. The results are shown in Table 7.5.

For application running under preallocation, about 50% of the swap space usage is reached. "dmm" has a similar swap space usage. The fact that "dmm" is 1.2 times faster than preallocation is that the dynamic memory management causes less access to the swapped region. Chomping has superior benefit under this case. Application with chomping is more than 4 times faster than preallocation. Less memory allocation in chomping contributed to the less usage of swap space. Because chomped variables are allocated as small blocks, they do not have to be constantly swapped in and out. For chomping, we also noticed a gradual increase of the swap space usage as the program runs. Under another measurement for chomping, we increased the computation 12 times. We observed the swap space usage reaches around 48.4% during approximately half way of the computation and stays stable at this level thereafter. But the timing result scales well, actually even slightly better than linear. This indicates the access to swapped region does not increase as the swap space usage grows. This case study also suggests a possible benefit for chomping in the future. As the gap of cache access speed and main memory access speed grows more towards the main memory and disk access gap, chomping could potentially have significant impact on program performance as suggested in this case.

7.3.4 Memory Utilization vs. Communication Costs

In order to examine the trade-offs between memory utilization and parallel communication costs, we utilize a large IBM Linux Cluster for the measurements presented in this section. Each test is conducted on 32 processors on the cluster.

Table 7.6 Mem vs. Comm under dmm on Linux Cluster

	memory usage (MB)		sync		time
	real	bc	points	time (s)	ratio(%)
comp greedy	372.352	174.464	32	3177.98	1
mem greedy	329.305	158.781	50	3179.24	1.0004

Table 7.7 Mem vs. Comm under chomping on Linux Cluster

	memory usage (MB)		sync		time
	real	bc	points	time (s)	ratio(%)
comp greedy	307.133	171.628	32	2987.95	1
mem greedy	299.743	164.721	50	2994.05	1.0020

Table 7.6 and Table 7.7 show the results for a typical large Loci application running under implicit time method. We noticed the difference of peak memory usage to be some what significant when running with "dmm" alone. However, the timing results are almost identical. In the memory greedy schedule, the synchronization points are about 1.6 times more than those in the computation greedy schedule. But the slowdown due to increased

synchronization points is virtually negligible in both tables. A possible explanation is that the application is computationally intensive, the additional communication startup costs do not contribute significantly to the total execution time.

Table 7.8 Mem vs. Comm under dmm on Linux Cluster (a small case)

	bc(MB)	sync points	time (s)	time ratio(%)
comp greedy	1.07	32	1269.59	1
mem greedy	0.92	52	1436.07	1.13

Table 7.9 Mem vs. Comm under chomping on Linux Cluster (a small case)

	bc(MB)	sync points	time (s)	time ratio(%)
comp greedy	1.08	32	1155.55	1
mem greedy	1.05	52	1699.33	1.47

In order to verify our hypothesis and study the extreme case that the increased synchronization points could have, we created another test. In this test, we selected a much smaller problem (more than 100 times smaller than the previous one), but running under the same configuration as the previous measurement. We choose such a problem with the hope that the parallel communication could be a major factor, if not dominant. The results are shown in Table 7.8 and Table 7.9.

In this case, we put the emphasis on the communication side because we want to study the potential effect of increases in synchronization frequency in a program. On the other side, this is a small problem, when distributed to each processor, each one gets even smaller dataset. The bean-counting peak memory is listed in the tables only for reference purpose to show the distinctions between computation and memory greedy scheduling. In fact, the memory bound is dominated by the size of program instructions and internal data-structure in this case. Note, the fact that the bean-counting peak memory for chomping is larger than the corresponding one for "dmm" is because at this level of memory size, the actual chomping size may be a dominant factor. We used 128KB for chomping size in this case and did not optimize the allocation of chomping blocks for small problems.

The timing results show that the increased synchronization points have a significant effect on the total execution time for such small problems. In the schedule, the synchronization points increased roughly 1.6 times from computation greedy to memory greedy. When running with chomping, the execution time increased about 1.5 times. The slow-down for "dmm" is less than this level, but we consider it to be relatively significant.

Consider the "dmm" case in Table 7.8, the bean-counting numbers show that roughly 14% of the memory saving is achieved from computation greedy schedule to memory greedy schedule, contrary, approximately same amount of performance improvement is achieved from memory greedy schedule to computation greedy schedule. Of course, this is an exaggerated case. But it does show that under certain circumstances, trade-offs do

exist between memory utilization and parallel communication costs. And these trade-offs can be utilized by the framework to maximize its flexibility.

The results in this section also suggest for computation intensive problem, the additional communication startup costs do not have noticeable effect on the application performance. Therefore for this type of applications, the memory greedy scheduling is preferred because it potentially saves memory without undue performance overhead.

7.4 Summary

This section presents extensive experimental results for various topics studied in this thesis. They range from the run-time property and behavior of algorithms developed in this thesis to various outcomes of these algorithms. Analyses and discussion of various implications of these results are also given in this chapter.

CHAPTER VIII

CONCLUSIONS

The study presented in this thesis proposed a high-level scheme for dynamic memory management for a declarative data-parallel programming system — the Loci framework. In addition to basic memory management, the proposed scheme also tries to take advantage of the cache memory subsystem to improve the application performance. As a side-effect of introducing dynamic memory management, this study also presented the existence of performance trade-offs in memory utilization and parallel communication costs. A balanced approach will require interactions between the memory and communication scheduling strategies.

The experimental results support the primary hypotheses proposed in this thesis. Application with dynamic memory management have a lower peak memory bound. Under certain configurations, especially when combining with chomping technique and memory greedy scheduling, the savings are relatively significant. Having dynamic memory management does not unduly affect the program execution time. In fact, under most cases, the overhead is negligible. By taking advantage of the cache memory subsystem, the chomping technique improves the application performance. The benefit depends on actual architecture and program configurations. On the memory side, chomping not only reduces

the theoretical memory bound, but also contributes to alleviate the memory fragmentation problem. The dynamic memory management scheme usually achieves maximum benefit when combining with chomping. Memory greedy scheduling reduces the memory bound and typically helps to alleviate the memory fragmentation problem. As expected, memory greedy scheduling also increases the amount of synchronization points in a parallel schedule. But we found, in computation intensive applications, the increased communication startup costs is typically negligible. Only in communication bounded applications, the number of synchronization points affect the execution performance. Thus, the trade-offs between memory utilization and parallel communication costs exist under certain circumstances. This suggests, only under certain circumstances, the system can take advantage of this type of trade-offs.

This presented study also provides an infrastructure for further exploring memory system management in declarative data-parallel programming systems and resource management in general.

8.1 For Future Research

Some of the discussions in chapter VII already suggested possible future work. In particular, we aim to provide a high-level management strategy in this thesis and start off by assuming that this strategy will work smooth with low-level details. Now by looking at some of the experimentation results, we found our assumption to be some what optimistic. It is time now to look back some of these issues. In particular, as shown in chapter VII,

we have two main issues: the dubious memory leak problem on SGI, and the irregular timing behavior on Linux. Identifying and solving these problems can greatly improve the practicability of the techniques developed in this thesis.

Second, the performance improvements by chomping is below our expectation. After measurements in chapter VII, we created a fictitious Loci program that is highly optimized for chomping. We found the performance improvements to be quite significant. We achieved 1.5 times speedup on Linux and 5 times speedup on the SGI. Although this is an exaggerated case and is unlikely to appear in real design, it suggests there are still rooms for improvements. In particular, we want to investigate the effect to chomping performance by using different scheduling algorithms that are optimized for cache performance. By this time, the chomping graph is scheduled using the default computation greedy algorithm. This may not be the best choice for scheduling chomping graph because it is not aware of preserving the cache benefit. In other words, it could schedule things that destroy the cache benefit. Thus, we want to investigate the effect of a cache-aware scheduling algorithm.

REFERENCES

[1] A. Aiken, M. Faehndrich, and R. Levien, "Better Static Memory Management: Improvements to Region-Based Analysis of Higher-Order Languages," *Proceedings: SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, 1995, pp. 174–185.

[2] G. Attardi, T. Flagella, and P. Iglio, "A customisable memory management framework for C++," *Software Practice and Experience*, vol. 28, no. 11, 1998, pp. 1143–1183.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object-oriented numerical software libraries," *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997, pp. 163–202.

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Composing High-Performance Memory Allocators," *Proceedings: SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, 2001.

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-Oblivious Algorithms," *Proceedings: The 40th Annual Symposium on Foundations of Computer Science, FOCS '99*, New York, NY, 1999.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995.

[7] D. Gay and A. Aiken, "Memory Management with Explicit Regions," *Proceedings: SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, 1998, pp. 313–323.

[8] D. Gay and A. Aiken, "Language Support for Regions," *Proceedings: SIGPLAN Conference on Programming Language Design and Implementation*, 2001, pp. 70–80.

[9] D. Grunwald, B. Zorn, and R. Henderson, "Improving the Cache Locality of Memory Allocation," *Proceedings: SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, 1993, pp. 177–186.

[10] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. J. Humphrey, J. Reynders, S. Smith, and T. Williams, "Array Design and Expression Evaluation in POOMA II," *Proceedings: International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Santa Fe, New Mexico, 1998, Springer-Verlag, pp. 231–238.

[11] A. LaMarca and R. E. Ladner, "The Influence of Caches on the Performance of Sorting," *Journal of Algorithms*, vol. 31, 1999, pp. 66–104.

[12] E. A. Luke, "Loci: A Deductive Framework for Graph-Based Algorithms," *Proceedings: Third International Symposium on Computing in Object-Oriented Parallel Environments*, San Fransisco, California, 1999, Springer-Verlag, pp. 142–153.

[13] N. Mitchell, L. Carter, and J. Ferrante, "Localizing Non-affine Array References," *Proceedings: Parallel Architectures and Compilation Techniques '99*, Newport Beach, California, 1999, IEEE Computer Society and IFIP Working Group 10.3.

[14] J. G. Siek and A. Lumsdaine, "The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra," *Proceedings: International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Santa Fe, New Mexico, 1998, Springer-Verlag, pp. 59–70.

[15] T. Sterling, D. Becker, D. Savarse, J. Dorband, U. Ranawak, and C. Packer, "Beowulf: A Parallel Workstation for Scientific Computation," *Proceedings: The 1995 International Conference on Parallel Processing*, 1995, pp. 11–14.

[16] M. Tofte and L. Birkedal, "A Region Inference Algorithm," *Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 5, July 1998, pp. 734–767.

[17] T. L. Veldhuizen, "Arrays in Blitz++," *Proceedings: International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Santa Fe, New Mexico, 1998, Springer-Verlag, pp. 223–230.

[18] P. R. Wilson, "Uniprocessor Garbage Collection Techniques," *Proceedings: International Workshop on Memory Management*, St. Malo, France, 1992, Springer-Verlag.

[19] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," *Proceedings: International Workshop on Memory Management*, Kinross, Scotland, 1995, Springer-Verlag.