Scalable Computing: Practice and Experience
Volume 7, Number 3, pp. 27–37. http://www.scpe.org
ISSN 1895-1767

DYNAMIC MEMORY MANAGEMENT IN THE *LOCI* FRAMEWORK

YANG ZHANG AND EDWARD A. LUKE*

Abstract. Resource management is a critical concern 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 paper presents a dynamic memory management scheme for a declarative high-performance data-parallel programming system—the *Loci* framework. In such systems, some sort of automatic resource management is a requirement. We present an automatic memory management scheme that provides good compromise between memory utilization and speed. In addition to basic memory management, we also develop methods that take advantages of the cache memory subsystem and explore balances between memory utilization and parallel communication costs.

Key words. Memory management, declarative languages, parallel programming, software synthesis

1. Introduction. In this paper we discuss the design and implementation of a dynamic memory management strategy for the declarative programming framework, *Loci* [5, 6]. The *Loci* framework provides a rule-based programming model for numerical and scientific simulation similar to the *Datalog* [12] logic programming model for relational databases. In *Loci*, the arrays typically found in scientific applications are treated as relations, and computations are treated as transformation rules. The framework provides a planner, similar to the FFTW [3] library, that generates a schedule of subroutine calls that will obtain a particular user specified goal. *Loci* provides a range of automatic resource management facilities such as automatic parallel scheduling for distributed memory architectures and automatic load balancing. The *Loci* framework has demonstrated predictable performance behavior and efficient utilization of large scale distributed memory architectures on problems of significant complexity with multiple disciplines involved [6]. *Loci* and its applications are in active and routine use by engineers at various NASA centers in the support of rocket system design and testing.

The *Loci* planner is divided into several major stages. The first stage is a dependency analysis which generates a dependency graph that describes a partial ordering of computations from the initial facts to the requested goal. In the second stage, the dependency graph is sub-divided into functional groups that are further partitioned into a collection of directed acyclic graphs (DAGs). In the third stage, the partitioned graphs are decorated with resource management constraints (such as memory management constraints). In the forth stage a proto-plan is formed by determining an ordering of DAG vertices to form computation super-steps. (In the final parallel schedule, these steps are similar to the super-steps of the Bulk Synchronous Parallel (BSP) model [13, 10, 2].) The proto-plan is used to perform analysis on the generation of relations by rules as well as the communication schedule to be performed at the end of each computation step in the fifth and sixth stages (existential analysis and pruning), as described in more detail in this recent article [6]. Finally the information collected in these stages is used to generate an execution plan in the seventh stage. Dynamic memory management is primarily implemented as modifications to the third and fourth stages of *Loci* planning.

2. Related Work. The memory system and its management has been studied extensively in the past. These studies are on various different levels. On the software level, memory management can be roughly categorized into allocation techniques and management strategies. Allocation techniques mostly deal with how memory is requested and returned to the operating system in order to efficiently satisfy application requests. Memory management strategies often study how and when to recycle useless memory. Allocation is usually performed by the "allocator," which is typically implemented as a library component (such as the `malloc` routine in the standard C library). The central themes in various allocation techniques are fragmentation and locality problems. If care is not taken, then the allocator could build up large internal fragmentation with significant inaccessible memory. The locality property in the allocator can greatly affect the cache and page misses and hence also contributes to the program performance. Wilson et al. [15] has an excellent survey for various allocation techniques. The memory management strategies can be subdivided mainly into two directions: one is to managing memory manually; while the other direction is to automatically reclaim useless memory. Manual memory management is usually performed by explicit programming. Programmer has full control over memory recycling. There has been much debate concerning various aspects of the advantages and disadvantages for

*Department of Computer Science and Engineering, and Computational Simulation and Design Center, Mississippi State University, Mississippi State, MS 39762. Questions, comments, or corrections may be directed to the first author at fz15@cse.msstate.edu

manual memory recycling. But a general consensus is that for large complex systems, a manual strategy is not encouraged due to its complex interaction with other software components. Automatic memory management frees the programmers from bookkeeping details of reclaiming memory and is typically a built-in feature in many modern languages such as *Java*, *ML*, *Smalltalk*, etc. The most prevalent technique for automatic memory recycling is "garbage collection" where the run-time system periodically reclaims useless memory [14]. Recent studies proposed "region inference" as another technique for automatic memory recycling. Region inference [11] relies on static program analysis and is a compile-time method and uses the region concept. The compiler analyzes the source program and infers the allocation. In addition to being fully automatic, it also has the advantage of reducing the run-time overhead found in garbage collection.

When designing the memory management subsystem for *Loci*, we are mostly interested in designing a memory management strategy and not in low level allocator designs. The programming model in *Loci* is declarative, which means the user does not have direct control of allocation. Also one major goal of the *Loci* framework is to hide irrelevant details from the user. Therefore we are interested in designing an automatic memory management scheme. Garbage collection typically works better for small allocations in a dynamic environment. While in *Loci*, the data-structures are often static; and allocations are typically large. Thus, the applicability of garbage collection to this domain is uncertain. Another problem of garbage collection is that the time required for collecting garbage cannot be predicted easily. While there has been research work on real-time garbage collection [4, 9] that attempt to address such issues, we find a memory management scheme without garbage collection to be straightforward and easy to reason about. Therefore instead of applying traditional garbage collection techniques, we have adopted a strategy that shares some similarities to the region inference techniques as will be described in the following sections. We also note interactions between the parallel scheduling of tasks and memory management strategies. Similar interactions have been observed in recent studies in continuous data streams [1] where it is demonstrated that operator scheduling order in the context of continuous data streams can affect overall memory requirements. They suggested a near-optimal strategy in the context of stream models. Although this is in a context that differs from our data-parallel programming domain, it shares some similarities with our approaches in balancing the memory utilization and parallel communication costs within the *Loci* framework.

3. Basic Dynamic Memory Management. In *Loci*, the aggregations of attributes found in scientific computing are treated as binary relations and are stored in *Loci* provided value containers. These value containers are the major source of memory consumption. Therefore the management of allocation and deallocation of these containers is the major focus of our memory management scheme. A simple way to manage the lifetime of these containers is *preallocation*. In this approach we take advantage of the *Loci* planner's ability to predict the sizes of the containers in advance. In the preallocation scheme, all containers are allocated at the beginning and recycled only at the end of the schedule. While this scheme is simple and has little run-time overhead, it does not offer any benefits for saving space. Scientific applications for which *Loci* is targeted tend to have large memory requirements. The primary goal of the management is therefore to reduce the peak memory requirement so that larger problems can be solved on the same system. Preallocation obviously fails this purpose.

After the user submits a request, the *Loci* planner generates a dependency graph. This graph describes the relationship between rules that obtains the specified goal. The dependency graph usually contains cycles caused by the specification of iteration, conditional execution blocks, and rule recursion. To simplify scheduling, the dependency graph is partition to a hierarchical graph where each level contains a DAG. Cycles have been removed in this graph and replaced by super-nodes that represent the semantics (e.g. iteration or recursion). This hierarchical graph is referred to as the multi-level graph. Thus, most of *Loci* scheduling is reduced to scheduling a DAG of rules. A simple approach to incorporating appropriate memory scheduling would be to incorporate relevant memory management operations into the multi-level graph. Then, when the graph is compiled, proper memory management instructions are included into the schedule and will be invoked in execution. We refer this process of including memory management instructions into the dependency graph as graph decoration. Thus memory management for *Loci* becomes the graph decoration problem. For example, Fig. 3.1 shows a decoration for a simple DAG. However, the multi-level dependency graph for a real application is likely to be complex. For example, multiple nested iterations and conditional specifications, recursions, etc. could also be involved. A global analysis of the graph is performed to determine the lifetime of all containers in the final schedule [16].

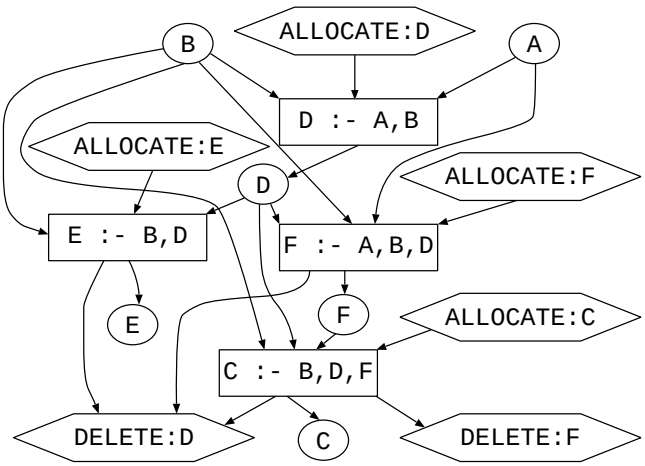


FIG. 3.1. *Memory Management by means of Graph Decoration*

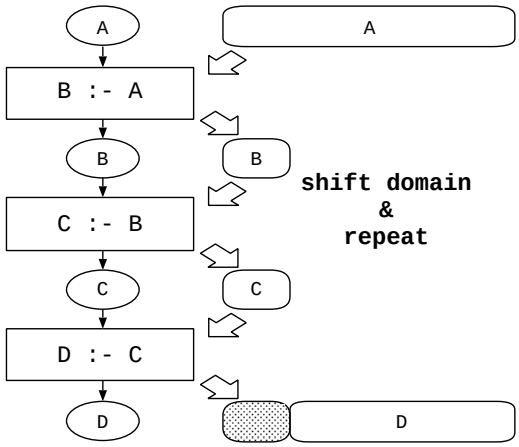


FIG. 4.1. *The Chomping Idea*

4. Chomping. Chomping is a technique we used in *Loci* to optimize the cache performance. The idea of chomping is borrowed from the commonly known loop scheduling technique: strip mining. In *Loci*, relations, the primary data abstractions, are collections of attributes that are stored in array-like containers that represent aggregations of values. Since these containers dominate the space consumed by *Loci* applications, they are ideal candidates for cache optimization by data partitioning. Data partitioning also creates further chance for memory savings in addition to the basic memory management implemented in *Loci*. Consider the rule chain in Fig. 4.1. Relation A is the source to the chain and D is the final derived relation; B and C are intermediate relations. We can break the rules in the chain into small sub-computations. In each of these sub-computation, only part of the derived relations are produced. This implies for any intermediate relations, only partial allocation of their container is required. Because these partial allocations can be made small, they enhance cache utilization and can further reduce memory requirements. Breaking computations into smaller intermediate segments not only reduces absolute memory allocation requirements, but also helps to reduce fragmentation by reusing a pool of small uniformly sized memory segments.

4.1. Chomping Implementation. The implementation of chomping extends the partitioning second stage in the *Loci* planner. In each level in the multi-level dependency graph generated by the *Loci* planner, all suitable rule chains for chomping are first identified. Then each chain is replaced by a special chomping rule and is handled separately in the graph compilation phase. The replacement is illustrated in Fig. 4.2. This allows smooth integration of chomping and dynamic memory management implemented in the *Loci* planner. As we presented in section 3, memory management in the *Loci* planner is implemented as a graph decoration

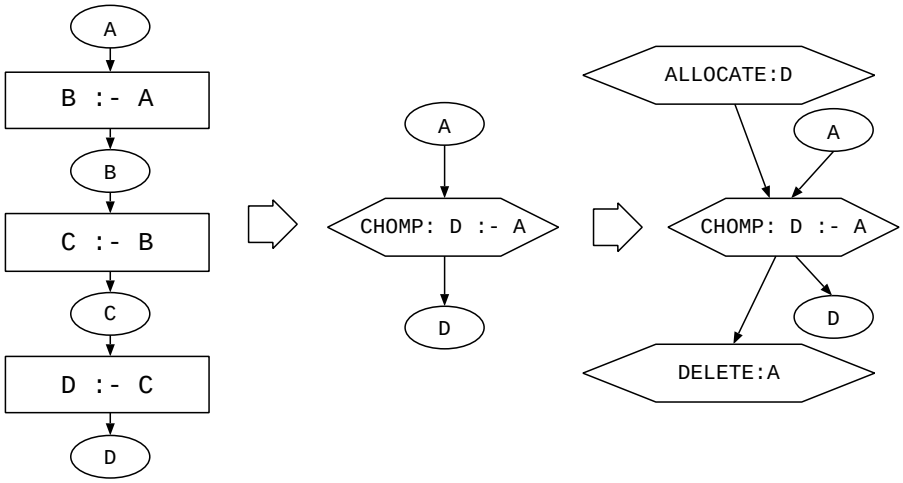


FIG. 4.2. *Implementation of Chomping*

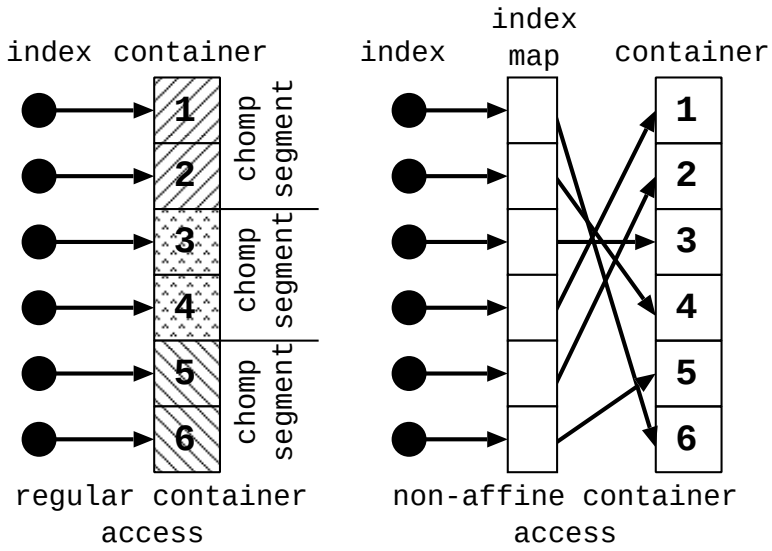


FIG. 4.3. *Memory Reference Patterns*

problem. The graph decorator does not have to know the chomping chain. For example, for the replaced graph in Fig. 4.2, the decorator can proceed as normal without the knowledge of the chomping chain. The memory allocation of the chomped relations B and C are handled internally in the chomping rule.

Obviously, the central problem in the implementation is how we identify suitable rule chains that can be chomped in a DAG in the multi-level dependency graph. Because of the existence of non-affine memory references in *Loci* rules, we cannot group arbitrary rules into rule chains that can be chomped. Consider the examples of memory access shown in Fig. 4.3. For a regular container access, there is a pre-determined access order to the domain of the container. For the case shown in Fig. 4.3, the access order to the domain is $[1, 2, 3, 4, 5, 6]$. Since this pattern is pre-determined, we can allocate memory for domain $[1, 2]$ first and perform the computation on this sub-domain of the container. Then we can shift this sub-domain by a distance of 2 and obtain the following sub-domain $[3, 4]$, and finally $[5, 6]$. Therefore the container and the rules associated can be chomped directly. For a non-affine container access, essentially we have the pattern of $A[B[i]]$. A is the container and B is the index map. The index map is often unknown until run-time. Therefore the access pattern to the container is also unknown until run-time. For the case shown in Fig. 4.3, the access order to the domain is $[6, 4, 3, 1, 2, 5]$. We have no direct way to allocate and shift sub-domains as in the regular container access. As

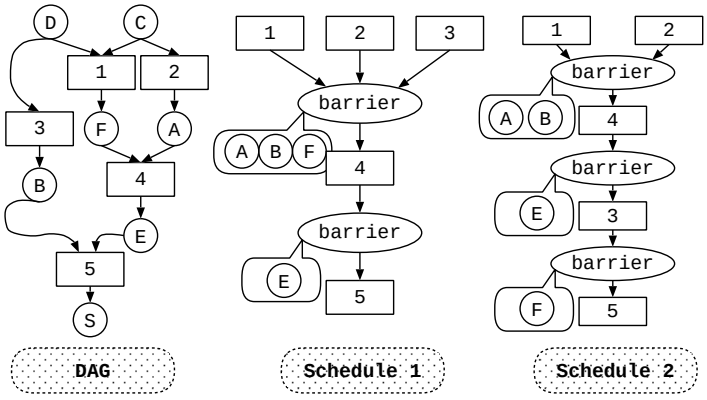


FIG. 5.1. *Different Scheduling for a DAG*

a result, the container and its associated rules cannot be chomped directly. In *Loci*, we use a heuristic search to identify suitable chains in the multi-level dependency graph and apply chomping only to them. The running time and the searching results of our algorithm are of satisfactory for large *Loci* applications. In section 6, we include an empirical evaluation of this heuristic chomping searching algorithm.

4.2. The Decision of Chomping Size. The total allocation size for chomped relations in a chomping rule is referred to as chomping size in the *Loci* planner. For example, the chomping size for the chomping chain in Fig. 4.1 is the total memory allocation for segments for relations B and C. Ideally, as in matrix blocking algorithms, the chomping size should be approximately the size of the data cache in order to utilize the cache performance. However, in *Loci*, we typically set it to be approximately half the size of the data cache. Chomping in *Loci* does not completely resemble typical cache optimization techniques such as matrix blocking algorithms. The source and target relations for a chomping rule chain is not chomped. They could be large and rules in the middle of the chomping chain could also have access (or non-affine access in the worst case) to the source relations. These potentially destroy the cache benefits obtained through chomping. We set the chomping size smaller than the data cache size with the hope to alleviate some of these problems. In our implementation, users can also specify a particular chomping size for the *Loci* planner. Chomping may create some further chances for program optimization, we will touch some of these in our conclusion section.

5. Memory Utilization and Parallel Communication Costs. In section 3, we transformed the memory management into a graph decoration problem. However the graph decoration only specifies a dependencies between memory management and computation. It is up to the *Loci* planner to generate a particular execution order that satisfies this dependence relationship. From the memory management point of view, the order to schedule allocation and deallocation affects the peak memory requirement of the application. On the other hand, the *Loci* planner can produce a data-parallel schedule. In the data-parallel model, after each super-step, processors need to synchronize data among the processes. From the communication point of view, different schedules may create different numbers of synchronization points. While the number of synchronization points does not change the total volume of data communicated, increased synchronization does reduce the opportunity to combine communication schedules to reduce start-up costs and latency. Thus with respect to parallel overhead, less synchronization is preferred.

Figure 5.1 shows the effect of different scheduling of a DAG. Schedule one is greedy on computation, a rule is scheduled as early as possible. Therefore schedule one has fewer synchronization points. Schedule two is greedy on memory, a rule is scheduled as late as possible. Therefore derived relations are spread over more super-steps, hence more synchronization points are needed.

A trade-off therefore exists in the *Loci* planner. In order to optimize memory utilization and reduce peak memory requirement, the planner will typically generate a schedule with more synchronization points, and therefore increase the communication start-up costs and slow down the execution. Attempting to minimize the synchronization points in a schedule results in a fast execution, but with more memory usage. Such trade-off 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 within limited

resources. On the other hand, if time is the major issue, then a computation greedy schedule is preferred, but users have to supply more memory to obtain speed. In the *Loci* planner, we have implemented two different scheduling algorithms. One is a simple computation greedy scheduling algorithm, which minimizes the total synchronization points. The other one is a memory greedy scheduling algorithm. It relies on heuristics to attempt to minimize the memory usage. Users of *Loci* can instruct the planner to choose either of the two policies.

The scheduling infrastructure in the *Loci* planner is priority based. *Loci* planner schedules a DAG according to the weight of each vertex. In this sense, scheduling policies can be implemented by providing different weights to the vertices. For a computation greedy schedule, we simply set the same weight for each vertex in the graph, since vertices with same weight will be scheduled together according to the graph topology. This effectively schedules all possible rules together to form a super-step and hence minimizes the barrier points needed to synchronize intermediate results among processes.

```
PrioGraph(gr)
 1  l ← NIL
 2  for vi ∈ V
 3       do a ← AllocNum(vi)
 4          d ← DelNum(vi)
 5          o ← TargetOutEdgeNum(vi)
 6          l ← Append(l, (vi, a, d, o))
 7  prio ← 0
 8  for i ← 1 to Length(l)
 9       do s ← l[i]
10          if s.a = 0
11             then p[s.vi] ← prio
12                  Erase(l, l[i])
```

```
 1  prio ← 1
 2  Sort(l, Ascend(a))
 3  StableSort(l, Descend(d))
 4  for i ← 1 to Length(l)
 5       do s ← l[i]
 6          if s.d ≠ 0
 7             then p[s.vi] ← prio
 8                  Erase(l, l[i])
 9                  prio ← prio +1
10  Sort(l, Ascend(o))
11  for i ← 1 to Length(l)
12       do s ← l[i]
13          p[s.vi] ← prio
14          prio ← prio +1
```

We also provide a heuristic for assigning vertices weight that attempts to minimize the memory utilization for the schedule. The central idea of the heuristic is to keep low memory usage in each scheduling step. Given a DAG with memory management decoration, rules that do not cause memory allocation have the highest priority and are scheduled first. They are packed into a single step in the schedule. If no such rules 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 deallocation. We schedule one such rule that causes most deallocations. If multiple rules have the same number of deallocations, 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 relations that it produces. This is based on the assumption that the more outgoing edges a relation has in a DAG, the more places will it be consumed, hence the relation will have a longer lifetime.

We used a sorting based algorithm in *Loci* for computing vertex priority based on the heuristics described above for memory minimization, which is shown in the procedure PrioGraph. Given a graph, we start off from building a list of statistical information for each vertex. Line 3 to line 5 compute the allocation number, the deallocation number, and the number of outgoing edges for all target relations respectively for every rule (for a relation, these numbers are all 0). Then all vertices that do not have allocation number get a priority of 0, which represents the highest priority. Then we sort the remaining list first according to the ascending order of the allocation number (line 2) and then the descending order of deallocation number (line 3). After this, all the remaining rules will be ordered according to their deallocation and allocation number. We assign appropriate priority to each rule that causes deallocation. The remaining rules are sorted again according to the number of outgoing edges for target relations (line 10) and priorities are assigned accordingly.

6. Experimental Results. In this section, we present some of our measurements for the work discussed in the previous sections. First of all, *Loci* planning is carried out at run-time. Our work in memory management incurs some additional costs to the planner. We performed a measurement first for the planner itself in order to evaluate the planning performance. Table 6.1 shows our measurement of various planning stages discussed in

previous sections. The measurement is performed on a typical Linux workstation for an average complex *Loci* application. We can conclude that the planning overhead is virtually negligible since typical running time for *Loci* applications range from hours to several days on large parallel machines. For substantially larger problems (e.g., a complex unstructured grid), the total planning time will increase correspondingly. But the planning for the work addressed in the paper only depends on the number of rules and relations in an application and does not relate to the input problem size. The measurements here should be a good suggestion for practical problems we are currently considering.

TABLE 6.1
Loci planner statistics

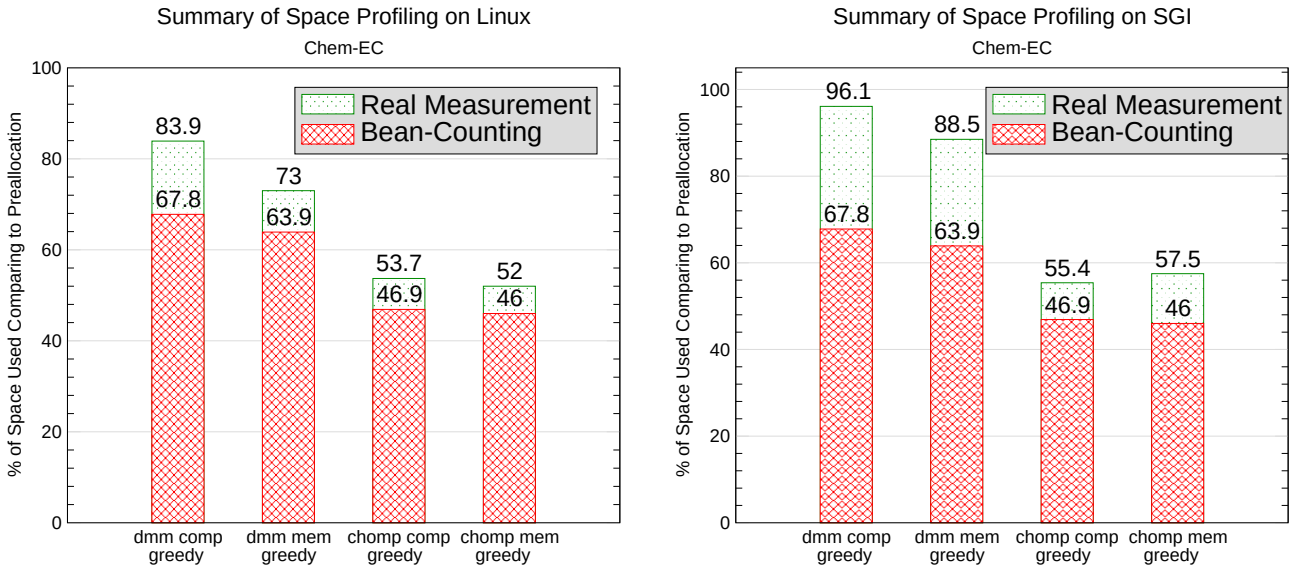
	unit: second
decoration	0.5402
chomping chain searching	0.3760
computation greedy schedule	0.0103
memory greedy schedule	0.1350
total *Loci* planning time	10.9706

We used the CHEM program as the benchmark in our profiling. CHEM [7, 8] is a finite-rate non-equilibrium Navier-Stokes solver for generalized grids fully implemented using the *Loci* framework. CHEM can be configured to run in several different modes, they are abbreviated as Chem-I, Chem-IC, Chem-E, and Chem-EC in the following figures and tables. An IBM Linux Cluster (total 1038 1GHz and 1.266GHz Pentium III processors on 519 nodes, 607.5 Gigabytes of RAM), and various Linux and SGI workstations are used in the measurement. In addition to taking the measurement of the real memory usage, we also record the bean-counting memory usage numbers. (By bean-counting we mean tabulating the exact amount of memory requested from the allocator. It is shown as a reference as we use GNU GCC's allocator in *Loci*.) In most of the measurements, we are comparing the results with the preallocation scheme mentioned in section 3, as the preallocation scheme represents the upper-bound for space requirement and the lower-bound for run-time management overhead.

We did extensive profiling of the memory utilization on various architectures. Figure 6.1(a) shows a measurement of Chem-EC on a single node on the Linux cluster. Figure 6.1(b) shows the same measurement on an SGI workstation. The "dmm" in the figure means the measurement was performed with the dynamic memory management enabled; "chomp" means chomping was also activated in the measurement in addition to basic memory management. As can be found from the figures, when combining with memory greedy scheduling and chomping, the peak memory usage is reduced to at most 52% (on Linux) of preallocation peak memory usage. The actual peak memory also depends on the design of the application. We noticed that for some configurations, the difference between the real measurement and the bean-counting is quite large. We suspect that this is due to the quality of the memory allocator. We also found that under most cases, using chomping and memory greedy scheduling improves the memory fragmentation problem. We suspect this is attributable to allocations that are much smaller and regular, thus the same allocations may be more effectively reused.

Figure 6.2(a) shows one timing result for chomping on a single node on the Linux cluster. Figure 6.2(b) shows the same measurement on an SGI workstation. The results showed different chomping sizes for different CHEM configurations. Typically using chomping increases the performance, although no more than 10% in our case. The benefit of chomping also depends on the *Loci* program design, the more computations are chomped, the more benefit we will have. The box in Fig. 6.2(a) and Fig. 6.2(b) shows the speed of dynamic memory management alone when compared to the preallocation scheme. This indicates the amount of run-time overhead incurred by the dynamic memory management. Typically they are negligible. The reason for the somewhat large overhead of Chem-I under "dmm" on Linux machine is unknown at present and it is possible due to random system interactions.

To study the effect of chomping under conditions where the latencies in the memory hierarchy are extreme, we performed another measurement of chomping when virtual memory is involved. To invoke virtual memory, we intentionally execute CHEM on a large problem such that the program had significant access to disk through



(a) Space Profiling on Linux

(b) Space Profiling on SGI

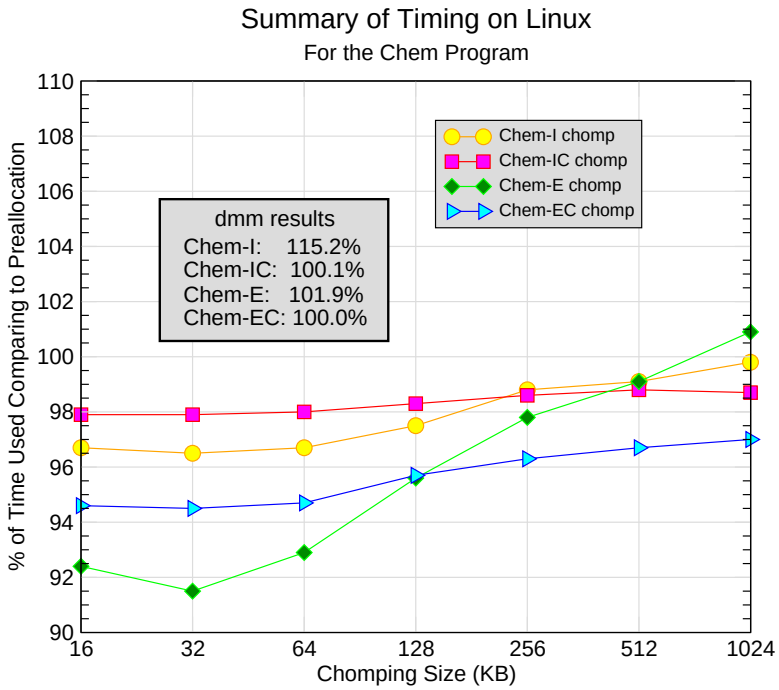
FIG. 6.1. *Space Measurement*

virtual memory. We found in this case, chomping has superior benefit. Schedule with chomping is about 4 times faster than the preallocation schedule or the schedule with memory management alone. However the use of virtual memory tends to destroy the performance predictability and thus it is desirable to avoid virtual memory when possible. For example, a large memory requirement can be satisfied by using more processors. Nevertheless, this experiment showed an interesting feature of chomping. Chomping may be helpful when we are constrained by system resources.

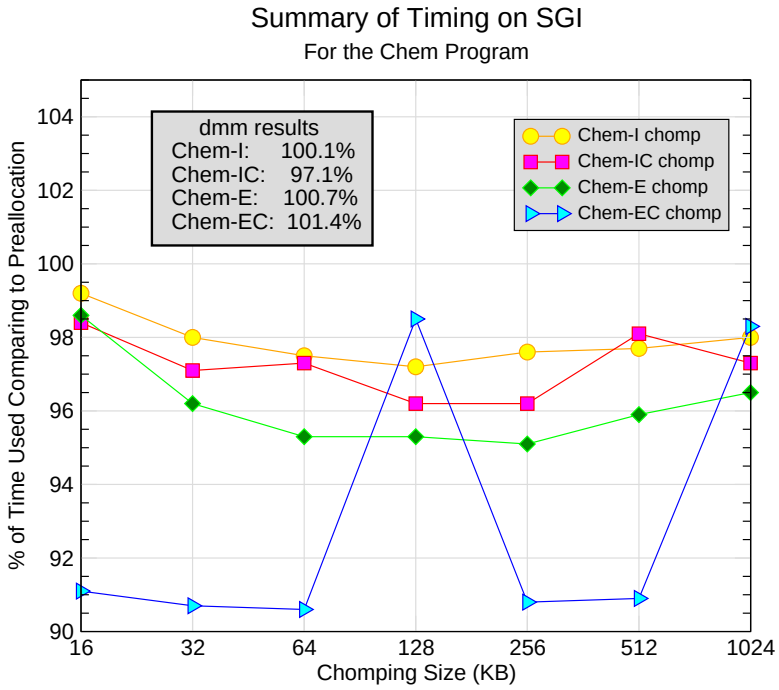
We are also interested to see how well the searching algorithm for chomping chain (as discussed in section 4.1) performs for real applications. We did a measurement for the searching for the CHEM program as shown in table 6.2. The "chomping candidates" in the table refers to relations in the program that do not involve non-affine accesses. As discussed in section 4.1, they represent the upper bound of the relations that we can possibly chomp. But the constraints of relations and rules in the graph may force us to discard some of them. We do not know whether the searching results are optimal or not. But the results shown in the table are close to the upper bound and we consider them to be good enough for practical use. We also took a measurement of the size of the total chomped relations. Interestingly, for this measurement, the percentage of size is larger than the percentage of number in total relations. This further shows the importance of chomping. If a *Loci* program is designed appropriately, from the memory management point of view, doing chomping alone would eliminate a large portion of memory requirement. Typically the peak memory is determined by the number of relations that needs to be in the memory simultaneously. If most of these relations can be chomped, then memory requirement can be potentially reduced greatly in addition to performance benefits.

TABLE 6.2
Statistics of Chomping

	CHEM-I	CHEM-IC	CHEM-E	CHEM-EC
total relations	192	196	162	166
chomping candidates	47	49	49	51
chomped relations	40	42	44	47
% of the size of chomped relations in total relations	32.25	32.39	44.74	51.03



(a) Timing on Linux



(b) Timing on SGI

FIG. 6.2. *Timing Measurement*

Finally we present one result of the comparison of different scheduling policies in table 6.3. The measurement was performed on 32 processors of our parallel cluster. We noticed the difference of peak memory usage between computation greedy and memory greedy schedule is somewhat significant, however the timing results are almost identical albeit the large difference in the number of synchronization points. We attribute this to the fact that CHEM is computationally intensive, the additional communication start-up costs do not contribute significantly

TABLE 6.3
Mem vs. Comm under dmm on Linux Cluster

	memory usage (MB)		sync		time
	real	bean-counting	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

to the total execution time. This suggests for computationally intensive applications, the memory greedy scheduling is a good overall choice, as the additional memory savings do not incur undue performance penalty. For more communication oriented applications, the difference of using the two scheduling policies may be more obvious. In another measurement, we artificially ran a small problem on many processors such that parallel communication is a major overhead. The results are presented in table 6.4. We found the synchronization points in the memory greedy schedule is about 1.6 times more than the one in computation greedy schedule and the execution time of memory greedy schedule increased roughly about 1.5 times. Although this is an exaggerated case, it provided some evidence that such a trade-off does exist. However, for scaling small problems, conserving memory resources should not be a concern and in this case the computation greedy schedule is recommended.

TABLE 6.4
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

7. Conclusions. This study presented a new dynamic memory management technique implemented in a novel declarative parallel programming framework, *Loci*. The approach utilizes techniques to improve both cache utilization and memory bounds. In addition, we studied the impact of memory scheduling on parallel communication overhead. Results show that memory management is effective and is seamlessly integrated into the *Loci* framework. By utilizing the chomping technique, which is similar to strip-mining in traditional loop optimizations, we were able to reduce both memory bounds and run times of *Loci* applications. In addition, we illustrate that the aggregation performed by *Loci* also facilitates memory management and cache optimization. We were able to use *Loci*'s facility of aggregating entities of like type as a form of region inference. The memory management is thus simplified as managing the lifetime of these containers amounted to managing the lifetimes of aggregations of values. In this sense, although *Loci* supports fine-grain specification [6], the memory management does not have to be at the fine-grain level. This has some similarity with the region management concept. The initial graph decoration phase resembles the static program analysis performed by the region inference memory management, although much simpler and is performed at run-time.

We also observed an interesting phenomenon with our chomping technique. While we expected it to increase performance and to a small extent reduce memory requirements, its benefits in memory reduction were greater than expected. Apparently this result comes from two sides. The first one is due to heap fragmentation: the uniformly sized chomps were more efficiently managed by the allocator. Obviously there are interesting and complex interactions between allocation policy and heap management. The SGI platform seems to have significant benefit from the improved memory fragmentation by chomping. The second reason is due to that the memory allocations for chomping is virtually negligible. Therefore the more relations are chomped, the less we pay for their memory space allocations. We found that we were able to chomp many relations in our CHEM program and the aggregate size of all these relations is large.

We also note that the performance gains in real applications achieved by using chomping is significantly less than potential gains. While the CHEM application saw a performance increase on the order of 10 percent in some cases, simple example programs that made significant use of the chomping facility saw performance boosts of as much as a factor of five. We suspect that the cache performance suffers from large non-chain access as well as non-affine accesses to other data in the middle of the chomping chain as discussed in section 4.2. We believe that these issues may be mitigated by identifying common non-affine references and factoring them out

of the computations through program transformations. However, this transformation may require replicating some work to achieve full effect, limiting the potential performance boost.

While we have demonstrated a trade-off between an efficient memory schedule and communication barriers in the parallel program, we have not provided an automated way of selecting the appropriate policy. This is currently under user control. It would be fairly simple to use a model-based approach to decide when a memory efficient schedule would cost more than some small percentage of overall run-time and then select the appropriate policy automatically. However, preliminary investigations into replicating work to eliminate communication barriers appears in many cases to eliminate most of the barriers introduced by the memory efficient schedule. While this replication technique is still under development, we believe, based on our preliminary results, that the combination of work replication and a memory efficient schedule may provide the best of both techniques and eliminate the need for a more sophisticated policy selection mechanism.

Acknowledgment. We thank the financial support from the National Science Foundation (ACS-0085969), NASA GRC (NCC3-994), and NASA MSFC (NAG8-1930). In addition we are grateful to the anonymous reviewers for their insightful suggestions and comments.

REFERENCES

[1] B. Babcock, S. Babu, M. Datar, and R. Motwani, *Chain: Operator scheduling for memory minimization in data stream systems*, in Proceedings of the ACM International Conference on Management of Data (SIGMOD 2003), San Diego, California, June 2003.

[2] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press, 2004.

[3] M. Frigo and S. G. Johnson, *FFTW: An adaptive software architecture for the FFT*, in Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 3, Seattle, WA, May 1998, pp. 1381–1384.

[4] H. Lieberman and C. Hewitt, *A real time garbage collector based on the lifetimes of objects*, Communications of the ACM, 26(6) (1983), pp. 419–429.

[5] E. A. Luke, *Loci: A deductive framework for graph-based algorithms*, in Third International Symposium on Computing in Object-Oriented Parallel Environments, S. Matsuoka, R. Oldehoeft, and M. Tholburn, eds., no. 1732 in Lecture Notes in Computer Science, Springer-Verlag, December 1999, pp. 142–153.

[6] E. A. Luke and T. George, *Loci: A rule-based framework for parallel multi-disciplinary simulation synthesis*, Journal of Functional Programming, Special Issue on Functional Approaches to High-Performance Parallel Programming, 15 (2005), pp. 477–502. Cambridge University Press.

[7] E. A. Luke, X. Tong, J. Wu, and P. Cinnella, *Chem 2: A finite-rate viscous chemistry solver – the user guide*, tech. report, Mississippi State University, 2004.

[8] E. A. Luke, X. Tong, J. Wu, L. Tang, and P. Cinnella, *A step towards "shape-shifting" algorithms: Reacting flow simulations using generalized grids*, in Proceedings of the 39th AIAA Aerospace Sciences Meeting and Exhibit, AIAA, January 2001. AIAA-2001-0897.

[9] S. M. Nettles and J. W. O'Toole, *Real-time replication-based garbage collection*, in Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation, Albuquerque, NM, June 1993, pp. 217–226.

[10] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl, *Questions and answers about BSP*, Scientific Programming, 6 (1997), pp. 249–274.

[11] M. Tofte and L. Birkedal, *A region inference algorithm*, Transactions on Programming Languages and Systems (TOPLAS), 20 (1998), pp. 734–767.

[12] J. Ullman, *Principles of Database and Knowledgebase Systems, Volume I*, Computer Science Press, 1988.

[13] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–111.

[14] P. R. Wilson, *Uniprocessor garbage collection techniques*, in Proceedings of International Workshop on Memory Management, St. Malo, France, 1992, Springer-Verlag.

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

[16] Y. Zhang, *Dynamic memory management for the Loci framework*, master's thesis, Mississippi State University, Mississippi State, Mississippi, May 2004.

Edited by: Frédéric Loulergue
Received: October 3, 2005
Accepted: February 1st, 2006