Dynamic Memory Management in the *Loci* Framework

Yang Zhang and Edward A. Luke

Department of Computer Science and Engineering,
Mississippi State University, Mississippi State, MS 39762, USA
{fz15,luke}@cse.msstate.edu

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.

1 Introduction

In this paper we discuss the design and implementation of a dynamic memory management strategy for the declarative programming framework, *Loci* [1, 2]. The *Loci* framework provides a rule-based programming model for numerical and scientific simulation similar to the *Datalog* [3] 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 [4] 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 [2]. *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 [5].) 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 [2]. 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. 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 [6] is the most prevalent automatic memory management technique. Useless memory blocks are treated as garbage and are recycled periodically by the run-time system. A non-traditional method for managing memory resources in the context of scheduling operators in continuous data streams[8] shows how scheduling order can effect overall memory requirements. They suggest an optimal strategy in the context of stream models. Region inference [7] is a relatively new form of automatic memory management. It 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. 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. 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.

3 Basic Dynamic Memory Management

In *Loci*, relations are stored in value containers. These 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.

Since *Loci* planner generates an execution schedule from the partitioned dependency graph (the multi-level graph), a simple approach to incorporating appropriate memory scheduling would be to incorporate relevant memory management operations into this graph. Then, when the graph is compiled, proper memory management instructions are included into the schedule that 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. The multi-level 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 schedule [9].

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 memory savings by data partitioning. Consider the rule chain in Fig. 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 the memory requirement. However, because of the existence of non-affine memory references, we cannot group arbitrary rules into rule chains that can be chomped. In *Loci*, we use a heuristic search to identify suitable chains in the multi-level graph and apply chomping only to them [9]. 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.

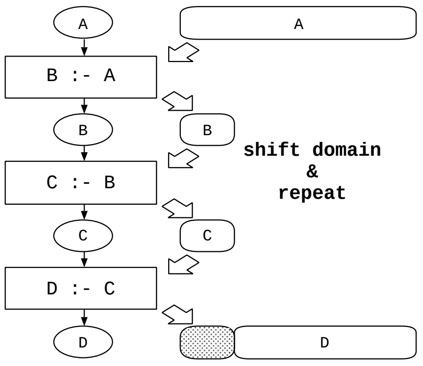


Fig. 1. The Chomping Idea

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, *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 2 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 less 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.

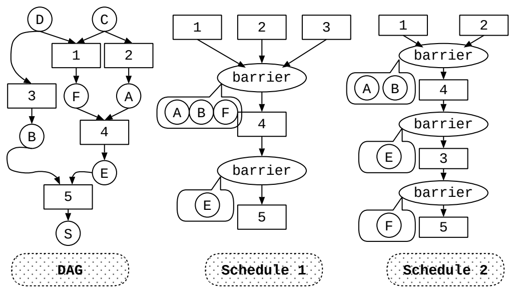


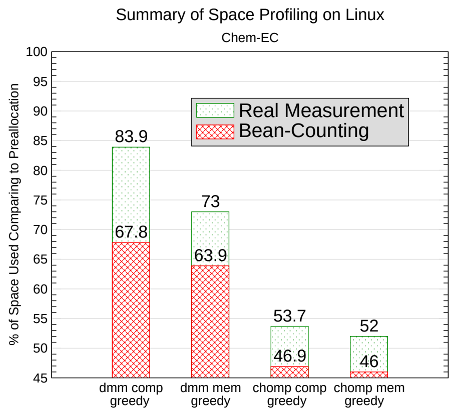
Fig. 2. Different Scheduling for a DAG

A trade-off therefore exists in the *Loci* planner. In order to optimize memory utilization and reduce the 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.

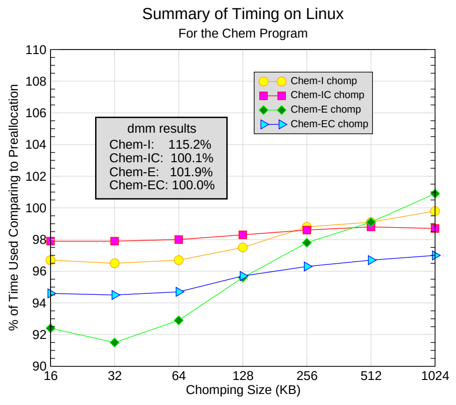
The scheduling infrastructure in the *Loci* planner is priority based. *Loci* planner schedules a DAG according to the weight of each vertex. In this sense, scheduling policies can be implemented by providing different weights to the vertices. We provide a heuristic for assigning vertices weight that attempts to minimize the memory utilization for the schedule. The central idea of the heuristic is to keep a 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 [9] in Loci for computing vertex priority based on the heuristics described above for memory minimization.

6 Experimental Results

In this section, we present some of our measurements for the dynamic memory management in *Loci*. The CHEM program is used as a benchmark. CHEM 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) is used in the measurement. In addition to take 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 the measurement, 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.



(a) Space Profiling on Linux



(b) Timing on Linux

Fig. 3. Space and Timing Measurement

We did extensive profiling of the memory utilization on various architectures. Figure 3(a) shows a measurement of Chem-EC on a single node on the cluster. 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 Fig. 3(a), when combining with memory greedy scheduling and chomping, the peak memory usage is reduced to at most 52% of preallocation peak memory usage. The actual peak memory depends also 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 will help to improve the memory fragmentation problem. Because in these cases, the allocations are possibly much smaller and regular.

Figure 3(b) shows one timing result for chomping on a single node on the cluster. The result shows different chomping size 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. 3(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" 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. We run CHEM on a large problem such that the program had significant access to disk through 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.

Table 1. 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

Finally we present one result of the comparison of different scheduling policies in table 1. 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 to the total execution time. This suggests for computational intensive application, 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. 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 trade-off does exist. However, for scaling small problems, memory resources should not be a concern and in this case the computation greedy schedule is recommended.

7 Conclusions

The study presented in this paper provides a dynamic memory management infrastructure for the *Loci* framework. We transformed memory management to a graph decoration problem. The proposed approach utilized 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 the memory management is effective and is seamlessly integrated into the *Loci* framework. Combining the memory management with chomping, the resulting schedule is typically faster and space efficient. The aggregation performed by *Loci* also facilitates the 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 [2], the memory management does not have to be at the fine-grain level. This has some similarity with the region management concept. The graph decoration resembles the static program analysis performed by the region inference memory management, although much simpler and is performed at run-time. The scheduling policies implemented in *Loci* are currently specified by users. As a future work, it is possible to extend this and make *Loci* aware of the scheduling policies itself. We imagine there are several different ways to achieve this. In *Loci*, we can estimate the overall computation and communication time and the memory consumption before the execution plan is run. Therefore we can infer an appropriate scheduling policy in *Loci* and thus does not require the user being aware of this choice. A more sophisticated way would be to generate two schedules (one for memory minimization and the other for communication overhead minimization) and switch between them at runtime. Since it is possible that some containers would be dynamically resized at runtime, the estimation at the scheduling phase could be imprecise. If we have two schedules, we can dynamically measure the cost at runtime and switch to an appropriate schedule when necessary. This scheme requires some amount of coordinations between different schedules and is much harder than the previous scheme. But as we observed, current *Loci* applications are typically computation bounded and therefore this feature is less critical.

8 Acknowledgment

The financial support of the National Science Foundation (ACS-0085969), NASA GRC (NCC3-994), and NASA MSFC (NAG8-1930) is gratefully acknowledged. In addition we would like to thank the anonymous reviewers for their excellent suggestions.

References

1. Luke, E.A.: Loci: A deductive framework for graph-based algorithms. In Matsuoka, S., Oldehoeft, R., Tholburn, M., eds.: Third International Symposium on Computing in Object-Oriented Parallel Environments. Number 1732 in Lecture Notes in Computer Science, Springer-Verlag (1999) 142–153
2. Luke, E.A., George, T.: 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 (to appear) available at: `http://www.erc.msstate.edu/~lush/publications/LociJFP2005.pdf`.
3. Ullman, J.: Principles of Database and Knowledgebase Systems. Computer Science Press (1988)
4. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing. Volume 3., Seattle, WA (1998) 1381–1384
5. Valiant, L.G.: A bridging model for parallel computation. Communications of the Association for Computing Machinery **33** (1990) 103–111
6. Wilson, P.R.: Uniprocessor garbage collection techniques. In: Proceedings of International Workshop on Memory Management, St. Malo, France, Springer-Verlag (1992)
7. Tofte, M., Birkedal, L.: A region inference algorithm. Transactions on Programming Languages and Systems (TOPLAS) **20** (1998) 734–767
8. Babcock, B., Babu, S., Datar, M., Motwani, R.: 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 (2003)
9. Zhang, Y.: Dynamic memory management for the Loci framework. Master's thesis, Mississippi State University, Mississippi State, Mississippi (2004)