Parallel Management of Large Dynamic Shared Memory Space: A Hierarchical FEM Application

Xavier Cavin* and Laurent Alonso**

ISA research team, LORIA***
Campus Scientifique, BP 239, F–54506 Vandœuvre–lès–Nancy CEDEX, France
Xavier.Cavin@loria.fr

Abstract. We show in this paper the memory management issues raised by a parallel irregular and dynamic hierarchical application, which constantly allocates and deallocates data over an extremely large virtual address space. First, we show that if memory caches data locality is necessary, a lack of virtual pages locality may greatly affect the obtained performance. Second, fragmentation and contention problems associated with the required parallel dynamic memory allocation are presented. We propose practical solutions and discuss experimentation results obtained on a cache–coherent non uniform memory access (ccNUMA) distributed shared memory SGI Origin2000 machine.

1 Introduction

The radiosity equation [Kaj86] is widely used in many physical domains and in computer graphics, for its ability to model the global illumination in a given scene. It looks like a Fredholm integral equation of the second kind, which can be expressed as:

$$f(y) = f_{\mathrm{e}}(y) + \int_{\Omega} k(x,y) f(x)\, dx \quad , \tag{1}$$

where f is the radiosity equation to determine. Generally, this unknown function is defined over an *irregular*, *non–uniform* physical domain Ω, mainly described in terms of polygonal surfaces (some of them having non–zero initial radiosity f_{e}).

Finding an analytical solution to (1) is not possible in general. Numerical approximations must be employed, generally leading to very expensive algorithms. The fundamental reason for their high cost is that each surface of the input scene may potentially influence all other surfaces via reflection.

A common resolution technique is the weighted residual method, often referred as "finite element method" (FEM). Early radiosity algorithms can be analyzed as FEM using piecewise constant basis functions. Later, hierarchical algorithms, inspired by adaptive N–body methods, have been introduced

* Institut National Polytechnique de Lorraine.
** INRIA Lorraine.
*** UMR 7503, a joint research laboratory between CNRS, Institut National Polytechnique de Lorraine, INRIA, Université Henri Poincaré and Université Nancy 2.

by [HSA91] to increase the efficiency of the computations. They are based on a multi–level representation of the radiosity function, which is *dynamically* created as the computation proceeds, subdividing surfaces where necessary to increase the accuracy of the solution. Since energy exchanges can occur between any levels of the hierarchies, *sub–computation times are highly variable and change at every step of the resolution.*

This dynamic nature, both in memory and computing resources, combined to the non–uniformity of the physical domain being simulated makes the parallelization of hierarchical radiosity algorithms a challenge, since straightforward parallelizations generally fail to simultaneously provide the load balancing and data locality necessary to efficient parallel execution, even on modern distributed shared memory multiprocessor machines [SHG95].

In a recent paper [Cav99], we have proposed appropriate partitioning and scheduling techniques for a parallel hierarchical wavelet radiosity algorithm, that deliver an optimal load balancing, by minimizing idle time wasted on locks and synchronization barriers, while still exhibiting an excellent data locality. However, our experiments seemed to show that this was still not sufficient to perform extremely large computations with optimal parallel performance. Indeed, dealing in parallel with a dynamically growing huge amount of memory (for a whole building simulation, it is not rare that more than 20 Gbytes may be required) is not free of problems to have it done in an efficient way. This is even more complicated since most of this memory management is generally hidden to the programmer. If this can be a great facility when all works well, it quickly becomes damageable when problems start to occur.

We show in this paper the two main causes of the performance degradation, and experiment practical solutions to overcome them on a 64–processor SGI Origin2000 ccNUMA machine. The first problem concerns the irregular memory access patterns of our application, which have to be handled within an extremely large virtual address space. The issues are discussed in Sect. 2, and we show how some SGI IRIX operating system facilities can help enhancing virtual pages locality and consequently reduce computation times. Parallel dynamic memory allocation is the second problem and comes in two different flavors: *fragmentation* and *contention*. We experiment in Sect. 3 available IRIX and public domain solutions, and propose enhancements leading to an efficient parallel memory allocator. Finally, Sect. 4 concludes and presents future work.

2 Efficient Irregular Memory Accesses within a Large Virtual Address Space

2.1 Understanding ccNUMA Architecture

In order to fully benefit from a computer system performance, it is really important to understand the underlying architecture. The SGI Origin2000 is a scalable multiprocessor with distributed shared memory, based on the Scalable Node 0 (SN0) architecture [Cor98]. The basic building block is the node board, composed of two MIPS R10000 processors, each with separate 32 Kbytes first level

(L1) instruction and data caches on the chip, with 32–byte cache line, and a unified (instruction and data), commonly 4 Mbytes, two–way set associative second level (L2) off–chip cache, with 128–byte cache line. Large SN0 systems are built by connecting the nodes together via a scalable interconnection network.

The SN0 architecture allows the memory to be physically distributed (from 64 Mbytes to 4 Gbytes per node), while making all memory equally accessible from a software point of view, in a ccNUMA approach [LL97]. A given processor only operates on data that are resident in its cache: as long as the requested piece of memory is present in the cache, access times are very short; on the contrary, a delay occurs while a copy of the data is fetched from memory (local or remote) into the cache. The trivial conclusion is that a program shall use these caches effectively in order to get optimal performance.

Obviously, if the shared memory is seen as a contiguous range of virtual memory addresses, the physical memory is actually divided into pages, which are distributed all over the system. For *every* memory access, the given virtual address must be translated into the physical address required by the hardware. A hardware cache mechanism, the translation lookaside buffer (TLB), keeps the 64×2 most recently used page addresses, allowing an instant virtual–to–physical translation for these pages. This allows a 2 Mbytes memory space (for the default 16 Kbytes page size) to be addressed without translation penalty. Programs using larger virtual memory (the common case) may refer to a virtual address that is not cached in the TLB. In this case (TLB miss), the translation is done by the operating system, in the kernel mode, thus adding a non–negligible overhead to the memory access, *even if it is satisfied in a memory cache.*[1]

2.2 Enhancing Virtual Pages Locality

Having an optimal data caches locality appears to be a necessary, but no sufficient, condition to get optimal performance, since a L1 or L2 cache hit may be greatly delayed by a TLB miss. It is really important, however, to understand that data caches locality does not necessarily implies virtual pages (i.e. TLB) locality. Indeed, the data are stored in memory caches with a small size granularity (128 bytes for the 4 Mbytes of L2 cache), and may thus come from a large number (much greater that 64 TLB entries) of different pages. Then, the application may exhibit an optimal data locality through these data and a poor TLB locality at the same time.

Unfortunately, this is the case for our hierarchical application, which by nature exhibits high data locality, but suffers from highly irregular data access patterns. Whatever the input data, our application affords very high cache hits rates of more than 95 % for both L1 and L2 caches, for *any number* of processors used [CAP98]. At the same time, it is clear that the irregular memory accesses towards a large number of virtual pages are responsible for many TLB misses, especially as the size of the input data increases.[2] This is confirmed by the analysis of the sequential run of the application with the default 16 Kbytes page

[1] See [Cor98] for the precise read latency times.

[2] Monitoring of L1, L2 and TLB misses is done with the IRIX `perfex` command.

size, where about 44 % of the execution time is lost due to TLB misses. Table 1 shows that when the number of processors used increases, the total number of TLB misses quickly falls down by 33 % with 16 processors, and then more slowly decreases. This seems to be due to the fact that using N processors allows to use $64 * N$ TLB entries at the same time. This suggests that next generations of MIPS processors should contain more TLB entries.

Increasing the number of TLB entries appears to enhance TLB locality. Nevertheless, for a fixed number of available TLB entries, an alternate solution is to increase the total memory space they can address, by simply telling the operating system to increase the size of a memory page, thanks to the IRIX `dplace` command. As shown by Table 1, doing this reduces the execution times, at least with at the small 16–processor scale. Indeed, using a larger number of processors multiplies the number of available TLB entries, thus reducing the TLB misses problem and the benefits of this solution.

Table 1. Impact of page sizes on TLB misses and execution times

		Processors						
	Page size	1	4	8	16	24	32	40
Execution time	16k	13 835	3 741	1 866	1 054	761	653	610
(s)	1m	10 813	–	1 538	891	701	612	590
TLB misses	16k	17 675	14 019	12 487	11 676	10 176	9 888	9 497
($\times 10^6$)	1m	5 479	–	3 939	3 769	3 813	3 842	3 688

3 Efficient Parallel Dynamic Memory Allocation

Since the memory required by our application has to be dynamically allocated all along the execution, an efficient dynamic memory allocator has to be employed, both to reduce fragmentation problems, which may cause severe memory waste, and to allow contention free parallel manipulations. As quoted in [WJNB95], "memory allocation is widely considered to be either a solved problem, or an insoluble one". This appears to be true for the common programmer using the default memory allocation package available on his machine. Once again, most of the time, this solution is not a bad one, but when the memory allocator runs badly, one discovers to be in front of a mysterious "black box".

The role of a memory allocator is to keep track of which parts of the memory are in use, and which parts are free, and to provide the processes an efficient service, minimizing wasted space without undue time cost, or *vice et versa*. Space consideration appears to be of primary interest. Indeed, worst case space behavior may lead to complete failure due to memory exhaustion or virtual memory trashing. Obviously, time performance is also important, especially in parallel, but this is a question rather of implementation than of algorithm, even if considerations can be more complicated.

We believe that it is primordial to rely on existing memory allocators, rather than to develop *ad hoc* storage management techniques, for obvious reasons of software clarity, flexibility, maintainability, portability and reliability. We focus here on the three following available ones:

1. the IRIX C standard memory allocator, which is a complete black box;
2. the IRIX alternative, tunable, "fast main memory allocator", available when linking with the `-lmalloc` library;
3. the LINUX/GNU libc parallel memory allocator[3], which is an extension of famous Doug Lea's Malloc[4], and is also parameterizable.

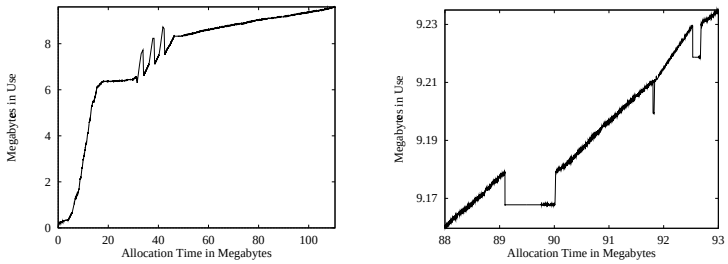
3.1 The Fragmentation Problem

Fragmentation is the inability to reuse memory that is free. An application may free blocks in a particular order that creates holes between "live" objects. If these holes are too numerous and small, they cannot be used to satisfy further requests for larger blocks. Note here that the notion of fragmented memory at a given moment is completely relative to further requests. Fragmentation is the central problem of memory allocation and has been widely studied in the field, since the early days of computer science. Wilson et al. present in [WJNB95] a wide coverage of literature (over 150 references) and available strategies and policies. Unfortunately, a unique general optimal solution has not emerged, because it simply does not exist. Our goal here is not to propose a new memory allocation scheme, but rather to report the behavior of the chosen allocators in terms of fragmentation. Algorithmic considerations have been put aside, since it is difficult to know the principles a given allocator is implemented on.

We have chosen to use the experimental methodology proposed in [WJNB95] to illustrate the complex memory usage patterns of our radiosity application: all allocation and deallocation requests done by the program are written to a file during its execution. The obtained trace only reflects the program behavior, and is *independent* of the allocator. Figure 1 shows the profile of memory use for a complete run of our radiosity application. Although it has been done on a small input test scene, it is representative of what happens during the computations: many temporary, short–live objects are continuously allocated and deallocated to progressively build the long–live objects of the solution (here for instance, 120 Mbytes of data are allocated for only 10 Mbytes of "useful" data).

Then, the trace is read by a simulator, which has first been linked with the allocator to be evaluated: this allows to precisely monitor the way it behaves, including the potential fragmentation of the memory. Unfortunately, on such small input test scenes, none of the three allocators suffers from fragmentation (86 % of the allocated memory is actually used). The fragmentation problem only occurs with large input test scenes, the computation of which can not (yet) be traced with the tools we use[5]. We can just report what we have observed

[3] See http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html
[4] See http://gee.cs.oswego.edu/dl/html/malloc.html
[5] At ftp://ftp.dent.med.uni-muenchen.de/pub/wmglo/mtrace.tar.gz



(a) Complete execution. (b) Zoom in.

Fig. 1. Profile of memory use in a short radiosity computation (534 initial surfaces, 20 iterations, 5 478 final meshes): these curves plot the overall amount of live data for a run of the program, the time scale being the allocation time expressed in bytes

during our numerous experiments: the standard IRIX C allocator appears to be the better one, while the alternative IRIX allocator leads to catastrophic failure with large input data on its default behavior (our experiments with the parameters have not been successful either); LINUX/GNU libc allocator is closer to the IRIX C allocator, although a little bit more space consuming.

3.2 Allocating Memory in Parallel

Few solutions have been proposed for parallel allocators (some are cited at the beginning of Sect. 4 in [WJNB95]). However, none of them appears to be implemented inside the two IRIX allocators, since the only available solution is to serialize memory requests with a single lock mechanism. This is obviously not the right way for our kind of application, which constantly allocates and deallocates memory in parallel. Generally, with this strategy, only one or two processes are running at a given moment, while all remaining ones are idle.

We first considered implementing our own parallel allocator, based on the *arena* facilities provided by IRIX. Basically, each process is assigned to its own arena of memory, and is the only one responsible for memory operations in it: we observed we could achieve optimal time performance, without any contention. Unfortunately, the allocation inside a given arena is based on the alternative IRIX allocator, which gives, as previously said, very poor results in terms of fragmentation. We then considered the LINUX/GNU libc allocator, which exhibits better space performance and provides parallelism facilities, based on similar ideas as ours. Unfortunately, a few implementation details greatly limit the scalability to four or eight threads, which is obviously insufficient for us. We thus fixed these minor problems to have the LINUX/GNU libc allocator more looks like our parallel version, and finally get a parallel allocator which proves to be (for the moment) rather efficient both in space and time performance.

4 Conclusion

We show in this paper the problems raised by the memory management of a large, dynamically evolving, shared memory space within an irregular and non–uniform parallel hierarchical FEM algorithm. These problems are not widely covered in the literature, and there are few available solutions. We first propose practical techniques to enhance memory accesses performance. We then study the characteristics of our application in terms of memory usage, and show that it is greatly suitable to fragmentation problems. Available allocators are considered and experimented to find which one gives the best answer to the request patterns. Finally, we design a parallel allocator, based on the LINUX/GNU libc one, which appears to give good performance in terms of time and space.[6] However, deeper insights, inside both our application and available memory allocators, will be needed to better understand the way they interact, and we believe this is still an open and beautiful problem.

Acknowledgments. We would like to thank the Centre Charles Hermite for providing access to its computing resources. Special thanks to Alain Filbois for the hours spent on the Origin.

References

[CAP98] Xavier Cavin, Laurent Alonso, and Jean-Claude Paul. Experimentation of Data Locality Performance for a Parallel Hierarchical Algorithm on the Origin2000. In *Fourth European CRAY-SGI MPP Workshop*, Garching/Munich, Germany, September 1998.

[Cav99] Xavier Cavin. Load Balancing Analysis of a Parallel Hierarchical Algorithm on the Origin2000. In *Fifth European SGI/Cray MPP Workshop*, Bologna, Italy, September 1999.

[Cor98] David Cortesi. Origin2000 (TM) and Onyx2 (TM) Performance Tuning and Optimization Guide. Tech Pubs Library guide Number 007-3430-002, Silicon Graphics, Inc., 1998.

[HSA91] Pat Hanrahan, David Salzman, and Larry Aupperle. A Rapid Hierarchical Radiosity Algorithm. In *Computer Graphics (ACM SIGGRAPH '91 Proceedings)*, volume 25, pages 197–206, July 1991.

[Kaj86] James T. Kajiya. The Rendering Equation. In *Computer Graphics (ACM SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, August 1986.

[LL97] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, Denver, June 1997. ACM Press.

[SHG95] Jaswinder Pal Singh, John L. Hennessy, and Annoop Gupta. Implications of Hierarchical N-body Methods for Multiprocessor Architectures. *ACM Transactions on Computer Systems*, 13(2):141–202, May 1995.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1995.

[6] Feel free to contact us if you want to evaluate this clone.