Experience with Memory Allocators for Parallel Mesh Generation on Multicore Architectures

Andrey N. Chernikov[1]
Christos D. Antonopoulos[2]
Nikos P. Chrisochoides[1]
Scott Schneider[3]
Dimitrios S. Nikolopoulos[3]

[1] Center for Real-Time Computing
Department of Computer Science
College of William and Mary
P.O. Box 8795, Williamsburg, VA 23187, USA
e-mail: {ancher,nikos}@cs.wm.edu

[2]Department of Computer and Communications Engineering
University of Thessaly
Glavani 37 & 28 Oktovriou, Volos, 38221, Greece
e-mail: cda@inf.uth.gr

[3]Center for High-End Computing Systems
Department of Computer Science
Virginia Tech
660 McBryde Hall, Blacksburg, VA 24061, USA
e-mail: {scschnei,dsn}@cs.vt.edu

Abstract

Scalable and locality-aware multiprocessor memory allocators are critical for harnessing the potential of emerging multithreaded and multicore architectures. This paper evaluates two state-of-the-art generic multithreaded allocators designed for both scalability and locality, against custom allocators, written to optimize the multithreaded implementation of parallel mesh generation algorithms. We use three different algorithms in terms of communication/synchronization requirements. The implementations of all three algorithms are heavily dependent on dynamically allocated pointer-based data structures and all three use optimized internal memory allocators based on application-specific knowledge. For our study we used memory allocators which are implemented and evaluated on two real multiprocessors with a multi-SMT (quad Hyperthreaded Intel) and a multi-CMP/SMT (dual IBM Power5) organization. Our results indicate that properly engineered generic memory allocators can come close or sometimes exceed (in sequential allocation) the performance of custom multi-threaded allocators. These results suggest that in the near future we should be able to develop generic multi-threaded allocators that can adapt to application charac-

teristics and increase productivity without compromising performance.

Introduction

The critical nature of memory allocation, especially in highly irregular applications, like parallel mesh generation, that make heavy use of dynamic data structures, forces application developers to build customized concurrent memory allocators. Clearly, the main argument for building a custom memory allocator is performance. Standard system allocators often exhibit high latency and poor scalability, for reasons which are hard to diagnose since they are almost always used as black boxes. The black box approach does not let application developers utilize their understanding of code behavior. Custom multithreaded memory allocators can be designed for a specific application's properties. For example, a custom allocator may: i) eliminate synchronization and shared memory pools altogether, if objects are allocated and deallocated uniformly across threads; ii) customize the object segregation strategy to the application's actual object allocation and access patterns; iii) change the semantics of deallocation, for example, by deallocating objects in batches, thus aggregating many costly memory allocation operations into one.

The major disadvantage of custom memory allocators is the substantial effort needed to design, implement and test them, especially in a multithreaded execution environment. Effective custom allocators require a deep understanding of applications and their memory allocation patterns—a non-trivial exercise in the case of irregular and adaptive applications. Custom allocators also tend to suffer from lack of portability both between applications and between platforms. Generic multithreaded memory allocators, on the other hand, are advantageous from a productivity perspective. Given the severe productivity obstacles in parallel computing, off-loading the complexity of memory allocation and hiding hard to gauge architectural details from the programmer are major steps towards improved productivity. This effort is relevant and timely, due to an acute need for rapid porting and prototyping of applications on multithreaded and multicore processors. These processors are becoming a commodity at a fast pace.

We evaluate custom and generic multithreaded allocators with three parallel guaranteed quality mesh generation codes. These codes (briefly referred to as PCDM, MPCDM and PDR) have been developed and extensively optimized over the last ten years. The codes stress different aspects of allocation, including both sequential and parallel allocation latency, locality and scalability in multithreaded environments. Their memory allocation, deallocation and usage patterns are highly irregular and unstructured. For realistic problem sizes, the codes generate tens of millions of objects of multiple sizes. All three codes rely heavily on custom memory allocators to mitigate the problems of standard system allocators. Alongside the evaluation of different allocators, we revisit the question of viability of generic multithreaded memory allocators for demanding applications, by examining the design aspects of multithreaded memory allocators that improve scalability and locality, while preserving good sequential performance.

Parallel Mesh Generation

Parallel mesh generation procedures in general decompose the original mesh generation problem into N_s smaller subproblems which are solved (i.e., meshed) concurrently using $P \ll N_s$ processors. The subproblems can be formulated to be tightly coupled, partially coupled, weakly coupled, or decoupled [7]. The coupling of the subproblems determines the intensity of the communication and the amount and type of synchronization required between the subproblems. Existing parallel mesh generation methods are based on either Delaunay triangulation, the advancing front technique or edge-subdivision methods. A complete review of all these methods appears in [7]. In this paper we use parallel guaranteed quality Delaunay mesh generation methods as our application testbed. The custom memory manager we build is specific for unstructured mesh generation but does not rely on a highly efficient compact representation of simplicial meshes as is the one presented in [3]; it is representative of any typical adaptive and irregular multithreaded application. For instance, in [12] the authors evaluate the parallelization of a dynamically adapting, unstructured mesh application. They mention the memory allocation-related problems that they observed, such as poor cache reuse due to pointer chasing, false sharing, and memory hot-spots. These problems are similar to those we encountered with the applications described in this paper, and are relevant to a wide class of unstructured codes. In this paper we take a closer look on the issue of multithreaded memory allocation [13], for dynamic and irregular codes like parallel mesh generation, and report our experience on trade-offs between custom and generic allocators.

Parallel Delaunay Refinement The Parallel Delaunay Refinement (PDR) code is a two-dimensional, partially coupled multithreaded mesh generator [4]. It is written in C++ using POSIX threads for parallelism. The algorithm implemented in the PDR code has the following characteristics: i) allows the construction of well-shaped elements with bounded minimal angle; ii) produces graded meshes, i.e., meshes with element size specified by a user-defined function; iii) termination and size optimality can be proved; iv) allows the use of custom point placement strategies (e.g., circumcenter, off-center, etc.); v) replaces the solution of a difficult domain decomposition problem with an easier data distribution approach without relying on the speculative execution model [6]; vi) offers performance improvement over the best available sequential software, on workstations with just a few hardware cores.

Parallel Constrained Delaunay Meshing The Parallel Constrained Delaunay Meshing (PCDM) software [5] is a weakly-coupled distributed memory two-dimensional mesh generator which uses domain decomposition [10] to divide the problem among the processes. It is written in C++ and uses MPI for communication among processes. The units of work are represented by subdomain objects.

Each subdomain contains the collections of the constrained edges, the triangles, and the points. Each triangle contains three pointers to its vertices and three pointers to the neighboring triangles. For the point insertion, we use the B-W algorithm. The constrained (boundary) segments are protected by diametral lenses [14], and each time a segment is encroached, it is split in the middle; as a result, a *split* message is sent to the neighboring subdomain.

Since PCDM follows the MPI process model, each process lies in its own address space and uses its own copy of a custom memory allocator. Therefore, PCDM is not expected to benefit from scalability or other optimizations of multithreaded allocators. However, its allocation intensity stresses the speed of sequential memory allocation and tests the adaptivity of multithreaded allocators to sequential allocation patterns. Furthermore, the small object sizes stress the capability of the allocator to exploit spatial locality. Both multiple triangles and multiple vertices can fit in a single L2/L3 cache line on our experimental platforms.

In addition to the distributed memory version of PCDM, we evaluate a medium-grain multithreaded implementation of PCDM (MPCDM) [1]. MPCDM uses multithreading within each domain. Threads create and refine individual cavities concurrently, using the B-W algorithm. MPCDM is synchronization-intensive mainly because threads need to tag each triangle while working on a cavity, to detect conflicts during concurrent cavity triangulation. The data for MPCDM are from an execution which uses a pipe cross-section model.

MPCDM adopts the memory pools for triangles and vertices used in PCDM, and uses one memory pool per thread instead of global memory pools. Each thread allocates only from its local pool, therefore no synchronization instructions are needed for accessing memory pools.

The Custom Memory Pool Data Structure

Meshing is an extremely memory-intensive application area. Hence, the performance of a particular meshing application is dependent on the performance of its underlying memory allocator. If the standard memory allocation facilities are insufficient to provide the necessary performance and scalability, application developers are forced to write their own customized allocators. The standard memory allocator backing `new` and `delete` on our experimental platforms imposed unacceptable overhead, which motivated developing a custom allocation scheme.

The three codes use the same memory pool class, which can be customized to allocate objects of different sizes. PCDM uses a custom allocator which was originally tuned for UltraSparc II-based uniprocessors. The original code makes extensive use of the C++ STL, which introduces its own layer of memory allocation for internal data structures.

At initialization, the memory pool class takes the size of the underlying object as parameter and at runtime it allocates blocks of memory which can accommodate a certain number of objects. Since triangles are the only type of objects that can be deleted in all three applications, the memory pool class does not provide a general-purpose facility to reuse deleted objects. Instead, the applications manage an additional vector to store the pointers to the memory addresses of the triangles which can be recycled. All methods of the memory pool class are implemented in a header file, so that the calls to these methods can be inlined by the compiler.

The memory pool class does not address such issues as aligning the objects to memory addresses and choosing appropriate block sizes. Furthermore, all memory pools of PDR and MPCDM are strictly thread-local; they do not require locks or other forms of synchronization since they do not serve concurrent accesses from multiple threads.

Performance Evaluation

Our experimental setting consists of two hardware platforms with layered shared-memory parallelism. The first is a Dell PowerEdge 6650 server, with 4 Hyperthreaded Intel processors. Each Hyperthreaded processor has two execution contexts. The second platform is an IBM OpenPower 720 node. The node has 2 IBM Power5 processors, each of which is a dual-core processor with 2-way SMT cores. Although our experimental platforms are small-scale shared-memory machines, they still provide valuable insight on scalability trends on systems with SMT and multicore components, as well as insights on the implications of multiprocessor memory allocators on both multithreaded codes and multi-process MPI codes. Our results are also relevant for clusters of small shared-memory systems, a still quite popular hardware platform for high-end computing.

As a representative example of the state-of-the-art in multithreaded memory allocation, and as a competitive design to compare against custom allocators, we select two recently developed allocators, Streamflow [13] and Google's Tcmalloc [8].

The allocators and their variants that we evaluate are shown in Table 1. In addition to the custom memory allocators hard-coded in each application, we evaluate the standard system allocator (labeled standard in the results), the multithreaded allocator included in Google's performance tools (labeled tcmalloc), Streamflow, and a hybrid scheme which uses the custom allocator for all objects and Streamflow's page manager for managing memory pools from which objects are allocated and recycled (labeled custom+pageman). This configura-

Allocator	**Characteristics**
standard	glibc, thread-safe for MPCDM and PDR, or optimized sequential for PCDM, 2^n object sizes
tcmalloc [8]	no headers, locks, 2^n object size classes
Streamflow	no headers, no locks, lock-free page block recycling, $4\|8\times$ object size classes
custom	application-specific
custom + page manager	custom allocator uses Streamflow's page manager for block allocations

Table 1: Memory allocators evaluated in the experiments.

tion tests whether a generic page manager can be beneficial even for custom allocators. The standard glibc allocator is configured differently for sequential and multithreaded execution in Linux, a property we discovered via experimentation. The default sequential allocator is based on Doug Lea's segregated object allocator, which segregates objects according to size, and uses object sizes which are powers of two [9]. However, when the C compiler detects that the POSIX threads library is used, it switches to a different thread-safe allocation scheme, which is slower—according to our experiments—than the default sequential allocator. Glibc adds metadata to each object, for linking objects to free lists and facilitating object recycling.

There are other well-known general purpose multithreaded memory allocators, such as Hoard [2] and Maged Michael's lock-free allocator [11], that we have not included in our study. We have focused on Streamflow and Tcmalloc because they have exhibited the best performance with these applications on our experimental systems. For a more comprehensive comparison of all four memory allocators (Hoard, Michael's, Streamflow and Tcmalloc), see [13].

We examine the performance of multithreaded allocators using MPCDM and PDR, with executions up to 8 threads on our two experimental platforms.

MPCDM requires the available processors to always be a power of two. Its input is the same pipe cross-section model as used with PCDM. The problem sizes were scaled to fit in available physical memory on each system, generating 8 million triangles on the Intel system, and 30 million triangles on the IBM system. For PDR, we use a model of the Chesapeake Bay as input, and generate 3 million triangles on the Intel system, and 11 million triangles on the IBM system.

Single-threaded performance of MPCDM stresses a memory allocator's latency, and for memory allocators designed mainly for scalability, their ability to adapt to an allocation pattern that does not require synchronization. The generic allocators perform reasonably close (within 7%) to the custom allocator in MPCDM. Streamflow is only 3% slower than the custom allocator in single-threaded executions. The results indicate that the generic allocators we evaluate, with the exception of standard, exhibit good sequential performance. Standard memory allocation shows the highest latency, primarily because the standard allocator uses a slower sequential allocation path when the code spawns threads via the POSIX library. The custom allocator is 13% faster than standard on the Intel system, and 15% faster on the IBM system. Streamflow adapts consistently well to single-threaded and multi-threaded execution with MPCDM on both platforms.

PDR's custom allocator outperforms the general purpose allocators on both platforms by a wide margin. On the IBM platform, PDR's custom allocator outperforms standard by 69%, streamflow by 58%, and tcmalloc by 37%. On the Intel platform, PDR's custom outperforms standard by 71%, streamflow by 30%, and tcmalloc by 34%. The application-specific allocator in PDR takes advantage of application knowledge in two ways. First, due to the workings of the algorithm, no synchronization is needed for memory allocation, therefore no expensive atomic instructions are imposed on the critical path of sequential allocation. Second, the custom allocator exploits the fact that memory usage constantly increases over the life of the application. When member objects of the critical data structures are no longer needed, their memory is not returned to memory pools. Instead, that memory is recycled in thread-local vectors, as new instances of those data structures are guaranteed to be needed very soon.

Recycling through local vectors involves only the manipulation of one pointer. Recycling through any generic memory allocator involves at least a function call and several pointer manipulations, since the allocator needs to look up page block metadata in order to locate the memory pool which should host the recycled object.

On the Intel platform, MPCDM with Streamflow performs on par with, or outperforms, the custom allocator by up to 2% in MPCDM. On the IBM platform, Streamflow is within 3% of the performance of the custom allocator. Tcmalloc achieves noticeably lower performance than the custom allocator, by 7–16% on the Intel platform and 4–7% on the IBM platform.

The reason for the difference between Streamflow and Tcmalloc is their synchronization mechanisms. MPCDM is an extremely synchronization intensive application; point-to-point synchronization is performed between threads approximately every 1μsec. While both Streamflow and Tcmalloc eliminate additional synchronization overhead on thread-local operations, when synchronization is needed, Streamflow uses non-blocking, lock-free algorithms while Tcmalloc uses locks. Non-blocking, lock-free algorithms allow finer-grain allocations because threads never have to busy-wait on a critical resource within the allocator.

On the IBM system, a 64-bit `compare&swap` operation takes 83μs, which is 157 cycles on the 1.65GHz processor. On Intel, a 64-bit `compare&swap` takes 137μs, or 274 cycles on a 2.0GHz processor. Since tcmalloc relies more on synchronization, it is more susceptible to the increased cost of atomic operations. Streamflow overcomes this problem due to its decoupled design of allocation operations.

We also observe that the page manager of Streamflow improves the performance of the custom allocator even further, by 3–5% on the Intel platform and by 2–7% on the IBM platform. This result indicates that customized and generic allocators can also work in synergy, with generic allocators responsible for managing large memory blocks and customized allocators responsible for managing small objects within memory blocks.

In MPCDM, the custom allocator achieves a speedup of up to 1.69 on the Intel system and up to 2.6 on the IBM system. The latter architecture is generally more scalable due to the use of dual- core processors, instead of SMT processors and a more scalable implementation of the SMT microarchitecture within the cores. On the other hand, we observe that absolute performance is noticeably higher on the Intel platform. Part of this difference is attributed to the quality of the compilers used. We used version 9 of the Intel compiler on the Intel platform and g++ on the IBM platform.

PDR is characterized by a wider disparity between allocators than MPCDM. On the Intel system, the standard allocator exhibits poor sequential performance (more than 70% slower than custom) and the worst scalability, achieving a maximum speedup of 1.7 versus a maximum speedup of 2.5 achieved with custom. Streamflow achieves a maximum speedup of 2.0, while its sequential performance is 30% lower than that of custom. Tcmalloc performs within 34% of custom with a single thread, and achieves a 2.0 speedup with up to 8 threads. The custom allocator scales well because it avoids synchronization altogether. This is possible because custom is implemented with knowledge of the algorithm's behavior. Further, the code of custom can be inlined and optimized with the rest of the application, while calling the external generic allocators inevitably includes function call overheads and makes inlining harder, even if the generic allocators are compiled together with the applications. Finally, as explained earlier, custom uses an application-specific zero-cost recycling policy which avoids entirely the overhead of returning and retrieving memory back to and from the allocator via sequences of `new` and `delete` calls. Objects are recycled with a single pointer bump. The overhead of function calls and pointer dereferencing negatively affects the performance of generic allocators, which have to recycle every object through a call to `delete` and a subsequent call to `new`.

Once again, we observe that merging the custom allocator with the page block and large object manager in streamflow, yields performance improvements, which reach up to 17% on the Intel platform. The custom allocator benefits from the page block caching capabilities of streamflow. We have verified with experiments that the page block manager in streamflow reduces the latency of both the custom allocator and the generic streamflow small object allocator via a drastic reduction of minor page faults and TLB misses.

The trends observed with PDR on the IBM system are mostly similar to the trends observed on the Intel system. We only outline the most important differences. The IBM system again exhibits better scalability than the Intel system, achieving speedups of 3.0–4.0 (versus

1.7–2.5 on the Intel system), due to the inherently better architectural scalability of the multicore and SMT-core design of the IBM Power5. The excessive cost of unnecessary synchronization hurts the performance of tcmalloc beyond 4 threads, rendering it inferior to streamflow. The gain from merging Streamflow's page manager with the custom allocator is wider than on the Intel system, giving rise to up to 27% better performance than page management directly from Linux.

Conclusions

This paper explored the performance and productivity implications of using generic multithreaded memory allocators for parallel mesh generation codes on emerging multiprocessors. We have investigated the merits and disadvantages of using generic memory allocators in the context of three applications with challenging and particularly demanding memory allocation patterns by providing a qualitative and quantitative comparison of custom and generic allocators. Our quantitative comparison used real SMT and multicore-based multiprocessors, and three real applications which are heavily dependent on dynamically allocated and managed data structures, producing finite element mesh sizes in the order of tens of millions of elements each.

The main findings of this paper are summarized as follows:

- Although custom memory allocators achieve on average the best performance in mesh generation codes, generic multithreaded memory allocators designed simultaneously for fast locality-aware sequential allocation, and scalable multithreaded allocation come very close to and occasionally outperform custom allocators. Generic allocators also have negligible deployment costs.
- Generic multithreaded allocators are efficient, only if they adapt well to both sequential and parallel object allocation/deallocation patterns. Our results stress the fact that sequential optimizations in a multithreaded memory allocator are critical for the overall performance of adaptive and irregular applications. In the case of parallel mesh generation, sequential optimizations in the allocator can improve performance by as much as a 70%.
- Generic and custom allocators can be used in synergy. We have built a generic page block manager to support both custom and generic allocators, by providing user-level caching and recycling of page blocks for both small and large objects, using a unified strategy. We show that using a page block manager to support custom allocators, we achieve substantial performance improvement on multithreaded and multicore processors, by reducing TLB misses and minor page faults.

Acknowledgments This work is supported by NSF (grants ITR-ACI-0312980, CAREER-CCR-0346867, MRI-CNS-0521381, ITR-ACI-0085969, CAREER-EIA-9972853, NGS-

EIA-0203974), and the Department of Energy (grant ECPI-DE-FG02-05ER25689). Filip Blagojevic implemented and optimized the multithreaded version of MPCDM.

References

[1] C. Antonopoulos, X. Ding, A. Chernikov, F. Blagojevic, D. Nikolopoulos, and N. Chrisochoides. Multigrain Parallel Delaunay Mesh Generation: Challenges and Opportunities for Multithreaded Architectures. In *Proc. of the 19th ACM International Conference on Supercomputing (ICS'2005)*, pages 367–376, Cambridge, MA, June 2005.

[2] E. Berger, K. Mckinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambridge, MA, November 2000.

[3] D. Blandford, G. Blelloch, D. Cardoze, and C. Kadow. Compact representations of simplicial meshes in two and three dimensions. In *Proceedings of the 12th International Meshing Roundtable*, pages 135–146, 2003.

[4] Andrey N. Chernikov and Nikos P. Chrisochoides. Generalized Delaunay mesh refinement: From scalar to parallel. In *Proceedings of the 15th International Meshing Roundtable*, pages 563–580, Birmingham, AL, September 2006. Springer.

[5] Andrey N. Chernikov and Nikos P. Chrisochoides. Algorithm 872: Parallel 2D constrained Delaunay mesh generation. *ACM Transactions on Mathematical Software*, 34(1), March 2007. In press.

[6] Nikos Chrisochoides and Démian Nave. Parallel Delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering*, 58:161–176, 2003.

[7] Nikos P. Chrisochoides. A survey of parallel mesh generation methods. Technical Report BrownSC-2005-09, Brown University, 2005. Also appears as a chapter in Numerical Solution of Partial Differential Equations on Parallel Computers (eds. Are Magnus Bruaset and Aslak Tveito), Springer, 2006.

[8] Google. Google performance tools. http://goog-perftools.sourceforge.net/.

[9] D. Lea. A Memory Allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[10] Leonidas Linardakis and Nikos Chrisochoides. A static medial axis domain decomposition for 2D geometries. *ACM Transactions on Mathematical Software*, 2007. In press.

[11] M. Michael. Scalable Lock-free Dynamic Memory Allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 35–46, Washington, DC, June 2004.

[12] Leonid Oliker and Rupak Biswas. Parallelization of a dynamic unstructured application using three leading paradigms. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, New York, NY, USA, 1999. ACM Press.

[13] S. Schneider, C. Antonopoulos, and D. Nikolopoulos. Scalable Locality-Conscious Multithreaded Memor Allocation. In *Proc. of the 2006 ACM SIGPLAN International Symposium on Memory Management*, Ottawa, Canada, June 2006.

[14] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22(1–3):21–74, May 2002.