Multithreaded Geant4: Semi-automatic Transformation into Scalable Thread-Parallel Software

Xin Dong[1], Gene Cooperman[1], and John Apostolakis[2]

[1] College of Computer Science, Northeastern University, Boston, MA 02115, USA
{xindong,gene}@ccs.neu.edu
[2] PH/SFT, CERN, CH-1211, Geneva 23, Switzerland
John.Apostolakis@cern.ch

Abstract. This work presents an application case study. Geant4 is a 750,000 line toolkit first designed in the mid-1990s and originally intended only for sequential computation. Intel's promise of an 80-core CPU meant that Geant4 users would have to struggle in the future with 80 processes on one CPU chip, each one having a gigabyte memory footprint. Thread parallelism would be desirable. A semi-automatic methodology to parallelize the Geant4 code is presented in this work. Our experimental tests demonstrate linear speedup in a range from one thread to 24 on a 24-core computer. To achieve this performance, we needed to write a custom, thread-private memory allocator, and to detect and eliminate excessive cache misses. Without these improvements, there was almost no performance improvement when going beyond eight cores. Finally, in order to guarantee the run-time correctness of the transformed code, a dynamic method was developed to capture possible bugs and either immediately generate a fault, or optionally recover from the fault.

1 Introduction

The number of cores on a CPU chip is currently doubling every two years, in a manner consistent with Moore's Law. If sequential software has a working set that is larger than the CPU cache, then running a separate copy of the software for each core has the potential to present immense memory pressure on the bus to memory. It is doubtful that the memory pressure will continue to be manageable as the number of cores on a CPU chip continues to double.

This work presents an application case study concerned with just this issue, with respect to Geant4 (GEometry ANd Tracking, http://geant4.web.cern.ch/). Geant4 was developed over about 15 years by physicists around the world, using the Booch software engineering methodology. The widest use for Geant4 is for Monte Carlo simulation and analysis of experiments at the LHC collider in Geneva. In some of the larger experiments, such as CMS [1], software applications can grow to a two gigabyte footprint that includes hundreds of dynamic libraries (.so files). In addition to collider experiments, Geant4 is used for radiation-based medical applications [2], for cosmic ray simulations [3], and for space and radiation simulations [4].

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part II, LNCS 6272, pp. 287–303, 2010.

Geant4 is a package with 750,000 lines of C++ code spread over 6,000 files. It is a toolkit with deep knowledge of the physics of particle tracks. Given the geometry, the corresponding materials and the fundamental particles, a Geant4 simulation is driven by randomly generated independent events. Within a loop, each event is simulated in sequence. The corresponding computation for each event is organized into three levels: event generation and result aggregation; tracking in each event; and stepping on each track. Geant4 stepping is governed by physics processes, which specify particle and material interaction.

The Geant4 team of tens of physicists issues a new release every six months. Few of those physicists have experience writing thread-parallel code. Hence, a manual rewriting of the entire Geant4 code base for thread parallelism was not possible. Geant4 uses an event loop programming style that lends itself to a straightforward thread parallelization. The parallelization also required the addition of the ANSI C/C++ `__thread` keyword to most of the files in Geant4. As described in Section 3.1, an automatic way to add this thread parallelism was developed by modifying the GNU C++ parser. Section 3.2 then describes a further step to reduce the memory footprint. This thread-parallel implementation of Geant4 is known as *Geant4MT*. However, this intermediate Geant4MT was found not to be scalable. When scaling to 24 cores, two important performance drains were found: memory allocation and writes to shared variables.

Custom memory allocator. First, none of the standard memory allocators scale properly when used in Geant4. Some of the allocators tried include the glibc default malloc (ptmalloc2) [5], tcmalloc [6], ptmalloc3 [5] and hoard [7]. This is because the malloc standard requires the use of a shared memory data structure so that any thread can free the memory allocated by any other thread. Yet most of the Geant4 allocations are thread-private. The number of futex calls in Geant4MT provided the final evidence of the importance of a thread-private custom memory allocator. We observed the excessive number of futex calls (Linux analog of mutex calls) to completely disappear after introducing our thread-private allocator.

Writes to shared variables. The second important drain occurs due to excessive writes to shared variables. This drain occurs *even when the working set is small*. Note that the drain on performance makes itself known as excessive cache misses when measuring performance using performance counters. However, this is completely misleading. The real issue is the particular cache misses caused by a write to a shared variable. Even if the shared variable write is a cache hit, all threads that include this shared variable in their active working set will eventually experience a read/write cache miss.

This is because there are four CPU chips on the motherboard *with no off-chip cache*, in the high performance machines on which we tested. So, a write by one of the threads forces the chip set logic to invalidate the corresponding cache lines of the other three CPU chips. Thus, a single write eventually forces three subsequent L3 cache misses, one miss in each of the other three chips.

The need to understand this unexpected behavior was a major source of the delay in making Geant4 fully scalable. The interaction with the malloc issue

above initially masked this second performance drain. It was only after solving the issue of malloc, and then building a mechanism to track down the shared variables most responsible for the cache misses, that we were able to confirm the above working hypothesis. The solution was then quite simple: eliminate unnecessary sharing of writable variables.

Interaction of memory allocator and shared writable variables. As a result of this work, we were able to conclude that the primary reason that the standard memory allocators suffered degraded performance was likely not the issue of excessive futex calls. Instead, we now argue that it was due to writes to shared variables of the allocator implementation. Our back-of-the-envelope calculations indicated that there were not enough futex calls to account for the excessive performance drain!

We considered the use of four widely used memory allocators, along with our own customized allocator for a malloc/free intensive toy program. Surprisingly, we observed a *parallel slowdown* for each of the four memory allocators. In increasing the number of threads from 8 to 16 on a 16-core computer, the execution was found to become *slower*!

Reasoning for correctness of Geant4. The domain experts are deeply concerned about the correctness of Geant4MT. Yet it challenges existing formal methods. First, the ubiquitous callback mechanism and C++ virtual member functions defined in Geant4 resist static methods. What part of the code will be touched is determined dynamically at the run-time. Second, the memory footprint is huge for large Geant4 applications, rendering dynamic methods endless. For an example, Helgrind [8] makes the data initialization too slow to finish for a representative large Geant4 application.

Because Geant4MT, like Geant4, is a toolkit with frequent callbacks to end user code, we relax the correctness requirements. It is not possible with today's technology to fully verify Geant4MT in the context of arbitrary user callbacks. Hence, we content ourselves with enhancements to verify correctness of production runs. In particular, we enforce the design assumption that "shared application data is never changed when parallel computing happens". A run-time tool is developed to verify this condition. This tool also allows the application to coordinate the threads so as to avoid data races when updates to shared variables occur unexpectedly.

Experience with Geant4MT. Geant4MT represents a development effort of two and a half years. This effort has now yielded experimental results showing linear speedup both on a 24-core Intel computer (four Nehalem-class CPU 6-core chips), and on a 16-core AMD computer (four Barcelona-class CPU 4-core chips). The methodology presented here is recommended because it compresses these two and a half years of work into a matter of three days for a new version of Geant4. By using the tools developed as part of this work, along with the deeper understanding of the systems issues, we estimate the time for a new project to be the same three days, plus the time to understand the structure of the new software and create an appropriate policy about what objects should be shared,

while respecting the original software design. The contributions of this work are four-fold. It provides:

1. a semi-automatic way to transform C++ code into working thread-parallel code;
2. a thread private malloc library scalable for intensive concurrent heap accesses to transient objects;
3. the ability to automatically attribute frequent sources of cache misses to particular variables; and
4. a dynamic method to guarantee the run-time correctness for the thread-parallel program

One additional novelty in Section 3.4 is an analytical formula that predicts the number of updates to shared variables by all threads, based on measurements of the number of cache misses. The number of shared variable updates is important, because it has been identified as one of two major sources of performance degradation. The experimental section (Section 4) confirms the high accuracy of this formula.

The rest of this paper is organized as follows. Section 2 introduces Geant4 along with some earlier work on parallelization for clusters. Section 3 explains our multithreading methodology and describes the implementation of multithreading tools. Section 4 evaluates the experimental results. We review related work in Section 5 and conclude in Section 6.

2 Geant4 and Parallelization

2.1 Geant4: Background

Detectors, as used to detect, track, and/or identify high-energy particles, are ubiquitous in experimental and applied particle physics, nuclear physics, and nuclear engineering. Modern detectors are also used as calorimeters to measure the energy of the detected radiation. As detectors become larger, more complex and more sensitive, commensurate computing capacity is increasingly demanded for large-scale, accurate and comprehensive simulations of detectors. This is because simulation results dominate the design of modern detectors. A similar requirement also exists in space science and nuclear medicine, where particle-matter interaction plays a key role.

Geant4 [9,10] responds to this challenge by implementing and providing a "diverse, wide-ranging, yet cohesive set of software components for a variety of settings" [9]. Beginning with the first production release in 1998, the Geant4 collaboration has continued to refine, improve and enhance the toolkit towards more sophisticated simulations. With abundant physics knowledge, the Geant4 toolkit has modelling support for such concepts as secondary particles and secondary tracks (for example, through radioactive decay), the effect of electromagnetic fields, unusual geometries and layouts of particle detectors, and aggregation of particle hits in detectors.

2.2 Prior Distributed Memory Parallelizations of Geant4

As a Monte Carlo simulation toolkit, Geant4 profits from improved throughput via parallelism derived from independence among modelled events and their computation. Therefore, researchers have adopted two methods for parallel simulation in the era of computer clusters. The first method is a standard parameter sweep: Each node of the cluster runs a separate instance of Geant4 that is given a separate set of input events to compute. The second method is that of ParGeant4 [11]. ParGeant4 uses a master-worker style of parallel computing on distributed-memory multiprocessors, implemented on top of the open source TOP-C package (Task Oriented Parallel C/C++) [12]. Following master-worker parallelism, each event is dispatched to the next available worker, which leads to dynamic load-balancing on workers. Prior to dispatching the events, each worker does its own initialization of global data structures.

Since then, many-core computing has gained an increasing presence in the landscape. For example, Intel presented a roadmap including an 80-core CPU chip for the future. It was immediately clear that 80 Geant4-based processes, each with a footprint of more than a gigabyte, would never work, due to the large memory pressure on a single system bus to memory. A potentially simple solution takes advantage of UNIX copy-on-write semantics to enhance the sharing of data further by forking the child processes after Geant4 initializes its data. However, the natural object-oriented style of programming in Geant4 encourages a memory layout in which all fields of an object are placed on the same memory page. If just one field of the object is written to, then the entire memory page containing that object will no longer be shared. Hence the copy-on-write approach with forked processes was rejected as insufficiently scalable.

3 Geant4MT Methodology and Tools

The Geant4MT follows the same event-level parallelism as the prior distributed memory parallelization has done. The goal of the Geant4MT is: given a computer with k cores, we wish to replace k independent copies of the Geant4 process with an equivalent single process with k threads, which use the many-core machine in a memory-efficient scalable manner. The corresponding methodology includes the code transformation for thread safety($T1$) and for memory footprint reduction($T2$), the thread private malloc library and the shared-update checker. The transformation work generates two techniques further used by other work: one is to dump the data segment to the thread local storage (TLS [13]) for thread safety, which later produces the thread private allocator; another is to protect memory pages for write-access reflection, which evolves to the run-time tool for shared-update elimination, correctness verification and data race arbitration.

3.1 $T1$: Transformation for Thread Safety

The *Transformation for Thread Safety* transforms k independent copies of the Geant4 process into an equivalent single process with k threads. The goal is

to create correct thread-safe code, without yet worrying about the memory footprint. This transformation includes two parts: global variable detection and global variable privatization. For C++ programs, we collect the information from four kinds of declarations for global variables, which are the possible source of data race. They are *"static" declarations, global declarations, "extern" declarations and "static const" declarations for pointers.* The last case is very special and rare: pointers are no longer constants if each thread holds its own copy of the same object instance.

A rigorous way to collect the information for all global variables is to patch some code in the C++ parser to recognize them. In our case, we change the GNU (g++ version 4.2.2) C++ parser source file *parser.c* to patch some output statements there. After the parser has been changed, we re-compile gcc-4.2.2. We then use the patched compiler to build Geant4 once more. In this pass, all concerned declarations and their locations are collected as part of the building process.

For each global variable declaration, we add the ANSI C/C++ keyword `__thread`, as described by $T1.1$ in Table 1. After this step, the data segment is almost empty with merely some `const` values left. As a result, at the binary level, each thread acts almost the same as the original process, since any variable that could have been shared has been declared thread-local. Naturally, the transformed code is thread-safe.

The handling of thread-local pointers is somewhat more subtle. In addition to the term *thread-local*, we will often refer to *thread-private* data. A thread may create a new object stored in the heap. If the pointer to the new object is stored in TLS, then no other thread can access this new object. In this situation, we refer to both the new object and its *thread-local* pointer as being *thread-private*. However, only the pointer is stored in TLS as a *thread-local* variable. The *thread-local* pointer serves as a remedy for TLS to support variables that are not plain old data (not *POD*) and to implement dynamic initialization for TLS variables. To make non-POD or dynamically initialized variables thread safe, we introduce new variables and transform the original type to be pointer type, which is a plain old data structure (POD). Then initialize variables dynamically before it is first used. Two examples $T1.2$ and $T1.3$ in Table 1 demonstrate the implementation.

A tool based on an open source C++ parser, Elsa [14], has been developed to transform global declarations collected by the patched parser. This tool works not only for Geant4 but also for CLHEP, which is a common library widely used by Geant4. The transformation $T1$ generates an *unconditionally thread-safe Geant4*. This version is called unconditional because any thread can call any function with no data race.

3.2 $T2$: Transformation for Memory Footprint Reduction

The *Transformation for Memory Footprint Reduction* allows threads to share data without violating thread safety. The goal is to determine the read-only variables and field members, and remove the __thread keyword, so that they become shared. A member field may have been written to during its initialization,

Table 1. Representative Code Transformation for $T1$ and $T2$

$T1.1$	*int global = 0;*	$\longmapsto$	*__thread int global = 0;*

$T1.2$	*static nonPOD field;*	$\longmapsto$	*static __thread nonPOD *newfield;*
			if (!newfield) //Patch before refer to, as indicated
			newfield = new nonPOD; //by compilation errors
			*#define field (*newfield)*
			*#define CLASS::field (*CLASS::newfield)*

$T1.3$	*static int var1 = var2;*	$\longmapsto$	*static __thread int *var1_NEW_PTR_ = 0;* *if (!var1_NEW_PTR_)* *{ var1_NEW_PTR_ = new int;* **var1_NEW_PTR_ = var2; }* *int &var1 = *var1_NEW_PTR_;*
$T2.1$	*class volume* *{ //large relatively read-only member field* *RD_t RD;* *//small transitory member field* *RDWR_t RDWR; };* *__thread vector<volume*> store;*	$\longmapsto$	*//dynamically extended by the main thread via the* *//constructor and replicated by worker threads* *1 __thread RDWR_t *RDWR_array;* *2 class volume* *3 { int instanceID;* *4 RD_t RD; };* *5 #define RDWR (RDWR_array[instanceID])* *6 vector<volume*> store;*

but may be read-only thereafter. The difficulty is to figure out for each sharable class (as defined in the next paragraph), which member fields become read-only after the worker threads are spawned. Below, such a field member is referred to as *relatively read-only.*

A *sharable class* is defined as *a class that has many instances, most of whose member fields are relatively read-only.* A *sharable instance* is defined as *an instance of a sharable class.* We take the left part of $T2.1$ in Table 1 as an example. The class "volume" is a sharable class that contains some relatively read-only fields, whose cumulative size is large. The class "volume" also contains some read-write fields, whose cumulative size is small. However, it is not clear which field is relatively read-only and which field is read-write.

To recognize relatively read-only data, we put all sharable instances into a pre-allocated region in the heap by overloading the "new" and "delete" methods for sharable classes and replacing the allocator for their containers. The allocator substitute has overloaded "new" and "delete" methods using the pre-allocated region of sharable instances. Another auxiliary program, the *tracer*, is introduced to direct the execution of the Geant4 application. Our tracer tool controls the execution of the application using the *ptrace* system call similarly to how "gdb" debugs a target process. In addition, the tracer tool catches segmentation fault signals in order to recognize member fields that are not *relatively read-only.* (Such non-relatively read-only data will be called *transitory data* below.)

Figure 1 briefly illustrates the protocol between the tracer and the target application. First, the Geant4 application (the "inferior" process in the terminology of ptrace) sets up signal handlers and spawns the tracer tool. Second, the tracer tool (the "superior" process in the terminology of ptrace) attaches and notifies the "inferior" to remove the "write" permission from the pre-allocated memory region. Third, the tracer tool intercepts and relays each segmentation fault signal to the "inferior" to re-enable the "write" permission for the pre-allocated

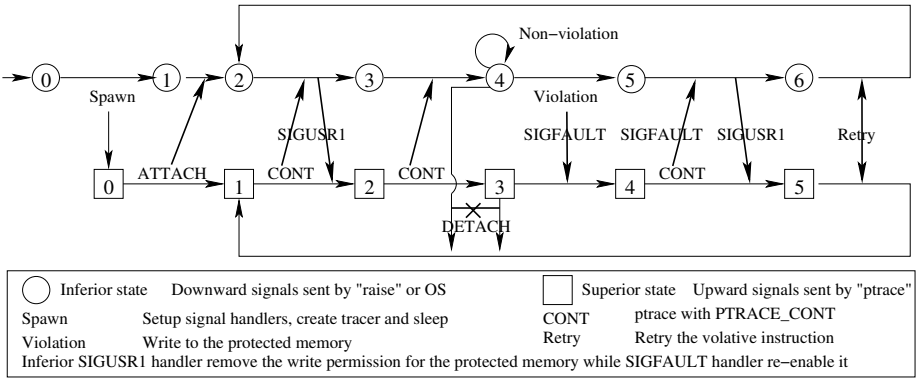

Fig. 1. Interaction between Inferior and Tracer

memory region, retry the instruction that calls the segmentation fault, and return to the second step. As the last step, the "inferior" will actively re-enable the "write" permission for the pre-allocated memory region and tell the tracer tool to terminate, which then forces the tracer tool to detach.

If a sharable class has any transitory member field, it is unsafe for threads to share the whole instance for this class. Instead, threads share instances whose transitory member fields have been moved to the thread-private region. The implementation is described by the right part of $T2.1$ in Table 1, whose objective is illustrated by Figure 2. In this figure, two threads share three instances for the sharable class "volume" in the heap. First, we remove the transitory member field set *RW-Field* from the class "volume". Then, we add a new field as an instance ID, which is declared by line 3 of $T2.1$. In Figure 2, the ID for each shared instance is 0, 1 or 2. Each thread uses a TLS pointer to an array of fields of type RW-Field, which is indexed by the instance ID. The RW-field array is declared in line 1 while the RW-field reference is redefined by the macro on line 5. As we can see from Figure 2, when worker thread 1 accesses the RW-Field set of instance 0, it follows the TLS pointer and assigns the RW-Field to 1. Similarly, worker thread 2 follows the TLS pointer and assigns the RW-Field to 8. Therefore, two threads access the RW-Field of the same instance, but access different memory locations. Following this implementation pattern, $T2$ transforms the *unconditionally thread-safe* version of Geant4 further to share the detector data.

For the $T2$ transformation, it is important to recognize in advance all transitory member fields (fields that continue to be written to). This leads to larger read-only and read-write memory chunks in the logical address space as a side-effect. Furthermore, this helps to reduce the total physical memory consumption even for process parallelism by taking advantage of copy-on-write technology. The internal experiments show that threads always perform better after the performance bottleneck is eliminated.

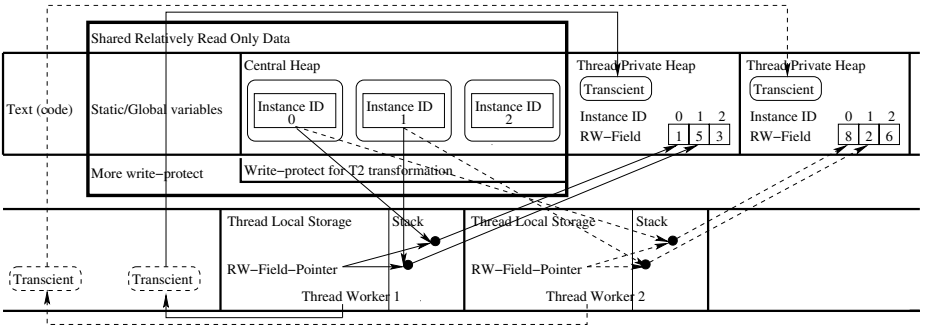

Fig. 2. Geant4MT Data Model

3.3 Custom Scalable Malloc Library

The parallel slowdown for the glibc default malloc library is reproducible through a toy program in which multiple threads work cooperatively on a fixed pool of tasks. The task for the toy program is to allocate 4,000 chunks of size 4 KB and to then free them. As the number of threads increases, the wall-clock time increases, even though the load per thread decreases.

An obvious solution to this demonstrated problem would be to directly modify the Geant4MT source code. One can pre-allocate storage for "hot", or frequently used, classes. The pre-allocated memory is then used instead of the dynamically allocated one in the central heap. This methodology works only if there is a small upper bound on the number of object instances for each "hot" class. Further, the developer time to overload the "new" and "delete" method, and possibly define specialized allocators for different C++ STL containers, is unacceptable. Hence, this method is impractical except for software of medium size.

The preferred solution is a thread-private malloc library. We call our implementation *tpmalloc*. This thread-private malloc uses a separate malloc arena for each thread. This makes the non-portable assumption that if a thread allocates memory, then the same thread will free it. Therefore, a thread-local global variable is also provided, so that the modified behavior can be turned on or off on a per-thread basis.

This custom allocator can be achieved by applying $T1$ to any existing malloc library. In our case, we modify the original malloc library from glibc to create the thread-private malloc arenas. In addition, we pre-initialize a memory region for each worker thread and force it to use the thread private top chunk in the heap. In portions of the code where we know that a thread executing that code will not need to share a central heap region, we turn on a thread-local global variable to use a thread-private malloc arena. As Figure 2 shows, this allows Geant4MT to keep both transient objects and transitory data in a thread-private heap region. Therefore, the original lock associated with each arena in the glibc malloc library, is no longer used by the custom allocator.

3.4 Detection for Shared-Update and Run-Time Correctness

The Geant4MT methodology makes worker threads share read-only data, while other data is stored as thread-local or thread-private. This avoids the need for thread synchronization. If the multithreaded application still slows down with additional cache misses, the most likely reason is the violation of the assumption that all shared variables are read-only. Using a machine with 4 Intel Xeon 7400 Dunnington CPUs (3 levels cache and 24 cores in total), we estimate the number of additional write cache misses generated by updates to shared variables via a simple model. This number would otherwise be difficult to measure.

Suppose there are $w \geq 2$ thread workers that update the same variable. The total number of write accesses is designated as n. In the simulation, write accesses are assumed to be evenly distributed. The optimal load-balancing for thread parallelism is also assumed, so that the number of updates is n/w per thread. Consider any update access u_1 and its predecessor u_0. One cache miss results from u_1 whenever the variable is changed by another thread between the times of u_0 and u_1. It is easy to see that any write access from another thread falls between u_0 and u_1 with probability w/n. Therefore, the probability that no update happens during this period is $(1 - w/n)^{(n-n/w)} \approx e^{(1-w)}$. Furthermore, one cache miss happens at u_1 with probability $1 - e^{(1-w)}$.

For the L1 cache, each core has its own cache, and it suffices to set w to be the number of cores. In the L3 case, there is a single L3 cache per chip. Hence, for L3 cache and a machine with multiple chips, one can consider the threads of one chip as a single large super-thread and apply the formula by setting the number of threads w to be the number of super-threads, or the number of chips. Similar ideas can be used for analyzing L2 cache.

The performance bottleneck from updates to shared variables is subtle enough to make testing based on the actual source code impractical for a software package as large as Geant4. Luckily, we had already developed a tracer tool for Transformation $T2$ (see Section 3.2, along with Figure 1). An enhancement of the tracer tool is applicable to track down this bottleneck. The method is to remove write permission from all shared memory regions and to detect write accesses to the region. As seen in Figure 2, all read-only regions are write-protected. Under such circumstance, the "tracer" will catch a segmentation fault, allowing it to recognize when the shared data has been changed.

This method exposes some frequently used C++ expressions in the code that are *unscalable*. An expression is *scalable* if given a load of n times execution of the expression, a parallel execution with k threads on k cores spends $1/k$ of the time for one thread with the same load. Some of the unscalable C++ expressions found in the code by this methodology are listed as follows:

1. *cout.precision(*);* Shared-updates to precision, even in the absence of output from *cout*.
2. *str = "";* All empty strings refer to the same static value using a reference count. This assignment changes the reference count.

3. *std::ostringstream os;* The default constructor for std::ostringstream takes a static instance of the locale class, which changes the reference count for this instance.

Whenever updates for shared variables occur intensively, the tracer tool can be employed to determine all instructions that modify shared variables. Note that since all workers execute the same code, if one worker thread modifies a shared variable, then all worker threads modify that shared variable. This creates a classic situation of "ping pong" for the cache line containing that variable. Hence, this results in unscalable code. The most frequent such occurrences are the obvious suspects for performance bottlenecks. The relevant code is then analyzed and replaced with a thread-private expression where possible.

The same tracer tool is sensitive to the violation of the $T2$ read-only assumption. So it works also for the production phase to guarantee the run-time correctness of Geant4MT applications. For this purpose, the tracer tool is enhanced further with some policies and corresponding mechanisms for the coordination of shared variable updates.

The tracer tool based on memory protection decides whether the shared data has ever been changed or not by Geant4MT. It serves as a dynamic verifier to guarantee that a production run is correct. If no segmentation fault happens in the production phase, the computation is correct and the results are valid. When a segmentation fault is captured, one just aborts that event. The results from all previous events are still valid. The tracer tool has zero run-time overhead in previous events in the case that the $T2$ read-only assumption is not violated.

A more sophisticated policy is to use a *recovery strategy*. The tracer tool suspends each thread that tries to write to the shared memory region. In that event all remaining threads finish their current event and arrive at a quiescent state. Then, all quiescent threads wait upon the suspended threads. The tracer tool first picks a suspended thread to resume and finish its current event. All suspended threads then redo their current events in sequence. This portion of the computation experiences a slowdown due to the serialization, but the computation can continue without aborting.

To accomplish the above recovery strategy, a small modification of the Geant4MT source code is needed. The Geant4MT application must send a signal to the tracer tool before and after each Geant4 event. When violations of the shared read-only data are rare, this policy has a minimal effect on performance.

4 Experimental Results

Geant4MT was tested using an example based on FullCMS. FullCMS is a simplified version of the actual code used by the CMS experiment at CERN [1].

This example was run on a machine with 4 AMD Opteron 8346 HE processors and a total of 16 cores working at 1.8 GHz. The hierarchy of the cache for this CPU is a single 128 KB L1 and a single 512 KB L2 cache (non-inclusive) per core. There is a 2 MB L3 cache shared by the 4 cores on the same chip. The cache line size is 64 bytes. The kernel version of this machine is Linux 2.6.31 and the compiler is gcc 4.3.4 with the "-O2" option, following the Geant4 default.

Removal of futex delays from Geant4MT. The first experiment, whose results are reported in the left part of Table 2, reveals the bottleneck from the ptmalloc2 library (the glibc default). The total number of Geant4 events is always 4800. Each worker holds around 20 MB of thread-private data and 200 MB of shared data. The shared data is initialized by the master thread prior to spawning the worker threads. Table 2 reports the wall-clock time for the simulation. From this table, we see the degradation of Geant4MT. Along with the degradation is a tremendously increasing number of futex system calls.

Table 2. Malloc issue for FullCMS/Geant4MT: 4 AMD Opteron 8346 HE (4×4 cores) vs. 4 Intel Xeon 7400 Dunnington (4×6 cores). Time is in seconds.

4 AMD Opteron 8346 HE CPUs						4 Intel Xeon 7400 Dunnington CPUs					
Number of	ptmalloc2			tpmalloc		Number of	ptmalloc2			tpmalloc	
Workers	Time	Speedup	Futex & Time	Time	Speedup	Workers	Time	Speedup	Futex & Time	Time	Speedup
1	10349	1	0, 0	10285	1	1	6843	1	0, 0	6571	1
4	2650	3.91	2.4K, 0.04	2654	3.87	6	1498	4.57	13K, 0.3	1223	5.37
8	1406	7.36	38K, 0.4	1355	7.59	12	1050	6.51	24M, 266	824	7.97
16	804	12.87	24M, 244	736	13.98	24	654	10.3	66M, 1281	496	13.25

After replacing the glibc default ptmalloc2 with our tpmalloc library in Geant4MT, the calls to futex completely disappeared. (This is because the tpmalloc implementation with thread-private malloc arenas is lock-free.) As expected, the speedup increased, as seen in Table 2. However, that speedup was less than linear. The reason was that another performance bottleneck still existed in this intermediate version! (This was the issue of writes to shared variables.)

Similar results were observed on a second computer populated with 4 Intel Xeon 7400 Dunnington CPUs (24 cores in total), running at 2.66 GHz. This CPU has three levels of cache. Each core has 64 KB of L1 cache. There are three L2 caches of 3 MB each, with each cache shared among two cores. Finally, there is a single 16 MB L3 cache, shared among all cores on the same chip. The cache line size is 64 bytes for all three levels. The kernel version on this machine is Linux 2.6.29 and the compiler is gcc 4.3.3 with the "-O2" option specified, just as with the experiment on the AMD computer.

The load remained at 4800 Geant4 events in total. The wall-clock times are presented in the right part of Table 2. This table shows that the malloc libraries do not scale as well on this Intel machine, as compared to the AMD machine. This may be because the Intel machine runs at a higher clock rate. On the Intel machine, just 12 threads produce a number of futexes similar to that produced by 16 threads on the AMD computer.

Analysis of competing effects of futexes versus cache misses. The performance degradation due to writes to shared variables tends to mask the distinctions among different malloc libraries. Without that degradation, one expects still greater improvements from tpmalloc, for both platforms. It was the experimental results from Table 2 that allowed us to realize the existence of this

one additional performance degradation. Even with the use of tpmalloc, we still experienced only a 13 times speedup with 24 threads. Yet tpmalloc had eliminated the extraordinary number of futexes, and we no longer observed increasing system time (time in kernel) for more threads. Hence, we eliminated operating system overhead as a cause of the slowdown.

Having removed futexes as a potential bottleneck, we turned to considering cache misses. We measured the number of cache misses for the three cache levels. The results are listed in Table 3 under the heading "before removal". The number of L3 cache misses for this case was observed to increase excessively.

As described in the introduction, this was later ascribed to writes to shared variables. Upon discovering this, some writes to shared variables were removed. The heading "after removal" refers to L3 cache misses after making thread-private some frequently used variables with excessive writes.

Table 3. Shared-update Elimination on 4 Intel Xeon 7400 Dunnington (4×6 cores)

Number of	Non-dominating statistics				Before removal		After removal		
Workers	# Instructions	L1-D Misses	L2 Misses	L3 References	L3 Misses	CPU Cycles	L3 Misses	Time	Speedup
1	1,598	24493M	402M	87415M	293M	1945G	308M	6547s	1
6	1,598G	24739M	630M	87878M	326M	2100G	302M	1087s	6.02
12	1,598G	24742M	634M	88713M	456M	3007G	302M	543s	12.06
24	1,599G	24827M	612M	88852M	517M	3706G	294M	271s	24.16

Estimating the number of writes to shared variables. It remains to experimentally determine if the remaining performance degradation is due to the updates to shared variables, or due to other reasons. For example, if the working set is larger than the available cache, this would create a different mechanism for performance degradation. If updates to shared variables is the primary cause of cache misses, then the analytical formula of Section 3.4 can be relied upon to correctly predict the number of updates to shared variables. The formula will be used in two ways.

1. The formula will be used to confirm that most of the remaining cache misses are due only to updates to shared variables. This is done by first considering five different cases from the measured data: L2/6 (L2 cache misses with 6 threads); L2/12; L2/24; L3/12; and L3/24. This will be used to show that all of the legal combinations predict approximately the same number of shared variable updates.
2. Given the predicted number of shared variable updates, this is used to determine if the number of shared variable updates is excessive. When can we stop looking for shared variables that are frequently updated? Answer: when the number of shared variable updates is sufficiently small.

Some data from Table 3 can be used to predict the number of shared variable updates following the first usage of the formula. The results are listed in Table 4. This table provides experimental validation that the predicted number of shared variable updates can be trusted.

Table 4. Prediction for Shared Variable Updates on 4 Intel Xeon 7400 Dunnington (4×6 cores)

Level of Cache	Number of Workers	Number of Cache Misses			Number of Threads or Super-threads (w)	Prediction for Shared Variable Updates (n)
		Single Thread	Multiple Threads	Additional		
L2 cache	6	402×10^6	630×10^6	228×10^6	3	$\approx 263 \times 10^6$
L2 cache	12	402×10^6	634×10^6	232×10^6	6	$\approx 230 \times 10^6$
L2 cache	24	402×10^6	612×10^6	210×10^6	12	$\approx 210 \times 10^6$
L3 cache	12	293×10^6	456×10^6	163×10^6	2	$\approx 258 \times 10^6$
L3 cache	24	293×10^6	517×10^6	224×10^6	4	$\approx 236 \times 10^6$

Effect of removing writes to shared memory. Table 3 shows that Geant4MT scales linearly to 24 threads after most updates to shared variables are removed. According to our estimate in Table 4, 260×10^6 updates to shared variables have been removed. As seen earlier, with no optimizations, only a 10.3 times speedup using 24 cores was obtained. By using tpmalloc, a 13.25 times speedup was obtained, as seen in Table 2. With tpmalloc and removal of excessive writes to shared variables, a 24.16 times speedup using 24 cores was obtained, as seen in Table 3. In fact, this represents a slight superlinear speedup (a 1% improvement). We believe this is due to reduced L3 cache misses due to sharing of cached data among distinct threads.

Results for different allocators with few writes to shared memory. The last experiment compared different malloc implementations using Geant4MT, after having eliminated previous performance bottlenecks (shared memory updates). Some interesting results from this experiment are listed in Table 5. First, ptmalloc3 is worse than ptmalloc2 for Geant4MT. This may account for why glibc has not included ptmalloc3 as the default malloc library. Second, tcmalloc is excellent for 8 threads and is generally better than hoard although hoard has better speedup for some cases. Our custom thread-private tpmalloc does not show any degradation so that Geant4MT with tpmalloc leads to a better speed-up. This demonstrates the strong potential of Geant4MT for still larger scalability on future many-core machines.

Table 5. Malloc Library Comparison Using Geant4 on 4 AMD Opteron 8346 HE (4×4 cores)

Number of Workers	ptmalloc2		ptmalloc3		hoard		tcmalloc		tpmalloc	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1	9923s	1	10601s	1	10503s	1	9918s	1	10090s	1
2	4886s	2.03	6397s	1.66	6316s	1.66	4980s	1.99	5024s	2.01
4	2377s	4.17	4108s	2.58	2685s	3.91	2564s	3.87	2504s	4.03
8	1264s	7.85	2345s	4.52	1321s	7.95	1184s	8.37	1248s	8.08
16	797s	12.46	1377s	7.70	691s	15.20	660s	15.02	623s	16.20

5 Related Work

Some multithreading work for linear algebra algorithms are: PLASMA [15], which addresses the scheduling framework; and PLUTO [16] and its variant [17], which addresses the compiler-assisted parallelization, optimization and scheduling. While the compiler-based methodologies are fit for the data parallelism existing in tractable loop nests, new approaches are necessary for other applications, e.g., commutativity analysis [18] to automatically extract parallelism from utilities such as gzip, bzip2, cjpeg, etc. The Geant4MT $T1$ transformation is similar to well-known approaches such as the private data clause in OpenMP [19] and Cilk [20]; and the SUIF [21] privatizable directive from either programmer or compiler. Nevertheless, $T1$ pursues thread safety in Geant4 with its large C/C++ code base containing many virtual and callback functions — a context that would overwhelm both the programming and the compile-time analysis.

The Geant4MT $T2$ transformation applies task-oriented parallelism (one event is a task) and gains some data parallelism by sharing relatively read-only data and replicated transitory data. While this transformation benefits existing parallel programming tools, it raises thread safety issues for the generated code. Besides our own tool for run-time correctness, other approaches are also available. Some tools for static data race detection are: SharC [22], which checks data sharing strategies for multi-threaded C via declaring the data sharing strategy and checking; RELAY [23], which is used to detect data races in the Linux kernel; RacerX [24], which finds serious errors in Linux and FreeBSD; and KISS [25], which obtains promising initial results in Windows device drivers. All these methods are unsound. Two sound static data race detection tools are LOCKSMITH [26], which finds several races in Linux device drivers and the method of Henzinger et al. [27], which is based on model checking. Sound methods need to check a large state space and may fail to complete due to resource exhaustion. All these methods, even with their limitations, are crucial for system software (e.g., an O/S) which requires strict correctness and is intended to run forever. In contrast, this work addresses the run-time correctness for application software that already runs correctly in the sequential case.

6 Conclusion

Multithreaded software will benefit even more from future many-core computers. However, efficient thread parallelization is difficult when confronted by two real-world facts. First, software is continually undergoing continuing active development with periodically appearing new releases to eliminate bugs, to enhance functionality, or to port for additional platforms. Second, the parallelization expert and the domain expert often have only limited understanding of each other's job. The methodology presented encourages black box transformations so that the jobs of the parallelization expert and the domain expert can proceed in a largely independent manner. The tools in this work not only enable highly scalable thread parallelism, but also provide a solution of wide applicability for efficient thread parallelization.

Acknowledgement

We gratefully acknowledge the use of the 24-core Intel computer at CERN for testing. We also gratefully acknowledge the helpful discussions with Vincenzo Innocente, Sverre Jarp and Andrzej Nowak. The many performance tests run by Andrzej Nowak on our modified software were especially helpful in gaining insights into the sources for the performance degradation.

References

1. CMS, http://cms.web.cern.ch/cms/
2. Arce, P., Lagares, J.I., Perez-Astudillo, D., Apostolakis, J., Cosmo, G.: Optimization of An External Beam Radiotherapy Treatment Using GAMOS/Geant4. In: World Congress on Medic Physics and Biomedical Engineering, vol. 25(1), pp. 794–797. Springer, Heidelberg (2009)
3. Hohlmann, M., Ford, P., Gnanvo, K., Helsby, J., Pena, D., Hoch, R., Mitra, D.: GEANT4 Simulation of a Cosmic Ray Muon Tomography System With Micro-Pattern Gas Detectors for the Detection of High-rmZ Materials. IEEE Transactions on Nuclear Science 56(3-2), 1356–1363 (2009)
4. Godet, O., Sizun, P., Barret, D., Mandrou, P., Cordier, B., Schanne, S., Remoué, N.: Monte-Carlo simulations of the background of the coded-mask camera for X- and Gamma-rays on-board the Chinese-French GRB mission SVOM. Nuclear Instruments and Methods in Physics Research Section A 603(3), 365–371 (2009)
5. malloc, http://www.malloc.de/en/
6. TCMalloc, http://goog-perftools.sourceforge.net/doc/tcmalloc.html
7. The Hoard Memory Allocator, http://www.hoard.org/
8. Instrumentation Framework for Building Dynamic Analysis Tools, http://valgrind.org/
9. Agostinelli, S., et al.: GEANT4–a simulation toolkit. Nuclear Instruments and Methods in Physics Research Section A 506(3), 250–303 (2003) (over 100 authors, including J. Apostolakis and G. Cooperman)
10. Allison, J., et al.: Geant4 Developments and Applications. IEEE Transactions on Nuclear Science 53(1), 270–278 (2006) (73 authors, including J. Apostolakis and G. Cooperman)
11. Cooperman, G., Nguyen, V., Malioutov, I.: Parallelization of Geant4 Using TOP-C and Marshalgen. In: IEEE NCA 2006, pp. 48–55 (2006)
12. TOP-C, http://www.ccs.neu.edu/home/gene/topc.html
13. Thread-Local Storage, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1966.html
14. Elsa: An Elkhound-based C++ Parser, http://www.cs.berkeley.edu/~smcpeak/elkhound/
15. Parallel Linear Algebra For Scalable Multi-core Architecture, http://icl.cs.utk.edu/plasma/
16. Bondhugula, U., Hartono, A., Ramanujam, J.: A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In: PLDI 2008, vol. 43(6), pp. 101–113 (2008)
17. Baskaran, M.M., Vydyanathan, N., Bondhugula, U.K.R., Ramanujam, J., Rountev, A., Sadayappan, P.: Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors. In: PPoPP 2009, pp. 219–228 (2009)

18. Aleen, F., Clark, N.: Commutativity Analysis for Software Parallelization: Letting Program Transformations See the Big Picture. In: ASPLOS 2009, vol. 44(3), pp. 241–252 (2009)
19. OpenMP, `http://openmp.org/wp/`
20. Cilk, `http://www.cilk.com/`
21. SUIF, `http://suif.stanford.edu/`
22. Anderson, Z., Gay, D., Ennals, R., Brewer, E.: SharC: Checking Data Sharing Strategies for Multithreaded C. In: PLDI 2008, vol. 43(6), pp. 149–158 (2008)
23. Voung, J.W., Jhala, R., Lerner, S.: RELAY: Static Race Detection on Millions of Lines of Code. In: ESEC-FSE 2007, pp. 205–214 (2007)
24. Engler, D., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: SOSP 2003, vol. 37(5), pp. 237–252 (2003)
25. Qadeer, S., Wu, D.: KISS: Keep It Simple and Sequential. In: PLDI 2004, pp. 149–158 (2004)
26. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In: PLDI 2006, vol. 41(6), pp. 320–331 (2006)
27. Henzinger, T.A., Jhala, R., Majumdar, R.: Race Checking by Context Inference. In: PLDI 2004, pp. 1–13 (2004)