Journal of Physics: Conference Series **396** (2012) 052029 doi:10.1088/1742-6596/396/5/052029

Creating and Improving Multi-Threaded Geant4

Xin Dong[1], Gene Cooperman[1], John Apostolakis[2], Sverre Jarp[2], Andrzej Nowak[2], Makoto Asai[3] and Daniel Brandt[3]

[1] College of Computer and Information Science, Northeastern University, Boston MA 02115, USA
[2] CERN, CH-1211 Geneva 23, Switzerland
[3] SLAC National Accelerator Laboratory, Menlo Park, CA 94025, USA.

E-mail: gene@ccs.neu.edu

Abstract. We document the methods used to create the multi-threaded prototype Geant4MT from a sequential version of Geant4. We cover the Source-to-Source transformations applied, and discuss the process of verifying the correctness of the Geant4MT toolkit and applications based on it. Tools to ensure that the results of a transformed multi-threaded application are exactly equal to the original sequential version are under development. Stand-alone or simple applications can be adapted within 1-2 working days. Geant4MT is shown to scale linearly on an 80-core computer. In the special case of a single worker thread on one core, 30% overhead has been observed. We explain the reasons for this and the improvements introduced to reduce this overhead.

1. Introduction

Today's large CPU clusters already encountered power and cooling supply issues such as energy inefficiency, growing energy demand and rising energy prices. Instead of increasing the number of nodes or the frequency of CPU chips, the next round of processing capacity improvements relies on many-core architectures. This leads to a more centralized memory system shared by many cores. Since modern systems consist of CPU cores that are already very fast, the computation of many fast cores has the potential to saturate the shared memory bus. Thread parallelism is expected to fully aggregate all CPU cores without overloading the central memory bus

To adapt Geant4 to run on emerging many-core platforms, the Geant4 collaboration started in 2007 to prototype extensions for multithreading i.e. to adapt the large code base of the toolkit (of order a million lines of code.) The Geant4 multithreading version (referred to below as Geant4MT) takes advantage of event-level parallelism similar to ParGeant4 [1][2] on distributed-memory multiprocessors that is implemented on top of Task Oriented Parallel C/C++ (TOP-C) [14]. On a computer with k cores, Geant4MT can replace k independent copies of a Geant4 process with a single process with k threads, which have a reduced memory footprint and also attain scalability.

In general, to convert a large code base as complex as Geant4, the process of adapting it to multithreading must solve two types of issues. The first is the elimination of race conditions; their detection is NP-hard, or more precisely, no efficient algorithm exists for checking data races in

polynomial time. The second is the significant additional manpower required to maintain a large code base when multithreading is introduced, in order to optimize the memory usage and to improve the performance.

A key goal is to ensure that results of the multi-threaded code are bit-compatible with sequential Geant4 when both start each event with the same random number engine in the same state. In order to meet this goal while taking advantage of task-oriented parallelism, the process to create Geant4MT is decomposed into different stages. In the first stage a version is created in which all data are thread-private (in thread-local storage) - i.e. a shared-nothing strategy is adopted; by construction this should guarantee thread safety, if carried out correctly.

In the second stage the memory footprint is reduced by sharing large data structures that are read-only after initialization (we refer to this as relatively read-only data); again thread safety is required for correctness. In a third stage, we ensure the correctness by splitting into two sub sub-stages: i) using a single worker thread plus the master and ii) ensuring thread independence when multiple workers are used. In the fourth stage performance is measured to identify any degradation; this is attributed to synchronization introduced in the second stage: either explicit or implicit synchronization. To eliminate any performance degradation we eliminate that synchronization, making threads more independent.

The significance and applicability of the work on the Geant4MT prototype is three-fold. The first is in the methodology for creating the multithreaded toolkit, which involves semi-automatic transformations of the source code. With this approach the work of parallelization experts and other Geant4 developers can proceed in a largely independent manner. Therefore, the methods of creating Geant4MT can serve as a basis of wide applicability for efficient multithreading. Secondly, Geant4 is used by many high-energy experiments and individual physicists to simulate detector response and other effects of the interactions of particles in matter, and they have immense and ever increasing demands on throughput. Thirdly, Geant4 is now in the process of being still more widely used by the medical community to estimate the energy and dose deposition in human voxel phantoms [15]. In all of the above cases it is possible, using Geant4MT, to share a very large geometry model between many threads. This enables simulations with a large number of threads to be carried out on a machine with many computing cores and a limited amount of memory.

It is possible to combine Geant4MT with the distributed memory version, ParGeant4. In this case Geant4MT enhances ParGeant4 to support a hybrid-memory model using multiple computers, each with multiple cores. To utilize a computer cluster, we have prototyped the promising approach of combining the distributed memory parallelization of ParGeant4 and the shared memory parallelization of Geant4MT. This synchronizes the global shared data on a per node basis following the TOP-C methodology and supplements it a shared-memory programming model within each node. Coupled with potential refinements to overlap communication and computation, this hybrid-memory parallelism could be used to maintain the scaling of simulation throughput in future extensions of Geant4MT to clusters and NUMA architectures.

The rest of this report is organized as follows. Section 2 introduces the Source-to-Source transformations used to create Geant4MT from a sequential version of Geant4. Section 3 discusses the process of testing and checking the correctness of Geant4MT and related debugging. Section 4 explains Geant4MT performance tools. Section 5 summarizes the Geant4MT methodology and the current process for creating releases. Section 6 discusses the results of Geant4MT performance benchmarking and its scalability.

2. Geant4MT Source-To-Source Transformation

Geant4MT follows the same event-level parallelism as the prior distributed-memory parallelization, ParGeant4, has done. Creating ParGeant4 for computer clusters did not require changes in the design or implementation of Geant4 kernel classes, only the development of extra classes and their use by ParGeant4 applications. In contrast to this, adding multi-threading for Geant4MT has required the modification of both the source code of the Geant4 kernel and the source code of Geant4 applications. The modification of the source code is done in two stages: (i) code modifications to achieve full thread independence (and thus thread-safety), together with the adoption of a thread-safe interface to CLHEP; and (ii) further code modification to reduce the memory footprint, coupled with the introduction of code for worker thread initialization and application parallelization.

2.1. Transformation for Thread Safety (TTS)

The objective of the code Transformation for Thread Safety (TTS) is to decouple completely the operations of all the threads. For this reason it must ensure that all the data, which would have been shared by different threads is identified and that the sharing is eliminated. It achieves this by transferring the contents of the writable part of the program's data segment into thread local storage (TLS). TTS includes two steps: global variable identification and global variable privatization. The privatization takes advantage of the ANSI C keyword __thread (part of the ANSI C99 standard). Correspondingly, all declarations qualified for the TLS keyword should be collected.

TTS consists of iterating over these two steps. Because each target application may have its own idiosyncrasies, one must iterate, observing what variables were not identified in the previous transformation. TTS does not terminate until all such variables are accounted for.

The essential spirit of TTS is to make all global and static variables thread-private. As a result TTS should generate code that is unconditionally thread-safe: any thread can call any function and guarantee the absence of data races. By replacing processes of the distributed ParGeant4 by independent threads, the Inter-Process Communication (IPC) required in ParGeant4 for synchronization is completely eliminated.

A rigorous way to collect the information for all global variables is to utilize a C++ parser to recognize them. In the Geant4MT case, we patched the GNU (GCC version 4.2.2) C++ parser. This patch does not change the output of the compilation. It simply collects all global declarations, the corresponding extern declarations, and all static declarations as a side effect of compilation.

Each qualifying declaration is adorned with the keyword __thread. After this step, at the binary level, the data segment is almost empty, only constant values remain. As a result, each thread behaves almost the same as the original process, since any variable that could have been shared has been declared thread-local. The transformed code is thread-safe even though the C++ standard does not make the same guarantee.

About ten thousand lines of Geant4 code are changed by TTS. In the version of the code modified by TTS, threads share only read-only data from the data segment – so they behave like independent processes. The result is then unconditionally thread-safe. Please refer to the technical note [3] for more detailed information on the Geant4MT TTS code modification. Since this version is unconditionally thread-safe and any thread can call any function with no data race, the event-level parallelism, originally applicable to processes on clusters, can be ported to Geant4MT threads.

The usage of pseudo random number generators (PRNG) Geant4, is changed in order to guarantee reproducibility. The sequential Geant4 uses the static interface of CLHEP in order to use a single generator instance to provide a stream of random numbers for all parts of the simulation. Two

important goals of Geant4MT are to have exactly the same results when given the same input, and to provide results that are bit-compatible with the sequential version. This creates two requirements for Geant4MT and its use of PRNGs. First, it is mandatory to maintain a separate PRNG instance for each thread. Second, the seeding and state of the random number generator for each thread must not depend on any global or static variable. Continuing to rely on the CLHEP static interface that is used in sequential Geant4 cannot satisfy these requirements. An early version of Geant4MT included a revised version of CLHEP to solve this issue. As an improved solution, Geant4MT provides a new static interface to a separate PRNG instance for each thread. This enables it to use the CLHEP unaltered, improving maintainability. In order to ensure that the results of a Geant4MT simulation do not depend on the number of threads, it is recommended that the PRNG be reinitialized at the start of each event with a state that is a function of a run identification number and the event number. (This is common practice in HEP experiment simulation.)

2.2. Transformation for Memory Footprint Reduction (TMR)

A variable may have been written to during its initialization, but may be read-only thereafter, and more specifically, during the computation for each task. This kind of variable is referred to as 'relatively read-only' and is sharable among threads. We can identify some classes as candidates to be shared between threads ('sharable') if they contain only relatively read-only fields, in particular if these fields occupy a large amount of memory. In a number of cases we identified classes with large relatively read-only data that also contain some transitory fields, whose total size is typically small.

The goal of the Transformation for Memory Footprint Reduction (TMR) is to determine the relatively read-only member fields, and to segregate and replicate other (writable) member fields if any exist. Enable a single instance of one class to refer to a different instance of another class in each thread. The current implementation has a drawback: it changes some data members from private to public so that threads can share instances of sharable classes.

The TMR's goal is to make threads share relatively read-only fields whilst replicating transitory fields. It iterates over three steps: (i) select some data structures which may be relatively read-only and check whether they are written to during the execution of each task; (ii) separate fields which are not relatively read-only (transitory data) from data structure and create thread private copies of them; and (iii) initialize relatively read-only data, and replicate the transitory data.

The developer of the multi-threaded application can choose on how many classes to apply the TMR transformations. TMR can terminate when the memory footprint has been sufficiently reduced to satisfy the design goals.

The first step of TMR is to figure out (or confirm) which member fields are relatively read-only for each candidate sharable class. For this purpose, we overload the 'new' and 'delete' methods for the class and its container. The objective is to put all instances of the class into a pre-allocated region in the heap. An auxiliary program, the superior, is introduced to direct the execution of the original sequential program. The relationship between the superior and the sequential program's process is similar to that between a debugger, such as gdb, and the process being debugged.

The protocol between the superior and the inferior is the following. Once the initialization stage of the program is finished, and before processing the first task, the superior notifies the inferior to remove write permission for the pre-allocated memory region. Any update to writable fields will then generate a segmentation fault, which is intercepted by the superior. The superior records each update instruction, and then notifies the inferior to recover write permission. The inferior retries the update instruction and continues.

Given a sharable class, if there are member fields that are not relatively read-only, this class must be split such that all non-read-only fields are moved to a new subclass (Figure 1). Worker threads replicate and hold their own copy of object instances for this subclass. However, there is only one copy of instances for the sharable class. This step of code modification maintains the thread safety from the previous code modification because TMR-modified sharable classes become relatively read-only. More detailed information concerning the Geant4MT TMR code modification for several geometric classes split is available in the CERN OpenLab technical notes [3]. Other split classes are changed in a similar way.

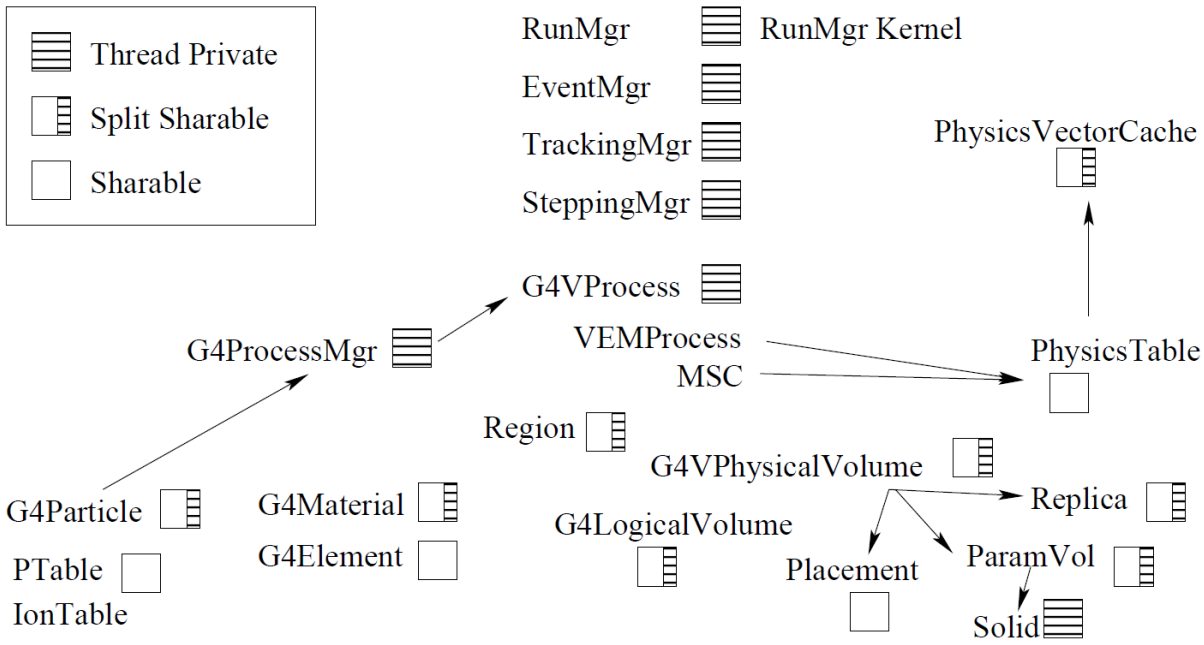

Figure 1: An overview of the design of Geant4 MT's shared, split and private classes. Classes in which large amount of data is stored are shared ("Sharable"): e.g. Physics-Table. To enable this some classes have one part shared, and another private ("Split Shareable"): e.g. a G4Particle type shares all constant properties and has a separate Process-Manager per thread. The remaining classes are replicated for each thread ("Thread Private"); this includes all physics processes and all classes dealing with tracking, event and run management.

Since TMR changed the fundamental data structure of Geant4MT, a new initialization phase for worker threads becomes necessary. Firstly the initialization phase for the master thread can be carried out in exactly the same way as the original sequential Geant4 program. After this the initialization of the worker thread mechanism is expected to (i) avoid to do anything for those classes which are completely shared and (ii) create and initialize subclass instances for classes which are partially shared. In addition, new methods are implemented to initialize object instances for each sharable class split. Furthermore, several manager class methods are supplemented with the code to direct the initialization for worker threads.

Geant4 applications are made multithreaded following a method similar to that used to adapt them to ParGeant4 on distributed-memory clusters. A new 'main' function and a thread function are implemented as wrappers to the application's original main function. The application's original main function is enhanced to coordinate the two initialization phases in Geant4MT: one for the master thread and another for worker threads. The first phase of initialization is completely sequential and

there is only one thread running. In the second phase, all worker threads act independently in parallel to initialize their own thread-private data by copying the relevant data from the master thread.

Worker threads are spawned in the event loop method (DoEventLoop) of Geant4MT's parallel run manager class, ParRunManager. This is derived from the default Geant4 run manager class, G4RunManager. In addition to the default Geant4 DoEvent Loop method, the Geant4MT DoEventLoop includes two pieces of extra code in order to (i) initialize and designate per event seeds and (ii) create and synchronize worker threads.

2.3. Adapting applications and handling output

The duty for handling of the output of a Geant4MT application is shared between Geant4MT and the application developer. Geant4MT handles the management of hit collections if the application uses the default Hits capability of the toolkit. Since hits are accumulated separately for each event, and only a single thread simulates all the tracks of the same event, the mechanism to manage these hit collections already exists in Geant4. Currently it remains a user's responsibility to manage the deletion of hits, if these are created using per-thread memory allocators.

The specialized output of hits for scoring flux, dose and related quantities has adapted in Geant4MT. It provides output already consolidated from all threads. This can be used as a template for the output of hits, for applications with moderate volume of hits data.

Handling of large per-event output on a system with limited memory presents challenges that are not currently addressed by Geant4MT. To overcome this, current practice for stand-alone applications is to create a separate output file for the user hits of each thread. Improving on this and providing solutions that can be adapted for experiment frameworks are topics for future development.

Other objects or files that are created in user code are the responsibility of the application developer. In particular users must adapt their own User Action classes, which are called at the start or end of each run (User Run Action) and each event (User Event Action) or after every step (User Stepping Action). The modifications would be usually restricted to a small set of changes, such as to accumulate sums of observables on each thread separately and to either store them separately or create additional code to sum them at the end. Such changes would not have to pose significant overheads if one takes advantage of the independence of simulation and hit creation. Users can choose to enhance the Geant4MT DoEventLoop to pack, unpack and/or accumulate this data, or handle this in their own action at the end of a Geant4 run (User Run Action.)

3. Geant4MT Testing, Debugging and Run-time Correctness

Given a Geant4 application, the corresponding multithreaded code is verified via several steps of comparison. There are three versions of the Geant4 application involved in the test: (i) the sequential specification version, (ii) the process parallel specification version and (iii) the multi-threaded version, whose correctness is the primary concern.

In order to obtain exactly the same results between parallel and sequential simulation we must ensure that all input, including the state of the random number engine, is the same at the start of each event. In the case that this is not guaranteed by the original application, we can adapt it using a simple method. To obtain the sequential specification version we replace the default run manager with a child class, which has a modified event loop method (DoEventLoop) to assign seeds on a per event basis. This is done in the same way as ParRunManager, the class for distributed memory parallelization does it. This revised sequential Geant4 application is the baseline sequential version for the multithreaded version.

The process parallel specification version is implemented via a modified run manager, COWRunManager, which is also derived from the default run manager. The DoEventLoop of this class is enhanced to (i) use fork to create new processes that act as workers in a way similar to the way that ParRunManager creates thread workers; (ii) assign per event seeds in the same way as ParRunManager does; and (iii) distribute events for process workers in a round robin fashion similar to what ParRunManager does for thread workers. If the process parallel specification version is qualified as a correct implementation for the multithreaded version in a single-worker case, then the former is qualified as a specification for the latter in a multiple-worker case.

The correctness of the process parallel specification version is verified by comparing its results in a single-worker case with the results generated by the sequential specification version. The criterion used to decide whether two versions of simulation results for the same event match is that two sets of tracks with per track stepping are identical. However, this criterion is costly because the output is huge in most cases when the tracking verbose is set to 1.

For many applications it is possible to verify accuracy with different criteria that are physical and easier to compare. These can use observable values which are accumulated within each event such as hits composed of energy deposit or the sums of values after several events - such as particle fluxes, doses or other scored quantities. In that case the criterion can be relaxed to the two sets of accumulated values being either a) identical in the case of single event quantities or b) as close to identical as can be expected due to the different ordering of floating point operations (for observables accumulated across events in different order.) Although these relaxed criteria are not strict, they are less computation intensive. Moreover, the relaxed criterion (a) for single event quantities is reliable if several quantities are monitored and statistics over a large number of events is gathered. If two sets of variables accumulated over many steps are the same, the likelihood that another pair simulations in the future will provide results that are significantly different should be small.

The bugs introduced during parallelization are typically dominant in the sense that those bugs diverge the stepping computations. Therefore, the accumulated number of steps serves as a unique, simple and powerful property for matching two simulation results. In addition other statistic variables can be employed as criteria to match simulations, depending on the choice of application developers.

After the process parallel version passes the correctness verification, the simulation results from the multithreaded version are compared with its simulation results using one of the criteria discussed above. Both versions are configured to use the same number of workers. Due to the fact that the output volume is large a script is used to separate the output from each worker and the compare the corresponding workers' outputs.

Generally, two rounds of comparison are necessary, one for a single-worker case; another for a multiple-worker case.

3.1. Debugging via Bisimulation

If two versions of simulation results do not match in the single-worker case, a bisimulation tool is applied to Geant4 applications to verify the following predicate:

For "any" Geant4 event, the execution via Geant4MT with one master thread and one worker thread must be identical with the execution via the original program.

Following this method, we compare the transformed program with the original version to locate an injected error that differentiates two versions of programs. The comparison verifies the multithreaded version against the original program to check whether they behave identically when they simulate the

same set of events, and thus particle tracks. This is exactly the proof via the bisimulation in which the original program is regarded as the specification and the multithreaded version is regarded as the implementation. Furthermore, the problematic instruction pair that leads to any divergence is the clue for the root cause of the error.

The tool used for debugging and comparison is URDB [4]. This adopts a temporal approach, allowing a user to rollback the execution of one, or in bisimulation model two programs. To achieve this it integrates the fast checkpoint-restart capability from the Distributed Multi-Threaded Check-Pointing (DMTCP) library [5]. It uses this to facilitate debugging by comparison. This tool simplifies this technique to a single-threaded case in which determinism is guaranteed if the application does not have uninitialized variables.

The interface of the bisimulation tool contains six windows for two debugging sessions, one for the original program and another for the multithreaded version. Each session has one DMTCP coordinator, one inferior under debugging and one gdb instance. Two important buttons are forward and binary. Forward is a large step striding: run several steps, if both programs stop at the same place, checkpoint and continue. Forward does not stop until two inferiors become diverse, which is followed by the binary search for where the programs start to diverge.

The place where a program stopped refers to a set of stack layers between the top function and the function that is being checked together with the current position in the top function. If two sets of stack layers and two current positions from two programs match, then the bisimulation tool can optimistically think that the computation of two programs also matches. This is an approximate method whose precision decreases along with a growing size of steps during the forward search or the binary search. In some cases, the instruction pair found might not be the first pair of instructions differentiating two executions and therefore it does not tell the root cause for the divergence of two programs. Under such circumstances, the size of stepping has to be reduced gradually until the earliest pair of instructions differentiating the two executions are found, indicating the root cause of the injected error.

It is very important for the bisimulation tool to take advantage of DMTCP when the computation is very large, i.e., there is a rather long initialization phase or the error occurs merely after executing for a very long time. In the case that the error happens very late, an efficient way is to checkpoint two gdb sessions to files after the initialization phase and later, at a time when the two programs have an identical behavior yet close enough to the time when the error is observed. As a result, each debugging round can begin with a restarting from the checkpointing files then replaying/comparing a relatively short execution. For programs that run a very short time, GDB checkpointing mechanism is also applicable for the bisimulation tool. No matter what checkpointing tool is selected, some minor change is required to echo the checkpointing filename in the DMTCP case, or to echo the identity of the checkpoint in the GDB checkpointing case.

3.2. Data Race Coordinator (DRC) for Runtime Correctness

If two versions of results for the single-worker case match yet two versions of results for the multiple-worker case do not, then the only reason is that threads influence each other via updating to shared data. To understand what data are shared, yet changed during the event simulation phase, Geant4MT extends the selective memory protection technique from TMR to guarantee the runtime correctness, which is defined as follows:

Unintended updates to shared data should be executed only by a single thread during that thread's current task, and during this time, no other thread should change any shared data.

The implementation is to enhance the superior with a recovery strategy to become DRC, which controls the inferior in a per-thread fashion. The recovery strategy is implemented by suspending each thread that is trying to update the shared memory region. Then, all remaining threads finish their current task and arrive at a quiescent state. Then, all quiescent threads wait upon the suspended threads. The superior first picks a suspended thread to resume and finish its current task. All suspended threads then redo their current tasks in sequence.

For the testing and debugging purpose, DRC needs only one worker thread, which is simpler and more efficient. Here, a conservative yet practical usage of DRC is to always use DRC to check the correctness for any multithreaded Geant4 application. When violations of shared data are rare, this policy has a minimal effect on performance (proportional to the number of writes). In this case the overhead is lightweight.

DRC can also serve 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. One naive policy is just to abort that event, when a segmentation fault is captured. The results from all previous events are still valid. This naive policy has zero runtime overhead in previous events until the runtime correctness is violated.

The hint to choose the naive policy or the recovery policy depends on how much confidence programmers have concerning Geant4MT and applications. If a multithreaded Geant4 application has been tested with coverage close to 100 percent, then the naive policy is sufficient. If the parallelized program has not been tested thoroughly, then the recovery policy will make the computation results trustworthy.

3.3. Test Coverage

The test for the Geant4MT kernel and the verification for its correctness are different from the test and verification for multithreaded Geant4 applications. This is not a parallelization specific issue. Instead, it is related to what test cases fit for the Geant4 kernel. Strictly speaking, the Geant4 kernel test and correctness verification requires a sufficient set of Geant4 applications as test drivers (The Geant4 kernel as a test stub). Correspondingly, the Geant4MT kernel test and correctness verification requires a sufficient set of multithreaded test drivers.

From another point of view, Geant4MT introduces new code into the Geant4 kernel. Therefore, the most important thing is to test this part of code, i.e., to guarantee that those shared variables are relatively read-only. To meet this goal a proper set of test cases for the Geant4MT kernel should be several multithreaded Geant4 applications that cover all usage of those shared variables and sharable instances. In both cases, most Geant4 kernel developers have to be involved into the Geant4 multithreading work to figure out a proper set of test drivers. This is not merely an engineering issue. It is generally an open problem for large parallel programs.

The test for the Geant4MT kernel is built on (i) a large application, (ii) several small applications and (iii) DRC. The large application adopted is FullCMS, which is a simplified version of the actual code used by the CMS experiment at CERN. This application covers almost all geometric classes and physics process classes whose physics tables are shared among threads. There are several small applications multithreaded as Geant4MT examples: N02, N03, A01, B01 and test1 for scorers. They are also employed as test cases. Avoiding test coverage issues, Geant4MT relies on bisimulation and DRC to figure out root causes whenever a sequential or parallel bug is triggered by any multithreaded Geant4 application. This is a practical solution for a software package as large as Geant4.

4. Tools for recovering Geant4MT Performance

The method described in previous sections produced an intermediate Geant4MT that was found not to be scalable. Two important performance drains were identified: (i) memory allocations/deallocations; and (ii) writes to shared variables.

4.1. Elimination the Bottleneck of the Central Heap

Firstly, none of the standard memory allocators scale well when used in the intermediate Geant4MT. Several allocators that were tried include the glibc default malloc (ptmalloc2), tcmalloc [6], ptmalloc3 [7] and hoard [8]. 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 (the arbitrating Linux system call that is activated when two locks collide) in the intermediate Geant4MT provided the final evidence of the importance of a thread-private custom memory allocator. The excessive number of futex calls does not disappear completely until a thread-private malloc library (TPMalloc) is introduced.

TPMalloc can be achieved by applying TTS to any existing malloc library. Geant4MT modifies the original malloc library from glibc to create thread-private malloc arenas. At the beginning of the thread function, each worker switches on the thread-private malloc arena and pre-initializes a memory region with a large top chunk in its private heap. Correspondingly, the private malloc arena is switched off at the end of the thread function. This is consistent with the event-level parallelism by which events and their simulation are completely independent and allows for a more efficient reduction of results generated in each thread.

4.2. Eliminating the Bottleneck of Shared-Updates

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 an excessive number of 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 multiple CPU chips on the motherboard with no off-chip cache, in the high performance machines on which the intermediate Geant4MT is tested. As a result 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 Geant4MT 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 the above working hypothesis was confirmed. The solution was then quite simple: eliminate unnecessary sharing of writable variables.

The performance bottleneck from updates to shared variables is subtle enough to make testing based on the actual source code impractical for Geant4, a million-line software package. DRC is able to recognize this bottleneck in an efficient and accurate way. Whenever updates for shared variables occur intensively, DRC 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 this shared variable. These simultaneous shared-memory updates create a classic situation of ``ping pong'' for the cache line containing that variable. Hence, they result in non-scaling 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.

This performance bottleneck elimination leads to an enhancement special for thread parallelism, i.e., the mechanism to demangle the standard output and error streams of Geant4MT per thread. This output privatization is implemented as a child class of G4coutDestination for each thread to redirect its output to a thread-private file. For the G4coutDestination child class, there is one instance per thread, which is associated to the corresponding private G4coutbuf/G4cerrbuf instance.

The Geant4MT work reveals the primary reason that the standard memory allocators suffered degraded performance was likely not the issue of excessive futex calls. Instead, it was due to writes to shared variables of the allocator implementation. A back-of-the-envelope calculation indicated that there were not enough futex calls to account for the excessive performance drain.

Comparing four widely used memory allocators with TPMalloc against a malloc/free intensive toy program, a parallel slow-down is observed from each of the four memory allocators: ptmalloc2, tcmalloc, ptmalloc3 and hoard. When increasing the number of threads from 8 to ~16 on a 16-core computer, the execution was found to become slower.

5. Geant4MT Engineering

Geant4MT starts from TTS to make all global and static variables thread local/private following a purely syntactic way. For those programs whose memory footprint is not a concern, TTS is already sufficient for thread parallelization. Both TTS and TMR have a potential to introduce unintended shared-updates. While TMR might get false positives concerning relatively read-only data recognition, TTS might invalidate constant variables that are generally not privatized.

Consider the original Geant4 kernel code as an example. A class may have only one instance if this class is defined as a singleton. While the sequential program regards the address of a singleton instance as a constant, the multithreaded program should no longer consider this pointer as a constant. Since each thread has a replicated instance for this singleton class, the pointer changes during task processing. Under such circumstances, DRC serves as a guard to eliminate unintended yet shared-updates, which are parallel bugs or potential performance bottlenecks.

TPMalloc is not trying to be a smart central heap implementation because a special optimization either introduces too much overhead or fails to cope with sufficient application scenarios. Instead, TPMalloc eliminates the central heap limit and makes the heap partitioned for threads. TPMalloc is still compatible with a central heap implementation when thread-private heaps are not used. Moreover, application developers are recommended to figure out what part of dynamic data has to be shared by threads, and to switch to thread-private heaps when the execution of threads passes through allocation/deallocation for this part of the data.

Using TPMalloc, the heap segment is not totally shared by threads. The heap partitioning is not a sufficient condition to guarantee scalability. To eliminate the central heap performance degradation, it is important to enforce the following strategy: each memory chunk must be deleted by the same thread that created it. This memory chunk privatization strategy allows each thread to bypass the shared central heap partition, i.e., to allocate and deallocate memory chunks in the thread-private heap

partition. Otherwise, it is hard to scale to 20 threads, as demonstrated by several previous studies that adhere to the standard malloc. These smart malloc implementations alleviate the central heap performance degradation for 20 threads or less. But, performance degradation still exists.

Correctness is as important as performance for parallelizing large programs such as Geant4. In other words, a parallelization following a radical method such as domain decomposition or vectorization has to be verified carefully. Apparently, the task-oriented parallelization facilitates the correctness verification that would otherwise be hard. Furthermore, Geant4MT addresses a bit-compatible parallelization based on the task-oriented parallelism, which allows programmers to verify the correctness for the parallelized code via the bisimulation tool.

5.1. Generation of a new Geant4MT Release

Geant4MT represents a development effort of two and a half years. The corresponding methodology compresses this two and a half years of work into a matter of several days for a new version of Geant4, as delineated below.

TMR is maintained as a patch. If there is no further requirement to reduce the memory footprint, then sharable classes and their corresponding relatively read-only fields will not be changed. Neither is the TMR patch.

TTS is almost automatic except for revisions and idiosyncrasies introduced in the new version of Geant4. To cope with these changes, TTS has to be modified to handle them. However, those may disappear in the future since Geant4 kernel developers always polish their code following the same Geant4 coding style, or they change their code to facilitate multithreading. So, we can "temporarily" revise the idiosyncrasies and maintain the change as a patch combined with the TMR patch.

Bisimulation is necessary only if there is a big change in the Geant4 kernel concerning the initialization for the detector and the underlying physics. In most cases, the initialization is stable and there is no change among several continuous Geant4 versions. Even though the new Geant4 release may include some minor changes to the initialization phase, the resulted change to the initialization phase for Geant4MT thread workers can still be very simple so that no bugs are introduced.

TPMalloc can be preloaded by Geant4MT applications. In addition to the standard malloc APIs, two more functions are implemented in TPMalloc: one for switching on the thread-private heap; another for switching it off. For these two functions, a stub library is implemented and linked to each Geant4MT application. If TPMalloc is not preloaded, then the standard malloc APIs are used. Otherwise, they are replaced with the TPMalloc implementation, which allows each thread to switch on/off its private heap dynamically.

Just as our experience on generating a multithreaded version for each recent Geant4 release is showing, the time spent on bisimulation, TPMalloc, the parallelization framework coding and DRC is very small. In most cases, they are just done once then reused. Since TTS does not consume much manpower, the multithreading cost for a new Geant4 release is primarily the TMR patch maintenance, followed by testing.

With a perfect TMR/TTS patch, the code transformation is directed by a shell script. The primary commands in this script are (i) patch the Geant4 source; (ii) build Geant4 to collect global and static variables; (iii) privatize those collected variables; (iv) build once again to collect warning messages; and (v) remove each variable declaration that is complained by the compiler such that the variable is not used. In practice, this is a three-day work consisting of one day for the patch maintenance, several hours (overnight) of code transformation and two days for testing the transformed code.

5.2. Geant4MT on NUMA

Similarly to the Geant4MT usage on computer clusters, the use of Geant4MT is also applicable to Non-Uniform Memory Access (NUMA) architectures. The NUMA model separates local memory from non-local memory for each processor. Accordingly, the NUMA-based parallelization is expected to take advantage of this asymmetry to aggregate both CPU cores and memory bandwidth. Following a simple way to port Geant4MT to NUMA, programmers just consider a NUMA computer as a cluster with a high bandwidth for inter-node communication. Meanwhile, they regard the CPU cores that share the same local memory system as a compute node.

A more sophisticated method is to exploit Geant4MT and optimize the corresponding thread parallelism for the NUMA asymmetry. More precisely, all threads are grouped into several subsets, each of them is associated to a fixed subset of CPU cores corresponding to the same local memory system. Threads in the same group take advantage of TMR to share part of application-level data while threads in different groups are completely independent. Moreover, inter-group communication is implemented via the same TOP-C callback mechanism yet in a more efficient manner based on the NUMA cache coherence protocol.

6. Geant4MT Performance Scalability

In this section we study the performance of Geant4MT, comparing it throughput on a many-core machine with that of the single-thread version, and the performance of the single-thread with a sequential application.

6.1. Scaling multiple-worker versus one-worker

Early measurements of the scalability of Geant4MT [9] showed good scaling between 1 and 24 nodes, even on hardware that had significant architectural performance limitations. We report here the extension of the benchmarking to newer systems with up to 40 cores.

The performance of Geant4MT was tested on a multi-socket server. The test system is a QSSC-S4R server on a Quanta platform. It has four LGA-1567 sockets, which can be occupied by up to four Xeon 7500/E7-4000 series processors, and two Boxboro-EX IOH chipsets. The memory configuration consist of 32 dual inline memory modules (DIMMs) of 4 GB each (128 GB in total), with 8 memory boards. The system was running 64-bit Scientific Linux CERN 5.6 (SLC5), which is based on Red Hat Enterprise Linux 5 (Server). The default SLC5 Linux kernel (version 2.6.18-238.9.1.el5) was used for all the measurements.

The four sockets of the platform contained four Westmere-EX class E7-4870 processors, each operating at 2.4 GHz, with SMT turned on. Each CPU therefore provided 10 cores (and 20 threads), bringing the platform total to 40 cores (80 threads).

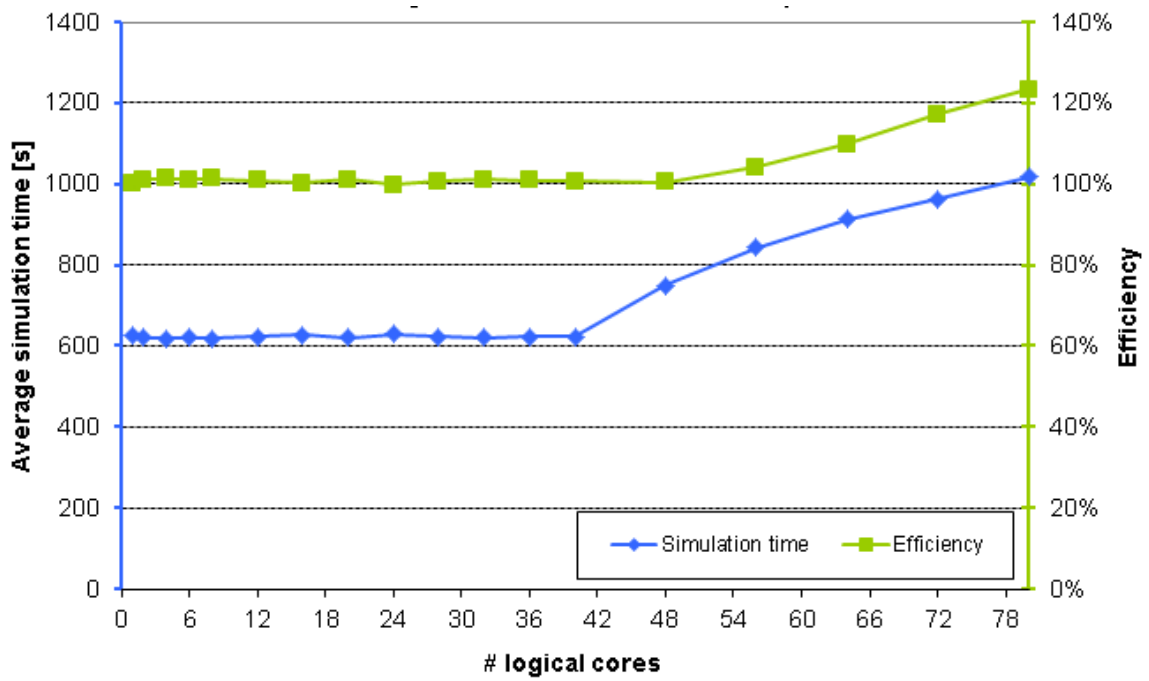

Figure 2: CPU time per worker thread for the ParFullCMS Geant4MT application running on a Westmere-EX 40-core system. The workload per logical core (thread) used was scaled with the number of cores: 100 events, each composed of one 300 GeV pion, were simulated per hardware thread.

The application used for benchmarking was ParFullCMSmt, in which the detector description is a snapshot of the CMS detector from 2008. The threads were pinned to the cores running them. The throughput measurement used was the average time per worker thread that it takes to process one 300 GeV pi- event in this predefined geometry. The hardware threading capability was activated and successfully used during the tests. After there were no more physical cores available, the jobs would be pinned to hardware threads, still maximizing the amount of physical cores used for maximum performance. We note also that the pinning was chosen so as to minimize the amount of sockets engaged. The tested framework was based on Geant4 4.9.2p01, CLHEP 2.0.4.2 and XercesC 2.8.0, and was compiled using the GCC 4.3.4 compiler.

As shown on Figure 2, the application tested very well up to 40 physical cores. The event loop efficiency under full physical core load was 100%, which corresponds to a perfect scaling factor of 40x compared to the single worker version. The efficiency curve grows past 40 cores to surpass 100%, since for thread counts higher than 40, expected scalability is fixed to 40x. Thus a final value of 123%, also shown on Figure 2, indicates that the system loaded with 80 threads of the benchmark yields 23% more throughput than a perfectly scaled serial version on 40 physical cores.

6.2. Overhead of single worker-thread version

In Geant4MT a slow-down of 30% is observed in benchmark applications with a single worker thread [11] when compared to the baseline sequential Geant4 application. The reason for this sequential slow-down is analyzed in this section.

One potential cause is the extra indirections Geant4MT introduces for each access to thread-private data members in classes which have both shared and thread-private parts.

Another potential cause is tied to mechanisms used to implement thread local storage. C++ compilers handle __thread by offsetting the address of each thread-local variable with a thread base-address [12] that is determined at runtime when the thread is created. Consequently, it is hard for compilers to support dynamic initialization/creation for thread-local variables. In other words, dynamic initialization/creation is supported only for non-thread-local variables. As a result, __thread is allowed only for types which contain only data, i.e. plain old data (POD) types. In addition any thread-local variable must be initialized with a static value that is evaluated during compilation.

For a global or static variable that is not POD or that is initialized by a dynamic value, its privatization introduces a corresponding pointer. This leads to a sequential slow-down when referring to this variable since its pointer must be fetched first. TMR has a similar performance issue. Nevertheless, the TTS/TMR slow-down is at most two times due to the following reason: firstly, the initialization code is executed only once; secondly, each privatized variable reference requires at most one more memory access to the pointer.

The TLS model for its run-time handling is also a potential source for a significant sequential slow-down. According to [12], a thread-local variable is defined by the module ID and the offset in the TLS block of that module, which is accessed via one of four TLS models: global-dynamic, local-dynamic, initial-exec or local-exec.

Following the global-dynamic model, the values for the module ID and the TLS block offset are determined by the dynamic linker at run-time and then passed to the __tls_get_addr function in an architecture-specific way, which leads to a dominant performance degradation.

The local-dynamic TLS model is an optimization for the general dynamic TLS model. If the compiler can recognize that the thread-local variable is defined in the same object it is referenced in, it is possible to generate code to reuse the module address and bypass the __tls_get_addr function. More precisely, the variable is accessed via the offset directly. Unfortunately, this optimization is not applicable to accessing thread-local variables in other modules.

An even better model is initial-exec, which directs compilers to generate code without using the __tls_get_addr function. Instead, the variable is accessed via the offset in the TLS block and one more offset computed using the architecture-specific formulas for the corresponding module. Similar to the global-dynamic case, an improved alternative, the local-exec model [13], is applicable to accessing thread-local variables in the executable itself.

The standard Geant4 compilation produces position independent code for use in dynamic libraries. This leads to the use of the global-dynamic model for thread local data.

We found that switching to the improved init-exec model reduced the overhead to 18%, compared to the sequential Geant4.

This slow-down remains an issue for Geant4MT. Even though porting applications to Geant4MT has been shown to be simple for stand-alone Geant4 applications, wider adoption will be limited by this slow-down. We will be seeking to reduce further, if possible to a negligible value.

6.3. Memory saving from sharing data

The savings in memory obtained by sharing the geometry description and physics process tables are substantial. Measurements of memory consumption on Linux systems are never straightforward. In particular vritual memory may contain parts that are not essential to the runtime, so the focus was on private memory consumption.

Our measurements of memory use were carried out using a Geant4 example application 'fullCMS', which uses a description of the CMS detector dating from 2008. The application reads the geometry model from a GDML file that contains the full geometry of the CMS detector. The single thread application footprint was 250MB. Each worker thread of Geant4MT required an additional 20MB (approximately). This enabled a multi-threaded fullCMS application with 24 workers to consume only 730 MB – an over 85% memory saving in this setup, which would scale with the number of workers.

This compares with an additional 70MB required by an additional process in Geant4MP, the separate multi-process parallelization of Geant4. In this setup, additional workers are created using fork, sharing objects that are not written (via the operating system's copy-on-write capability.) A significant part of this overhead is due to classes, which mix data which is read-only during simulation, with data which is read-write. The tables contain a large array of data, which is initialized before the event loop, and is read-only after that. A small set of data members are used to cache the last location and value, and are read-write. Improvements in the memory layout of these physics tables of Geant4 processes have been carried to separate the memory areas of the read-write data. We are now seeking to measure the revised overhead in the multi-process case in a realistic application after this improvement.

The thread private solution proposed is novel in the sense that it brings data sharing and protection to a single data member level. This gives a guarantee of the maximum reasonable achievable memory savings while preserving correctness, consistency and (to a large extent) performance.

Secondly, such significant memory savings give hope to seeing floating-point heavy simulation workloads running on memory-limited future accelerators, such as the Intel MIC.

7. Conclusions

Geant4MT adapts Geant4 to scale linearly on multi-core architectures. It was demonstrated to scale from a single worker to 40 or 80 worker threads on a 40-core computer. In the transition from sequential Geant4 to a single worker thread and one core, Geant4MT was found suffer an overhead initially of 30%. The underlying reasons for this overhead were described, and an improvement was found to reduce it to 18%.
The thread parallelization of Geant4 is accomplished by a four-stage methodology: (i) creation of a thread-private, shared-nothing version; (ii) reduction of the memory footprint by sharing large data structures; (iii) assuring correctness of the multi-threaded code; and (iv) performance measurements and tuning to identify and reduce overhead due to thread synchronization.

8. Acknowledgements

The work of the X. Dong and G. Cooperman was partially supported by the National Science Foundation under Grant CCF 0916133.

9. References

[1] G. Cooperman, L. Anchordoqui, V. Grinberg, T. McCauley, S. Reucroft, and J. Swain, "Scalable Parallel Implementation of Geant4 Using Commodity Hardware and Task Oriented Parallel C," arXiv: hep-ph/0001144, Jan. 2000.

[2] V. H. N. Gene Cooperman, "Parallelization of Geant4 Using TOP-C and Marshalgen," in Fifth IEEE International Symposium on Network Computing and Applications (NCA'06), pp. 48-55.

[3] X. Dong. Multi-threaded Geant4 with Shared Detector. Cern Openlab Technical Note 2008-03. 2008.

[4] A. M. Visan, A. Polyakov, P. S. Solanki, K. Arya, T. Denniston, and G. Cooperman, "Temporal Debugging using URDB," Operating Systems; Software Engineering, Oct. 2009, also available as arXiv:0910.5046v1.

[5] J. Ansel, K Arya, and G. Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In: Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS '2009), pp. 1-12., 2009.

[6] Google Performance Tool. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[7] Wolfram Gloger's Malloc Homepage. http://www.malloc.de/en/.

[8] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. SIGPLAN Not. vol. 35(11), pp. 117–128., 2000.

[9] X. Dong, G. Cooperman, J. Apostolakis, in Euro-Par 2010 - Parallel Processing, vol. 6272, edited by P. D.Ambra, M. Guarracino, and D. Talia, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 287-303.

[10] Intel Many Integrated Core (MIC) Architecture. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[11] Philippe Canal. Geant4MT Performance, *private communication*.

[12] U. Drepper. Elf Handling for Thread-Local Storage. http://www.akkadia.org/drepper/tls.pdf.

[13] A. Oliva and G. Araujo, "Speeding Up Thread-Local Storage Access in Dynamic Libraries", in GCC Developers' Summit 2006, 2006, pp. 159-178.

[14] G. Cooperman, TOP-C: A Task-Oriented Parallel C Interface: in 5th International Symposium on High Performance Distributed Computing (HPDC-5), IEEE Press, 1996, pp. 141-150.

[15] J. Schümann, H. Paganetti, J. Shin, B. Faddegon, and J. Perl, Efficient voxel navigation for proton therapy dose calculation in TOPAS and Geant4: in Phys Med Biol. 2012 Jun 7;57(11):3281-93. Epub 2012 May 9.