Parallelizing ATLAS Reconstruction and Simulation: Issues and Optimization Solutions for Scaling on Multi- and Many-CPU Platforms

C Leggett\(^1\), S Binet\(^2\), K Jackson\(^1\), D Levinthal\(^3\), M Tatarkhanov\(^1\), Y Yao\(^1\)

Abstract. Thermal limitations have forced CPU manufacturers to shift from simply increasing clock speeds to improve processor performance, to producing chip designs with multi- and many-core architectures. Further the cores themselves can run multiple threads as a zero overhead context switch allowing low level resource sharing (Intel Hyperthreading). To maximize bandwidth and minimize memory latency, memory access has become non uniform (NUMA). As manufacturers add more cores to each chip, a careful understanding of the underlying architecture is required in order to fully utilize the available resources.

We present AthenaMP and the Atlas event loop manager, the driver of the simulation and reconstruction engines, which have been rewritten to make use of multiple cores, by means of event based parallelism, and final stage I/O synchronization. However, initial studies on 8 and 16 core Intel architectures have shown marked non-linearities as parallel process counts increase, with as much as 30% reductions in event throughput in some scenarios. Since the Intel Nehalem architecture (both Gainestown and Westmere) will be the most common choice for the next round of hardware procurements, an understanding of these scaling issues is essential. Using hardware based event counters and Intel's Performance Tuning Utility, we have studied the performance bottlenecks at the hardware level, and discovered optimization schemes to maximize processor throughput. We have also produced optimization mechanisms, common to all large experiments, that address the extreme nature of today's HEP code, which due to it's size, places huge burdens on the memory infrastructure of today's processors.

1. Introduction
The days of easy performance gains from ever faster CPU frequencies are over. Heat dissipation due to high transistor density has become a serious issue, and without expensive active cooling, or other tricks such as Intel's TurboBoost, commodity CPU frequencies won't rise much above 3 GHz. Instead, CPU manufacturers are using this ever increasing transistor density to put multiple cores on each CPU as simply increasing clock speed is unfeasible. Furthermore, multi-threading techniques are being implemented at the hardware level to make each physical core look like multiple virtual cores to the operating system. The current generation of commodity processors have up to 6 cores per CPU, and 4 CPUs per box. Before long we will start seeing 32 and 64 cores in a desktop enclosure. RAM prices have not kept pace with the increasing density of CPUs, and while the ratio of RAM/CPU on motherboards has increased in recent years, the RAM/core ratio has dropped. Since ATLAS

\(^1\) Lawrence Berkeley National Lab
\(^2\) LAL
\(^3\) Intel

Published under licence by IOP Publishing Ltd
reconstruction software consumes approximately 1.5 GB of memory, providing sufficient physical RAM to naively distribute one job to each core will become prohibitively expensive. A more efficient mechanism to make use of all the cores must be found.

In this paper we will present the measures that have been taken to parallelize ATLAS reconstruction, and its scalability into the many-core world, as well as more general issues that were discovered that are endemic to all large object oriented code bases, and possible solutions to them.

2. Levels of Parallelism

There are several different levels of parallelism that can be implemented. The first is job based, where multiple independent jobs are run in parallel. Though this is the easiest to implement, and requires no rewriting of code, it offers the least amount of data sharing, and can quickly saturate the RAM if the processes are large, as is the case with ATLAS code. The name for this procedure in ATLAS is AthenaMJ, Athena being the name of the standard single process reconstruction code.

The next step is event based parallelism, where a mother process performs the configuration and initialization of the job, then farms out individual events to the worker nodes. At the end of the job, the mother process reestablishes control, and combines the outputs. This scheme, called AthenaMP, requires some changes to the framework, but for the most part is transparent to users. We can further improve on this design by parallelizing the I/O, which reduces the overhead at the end of the job due to the merging of the outputs from each worker.

In sub-event parallelism, we divide the detector into independent regions of interest, such as the calorimeter or the muon chambers, and assign individual worker nodes to process the data for each region. This requires considerably more code modification, both at the framework and the user level.

Finally, we have true algorithmic parallelism, where techniques such as loop unrolling, vectorization, and use of SIMD instructions are implemented. This is the hardest to do, and requires major code modification at the user level, as well as detailed understanding of the parallel capabilities of the compiler and underlying hardware. We are very leery of taking this step, as given the large ATLAS code base, it will require a huge effort, though it may be possible for a limited number of critical algorithms.

This paper will focus on AthenaMP, as it offers the greatest gains for the least amount of impact on the user code base.

3. Event Based Parallelism

3.1. AthenaMP

Fig. 1 shows how AthenaMP is configured. The mother process performs the configuration and initialization of the job, then processes the first event before forking off multiple workers to handle the subsequent events. The first event is very important, as much static configuration information is read in during the first event, such as geometry, databases, and detector description. By forking the workers after this, we maximise the amount of memory shared between workers, due to the OS's shared memory policy of copy on write. As long as a block of memory, with the granularity of a memory page, remains unchanged, it can be shared between multiple processes.
Forking off workers after the first event and sharing memory between processes saves on average 500 MB of memory per process. Since a single processor job consumes 1.5 GB of memory, this allows us to run a 16 process job (a typical reconstruction node), with only 16 GB of physical RAM. This is demonstrated in Fig. 2, where we show the memory footprint per process vs. the number of processes on a node. The reason the first data point is higher than that of a standard, single process run, is that a 1 process AthenaMP job really has 2 processes – the mother and the worker, which results in some extra overhead.

Comparing the total event throughput for AthenaMJ vs AthenaMP (see Fig. 3), fairly good tracking can be seen until the physical memory limit of the machine is reached, and the AthenaMJ job starts swapping to disk. For scenarios that are not memory limited, AthenaMP will always be a little slower than AthenaMJ, as there is extra overhead due to the mother process, as well as the merging of the output files at the end of the job. This difference will increase with the number of processes, as the
final merging becomes more time consuming. This can be reduced by implementing parallel I/O techniques.

Event queues have also been implemented to speed up multiprocess jobs. Since not all events take the same amount of time to process, allocating a fixed number of events to each worker at the start of the job will result in the non-optimal situation of the mother process waiting at the end of the job for the slowest worker to finish. If instead each worker dynamically requests a new event from the queue on an as-needed basis, a more load balanced schedule is achieved, resulting in increased overall performance. Currently, the event queue only distributes run and event numbers—the child processes still implement independent data converters to read the events from disk. Studies are underway to implement a separate process that distributes entire events on request to the workers, reducing their overhead of multiple copies of the data converters.

3.2. Hyperthreading and CPU affinity

With the release of the Nehalem CPU, Intel has reintroduced its form of symmetric multi-threading into its processor lineup, calling it hyperthreading. By duplicating certain parts of the processor, but not the majority of it, multiple instructions from different sources can be interleaved in the instruction pipeline, and this is presented to the OS as two virtual cores for each physical core. As long as a resource is not saturated, this is a cheap way of doubling the number of processors in each box. In Fig. 4 the results of running AthenaMP with hyperthreading enabled on an 8 core node can be seen. As long as there are more processes than cores, and the physical memory available on the system is not exhausted, it is worthwhile to leave hyperthreading enabled.

![Figure 4: Event Throughput with Hyperthreading](image1)

![Figure 5: Event Throughput with Affinity Pinning](image2)

The linux process scheduler available in the 2.6.32 (and earlier) kernel is not very smart. It will migrate processes from one core to another during the execution of a single job. Furthermore, with hyperthreading enabled, it cannot distinguish between virtual cores on the same physical core, and on different cores, and will stack multiple processes on the same core while leaving other physical cores empty. This obviously causes serious performance degradation, as large amounts of data need to be copied, especially if the job is moved across physical CPU boundaries. This is why differences in the event throughput rates can be seen in Fig. 4. In Fig. 5 the results of pinning the jobs (again on an 8 core node) to the most optimal cores can be seen. By distributing processes equally across all cores, and pinning them there for the lifetime of the job, real world gains of up to 20% can be achieved (the above plots do not include the effect of data merging during finalization).
4. Optimizing Large OO Code Bases

There are a large number of tools available on the linux platform to study performance and resource usage, such as sar (for I/O to disk and system loads), vmstat (for memory performance), IPM (time spent in I/O vs computation), and numastat and numactl (reporting and controlling NUMA memory settings). The most detailed and useful tool, however, is the Intel Performance Tuning Utility or PTU. It uses a linux kernel module to provide a sampling profiler, which captures information from the hardware counters that are available on Intel chips. There are literally thousands of different counters that can be profiled, with different information available on various processor families. PTU is by far the most accurate and detailed tool for understanding what's occurring at the hardware level.

Due to the sheer number of counters available however, and the difficulty in understanding exactly what they mean, a hardware and microcode expert is really needed to determine exactly what to profile. Assembling a profile schema is not a task for the average user, however, once such a profile has been created, non experts can easily make use of the information it provides. PTU was used to profile a typical ATLAS reconstruction run, and the distribution of memory accesses was analyzed. In Fig. 6 we see the results. The Nehalem chip has 3 caches: a very fast 32 KB L1 cache attached to each core, a slightly slower 256 KB L2 cache, also attached to each core, and a 8 MB L3 cache that is shared between all cores on a CPU. The address of each memory access was recorded for the run, and accumulated in 32 KB bins. These address access count bins were sorted by increasing access frequency, and then plotted on a log-log scale (left axis). The S-shaped curve, labeled “Percent Usage” (right axis) is the running, integrated sum of the memory access data points. From this curve, we can

![Graph showing memory usage distribution.](image-url)
see that 6.5% of reconstruction code is capable of residing in the L1 cache. When a L1 cache miss occurs (the data/instruction being sought by the processor is not found therein), a 6 cycle penalty is incurred, and the L2 cache is checked. 30% of the code base can fit inside L2, but an L2 miss will engender a 38 cycle penalty. Since the L3 cache on the Nehalem chip is 8Mb, about 95% of the ATLAS reconstruction code can, in theory, fit inside it. However, since this cache is shared between all cores on the CPU, if multiple processes are running, it is unlikely that maximal memory sharing will occur between all processes, ie they will not all have the same information stored in the L3 cache. We have profiled runs that show approximately 75% of the code resident in L3, allocating an average 2MB of L3 to each core. If an L3 miss occurs, and the processor must go to DRAM to find an instruction, there is a 200 to 360 cycle penalty, depending on whether the RAM is physically associated with the CPU, or if the QPI (Quick Path Interconnect) must be used to fetch data from RAM attached to another CPU in a multi-cpu configuration.

The conclusion here is that anywhere between 5% and 25% of ATLAS reconstruction instructions will result in L3 misses, resulting in large penalties. While ATLAS has a larger memory footprint than most, it is not the only experiment at LHC to have this issue.

4.1. Optimization Pitfalls

In days of yore, inlining often executed code was the optimization advice of choice, but that is no longer necessarily the case. Inlining increases the size of the binary, and can make ifetch misses more costly, if the processor has to fetch data from outside the cache. So, even if there are fewer misses, having to go to a few times to DRAM instead of many times L2 or L3 to satisfy a request can result in significantly slower code.

Large code built out of many small methods, which is typical for our Object Oriented codes, can result in totally flat cycle profiles. It can take thousands of functions to account for 80% of the clock cycles sampled, so optimizing just a few “hot” functions is no longer sufficient. In order to have a noticable effect, thousands of functions would have to be optimized.

Furthermore, the very act of calling a function results in added instructions. Each time a function is called, a penalty is incurred for a call and return. In order to use position independent code in shared libraries, the preferred solution for building large OO code bases, trampolines are required in order to fully resolve symbols, adding even more instructions. Local registers must be freed and restored at the start and end of every function call, and setting and reading function arguments adds yet more overhead. Using virtual functions and function pointers, as is the recommended OO style for implementing abstract interfaces, causes binaries to suffer from even greater instruction overhead penalties. Since there are so many functions called, the naïve approach of inlining them all, which might otherwise seem to be the solution, will result in a massive increase in binary size with resultant L3 misses, which will in fact greatly decrease performance. Only targeted inlining would be effective in this scenario.

Running ATLAS reconstruction and monitoring the execution with PTU, we observed that the average function size is just 32 instructions. The overhead for merely calling a function can be as large as 12 instructions if the called function resides in a different shared library. ATLAS reconstruction code has over 2500 shared libraries, resulting in constant access across library boundaries. While having many independent shared libraries makes development easier, scattering often executed functions across multiple libraries results in significant performance penalties, in this case resulting in huge overheads where on average, 30% of a function's cycles are devoted to the mechanics of accessing across shared library boundaries.
Given the large size of ATLAS code, it is not unusual for cache misses at all levels to occur. PTU has also shown us that ATLAS reconstruction suffers from massive instruction starvation, where 20% of all cycles are wasted waiting for instructions to be fetched.

ATLAS is not unique in suffering from these issues. In fact, classic OOP code normally has a large number of small functions integrated together and invoked by algorithms. This produces large inefficiencies with the current (and near future) generation of processors from Intel and AMD. PTU can be used to identify signatures for these inefficiencies, and our investigations have shown that some of the most useful ones are:

- low ratio of \( \text{instruction\_retired} / \text{call\_retired} \)
- high \( \text{call\_retired} / \text{branch\_retired} \)
- high \( \text{indirect\_call} / \text{call\_retired} \)
- high \( \text{uops\_issued.core\_stall\_cycles} - \text{resource\_stalls.any} \)
- high \( \sum \text{latency(source)} \times \text{ifetch\_miss(source)} \)

All of these inefficiencies are present in ATLAS code, as well as other LHC code bases. ATLAS tends to suffer more, however, due to its large memory footprint.

4.2. Optimization Strategies

We can begin to address these issues by first identifying the very hottest functions, and inlining them. Then, by performing a social networking analysis on running code to determine the function call graph hierarchy and frequency, and then associating costs to the calls based on location and parameters (eg number and type of function arguments, return type, virtualness, etc), we can identify clusters of active and costly calls. These clusters can be ordered by their total costs, and functions counts in the hot clusters can be reduced by explicitly inlining. This must be done judiciously, as if too many functions are inlined, then the binary size will grow beyond what can be cached, and performance will suffer. Obviously this will require some changes to the code base.

The next optimization approach involves class organization. Currently, shared objects are grouped into libraries by “domain”, eg tracking, I/O, calorimeter, core routines, etc. This results in many functions being called across shared library boundaries. If these objects are reordered in libraries such that hot functions are co-located in the same library, the overhead due to cross library calls can be significantly reduced. This can be done at link time, after a network analysis has been performed. Further optimization can be accomplished using features in newer compilers that do cross object optimization.

This style of runtime analysis needs to be performed on standard job profiles, such as common reconstruction or simulation jobs. If the results of the analysis on the different job profiles suggest contradictory optimizations, then either separate builds must be made for each style, or hard choices must be made as to which to optimize at the expense of the other.

Finally, the newest gcc standard implements a new attribute named `final`, which, when applied to a class, prevents subtyping, and when applied to a virtual member function, prevents subtypes from overriding that function. The appropriate use of this attribute will permit the compiler to generate more efficient code, reducing the overhead for virtual function calls. Other hints to the compiler such as `likely/unlikely` for branch prediction can also be of value.

---

https://openlab-mu-internal.web.cern.ch/openlab-mu-internal/00_News/News_pages/2010/10-15_Meeting_David_Levinthal/10-15_Meeting_David_Levinthal.htm
5. Conclusions

There are no quick fixes or magic bullets that will enable us to see major improvement in code execution speed. Both ATLAS, and the HEP community in general, as well as an increasing number of large commercial applications, suffer from problems related to their enormous complexity, and the style of programming that has been used, as well as the physical limitations of the hardware. There are, however, certain steps that can be taken to address the problems.

5.1. Short term solutions

For the short term, performing runtime network analysis on standard jobs, such as reconstruction, and restructuring the code to closely group or inline hot functions is the first step. We can also make use of new compiler features such as cross object optimization, and special attributes such as final or likely to improve the compiler's output.

5.2. Long term solutions

Since the HEP community is not facing these issues alone (Oracle, IBM, Google and many other commercial enterprises all have similar styles of code bases, and are just beginning to realize that there is a problem), we can expect to see more pressure placed on both the hardware providers, as well as the compiler developers, to find solutions. The main issue at this point is identifying exactly where the problem lies. New tools and analysis techniques are needed. Current tools either fail to show the exact source of the problems, or are not suited to large scale deployment - despite the incredible usefulness of PTU, it is not easy for the layman to interpret its results.

We need to drive these optimization techniques into the compilers and linkers themselves.

Changes at the hardware level, both for monitoring and for optimization, would also improve the situation. This is already happening - new counters have been introduced into Intel's Westmere and Sandy Bridge chipsets which will make profiling more useful. If we can point to specific inefficiencies in the hardware or the microcode, there is a very good chance that the manufacturers will implement solutions.

In many ways, we, the HEP community, are at the forefront of High Performance Computing technology, but the big companies are not far behind. While we do not purchase enough CPUs to have a serious say in the hardware manufacturers' direction, if we can show them exactly what's wrong, such as increasing cache sizes, or providing additional hardware counters to monitor certain events, and point out that their large customers are also beginning to suffer from the same issues, they will listen.