Event Reconstruction for Many-core Architectures using Java

Norman A. Graf
SLAC National Accelerator Laboratory, 2575 Sand Hill Road, Menlo Park, CA, 94025, USA
E-mail: Norman.Graf@slac.stanford.edu

Abstract. Although Moore’s Law remains technically valid, the performance enhancements in computing which traditionally resulted from increased CPU speeds ended years ago. Chip manufacturers have chosen to increase the number of core CPUs per chip instead of increasing clock speed. Unfortunately, these extra CPUs do not automatically result in improvements in simulation or reconstruction times. To take advantage of this extra computing power requires changing how software is written. Event reconstruction is globally serial, in the sense that raw data has to be unpacked first, channels have to be clustered to produce hits before those hits are identified as belonging to a track or shower, tracks have to be found and fit before they are vertexed, etc. However, many of the individual procedures along the reconstruction chain are intrinsically independent and are perfect candidates for optimization using multi-core architecture. Threading is perhaps the simplest approach to parallelizing a program and Java includes a powerful threading facility built into the language. We have developed a fast and flexible reconstruction package (org.lsim) written in Java that has been used for numerous physics and detector optimization studies. In this paper we present the results of our studies on optimizing the performance of this toolkit using multiple threads on many-core architectures.

1. Introduction
Moore’s Law predicting a periodic doubling of transistor density on integrated circuits has in the past resulted in a concomitant doubling in the speed of microprocessors, and therefore, a “free” reduction in the time required for programs to run. That is, the same code, or even executable program, could be made to run faster simply by buying newer computers. Moore’s law still holds, but, for a variety of reasons, the clock-speed of CPUs fell off the doubling curve several years ago. Therefore, one can no longer automatically get improved performance from newer CPUs. In fact, the current trend towards multi- or many-core processor architectures has seen a decrease in the clock speed of newer processors. There are no “silver bullet” solutions to adapt existing software to the new computing hardware: no compiler switch to optimize for many-core, and no libraries to simply link against to give concurrency. The need for increases in the speed of simulation or reconstruction programs, coupled with the much larger memory footprint of executables and events, therefore requires a paradigm shift in coding. In this paper we investigate the possibility of using a multi-threaded approach to event reconstruction to take advantage of the many-core architectures which are becoming more and more prevalent.
2. High Energy Physics Event Reconstruction

Currently, High Energy Physics (HEP) employs run and event-based parallelism to process events. Experimental conditions are usually fairly static per run (often by construction), so one can set up geometry, and query calibration databases once and access this information throughout the reconstruction job. The program then processes each event serially and independently. Parallelism is achieved by loading the program, run conditions and events into local memory on many individual CPUs, with one event being processed serially on one core or hardware thread. The memory footprint of HEP executables, detector conditions, and increasingly so for events, especially those for collider detector experiments such as ATLAS or CMS at the LHC, is, however, becoming significantly larger. The trend towards many CPUs sharing common memory means that having multiple, independent copies of programs per CPU does not scale. Here, we investigate whether parallelism within an event reconstruction holds promise for the future, that is, instead of devoting one CPU per event, we assign multiple CPUs to independent aspects of the reconstruction within a single event.

2.1. Event Reconstruction Parallelism

“The real performance payoff of dividing a program’s workload into tasks comes when there are a large number of independent, homogeneous tasks that can be processed concurrently.” [1] Modern collider detectors are enormously complex, but modular, and a number of reconstruction tasks can be easily identified as independent. For example, digitization, clustering, and centroid calculation of hits in silicon trackers is easily factorizable, either by subsystem (barrel vs endcap), by layer, or even by individual silicon wafer. Clustering of calorimeter cells is similarly separated by subsystem (e.g. barrel and endcap, or electromagnetic versus hadronic calorimeters), or by non-adjacent modules. This need not be limited to low-level detector elements. It can be extended to higher level reconstruction such as jet flavor-tagging, where vertex finding and fitting is done only on a set of tracks associated to the jet under consideration, or where soft-lepton association is only done on jet constituents on a jet-by-jet basis.

2.2. Event Reconstruction using org.lcsim

A fast and flexible event reconstruction and analysis framework has been developed for the International Linear Collider physics and detector response simulations. The org.lcsim [2] toolkit is written in Java and provides a software framework featuring plug and play reconstruction drivers. The components are fully modular and are available for tasks from digitization of tracking detector signals through to cluster finding, pattern recognition, track fitting, jet-finding and analysis. Runtime configuration and control is provided by an easily-edited xml file. It provides a perfect development environment to study the feasibility of a multi-threaded approach to event reconstruction by easily implementing and testing different multi-threaded event reconstruction strategies.

3. Concurrency in Java

Java provides both a very powerful, but simple-to-use, object-oriented language for development and transparent cross-platform portability for deployment. In addition, Java has included support for concurrency since its inception, and this support has improved over time. These features make it easy to develop, implement and study various approaches to concurrent event reconstruction. In contrast, the ISO C++ standard does not mention threads. Usual C++ coding solutions involve non-portable, platform-specific concurrency features and/or libraries. The boost package provides a possible common solution for HEP. Additionally, the C++0x draft offers threads, but it remains a draft, and compliant compilers are many years in the future. The motivation for this exercise is to study the concept of concurrent intra-event processing in
an environment currently supportive of this approach (Java), and apply successful solutions if, and when, needed and supported in C++.

3.1. Thread Class
The most basic, and oldest, threading method in Java is to extend the base class Thread.

```java
class ThreadTask extends Thread { public void run() { ... } }
```

However, this paradigm is extremely limited in that the run() method accepts no arguments, returns no values and cannot throw checked exceptions, and the user class cannot extend another base class, since Java does not allow multiple inheritance.

3.2. Runnable Interface
A better way to create a thread in Java is to implement the Runnable interface.

```java
public interface Runnable<V> { public void run(); }
```

```java
class RunnableTask implements Runnable extends UsefulBaseClass {
    public void run(){ ... }
}
```

which is called by passing it to a Thread:

```java
Runnable runnable = new RunnableTask();
Thread t = new Thread(runnable);
t.start();
t.join(); // blocks until task completes
```

To get a value back from the now-completed task, one must use a method outside the interface and wait for some kind of notification message that the task completed.

3.3. Callable Interface
The new Callable interface is much like Runnable but overcomes the two drawbacks with Runnable, namely its user method now returns a typed value and it can throw checked exceptions, namely,

```java
public interface Callable<V> { V call() throws Exception; }
```

```java
class CallableTask implements Runnable extends UsefulBaseClass {
    public Object call() { ... }
}
```

An ExecutorService, an asynchronous task handler which creates, manages, and runs thread pools, is used to execute Callable instances. Executor has three factory methods: `newSingleThreadExecutor()` which provides a single thread with an unbounded queue for tasks, `newFixedThreadPool(int nThreads)` with a specified maximum thread pool size and an unbounded task queue (if a thread dies, a new one will be created to replace it), and `newCachedThreadPool()` which provides an open-ended number of threads, which grows and shrinks on demand and which caches threads for short periods of time for re-use.

3.4. Thread-safe Collections
With multiple threads running, one needs to worry about concurrent access both for reading collections from and writing collections to the event. The collection classes in java.util which are commonly used are not currently synchronized by default. However, synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary Java collection.
For instance the following Collection

```java
List<Type> list = new ArrayList<Type>();
```

becomes simply

```java
List<Type> list = Collections.synchronizedList(new ArrayList<Type>());
```

Of course, this extra safety incurs a concommitant cost in speed, so it is an optimization process to balance the gain in speed resulting from extra processors against this overhead.

### 3.5. Code Migration

We present here a simple example of clustering calorimeter hits by serially processing the hits in various hit collections keyed by subdetector:

```java
// the map containing the calorimeter hits keyed on subdetector name
Map<String,List<CalHit>> hitmap=new HashMap<String,List<CalHit>>();
// a container to hold resulting clusters
List<Cluster> clusterList = new ArrayList<Cluster>();
// A Clusterer to cluster hits
Clusterer c = new Clusterer();
// Loop over available collections of hits, return a set of clusters for each
for (String s : keys) {
    List<Cluster> clusters = c.cluster(s, hitmap.get(s));
    clusterList.addAll(clusters); }
```

This code is then easily transformed into the following multi-threaded example, where for illustrative purposes we show how easy it is to set up a thread pool limited by the number of hardware threads currently available. One could, of course, use other considerations to set the pool size. This example creates one clustering thread per hit collection, each of which adds its clusters to a thread-safe collection of clusters. The ExecutorService returns a Future object for each thread, allowing one to not only check if the computation is complete, but also to wait for its completion, and to retrieve the result of the computation.

```java
// how many processors are available?
int nProcessors = Runtime.getRuntime().availableProcessors();
// create a fixed number of threads for processing
ExecutorService threadExecutor = Executors.newFixedThreadPool(nProcessors);
// the map containing the calorimeter hits keyed on subdetector name
Map<String, List<CalHit>> hitmap = new HashMap<String, List<CalHit>>();
// a thread-safe container to hold resulting clusters
List<Cluster> clus = Collections.synchronizedList(new ArrayList<Cluster>());
// a collection to hold the clustering tasks
Collection<Callable<List<Cluster>>> tasks = new LinkedList<Callable<List<Cluster>>>();
// create one task per subdetector
for (String s : keys) {
    tasks.add(new CallableClusterer(s, hitmap.get(s), clus));}
// process all tasks
List<Future<List<Cluster>>> futures = threadExecutor.invokeAll(tasks);
```
4. Analysis Process
There is clearly a very large phase space for optimization, and one needs to constantly balance
the granularity of the threaded tasks with the overhead of each thread, plus the cost of thread-safe data structures. For instance, is wafer-level threading realistic for a silicon tracker with ten thousand wafers? The modular architecture of the org.lcsim framework, coupled with the ease with which the existing serial code can be rewritten to target the multi-threaded support built into the Java language, make this exercise at least tractable. The use of a runtime control system based on xml to provide input to the various reconstruction drivers makes it easy to script and automate the systematic analysis of different reconstruction strategies. In the end, Amdahl’s law limits the maximum gain, since not all tasks lend themselves to concurrent processing. In order to conduct such optimizations in an efficient way, one needs robust tools to monitor threads, CPU and memory. Fortunately, Java provides such tools.

4.1. JConsole
JConsole is the Java Monitoring and Management Console which provides comprehensive
monitoring and management support of applications running on the Java platform. The GUI
provides convenient access to information about the performance and resource consumption of applications running either locally or remotely. An overview tab provides global information such as heap memory usage, the number of running threads, and total CPU usage. Most relevant to this study, however, is the MBeans pane which provides detailed information for all the running threads. MBeans are managed beans, i.e. Java objects that represent resources to be managed. JConsole provides a number of default MBeans with which one can easily monitor the state of all running threads, and detect if any threads are deadlocked. For each thread one can query the CPU consumption, stack trace, the monitor lock that the thread is currently blocked on, if any, and which thread is holding that lock, and thread contention statistics. JConsole is an extremely functional tool as-is, but one can extend the functionality by either implementing custom MBean classes, or coding to the JConcole plug-in API.

4.2. Testing on multi-core systems
The initial code development was done on a Dell Personal Computer running Windows7 on
an Intel Core i7 with hyperthreading enabled, giving 8 cores, with 12GB of RAM. The code
was then deployed to two dedicated test machines. The first featured dual Intel 'Westmere’
6-core CPUs. Intel’s hyperthreading feature was enabled which doubles the number of hardware threads from 12 to a total of 24, 48GB of RAM was available. The second machine featured
dual AMD 12-core CPUs, for a total of 24 hardware threads, with 64GB of available RAM. In Figure 1 we show the measured speedup factor for a CPU-intensive silicon detector signal
digitization process as a function of the number of hardware threads devoted to these tasks. We see the expected behavior, namely, a linear increase in the beginning, a fall-off due to Amdahl’s law, followed by saturation when the number of hardware threads has been reached.

5. Summary
HEP event reconstruction is inherently modular and lends itself well to a multi-threaded
approach. The modular approach to generic reconstruction intrinsic to the org.lcsim simulation
and analysis framework was easily modified to accommodate multi-threaded reconstruction.
Java’s built-in support for concurrent processing and tools to monitor results make coding and analysis very straightforward. The current work is presented as a proof-of-concept study of intra-event reconstruction parallelism. The process has just begun, we are still learning the intricacies of multi-threaded reconstruction in a many-core environment, and we are interested in collaborating with others. At present, our group is motivated by curiosity, not by need. Events produced at an electron-positron collider are small enough, and the existing Java code runs fast.
enough that current serial reconstruction was more than adequate for the International Linear Collider Letter of Intent Monte Carlo data challenge exercise, which involved the simulation, reconstruction and analysis of many tens of millions of events. Furthermore, job submission environments, e.g. lsf batch or Grid protocols, target individual processors, so do not (yet) benefit from multi-core CPUs. We anticipate, however, that the experience and lessons learned from studies of threaded event reconstruction in Java will be applicable to C++ reconstruction if and when it is needed and supported.

Acknowledgments
I would like to thank Tony Johnson for stimulating discussions and assistance during the course of this study. Thanks to Stuart Marshall and Yemi Adesanya at the Kavli Institute for Particle Astrophysics and Cosmology for granting access to their multi-core computers for testing. This work was supported in part by the US Department of Energy, Office of High Energy Physics.

References
[1] Goetz B 2006 Java Concurrency in Practice (Boston: Addison-Wesley Professional)
[2] http://lcsim.org See also Graf N org.lcsim: Event Reconstruction in Java these Proceedings.