Data processing in the wake of massive multicore processors

J Kowalkowski

Scientific Computing Division, Fermi National Accelerator Laboratory

E-mail: jbk@fnal.gov

Abstract. Developments in concurrency (massive multi-core, GPU, and architectures such as ARM) are changing the physics computing landscape. This paper will describe the use of GPU and massive multi-core, and the changes that result from massive parallelization and the impact on data processing.

Major HEP event-processing framework software runs within the changing computing environment. These frameworks have been evolving to accommodate the changes. The framework changes need to go quite a bit further, to better handle coprocessors with alternative architectures.

1. Introduction

The HEP software frameworks are evolving to accommodate changes in large-scale computing environments. The purpose of this paper is to explore the question of whether or not this evolution is going far enough given the current movement in computing towards massive multicore, better known as many-core systems. The focus will be how many-core computing affects our software systems. Currently the best examples of many-core are the NVIDIA GPGPU and the Intel MIC Xeon Phi coprocessors[1].

1.1. Reminders

The computing changes are real. They have been recognized everywhere for a while now, as evident from the R&D groups that have popped up within CERN, FNAL, and LBNL. The concurrency forum[2] and CERN OpenLab have also appeared. These groups are at least partly addressing the many-core coprocessor issue. The concurrency forumhas done an excellent job at tracking the major software infrastructure efforts and making current information available on the web.

The computing changes are fundamental. The focus is on large core counts: cores with reduced memories, reduced instructions, and much greater vector processing capabilities. The larger leadership class supercomputers show this movement, along with other important features such as multiple high-speed interconnects. Even the smaller footprint tablets have multicore ARM processors with integrated GPUs.

Our colleagues within the accelerator modeling, LQCD, and Computational Cosmology areas have embraced these newer archtectures. They have well-established relationships with the

[1] For the purposes of this paper accelerator or coprocessor can be used interchangably

[2] See http://concurrency.web.cern.ch/

Rank	Name	Total Cores	Processor	Mflops/Watt	Coprocessor
1	Tianhe-2 (MilkyWay-2)	3120000	Intel Xeon E5-2692 12C 2.2GHz	1901.54	Intel Xeon Phi 31S1P
2	Titan	560640	Opteron 6274 16C 2.200GHz	2142.77	NVIDIA K20x
3	Sequoia	1572864	Power BQC 16C 1.600GHz	2176.58	None
4	K computer	705024	SPARC64 VIIIfx 8C 2.000GHz	830.18	None
5	Mira	786432	Power BQC 16C 1.600GHz	2176.58	None
6	Stampede	462462	Xeon E5-2680 8C 2.700GHz	1145.92	Intel Xeon Phi SE10P
7	JUQUEEN	458752	Power BQC 16C 1.600GHz	2176.82	None
8	Vulcan	393216	Power BQC 16C 1.600GHz	2177.13	None
9	SuperMUC	147456	Xeon E5-2680 8C 2.700GHz	846.42	None
10	Tianhe-1A	186368	Xeon X5670 6C 2.930GHz	635.15	NVIDIA 2050

Figure 1. Excerpt from the Top500 Supercomputing Sites highlighting qualities of the top ten ranked based on the LINPACK benchmark

US HPC[3] communities. This adventure is largely just starting for many of the HEP experimental software infrastructure teams.

1.2. Architecture changes

1.2.1. Mainstream computing in HEP The current computing environment, with its heavyweight cores, can be simply described as a golden era. One of its end products are multi-gigabyte user applications. Take a look at, for example, Adobe acroread, Microsoft Office and Eclipse. Each of these serves a useful purpose, but one cannot help but be surprised at their enormous size and complexity of installation. HEP has not been immune to this.

Heavyweight cores do make life easy with seemingly endless memory and wonderful high-level languages. Now we experience mostly a single OS and a single architecture. Has this spoiled developers to some degree? The heavyweight microprocessors in use are beautiful in design and function: instruction scheduling, branch guessing, and sophisticated caching. Unfortunately the amount of work now required for the computational output has become too high given limited compute resources and an ever-increasing needs for processing [1]. As a result, the field has been ripe for alternative technology to squeeze in. The underlying machinery that has made life as a software developer so straightforward, essentially hiding much of the complexity and helping our productivity, has also created gluttons. Developers have become somewhat spoiled; it is amazingly easy to use up all the available resources. For a long time, the doubling of available hardware resources every two years was great and made this type of thinking easy. The focus on object-oriented C++ and needless abstraction layers did not help.

1.2.2. On the horizon What is now on the horizon for HEP is already out there in other areas of scientific computing, and not far beyond the reach of HEP. The Top500 supercomputer listing is a good place to look for computing trends because it is an indicator of where research money is being spent. It also shows who is driving change and innovation. Figure 1 shows a few attributes of the top ten machines. Note that four of the ten heavily utilize many-core coprocessors. Others are fairly special-purpose machines. Note also the modest multicore count on all the top ten. This multicore count has remained at a similar level for years. Megaflops/watt is now included as a relevant quality. Although megaflops is of limited value within HEP applications, a better measure of performance can be developed for evaluting computational effieciency.

[3] High Performance Computing

1.2.3. Available now Table 1 highlights key features of the currently available many-core processors that directly affect software frameworks and event processing applications. For the current K20, the key features that directly affect the ability to develop code (inhibiting or enabling) that does useful work are still the low memory available per thread and the simultaneous compute capacity of > 2500 cores. A common MIC Phi processor has a total of 240 threads available across the 60 cores. The memory is still very much limited to about 32MB/thread (based on 8GB total). There is also a substanial VPU with 512 bit SIMD registers.

For the K20, the thread count is not a straightforward calculation. The threads cooperate and the programming view is somewhat abstracted from the hardware view through the use of blocks and grids. As an example, at full occupancy, assuming full occupancy for an algorithm, 1024 threads (the maximum per block) can use 32 registers and share about 24KB of high-speed memory.

	Phi	K20
Power	300W	235W
Cores	60	2500
Threads	$60 * 4 = 240$	$2048 * 13 = 26624$
Memory/thread	$8GB/240 = 33MB$	$32reg + 24kB$ shared

Table 1. Interesting properties of two modern many-core processors

1.2.4. The past It is worthwhile looking back at the time before the x86 era. In these earlier days one needed to know nearly everything about the processor that was being used. To get anything useful done required a good strong effort and many hours with the processor references manuals. This was especially true of someone working in the embedded computing area. It was common practice to write mini-operating systems or standalone barebones programming environments for these processors. Maybe what we are seeing is a return to this kind of environment. There was not much fear of computing then and hopefully the sort of detailed computing work required to operate many-core will also be rewarding.

2. Software issues in general

2.1. On the way to many-core

With many-core computing comes a shift in the way software is organized and the way algorithms are designed and implemented. Before entering further into many-core, one last detail needs be be mentioned about current-era applications with regards to software development. What has the modern heavyweight processor done for software?

During this era Java came about. Javascript is now everywhere. For HEP, C++ is still here and strong. C++ remains unique in trying to be broadly accommodating with programming styles and efficiency. What do people outside HEP say of its use? Some say its a thing of the past because it is too complex and requires too much thinking because of its strict typing. Are they correct? If the language is used poorly, or used as the language was in the mid-90s, than the answer is yes.

With all the complexity of the current heavyweight core, it can be difficult to quantify performance in a way that attributes work done to the various internal components. It is easy to measure many things, but not easy turning the collected data to valuable information that helps us predict how non-trivial changes affect code performance [jbk-ref].

2.2. Development in this future context

There is still a clash of thinking with other organizations. Working with the HPC community can be challenging. HEP codes have no single computational kernel that can be unrolled, unfolded, turned to assembly language, expanded from 500 lines of code to 15,000 lines in order to increase its performance on specialized architectures. They can also quickly note that the key to better performance is the removal of all Object Oriented programming, largely because it seems that early C++ code used a lot of virtual function calls within tight loops.

The architectures and integration are different on these many-core processors. The compromises in programming flexibility for lower power and the need for greater computational load are leading to problems for HEP applications. In the end, HEP applications are still a mismatch for many-core architectures. This is partly because of the many-core expectations of operating on a regular grid using lots of vector operations, and partly because of the habits HEP has developed and the problems that we have encountered. In many cases, HEP codes have been classified as irregular. Many pattern recognition, clustering, and decision analysis problems fit this description [2]. Slow progress has been experienced for many-core HEP applications due to the immaturity of the software and hardware for general purpose scientific computing. Many-core architectures, with their segmented memories, large vector units have not traditionally been geared for these types of problems. This has made progress for showing reasonably good value slow. The latest version of these processors, however, show good signs of moving in the HEP direction.

2.3. Why all the excitement?

Are the massively parallel many-core platforms really all that interesting? Are they something we need to be worried about? From the software infrastructure point of view, they ought to be something we care about and something to consider carefully. Such specialized hardware is becoming widely available and can provide a cost effective way to accomplish more work. The heavyweight processor, on the other hand, has remained somewhat similar in architecture and in platform requirements. Software frameworks are used to define HEP applications, so their organization and design is critical. HEP has very large collaborations that successfully work together. To a large extent, this is due to the software infrastructure components which have enabled so much sharing of code. A few of the key requirements that change with many-core are choosing the just the right unit work when many different processors are involved, and extra steps that are likely needed such as ordering and summarizing of partial results.

3. Software frameworks on many-core

The standard HEP event-processing framework has been an essential organ for development, data reduction, and analysis. To a large extent, the frameworks that are in operation still have the processing model shown in the diagram on the left of Figure 2. For C++ frameworks, this architecture was in use before 2000, although the facilities used and the capabilities of the current systems are far more advanced than the early ones. The implementations of scheduling and interactions amongst the components differs by experiment, but all of them largely fit this definition, with ROOT as the main I/O format for the readers and writers. The boxes in the modules and services sections represent the different flavors of plugin. The impact on these frameworks and the various plugins due to many-core processors will be large (as it currently is for multicore), partly because it is where major APIs and protocols exist. The design of interfaces will directly impact application performance.

For the remainer of this section, the assumptions for data sizes and processing times listed in Table 2 will be used. Any bandwidth estimates will be based on these numbers and only account for the input data rates.

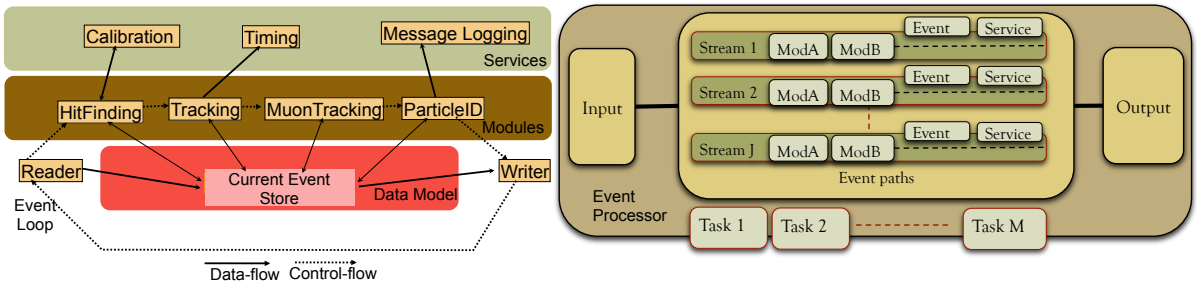

Figure 2. Standard event processing frameworks: Software that coordinates the processing of indendent collision events using pluggable reconstruction, filtering, and analysis modules. Modules add data to and retrieve data from one event. View before (left) and after (right) reorganization for use in multicore systems

$\sim 10 - 30$ sec	Reconstruction time
1-20Hz	Skim job event rate
1MB	Reconstruction event size
150KB	Analysis Object Data (AOD) event size

Table 2. Data assumptions based on CMS run 1 approximations

3.1. The current sitation

Many of the frameworks are being changed from serial processing to parallel processing to better accommodate multicore nodes. A simplified view and typical set of changes are depicted in the right diagram of Figure 2. These changes permit multiple events, shown as multiple processing streams in the diagram, to be active at one time. They also allow for parallelism at the module level where there are no data dependencies, and within the module at the user-written algorithm level. Streams provide serial paths for producer modules[4].

A key design decision that many framework developers have made is to more towards task-oriented processing to help manage multithreading. Intel Threading Building Blocks (TBB) [3] has been the tookit of choice for many projects. The key principle is to assign work to tasks, and allow the system to efficiently schedule tasks to run within threads assigned to cores. A common task model allows for both framework-generated work and algorithm-generated work to be mixed and scheduled. The input and output interfaces remain special, because the system facilities behind them have different capacities from the available computational resources.

Running the reorganized framework on a current multicore cluster nodes can have several advantages over the older single serial processs per core model. There is now one process capable to utilizing all resources on a node. This will reduce memory use, reduce dataset processing latencies (mainly due to the finer task queuing), and allow for multithreaded algorithm contributions. These multicore changes do not greatly affect the overall throughput of a standard cluster node as shown in Table 2.

3.2. Future direction

The left diagram of Figure 3 shows several of the important attributes of a many-core processor. A processor such as the K20 can efficiently schedule work within hardware-assisted threads to available cores. Each group of threads has a block of shared memory. All threads have access to a larger global memory (usually 6–10GB). The simple design of the processing units synchronizes execution of instructions across an entire block of threads, making programming more difficult.

[4] A producer adds derived data to an event

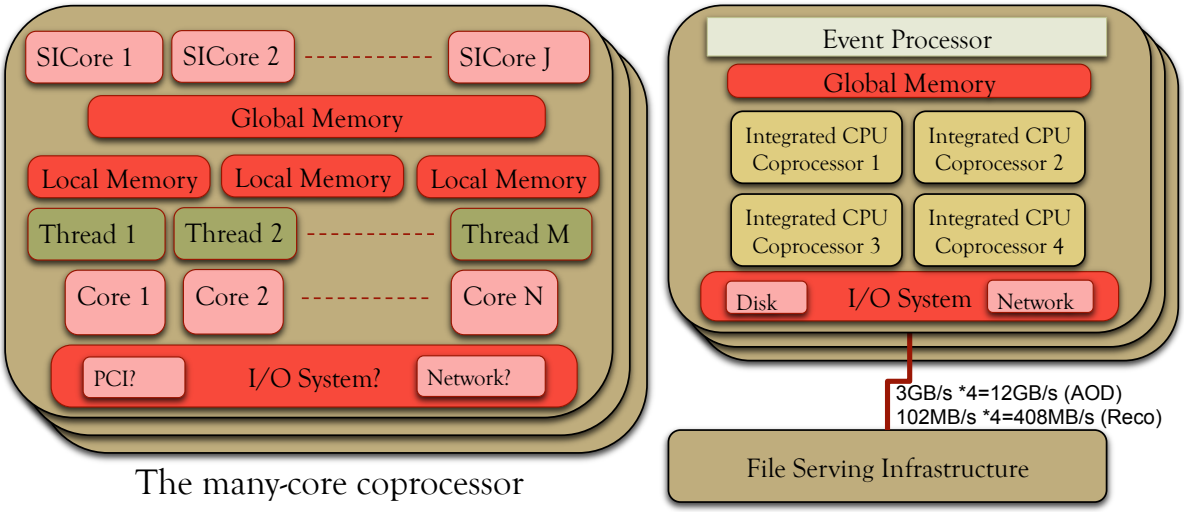

Figure 3. The future many-core processor with serial integrated cores (left) and a possible setting within a compute node alongside the event processing framework (right).

This is also true of the vector operations available on the Xeon Phi. Table 1 highlights many of the key many-core properties relevant for developers. In addition, the current processor boards sit on a PCI bus, making data transfer costs the major bottleneck in performance.

Current systems that employ the K20 GPU or the Xeon Phi typically have between two and four processor boards per node attached to the PCI bus. Such a node is capable of $5-10\times$ the compute capacity of a current 32 multicore node. Using the typical CMS event processing numbers, the bandwidth into the node may be as high as 3GB/s (AOD) or 102MB/s (Reco). Such performance requirements could easily exceed the network capabilities of current clusters. Many of the processor boards can be hooked directly into the external network, but that requires yet more software infrastructure changes. A node with this compute capacity will require low overhead scheduling and coordination within the software frameworks. The serial heavyweight cores will do the coordination, aid in computational codes where appropriate, and participate in I/O activities. It will be easy to overload the processor that is easiest to program, which is certainly going to be the coordination processor. Data exchange and sharing within and between computational areas (GPU processor or Phi processor) is likely to increase, meaning that a portion of our coordination software may need to be present on the light-weight cores.

With the increase in computational capabilities of individual nodes, will the software frameworks become distributed? Even within a node, they will need many of the properties of distributed frameworks: coordination of separate memories and load balancing across different kinds of resources. Even the sharing of data within the Event Data Model (EDM) will be affected by multiple, distinct memories.

Serial integrated cores (SICores) are also shown in Figure 3, in addition to the many-core features. This is a likely future direction for the many-core processor, where a version of the heavyweight multicore processor is available directly on the same chip. The ARM or PowerPC architecture are good candidates for these SICores since they are already known for low power, system-on-chip use, and embedded applications. The combination of many-core and multicore will form a powerful hybrid processor platform.

One possible view of a hybrid node is shown in the right diagram of Figure 3. A node with four hybrid processors could be capable of a $20-40\times$ increase in performance over a standard multicore cluster node. Unfortunately the bandwidth into such a node might have be 12GB/s

(AOD) or 408MB/s (reco) just to keep good processor utilization.

Event Size@Speed	1-core	32-core	256-core	512-core	>1024-core
150KB@20Hz	3MB/s	96MB/s	768MB/s	1.5GB/s	3GB/s
1MB@.1Hz	100KB/s	3.2MB/s	25MB/s	51MB/s	102MB/s

Table 3. Impact on moving toward high compute capacity nodes on input data bandwidth

Commercial products already exist with some of these features. There is the NVIDIA Tegra 4 [4] used in hand-held devises. It has 72 GPU cores and a quad-core ARM processor in one package. The Dell Copper [5] can hold up to 12 sleds in a 3U chassis, with each sled containing four quad-core ARM processors. The Cray XC30 blade [6] comes in three flavors: GPGPU, MIC, or multicore. Each blade has several processors on it. Customers pick and choose blades depending on the workload type. The blades plug directly into a backplane with builtin high-speed networking.

4. Programming issues

4.1. Aspects of programming many-core

There are special programming languages and libraries available to help developers use the many-core hardware effectively. The main tools for the common processors are CUDA for the K20 and Intel Cilk Plus, and the special intrinics and pragmas available in the Intel icc compiler for the Xeon Phi. Because of the specialized nature of these compute resources, these language extensions are likely to be permanent. These machine-specific programming features complicate every aspect of programming from coding (perhaps three or more implementations with partial overlap may be needed) to validation and testing. Each of these systems will embed the co-processor code into the host executable using cross-compilation tools. This will affect the development tool suites, release management, and moving to new compiler versions. It is easy to look at the Intel MIC and conclude that because there is overlap and commonality of the basic x86 instruction set, that development will surely be easier. With all the assists available in the programming tools and the internal hardware differences, it is not clear whether this compatibility will be as helpful as it appears.

The news is not all bad. GPGPU programming has come a long way in a very short time. The newer systems with compute capability 2.x make programming a bit smoother even though the constraints in using the cores and memory hierarchy are very similar to prior systems. The 3.0 capable systems go even further. Hyper-Q and dynamic parallelism capabilities are important for framework and algorithmic development. Hyper-Q allows many processing streams to be active at the same time. Dynamic parallelism allows GPU kernel functions to start more GPU kernels without returning control to the host processor. Overlapping transfers with computation boosts performance. There is still plenty of work left in testing the limits and flexibility of using this technique to see how an event-processing framework might take full advantage of them. Both are necessary to keep the GPU busy without incurring communications and call overheads. The dynamic parallelism feature will now permit elements of the software framework scheduling to occur within the GPU.

Movement towards the hybrid nodes described earlier will affect nearly all the subsystems of an event-processing framework. The increase in compute capability will stress the I/O systems. Adding distributed processing features to the framework will affect scheduling of work, the event data model (EDM), configuration, and organization of services. For scheduling, TBB will help provided that data dependencies are made more visible. For the EDM, features such as delayed reading and loading will need to be looked at closely. For configuration, the specification of boundaries (where elements are assigned) and constraints (what resources can be used) will need

expansion. Steps for closing out data runs and other logical data grouping will need additional processing and synchronization to perform reductions (sums and aggregations) across active events.

4.2. Already-visible software issues

Initial developments using available use-cases for reconstruction and event filtering on GPGPU many-core processors has revealed many issues. The considerations necessary to address the small hierarchical memories are perhaps the biggest. Bank access patterns are visible in the code and an algorithm developer must understand how to best traverse memory efficiently. Small changes in access patterns can yield wildly difference performance. Fortunately many of these access tricks also yield improvements in the main multicore processors. Doing odd things like recalculating results rather than caching them can yield performance improvements. Tool-chain-specific code blocks using CUDA or language extensions, or even different algorithm encodings with "#defines" or equivalent may be necessary. The framework or user-written algorithms will need to be aware when the lightweight HPC cores are better to use than the serial cores. The bigger problem is the need to choose where modules or algorithm phases are to be run. In prototype code, this is made easy by simplifying assumptions, such as hardcoded choices (for processor) and dedicated nodes. Eventually bodies of user code (modules or algorithms) will need to be scheduled where they will run best.

Collecting additional summary data and diagnostic information during algorithmic work will no longer be straightforward. This includes data for validating algorithms. It is currently easy in serial processing mode to tack extra variables onto the module data area (modules are C++ objects), or onto temporary data structures that are active during the module processing. Many times this extra data has associated special logic used to add to variable-length data structures. Such deviations from common instruction flow will likely lead to deviations in work load and therefore lead to performance decreases in lightweight cores.

Using the task model for parallelism (through TBB) has been helpful. It is a good direction that everyone is going towards and it fits naturally with a higher-level interfaces for dividing up work in loops. It does not alleviate the need for thoughtful construction of data structures for the tasks to operate on and careful division of work into stages that are relevant for the available processor types. It does alleviate the need for locking in user code when used well. Fortunately many of the optimizations made to improve performance for many-core have also benefited in the multicore environment.

5. A complete view

Figure 4 pulls together many of the thoughts from this paper and shows a possible outcome of moving towards more specialized hardware within nodes and facilities available within a cluster. The hybrid server cluster towards the top contains the many-core and multicore nodes to handle heavy computational load. The big data management block towards the bottom contains the essential and specialized infrastructure services to perform all data handling functions, including file and database input and output, data filtering and selection, and movement to and from the computational resources.

The user of such a system carries out the steps of developing a framework configuration and requesting a collection of coordinated resources to perform a physics task. Launching of the job through framework services involves placing the application components where they are best suited to run (computational or external storage), using the specialized hardware that is available. In this configuration, complexity of storage components and disk access is removed from the computational elements, thus simplifying both the hardware and the software; only a network streaming I/O service module is now needed in the framework. Data writing is delegated to nodes where file and storage system access is best handled. Data selection is delegated to nodes

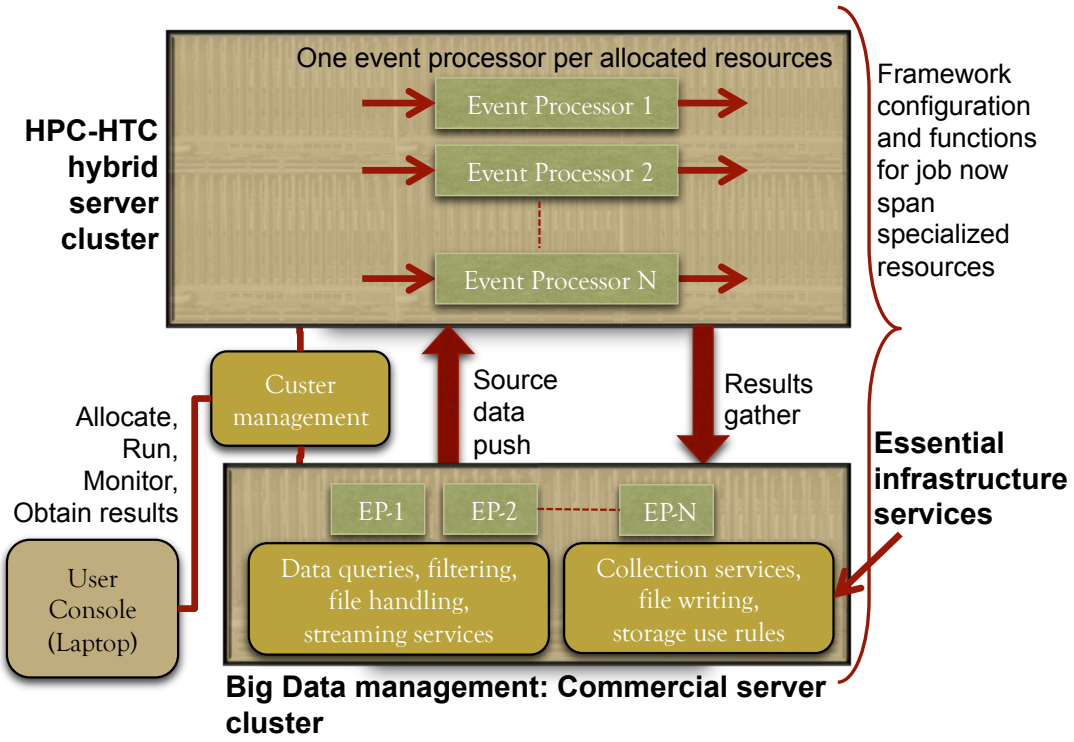

Figure 4. A cluster of the future with specialized resources for I/O and computations, with highly distributed framework component.

designed for decision logic closely attached to the data catalogs and other indexing systems. In this arrangement, some of the work is now done at the cluster level that traditionally was done (with good results) at the node level within an event-processing application.

6. Summary

A mixed computing environment is inevitable if we are going to make use of high-performance cores. Using these more specialized cores will be necessary to increase computational efficiency and to keep computing cost effective. The expression of the work to be done may need to be expanded to aid in the scheduling of work on a node that has 10× the capacity of a current node. The cooperation of multiple events, multiple algorithms, and multiple nodes may require more convenient ways to express the problem being handled. All the asynchronous processing of events will cause a need for final reduction stages to properly order data as it is processed, and provide statistical summaries. A change in strategy for algorithm development has already started and consideration for load balancing will need to be incorporated in the frameworks. The development strategy will need to accommodate both many-core and multicore. It is not all that clear if the I/O subsystem and the event data representation (EDM) and other data handling functions will be adequate for computational nodes described in this paper. For these nodes, simpler streaming-based methods of I/O directly using a high-speed network may be more appropriate, bypassing intermediate disk access wherever possible.

Both industry and research are moving towards performance increases through processor specialization and power savings. This will keep operational costs constant or perhaps even lower them. These changes translate to more independent components within the HEP software infrastructure and therefore mean more complexity. More than ever we have a need for continued cooperation from the different development teams, especially outside HEP, who do not have the

rich data structures or the advanced collaborative environments for sharing and development code, but do have the parallel processing experience. As in previous computing eras, HEP developers must be good at many levels of software development. The hope is that there is not only room for this broad level of involvement in the highly specialized world of today, but that there are also rewards to be had for entering into this sort of work.

References

[1] Brown 2010 Scientific grand challenges: Crosscutting technologies for computing at the exascale report from the workshop held February 2010 URL http://extremecomputing.labworks.org/crosscut/PNNL_20168.pdf

[2] Monteiro P and Monteiro M P 2010 A pattern language for parallelizing irregular algorithms *Proceedings of the 2010 Workshop on Parallel Programming Patterns* ParaPLoP '10 (New York, NY, USA: ACM) pp 13:1–13:14 ISBN 978-1-4503-0127-5 URL http://doi.acm.org/10.1145/1953611.1953624

[3] Reinders J 2007 *Intel threading building blocks* 1st ed (Sebastopol, CA, USA: O'Reilly & Associates, Inc.) ISBN 9780596514808

[4] NVIDIA 2013 Nvidia tegra processors URL http://www.nvidia.com/object/tegra-4-processor.html

[5] Trader T 2012 URL http://www.hpcwire.com/hpccloud/2012-05-29/dell_enters_hyperscale_arm_race.html

[6] Hemsoth N 2013 URL http://www.hpcwire.com/hpcwire/2013-10-01/cray_cascades_over_gpu_coprocessor_edge.html