GePSeA: A General-Purpose Software Acceleration Framework for Lightweight Task Offloading*

A. Singh[1] P. Balaji[2] W. Feng[1]

[1]Dept. of Computer Science
Virginia Tech
{ajeets, feng}@cs.vt.edu

[2]Mathematics and Computer Science
Argonne National Laboratory
balaji@mcs.anl.gov

Abstract

Specialized hardware accelerators have helped to improve application performance for many years. And as we scale to hundreds and thousands of cores, complex tasks, such as advanced application-specific processing, need to be offloaded to these accelerators in order to achieve better performance scalability. However, such specialized hardware accelerators also tend to be complex and add unnecessary expense to the system. Multi-core processors, on the other hand, have already become ubiquitous in supercomputers, cluster computers, data-centers and even personal computers. Thus, in this paper, we take advantage of the obvious benefits of multi/many-core architectures and propose *GePSeA*, a general-purpose software acceleration framework that uses the spare cycles of an available core in a multi-core equipped node to "onload" complex application-specific tasks. Specifically, our GePSeA framework provides a lightweight process that acts as a helper agent to the application by executing application-specific tasks asynchronously and efficiently. Together with the detailed design of GePSeA, we also present a case study with *mpiBLAST*, an open-source computational biology application, and demonstrate significant application-level benefits.

1 *Introduction*

Specialized hardware-based accelerators have beed used to improve the performance of high-end computing systems for many years. While such accelerators continue to grow in performance and complexity, the requirement for even more advanced processing has grown as well. For example, with systems scaling to hundreds of thousands of cores available today, more and more complex tasks, such as advanced application-specific processing, are required to be offloaded. However, the manufacturing complexity of such hardware units increases exponentially with the number and complexity of tasks they offload.

On the other hand, multi- and many-core processors are being increasingly deployed in clusters as well as in the general desktops and laptops. Quad-core and Hex-core processors are considered commodity today; Intel's 16-core Larrabbee processor and 80-core terascale processors will be arriving in the near future as well. Similarly, processors with symmetric multi-threading (SMT) capabilities such as the Intel eXtreme processors [5] and the SUN Niagara processors [11] with up to 512 threads in a single physical box are already available today, with plans for up to 2048 threads within the next year. These trends point to the fact that today, and more so in the future, each physical node will have a massive number of processing units which, for some tasks, can be viewed as a low-cost alternatives to expensive hardware accelerators.

Consequently, taking advantage of the obvious benefits of multi/many-core architectures, we propose, design and evaluate *GePSeA*—a lightweight general purpose software acceleraion framework that dedicates a small subset of the available cores on a multi-core equipped node to "onloading" complex application-specific tasks. Specifically, the GePSeA framework provides a number of utility components providing different functionalities together with a generic interface for applications to utilize them through simple plugins. For this paper, we present three categories of utility components, namely (i) data management components, (ii) memory management components and (iii) synchronization and coordination components. However, the general purpose nature of our framework allows it to be extended to other categories as well. In our design, different utility components are compiled together into a lightweight process referred to as a *software accelerator*. This process acts as a helper agent to the application by executing lightweight application-specific tasks asynchronously and efficiently.

The GePSeA framework does not aim to replace tasks that dedicated hardware-based accelerators such as GPGPUs and Cell processors specialize in. Instead, it utilizes the existing hardware-offloaded features of such accelerators, but extends them by onloading more complex application-specific functionality that cannot be easily offloaded. The general purpose processing capabilities of multi-core architectures allow our proposed architecture to be designed in a flexible, extensible and scalable manner, while benefiting from the reducing costs of each processing core.

Together with the detailed design of GePSeA, in this paper, we also present a case study with *mpiBLAST*, a popular open source computational biology application. Specifically, we present details on the various tasks onloaded with our framework and demonstrate more than 205% improvement in the overall application performance.

The remaining part of the paper is organized as follows. In

*This work was supported in part by NSF STTR Grant IIP-0741004 and the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

Section **??**, we present an overview of computational and communication accelerators and the mpiBLAST application. We present the design of our proposed software accelerator framework in Section 3. The case study with mpiBLAST is presented in Section 4 followed by a detailed experimental evaluation in Section 5. Other literature related to our paper is described in Section 2 and the conclusions and future work are presented in Section 6.

2 *Related Work*

While there have been many recent research efforts both on developing hardware accelerators [17, 14, 10, 15] and developing frameworks and libraries [3, 4, 7] for parallel and distributed computing, our work on design and development of generic framework for software acceleration for parallel applications is distinct from others.

Frameworks for parallel programming [3, 4] in the existing literature are collection of library functions intended to make programming easy for the programmers. These frameworks and libraries are used for the development of parallel applications while our framework is to build software accelerators for the applications. Our framework does not provide libraries such as mpich, OpenMPI etc for application development.

In [13, 9], the authors study several techniques that can be used for overlapping I/O, computation and communication in parallel scientific applications written using MPI and MPI-IO. Using our software accelerator, parallel applications can offload any lightweight task including disk I/O and communication.

In [16] authors developed a communication engine to exploit the core in multi-core systems using various multi-threading techniques. Authors plan to integrate this with the MPICH2 in future. This work is close to our research work. Software accelerator framework is however distinct from the communication engine. Our framework provide mechanism to build application specific plugins from the generic lower layers of our framework which can then be executed by accelerator. It not required use to use MPICH for the application to use our framework.

In summary, our work differs from existing literature with respect to its capabilities and underlying architecture, but at the same time, forms a complementary contribution to other existing literature that can be simultaneously utilized.

3 *The GePSeA Framework Design*

The GePSeA framework is designed to function as independently of the application as possible. Much like existing hardware accelerators, GePSeA onloads several utility functions on a dedicated unit (subset of cores in this case) and provides simple interface that applications can use to take advantage of such functionality. These tasks are executed asynchronously allowing the application to continue its respective processing. This design is illustrated in Figure 1.

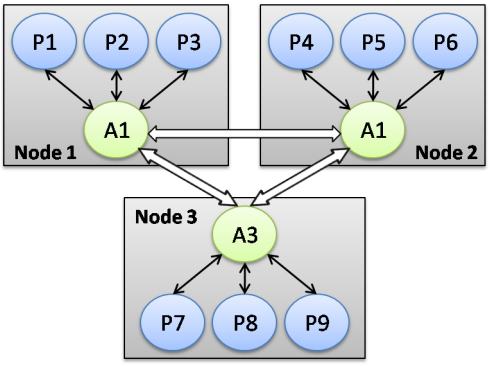

Figure 1: Using accelerator for parallel applications

While GePSeA is a general purpose framework that allows it to "onload" any task, in this paper we concentrate on three categories of components, namely: (a) data management components, (b) memory management components and (c) synchronization and coordination components as illustrated in Figure 2. In this section, we describe the details of each of these categories as well as some of the components within these categories.

3.1 *Data Management Utilities*

The data management utilities primarily deal with moving I/O data associated with the application. This includes input data, output data and transitional meta-data that is created and destroyed during application execution. We present four utilities within this category that the GePSeA framework provides: (i) distributed data caching, (ii) data streaming service, (iii) distributed data sorting and (iv) data compression engine. It is to be noted that while this paper presents software-based designs for each of these utilities, this does not preclude GePSeA from taking advantage of hardware implementations when they are available. When such hardware units are available, GePSeA would simply replace these modules with them while retaining the interface it exposes to applications. Thus, from an application's perspective, the actual implementation of these utilities does not matter.

Distributed Data Caching: This component provides caching capability for the entire input database on the overall system memory. The input data used by many applications (terabytes in size) is typically several times larger than the amount of memory on each node (gigabytes in size). However, for large-scale systems, the total system memory is tens of terabytes in size which, on the whole, is sufficient to cache the entire input data in many cases. The distributed data caching component performs this task by trapping I/O calls, reading the entire input data into the system memory and responding to I/O requests from the distributed memory cache, instead of from the disk or file-system.

One of the primary challenges in distributed data caching is

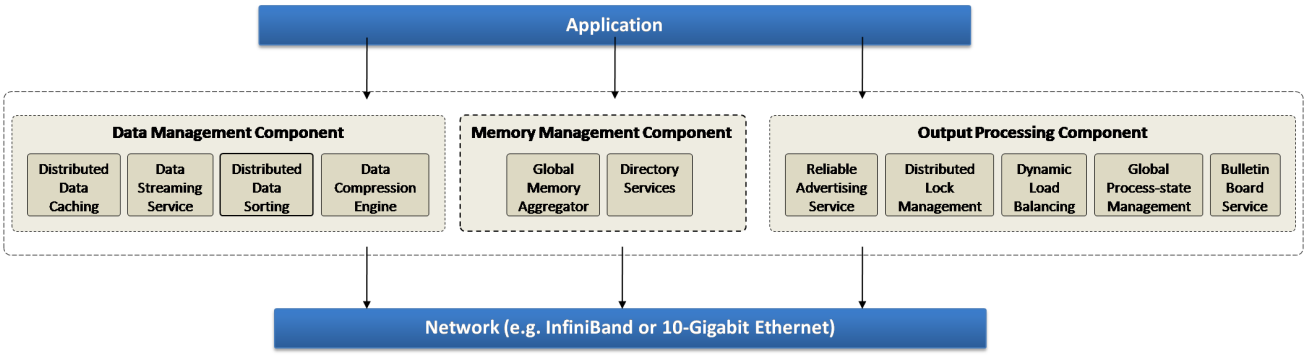

Figure 2: GePSeA Architecture

the addressability of the data. Specifically, should the application be aware of the actual locality of the data segment or should this information be hidden and handled internally by the data caching component? Typically, for most scalable applications, I/O transactions involve moving reasonably large amounts of data (at least a few kilobytes, but most times several megabytes). Thus, the overhead added due to trapping I/O calls and automatically figuring out the location and fetching data is typically not a major portion of the overall I/O cost. Further, implicitly managing data locality makes it simpler and intuitive for applications to use this component as well. Thus, we chose to hide data locality for component. Data movement is completely handled by this component through appropriate communication that is initiated and terminated within the components and never exposed to the applications.

Data Streaming Service: While distributed data caching moves data from the file-system to system memory thus reducing I/O overhead, the amount of time taken for data movement is still large, especially when the amount of computation per data unit is not much (e.g., in search algorithms). Thus, to keep the application fed with data, techniques such as prefetching are very useful. The data streaming service handles these aspects by swapping out unused data with fresh data that that application would use. This component includes distributed coordination with other instances of the GePSeA helper agents, in order to minimize duplication of data (i.e., data is swapped between two nodes instead of replicating and utilizing more memory than needed).

Another important aspect to note in this service is that since it is completely handled by the GePSeA helper agents, such prefetching and swapping is done in a completely asynchronous manner without disturbing the application.

Data Compression Engine: As the name suggests, this component deals with compressing/decompressing data. However, together with regular byte-stream compression, this engine also provides capabilities for application-specific compression as presented in our previous work [12]. Specifically, this engine can either view the data as a stream of bytes, or as high-level application-specific objects and converts them to meta-data that is much smaller in size. Once communicated over the network, this meta-data is reconverted back to the actual data.

3.2 *Memory Management Utilities*

Memory management utilities deal with handling system memory allowing applications with access to the entire memory in the system, rather than just the local nodes memory.

Global Memory Aggregator: Operations like caching, indexing, searching, etc. can all perform better with larger memory. Since the cost of remote memory access can be and is typically much lower than the cost of disk access, these components can effectively utilize the free memory available on other nodes. The global memory aggregator efficiently allows applications to utilize the memory on the entire cluster instead of just their local memory. Specifically, this primitive presents a global address space to its upper layers and maintains a mapping of the locations on the global address space with the actual node and physical address of these locations. Unlike the distributed data caching service, this component does not perform the global address translation to the actual physical address and physical node transparently; this is because memory accesses are typically much smaller in size than I/O accesses and are expected to have very little overhead. Thus, applications explicitly control and manage data placement on the system. Data movement, however, is completely handled by the global memory aggregator.

3.3 *Coordination and Synchronization Utilities*

For applications using a large number of processes, coordination and synchronization between the processes can be a major portion of the application execution. This category deals with utility components that improve such tasks.

Global Process State Management: Managing the data processing performed by application processes requires sharing certain information about each node, such as whether the process on that node is idle and waiting for communication, what fragment of the data it currently hosts, and various others, among the different nodes in the system. The performance and scalability of applications largely depends on how effi-

ciently such a global process-state is maintained. Thus, the global process-state management primitive aims at maintaining an up-to-date and complete information about the status of the different nodes in the cluster.

Bulletin Board Service: This task provides an addressable memory that can be read or written to by any other node in the cluster system. The bulletin board itself is distributed memory placed on different nodes in the system. However, from an application's perspective, this is a contiguous chunk of memory that is available to publish information. Together with efficient movement of data, this component also handles the synchronization required in order to avoid data corruption.

Reliable Advertising Service: The reliable advertising service primarily deals with reliable and efficient ways of distributing information across the entire system. Where available, this task can internally utilize the unreliable multicast features provided by networks such as InfiniBand, while providing software reliability on top of it. On other architectures, such reliability is handled using reliable protocols such as TCP/IP. This task also includes various other capabilities such as protection against overwrite (i.e., two continuous messages from the same host will ensure that the first message is read by the host before the second is delivered), host transparent advertising (i.e., the remote host does not have to actively provide a buffer to receive the advertisements from other nodes), advertisement filtering (i.e., getting rid of irrelevant advertisements) and various others that need to be handled efficiently.

Distributed Lock Management: A distributed lock manager allows lock-based synchronization between multiple nodes to avoid various race conditions while accessing shared resources. While it is possible to provide such locking capability using the atomic operation features provided by high-speed networks such as InfiniBand, the GePSeA helper agents can enhance these features by providing capabilities such as request queuing and group-wise shared locks, that cannot be easily provided in hardware. For environments where such networks are not available, the GePSeA agents can perform both the atomic operations as well as the request queueing and shared locks.

Dynamic Load Balancing: Load imbalance among the nodes of the cluster can result in a serious bottlenecks which can limit the scalability of parallel applications. The dynamic load balancing primitive provide mechanisms to balance the load on each node using the reliable advertising service component to keep track of availability of all the nodes in the system. In this approach, each node announces its availability when idle to an elected node called the *scheduler*. The scheduler then assigns the work to the nodes. Each node periodically queries the scheduler to obtain the information about the work assignment to itself and to all the other nodes in the cluster. Multiple schedulers are also possible to avoid hot-spot effects, but are not studied in this paper, but will be handled in the future.

4 *Case Study with mpiBLAST*

The GePSeA framework is a general purpose software accelerator that can be used by many applications. To demonstrate its capabilities, in this section, we describe a case study using a popular computational biology genetic sequence-search application called mpiBLAST. We present a brief overview of mpiBLAST in Section 4.1 followed by the description of the integration of mpiBLAST with GePSeA in Section 4.2.

4.1 *mpiBLAST Overview*

mpiBLAST [6] is one the most popular genetic sequence-search applications used in computational biology. It internally uses the NCBI BLAST toolkit [1], the de facto "gold standard" for sequential pairwise sequence-search that is ubiquitously used in biomedical research. The BLAST tool searches one or multiple input query sequences against a database of known nucleotide (DNA) or amino acid sequences. A similarity score is calculated for each close match based on a statistical model. The similarity of the comparison is measured by the match with the highest score. As a result, the sequences in the database that are most similar to the query sequence are reported, along with their matches scored beyond a certain threshold. Therefore, the BLAST process is essentially a top-k search, where k can be specified by the user, with a default value of 500.

The core of the mpiBLAST algorithm is based on *database segmentation*. Before the search, the raw sequence database is formatted, partitioned into fragments, and stored in a shared storage space. mpiBLAST then organizes parallel processes into one master and many workers. The master breaks down the search job in a Cartesian-product manner and maintains a list of unsearched tasks, each represented as a pair of a query sequence and a database fragment. Whenever a worker becomes idle, it asks the master for a unsearched task and copies the needed fragment to its local disk (if the fragment has not been cached locally) and performs a BLAST search on its assignment. Upon finishing a task, a worker reports its local results to the master for centralized result merging. Once the results of searching a query sequence against all database fragments have been collected, the master calls the standard NCBI BLAST output function to format and print out results of this query to the output file. By default, those results contain the top 500 database sequences with highest similarity to the query sequence along with their matches. The above process repeats until all tasks have been searched. With database segmentation, mpiBLAST can deliver super-linear speed-up when searching sequence databases larger than the memory of a single node. In recent developments, mpiBLAST has evolved to use a more scalable parallel approach that allows different queries to be concurrently searched as well [8].

mpiBLAST like most parallel sequence search algorithms follow scatter-search-gather model. The *scatter* stage consists of query and/or database segmentation. In the *search* stage, each

worker searches the query against the assigned database portion. Finally, the *gather* stage consists of merging of output results from individual workers.

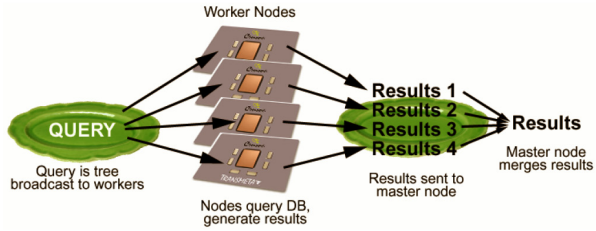

Figure 3: Scatter-search-gather model of parallel sequence search applications[2]

4.2 *mpiBLAST over GePSeA*

Figure 4 shows components at each layer of accelerator framework used by the mpiBLAST application. We developed asynchronous output consolidation, runtime output compression and hot-swap database fragments plugins for mpiBLAST. Using these plugins mpiBLAST can offload considerable amount of work to accelerator that can be executed in parallel. Though we developed these plugins for mpiBLAST, we believe they can be used by other sequence search applications such that follow scatter-search-gather model described in the previous section.

4.2.1 *Asynchronous Output Consolidation Plugin*

This plugin utilizes the processing capability of the accelerator to merge/sort the data that is distributed across all multiple nodes in parallel. This is an important capability since result merging can be done asynchronously by the accelerators while the applications can continue their respective application processing. For example, if the first node has gotten its results earlier than the other nodes, it does not have to wait for till the others are done to perform the merge. Instead, it can hand over this data to the accelerator; this accelerator can wait for the other nodes and sort the data incrementally as the other nodes finish their task.

To remove the bottleneck due to single writer, each accelerator has the capability to write the output results directly to the output file on a shared storage. Accelerator writes the results into a separate file for each query assigned to it by the *scheduler*. After all the queries have been processed the result files are sorted and merged into a single output file. This work of merging and writing output data is divided fairly among the accelerators using load balancing primitive.

4.2.2 *Runtime Output Compression Plugin*

The data compression engine provides capabilities to compress the actual data. We conducted tests on the compressibility of the BLAST output and found that when the output was in the standard pairwise alignment text format that the output could be compresses to less than 10 percent of its original size

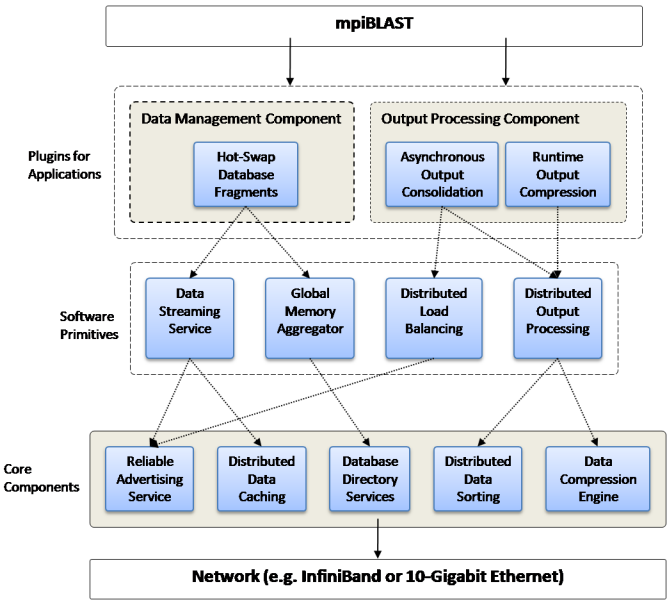

Figure 4: Accelerator Framework used by mpiBLAST

using gzip. This is mainly due to the redundancy found in the BLAST output. The runtime output compression can be used to compress the output before transfer thereby significantly reducing the transfer time.

4.2.3 *Hot-Swap Database Fragments Plugin*

Currently, in sequence search applications, the database is pre-partitioned into a number of fragments that are distributed over different nodes. Normally worker node searches the database it holds. However balancing the work load on each worker may require a node to hot-swap the portion of the database it is currently processing with the other portions on-demand. This feature swaps the database fragments asynchronously across the nodes while hosts can continue their respective application processing.

5 *Experimental Evaluation*

In this section, we evaluate the accelerator for mpiBLAST and do a performance analysis. For all our experiments, we used the GenBank **`nr`** database, a protein type repository frequently searched against by bioinformatics researchers. The size of the raw **`nr`** database is nearly 1 GB, consisting of 1,986,684 peptide sequences. We used ICE cluster residing in Synergy Lab at Virginia Tech for our experiments. ICE cluster has 9 nodes with each node equipped with 2 Dual-Core AMD Opteron Processor 2218. Therefore each node has 4 cores. Memory and cache size on each node is 4 GB and 1024 KB respectively. The interconnect is 1 Gbps Ethernet.

For our experiments we pre-partitioned the **`nr`** database into 8 fragments using **`mpiformatdb`** utility. For most experiments the input query sets containing different number of sequences were chosen randomly from **`nr`** database. For some

experiments pseudo-random query sets were chosen in order to have a better control over the output size and to better analyze the efficacy of specific features of our accelerator. Our experimental methodology include running mpiBLAST with accelerator and without the accelerator on a given set of processors on the ICE cluster for the same set of input query set.

5.1 *Running accelerator on existing core*

By existing core we mean a core that is already running some application process. Each node of the cluster have total of 4 cores. We ran one worker process on each of these 4 cores with each worker process explicitly bound to a core using **`physcpubind`** utility available on NUMA machines. Our accelerator (one per node) ran on one of these 4 cores as per the scheduling strategy of the operating system. We conducted the experiments running mpiBLAST with and without accelerator for 8, 16, 24 and 36 workers, each running on a separate core. For this experiment 300 input query sequences were randomly chosen from the **`nr`** database.

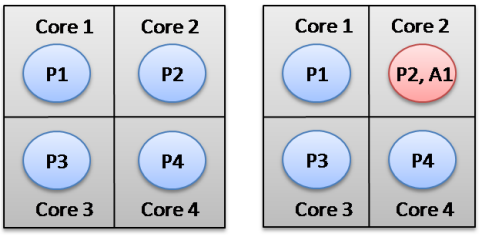

Figure 5: (a) Each worker process runs on a separate core (b) Core 2 shared between worker process P2 and accelerator A1

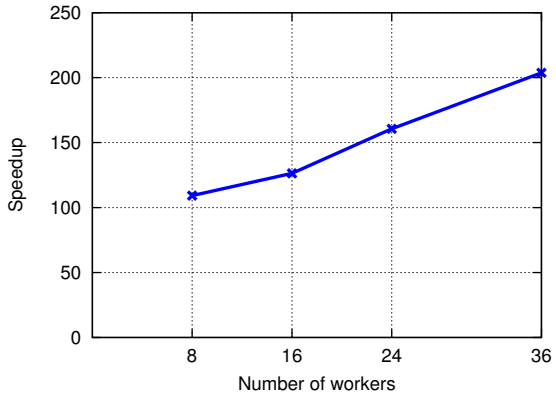

Figure 6: Speedup when accelerator run on existing core

Our results, see Figure 6, show that there is a significant performance improvement using the accelerator. With 36 workers we observed as much as 205% speedup. Speedup increases with increase in number of worker processes. The main reasons for this can be attributed to overlapping of worker computation and communication (between worker and writer). Offloading of result merging and writing tasks to accelerator also contributed significantly for this speedup. Results are particularly interesting since we are running accelerator on an oversubscribed core (against running exclusively on a spare core thereby committing it additional system resources) and still observe significant improvement.

5.2 *Running accelerator on a spare core*

By spare core we mean core that does not run any application process. In this experiment, on a node, we ran 3 worker processes each bound explicitly to a core and the accelerator bound explicitly to the fourth spare core. Therefore for 9 nodes, we have total of 27 workers with one accelerator on each node. Like the previous case, from Figure 8 we see significant speedup of mpiBLAST with maximum of 1.68 for 27 workers for the same reasons. We noticed that while the CPU utilization of worker process is nearly 100%, CPU utilization of accelerator is only 2-5%. Therefore running it exclusively on a spare core result in under-utilization of system resources.

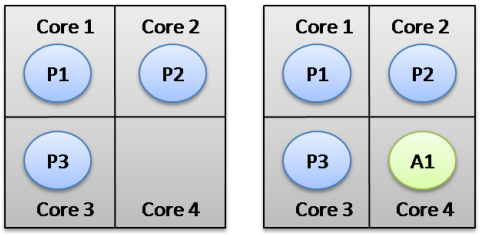

Figure 7: (a) Each of 3 worker processes runs on separate core, core 4 unused (b) Each of 3 worker processes runs on separate core, accelerator A1 on fourth spare core

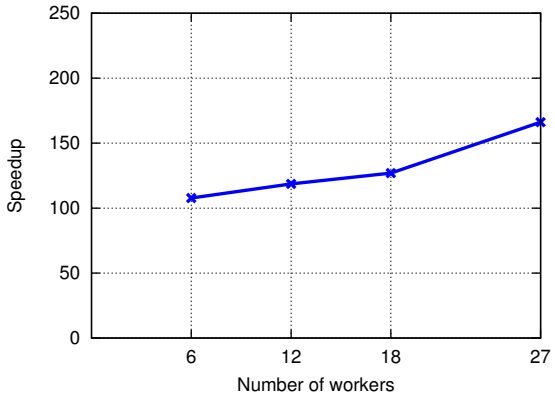

Figure 8: Speedup when accelerator runs on spare core

5.3 *Running accelerator on a spare core with unequal number of worker processes*

We use the data from above two experiments to analyze the effect of running mpiBLAST without accelerator with 4 worker processes running on 4 cores of a node against running mpiBLAST with accelerator with 3 workers nodes running on 3 cores of a node and accelerator running on the fourth spare

core. For example, for 9 node configuration, we compare between 36 mpiBLAST worker processes running on 36 different cores without accelerator against 27 mpiBLAST worker processes with an accelerator on each node. Even with one worker less per node, we still see speedup as much as 1.4 with 36 workers. Refer to Figure 10. Even though accelerator utilizes only 2-5% CPU cycles, it still performs better when replaced by a worker whose CPU utilization is close to 100%.

As mentioned earlier that running accelerator on an spare core result in under-utilization of CPU but even then it performs better than being replaced by a worker process.

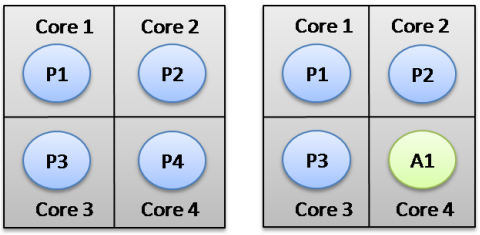

Figure 9: (a) 4 worker processes per node running on separate cores (b) 3 worker processes and an accelerator per node running on separate cores

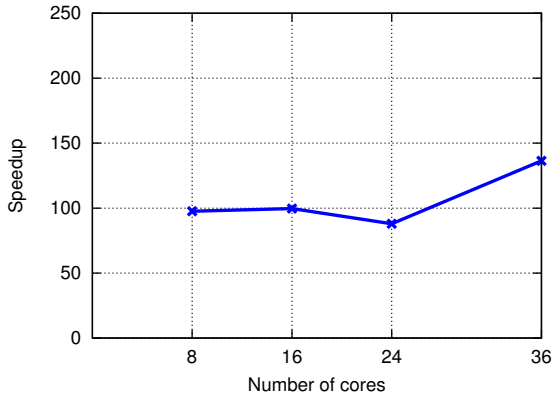

Figure 10: Speedup using accelerator with different number of workers

5.4 *Worker search time*

We analyzed the variation of *search time* and the *non-search time* of each worker with increase in number of worker processes. For mpiBLAST, search time refers to the computation time while non-search time refers to non-computation time. We observed that for a fairly large number of input query sequences, percentage of *search time* of a worker to total worker time decreases rapidly from 92.2% to nearly 71% as shown in Figure 11. However mpiBLAST with accelerator reported over 99% of the total worker time as search time on a consistent basis.

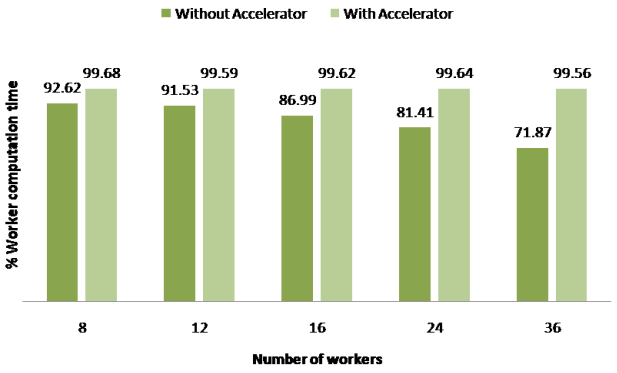

Figure 11: Worker search time as a percentage of total worker time with and without accelerator

5.5 *Asynchronous output consolidation*

We tested distributed output consolidation feature provided by accelerator as described in section 4.2.1. We compared it with result consolidation done by only one accelerator in the cluster statically assigned. Results are shown in Figure 12.

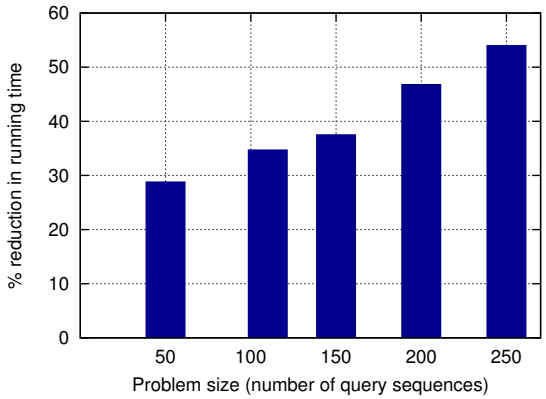

Figure 12: Reduction in running time of mpiBLAST due to asynchronous result consolidation feature

5.6 *Dynamic load balancing*

We tested dynamic load balancing functionality provided by accelerator as described in the design section. We compared dynamic load balancing with static allocation of result merging and writing assignment. We see average of 14% of improvement from the Figure 13. With highly *uneven* queries this difference could be very high.

5.7 *Runtime output compression*

We tested compression functionality provided by accelerator as described in section 4.2.2. Negative values Figure 14 shows increase in running time of mpiBLAST using the compression engine. This is contrary to our expectations. However, for compression at run time to be effective, network la-

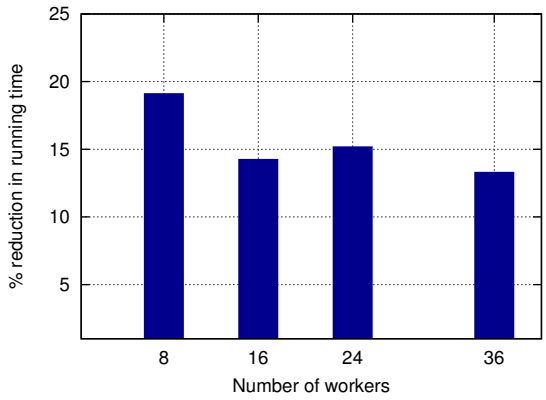

Figure 13: Reduction in running time of mpiBLAST due to dynamic load balancing

tency must exceed the time required to compress and uncompress the data. We believe that size of result data generated by our experiments is not large enough to test for compression engine to make positive impact on the running time. However we do observe a that running time decreases with increase in worker processes.

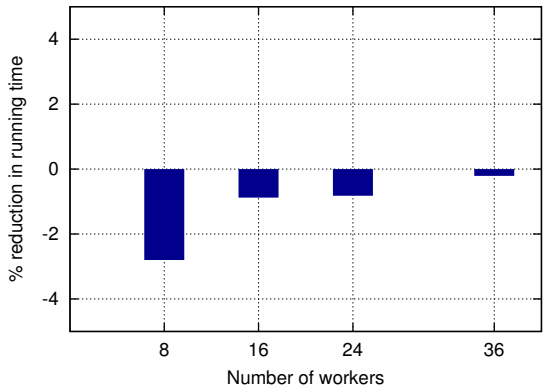

Figure 14: Effect of using compression engine to compress the output at runtime

6 *Conclusions and Future Work*

In this paper we presented a lightweight pluggable framework that dedicates a small subset of the available cores on a multicore equipped node to "onload" complex application-specific tasks. The proposed framework does not aim to replace tasks that dedicated hardware-based accelerators such as GPGPUs and Cell processors specialize in. Instead, it utilizes the existing hardware-offloaded features of such accelerators, but extends them by onloading more complex application-specific functionality that cannot be easily offloaded. Together with the detailed design of our lightweight pluggable framework, we also presented a case study with *mpiBLAST*, a popular open source computational biology application. Specifically, we presented details on the various tasks onloaded with our framework and demonstrated more than 205% improvement in the overall application performance.

Our future work would involve extending our framework to support new abstract features so that the accelerator can be used by more applications. We also plan to use our framework with other parallel applications and also conduct conduct extensive performance and usability studies.

References

[1] S. Altschula, W. Gisha, W. Millerb, E. Meyersc, and D. Lipmana. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3), 1990.

[2] Jeremy Archuleta, Eli Tilevich, and Wu chun Feng. Maintainable Software Architecture for Fast and Modular Bioinformatics Sequence Search . In *Proc. of the 23rd IEEE International Conference on Software Maintenance*, 2007.

[3] Olivier Aumage, Guillaume Mercier, and Raymond Namyst. MPICH/Madeleine: A True Multi-Protocol MPI for High-Performance Networks. In *Proc. of 15th International Parallel and Distributed Processing Symposium*, 2001.

[4] F. Bertrand and R. Bramley. DCA: A distributed CCA framework based on MPI. In *Proc. of High-Level Parallel Programming Models and Supportive Environments*, 2004.

[5] Intel Corporation. Intel pentium processor extreme edition. `http://www.intel.com/products/processor/pentiumXE/index.htm`.

[6] A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiblast. In *International Conference on Linux Clusters: The HPC Revolution 2003*, 2003.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of Symposium on Operating System Design and Implementation*, 2004.

[8] M. Gardner, W. Feng, J. Archuleta, and X. Ma H. Lin. Parallel Genomic Sequence-Searching on an Ad-Hoc Grid: Experiences, Lessons Learned, and Implications . In *IEEE/ACM SC2006: The International Conference on High-Performance Computing, Networking, and Storage*, 2006.

[9] Mao Jiayin, Song Bo, Wu Yongwei, and Yang Guangwen. Overlapping Communication and Computation in MPI by Multithreading. In *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications*, 2006.

[10] R. Luthy and C. Hoover. Hardware and software systems for accelerating common bioinformatics sequence analysis algorithms . In *Biosilico, 2(1)*, 2004.

[11] Sun Microsystems. Niagara 2: The next evolution in coolthreads technology. `http://www.sun.com/processors/niagara`.

[12] J. Archuleta P. Balaji, W. Feng and H. Lin. ParaMEDIC: Parallel Metadata Environment for Distributed I/O and Computing . In *IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2007.

[13] Christina M Patrick, SeungWoo Son, and Mahmut Kandemir. Enhancing the Performance of MPI-IO Applications by Overlapping I/O, Computation and Communication . In *Proc. of Symposium on Principles and Practice of Parallel Programming*, 2008.

[14] IBM Research. The cell project at ibm research. `http://www.research.ibm.com/cell/`.

[15] R. K. Singh, W. D. Dettloff, V. L. Chi, D. L. Hoffman, S. G. Tell, C. T. White, S. F. Altschul, and B. W. Erickson. BioSCAN: A Dynamically Reconfigurable Systolic Array for Biosequence Analysis. In *Research on Integrated Systems*, 1993.

[16] F. Trahay, E. Brunet, A. Denis, and R. Namyst. A multithreaded communication engine for multicore architectures. In *Proc. of IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[17] General-purpose computation using graphics hardware. `http://www.gpgpu.org`.