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

Hardware-acceleration techniques continue to be used to speed-up the execution of scientific codes. To do so, software developers identify portions of these codes that are amenable for offloading and map them to hardware accelerators. However, offloading such tasks to specialized hardware accelerators is non-trivial. Furthermore, these accelerators can add significant cost to a computing system.

Consequently, we propose a framework called GePSeA (General Purpose Software Acceleration Framework), which uses a small fraction of the computational power on multi-core architectures to "onload" complex application-specific tasks. Specifically, GePSeA provides a lightweight process that acts as a helper agent to the application by executing application-specific tasks asynchronously and efficiently. We then apply the GePSeA framework to a real application, namely *mpiBLAST*, an open-source computational biology application, and demonstrate significant application-level benefits.

1 *Introduction*

Hardware-acceleration techniques continue to assist in improving the performance of scientific codes. These accelerators have traditionally focused on speeding-up the computationally intensive portions of an application such as Fourier transformations, finite-element methods (FEMs), and dense linear algebra. A secondary benefit of hardware accelerators is their capability to serve as dedicated asynchronous engines, as compared to general-purpose CPUs that are designed to be shared between multiple processes.

As high-end computing systems and their associated applications continue to grow in scale and complexity, the need to offload more complex tasks, like asynchronous data management, becomes increasingly important. However, such tasks are orthogonal to the primary purpose of existing hardware accelerators. Consequently, offloading these tasks to hardware accelerators would result in a significant increase in the complexity and overall cost of a computing system. On the other hand, multi- and many-core architectures have become ubiquitous with quad- and hex-core processors already available and 16- and 80-core processors on the roadmap [6, 16]. 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 low-cost alternatives to expensive hardware accelerators.

In our previous work, we proposed ProOnE [11], a protocol onload engine that utilizes a small fraction of the computational power of multi-core architectures and allows for efficient onload of communication-related aspects in large-scale systems. In this paper, we extend our previous design and propose GePSeA[1] — a general-purpose, software-acceleration framework, which can also onload more complex application-specific tasks. Specifically, GePSeA provides a number of utility components that support different functionalities as well as a generic interface for applications to utilize these components through simple plug-ins. While this paper presents three categories of utility components: (i) data-management components, (ii) memory-management components, and (iii) synchronization and coordination components, the general-purpose nature of GePSeA allows it to be extended to other categories as well. In our design, different utility components are aggregated 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.

Like ProOnE, GePSeA does *not* aim to replace tasks that dedicated hardware-based accelerators such as GPUs and Cell processors excel at. Instead, it leverages the existing hardware-offloaded features of such accelerators and extends them by onloading more complex, application-specific tasks that cannot be easily offloaded. We demonstrate the efficacy of GePSeA by applying it to *mpiBLAST*, an open-source bioinformatics application and delivering a performance improvement with 2.05x speed-up.

The rest of the paper is organized as follows. Section 2 presents background and related work on accelerators and frameworks for parallel programming, followed by the design of our proposed software accelerator framework, i.e., GePSeA, in Section 3. We then present a case study with mpiBLAST in Section 4, followed by a detailed experimental evaluation in Section 5. Section 6 presents our conclusions and discussion for future work.

*This work was supported in part by National Science Foundation (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.

[1]GePSeA is pronounced like the word *gypsy*.

2 *Background and Related Work*

While there exist many research efforts on developing specialized hardware accelerators [12, 15, 17, 19], very little exists with respect to software accelerators. However, there has been considerable interest in the research community to develop multi-core-aware applications [18]. In this paper, we show how our GePSeA framework provides software-based acceleration to dramatically improve the performance of our case-study application, mpiBLAST.

Our work is distinct from other frameworks and libraries [3, 5, 8] available for the development of parallel applications. Frameworks [3, 5] and libraries such as MPICH2 and OpenMP are collections of library functions that are intended for the quick development of parallel programs. In contrast, our framework is not intended for the development of parallel applications; rather, it is for the development of software accelerators that assist parallel applications.

In [10, 14], the authors study several techniques that can be used for overlapping I/O, computation, and communication in parallel applications written with MPI and MPI-IO. Using GePSeA, applications can offload any of the aforementioned tasks, including disk I/O and communication.

In [18], the authors present a communication engine to exploit the cores in multi-core systems using various multi-threading techniques. The authors plan to integrate their engine with MPICH2 in the future. This work is most closely related to ours. However, GePSeA differs in that it provides an infrastructure to quickly build *application-specific software accelerators* for any application (whether or not using MPICH2) as well as to efficiently schedule onloaded tasks to maximize the use of a compute node's system resources.

In summary, our work differs from the existing literature with respect to its capabilities and underlying architecture, and 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 a 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. Figure 1 illustrates the design.

While GePSeA is a general-purpose framework that allows arbitrary tasks to be "onloaded," we focus on three categories of components in this paper: (a) data-management components, (b) memory-management components and (c) synchronization and coordination components, as shown in Figure 2. We describe the details of each of these categories as well as some of the components within these categories below.

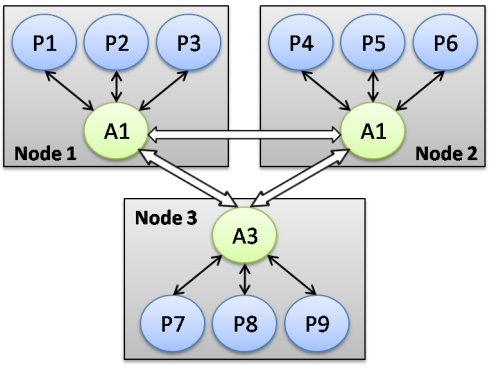

Figure 1: Using Accelerators for Parallel Applications

While the basic idea of some of these components is not new, e.g., [20, 13], our work focuses on providing such functionality in the context of a general-purpose software accelerator, which presents its own unique challenges.

3.1 *Data-Management Utilities*

The data-management utilities deal with moving I/O data associated with the application, including input data, output data, and transitional meta-data that is created and destroyed during application execution. We present four utilities within this category: (i) distributed data caching, (ii) data streaming service, (iii) distributed data sorting, and (iv) data compression engine. We note 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 the hardware ones 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 the caching capability for an entire input database across all of 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 filesystem.

The main challenge in distributed data caching is 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? For most scalable applications, I/O transactions involve moving reasonably large amounts of data (e.g., a few megabytes at a time). So, the overhead added due

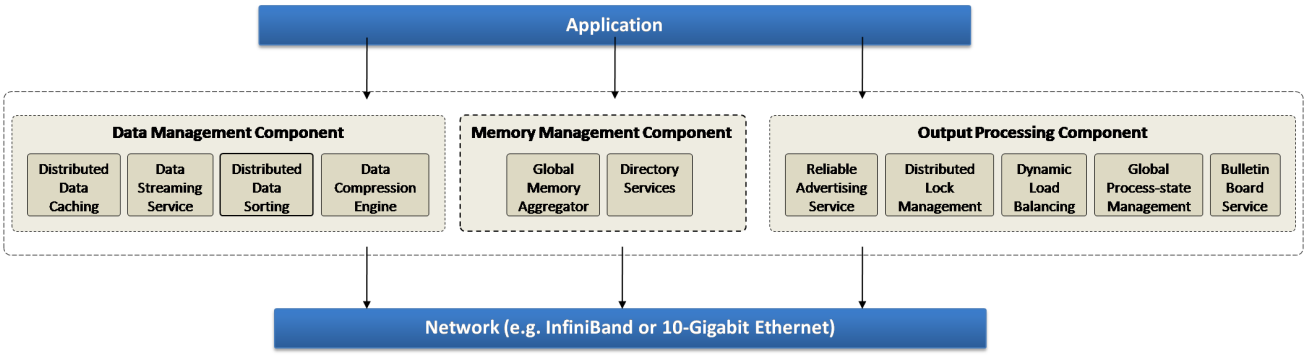

Figure 2: GePSeA Architecture

to trapping I/O calls, automatically figuring out the location, and fetching data is typically not a major portion of the overall I/O cost. Furthermore, implicitly managing data locality makes it simpler and intuitive for applications to use this component. Thus, we chose to hide data locality. 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 filesystem 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 small (e.g., in search algorithms). Thus, to keep the application fed with data, techniques such as prefetching are needed. The data streaming service handles this by swapping out unused data with fresh data that the 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). In addition, because this coordination is completely handled by the GePSeA helper agents, the 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 (de)compressing data. However, together with regular byte-stream compression, this engine also provides capabilities for application-specific compression, as presented in our previous work [4]. Specifically, this engine can either view the data as a stream of bytes or as high-level application-specific objects that are converted to meta-data that is much smaller in size. Once communicated over the network, this meta-data is converted back to the actual data.

3.2 *Memory-Management Utilities*

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

Global Memory Aggregator: Operations like caching, indexing, and searching can all perform better with larger memory. Since the cost of remote memory access 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 or what fragment of the data it currently hosts. The performance and scalability of applications largely depends on how efficiently 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 the

efficient movement of data, this component also handles the synchronization required in order to avoid data corruption.

Reliable Advertising Service: The reliable advertising service addresses 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. This task also includes 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), and 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 race conditions while accessing shared resources. While such locking capability can be provided using the atomic operations 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 bottlenecks that can limit the scalability of parallel applications. This component provides 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 can be used to avoid hot spots but are outside the scope of this paper.

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, we describe a case study using a popular sequence-search application called mpiBLAST.

4.1 *mpiBLAST Overview*

mpiBLAST [7] is one the most popular 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, as shown in Figure **??**. 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 [9].

mpiBLAST, like most parallel sequence-search algorithms, follows a 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.

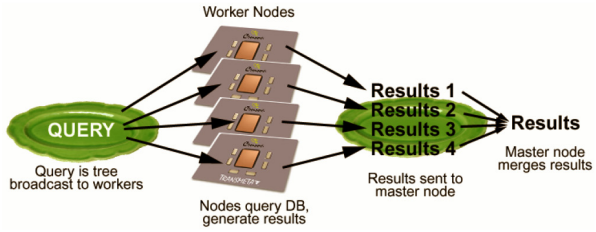

Figure 3: Scatter-Search-Gather Model of Parallel Sequence Search Applications [2]

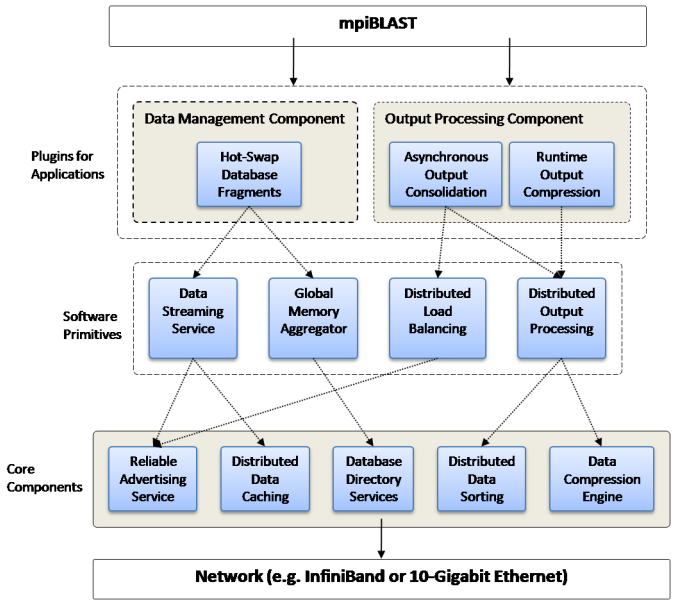

Figure 4: Accelerator Framework used by mpiBLAST

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 plug-ins for mpiBLAST. Using these plug-ins mpiBLAST can offload considerable amount of work to accelerator that can be executed in parallel. Though we developed these plug-ins 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 Plug-In*

This plug-in 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 Plug-In*

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 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 Plug-In*

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 two dual-core AMD Opteron 2218 processors. Thus, 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 NIH GenBank **`nr`** database into 8 fragments using the **`mpiformatdb`** utility. For most experiments, the input query sets containing different number of sequences were chosen randomly from **`nr`** database. For the other experiments, pseudo-random query sets were chosen in order to better control the output size and to better analyze the efficacy of specific features of our accelerator.

Our experimental methodology includes running mpiBLAST *with* an accelerator and *without* an accelerator on a given set of processors on the ICE cluster for the same set of input query set.

5.1 *Accelerator on an Existing Core*

By an existing core, we mean a core that is already running some application process, e.g., worker process. Each node of the cluster has a 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 the **`physcpubind`** utility available

on NUMA machines. Our accelerator (one per node) ran on one of these 4 already-occupied cores, as per the scheduling strategy of the operating system, as shown in Figure 5.

We conducted the experiments running mpiBLAST *with* and *without* an accelerator for 8, 16, 24 and 36 workers, where each worker ran on a separate core. For this experiment, 300 input query sequences were randomly chosen from the **`nr`** database.

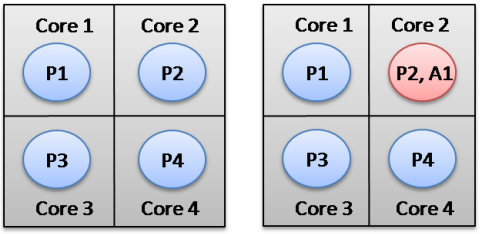

Figure 5: (a) Each worker process runs on a separate core. (b) Core 2 is shared between worker process P2 and accelerator A1

Figure 6 shows significant performance improvement using the software accelerator. For 36 workers, we observed as much as a 2.05x speed-up with an accelerator (per node) against no accelerator. The speed-up increases with increase in worker processes. This improvement is attributed to the overlapping of worker computation and communication (between worker and the writer). Offloading of result merging and writing tasks to an accelerator also contributes significantly to this speed-up. These results were particularly interesting as we are running the accelerator on an oversubscribed core (against running exclusively on a spare core thereby committing it additional system resources) and still observe significant improvement.

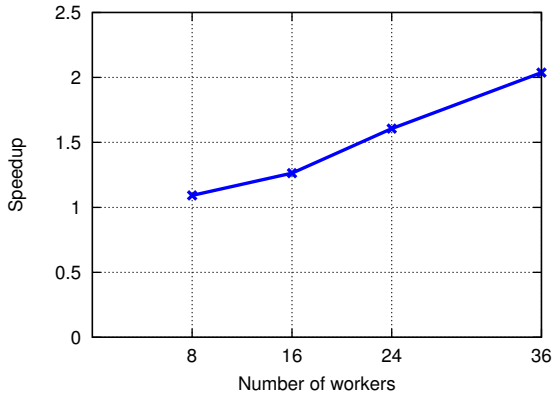

Figure 6: Speed-up: Accelerator on an Existing Core

5.2 *Accelerator on a Spare Core*

By spare core, we mean a core that does *not* run any application process. In this experiment, on a given node, we run 3 worker processes each bound explicitly to a core with 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. Figure 8 shows significant speed-up of mpiBLAST with a maximum of 1.68x for 27 workers. We noticed that while the CPU utilization of a worker process is nearly 100%, the CPU utilization of an accelerator is only 2-5%. Therefore, running the accelerator exclusively on a spare core results in under-utilization of system resources.

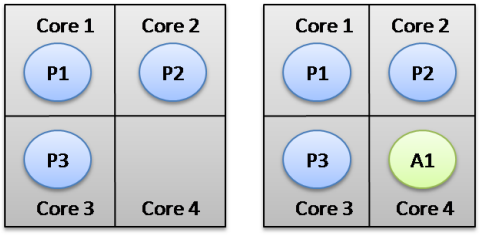

Figure 7: (a) Each of 3 worker processes runs on separate core, core 4 unused (b) Each of 3 worker processes runs on separate core, accelerator A1 on fourth spare core

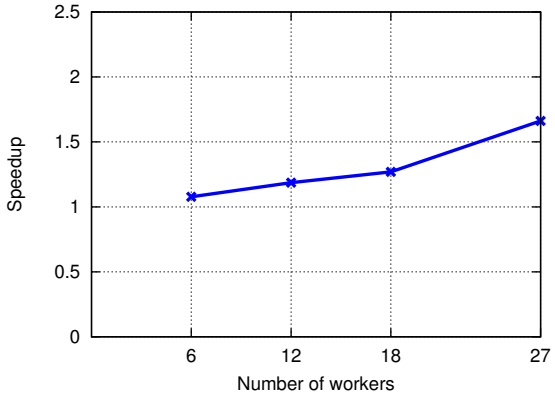

Figure 8: Speed-up: Accelerator on a Spare Core

Next, we use the data from the above experiments to analyze the effect of running mpiBLAST in the following per-node configurations: (1) 4 worker processes running on 4 cores of a node with no accelerator versus (2) 3 worker processes running on 3 cores and an accelerator running on the fourth spare core. Thus, for a 9-node configuration, we compare between 36 mpiBLAST worker processes running on 36 different cores without an accelerator versus 27 mpiBLAST worker processes with an accelerator on each node. Even with one worker less per node, we still see speed-up as much as 1.4x 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.

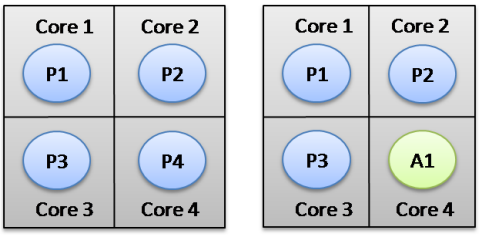

Figure 9: (a) 4 worker processes per node running on separate cores (b) 3 worker processes and an accelerator per node running on separate cores

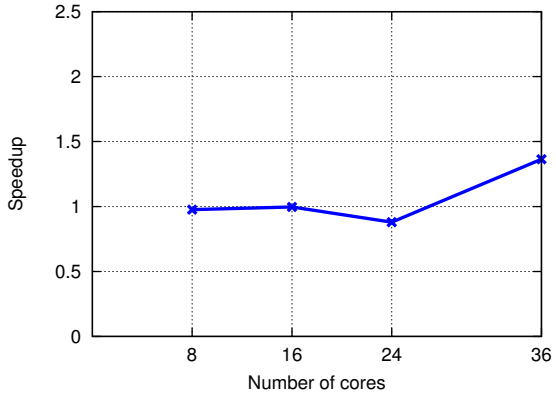

Figure 10: Speed-up using accelerator with different number of workers

5.3 *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 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 consistently.

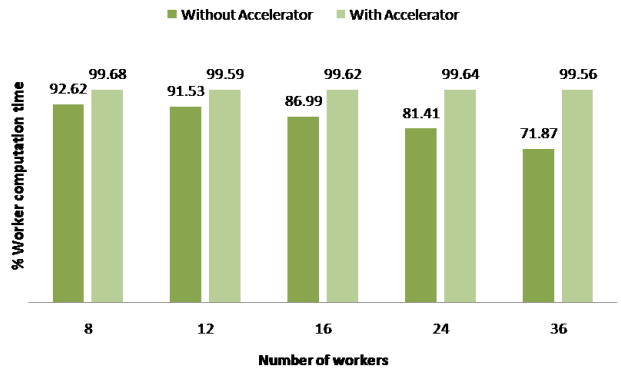

Figure 11: Worker search time as a percentage of total worker time with and without accelerator

5.4 *Asynchronous Output Consolidation*

We tested distributed output consolidation feature provided by accelerator as described in section 4.2.1. In this experiment we compared output result consolidation done by only one accelerator in the cluster (chosen statically) against result consolidation done by each accelerator in the cluster. Results are shown in Figure 12.

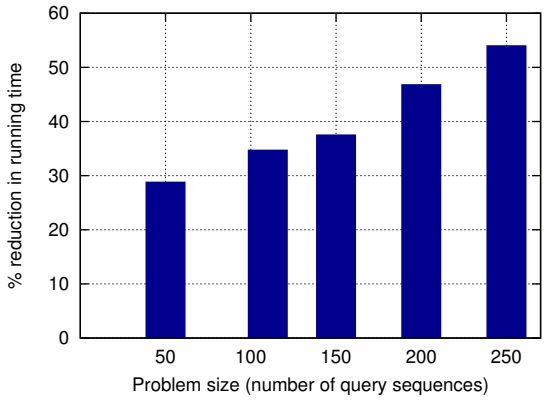

Figure 12: Reduction in running time of mpiBLAST due to asynchronous result consolidation feature

5.5 *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 close to 14% of performance improvement from the Figure 13. With highly *uneven* queries this difference could be very high.

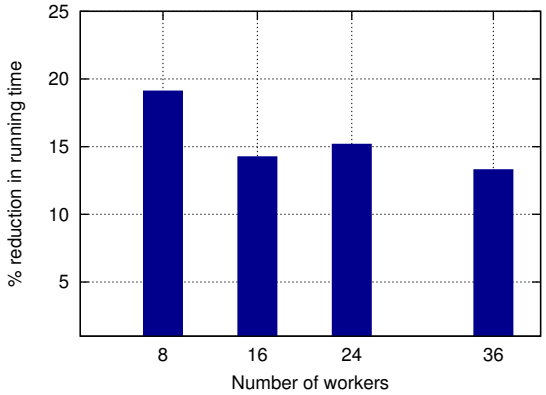

Figure 13: Reduction in running time of mpiBLAST due to dynamic load balancing

5.6 *Run-Time Output Compression*

We tested compression functionality provided by accelerator as described in section 4.2.2. Negative values Figure 14 signify increase in running time of mpiBLAST using the com-

pression engine. This is contrary to our expectations. However, for compression at run time to be effective, network latency 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 make positive impact on the running time. However we do observe a that running time decreases with increase in worker processes.

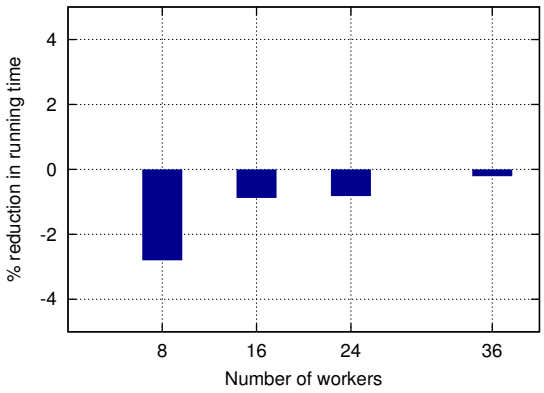

Figure 14: Effect of using compression engine to compress the output at runtime

6 *Conclusion and Future Work*

In this paper, we presented a general-purpose software acceleration framework called *GePSeA* that dedicates a small subset of the available cores on a multi-core 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 GePSeA, we also presented a case study with *mpiBLAST*, an open-source computational biology application and demonstrated more than 205% improvement in the overall application performance.

Our future work would involve extending GePSeA 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] P. Balaji, W. Feng, J. Archuleta, 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.

[5] 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.

[6] Intel Corp. Tera-scale computing.

[7] 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.

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

[9] 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.

[10] 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.

[11] P. Lai, P. Balaji, R. Thakur, and D. K. Panda. ProOnE: A General Purpose Protocol Onload Engine for Multi- and Many-Core Architectures. Technical report, Argonne National Laboratory, 2009.

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

[13] S. Narravula, A. Mamidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations. In *Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, 2007.

[14] 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.

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

[16] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.

[17] 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.

[18] 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.

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

[20] K. Vaidyanathan, S. Narravula, P.Lai, and D. K. Panda. Optimized Distributed Data Sharing Substrate in Multi-Core Commodity Clusters: A Comprehensive Study with Applications. In *Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, 2008.