View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Computer Science Technical Reports @Virginia Tech

A Pluggable Framework for Lightweight Task Offloading in Parallel and Distributed Computing *

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

Multicore processors have quickly become ubiquitous in supercomputing, cluster computing, datacenter computing, and even personal computing. Software advances, however, continue to lag behind. In the past, software designers could simply rely on clock-speed increases to improve the performance of their software. With clock speeds now stagnant, software designers need to tap into the increased horsepower of multiple cores in a processor by creating software artifacts that support parallelism.

Rather than forcing designers to write such software artifacts from scratch, we propose a pluggable framework that designers can reuse for lightweight task offloading in a parallel computing environment of multiple cores, whether those cores be co-located on a processor within a compute node, between compute nodes in a tightly-coupled system like a supercomputer, or between compute nodes in a loosely-coupled one like a cloud computer. To demonstrate the efficacy of our framework, we use the framework to implement lightweight task offloading (or software acceleration) for a popular parallel sequence-search application called mpiBLAST. Our experimental results on a 9-node, 36-core AMD Opteron cluster show that using mpiBLAST with our pluggable framework results in a 205% speed-up.

1 Introduction

Multi- and many-core architectures have become an undeniable reality in high-end computing with nearly every system in the *Top500 List* being multi-core capable. Quad-core and hex-core processors are considered commodity today; simultaneous multi-threading (SMT) processors capable of supporting as many as 64 threads in hardware are also currently available. This trend is only expected to increase in the future as evidenced by Intel's announcements to release the 16-core Larrabee processor in 2009 and the 80-core Terascale processor in 2011 and Sun's announcement to release the next generation of Niagara, which will support up to 2048 threads within the same physical node. On the other hand, hardware accelerators are also becoming increasingly common in high-end computing due to their enormous processing capabilities. However, tapping into these capabilities can incur significant enough overhead to require coarse-grained computation to offset the overhead. In light of these

*This work was supported in part by a Small Business Innovation Research and Small Business Technology Transfer Program of the National Science Foundtion 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.

two trends, we address the obvious question of whether or not a core in a general-purpose, multi-core processor can be used to deliver accelerator-like performance for lightweight tasks.

Therefore, we propose a novel framework that enables lightweight task offloading for parallel applications. Our framework uses a modular design, consisting of three layers — plug-ins, primitives, and core components. These layers compile into a lightweight process, which we refer to as a *software accelerator*. This process acts as a helper agent to the application that executes lightweight application-specific tasks asynchronously and efficiently.

Together with a detailed description of our framework design, this paper also presents a case study with the mpiBLAST bioinformatics application, an open-source parallelization of pairwise sequence search.. Our experiments reveal that applications such as mpiBLAST can significantly benefit from such techniques and demonstrate a 205% speedup.

2 Background

In many parallel applications, data processing is done in a cycle that typically involves three steps. First, input data is assembled from different sources and pre-processed. Second, the actual computation is performed on the input data. Third, the output data from the previous step from different sources is consolidated to obtain single output. Then, a new cycle may begin with a different set of input data.

In this paper, we call the time spent by the application in the second step of each cycle as the application's *computation time* and all other time as *non-computation time*. This non-computation time includes application communication and synchronization time as well as the time spent on doing data management related work such as input and output data processing.

For a small number of compute nodes, the non-computation time of an application may be negligible compared to the computation time. However, as the number of compute nodes increases, synchronization overhead as well as the time to manage application data increases rapidly. Therefore, overlapping the rKcomputation tasks with non-computation tasks by offloading these tasks to helper agents, who can execute these tasks asynchronously and efficiently, can provide sufficient acceleration to the application. For example, while the application is in the computation stage of a particular cycle, a helper agent can asynchronously fetch the input data from different sources for the next cycle, therby eliminating any application stalls.

We developed a generic framework that enables parallel applications to offload non-computation lightweight tasks to the helper agent, which we refer to as a software accelerator. Our research goal is to develop a framework that can be used by any parallel application requiring minimum code change. We present the design of the our framework in the next section.

3 Accelerator Framework Design

Figure 4 shows the layers of our framework for software acceleration. At the top of our framework is the parallel application. The bottom consists of current generation system-area networks such as InfiniBand and 10-Gigabit Ethernet. The intermediate

three layers of our framework represent many important functionalities used by parallel applications. Each of these layers consists of several design components, as represented by the various shaded boxes.

Our software accelerator is a application-helper process that implements the framework layers shown in Figure 2. As mentioned, using our accelerator, applications can offload certain lightweight tasks to the accelerators who then execute these tasks asynchronously and efficiently while applications can continue with their respective processing. Each node in the cluster runs exactly one accelerator that services all the application processes running on that node. On multi-core systems, the accelerator runs on an under-utilized core, thereby effectively utilizing the system cores.

Each application process on a node registers itself to the accelerator running on that node at the application start-up time. Figure 1 shows three-node cluster with each node running three application processes. Application processes are connected to the accelerator running on that node. Also, the accelerator on each node is connected to accelerators running on all the other nodes of the cluster. We now describe the design of various components our framework.

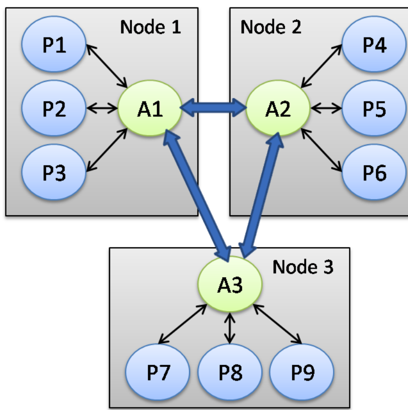


Figure 1: Application-Accelerator Interconnections

3.1 Application Plug-Ins

This layer resides between the application and lower layer called the software primitives. Application plug-ins are primarily designed to be simple and abstract utilities that can be utilized by the applications. These plug-ins are high-level services that are developed over services provided by the software primitives. Our framework provide infrastructure to quickly build application specific plug-ins using underlying software primitives.

We broadly classify the application plug-ins into three different areas, namely input processing, application data management and output processing. Each of these area can have any number of plug-ins depending on application need.

3.1.1 Input Processing

Input processing plug-ins primarily deal with the processing of input data. For example, plug-in can be designed to prefetch the input data for the application. More complex features such as dynamic group management and database pruning can also be

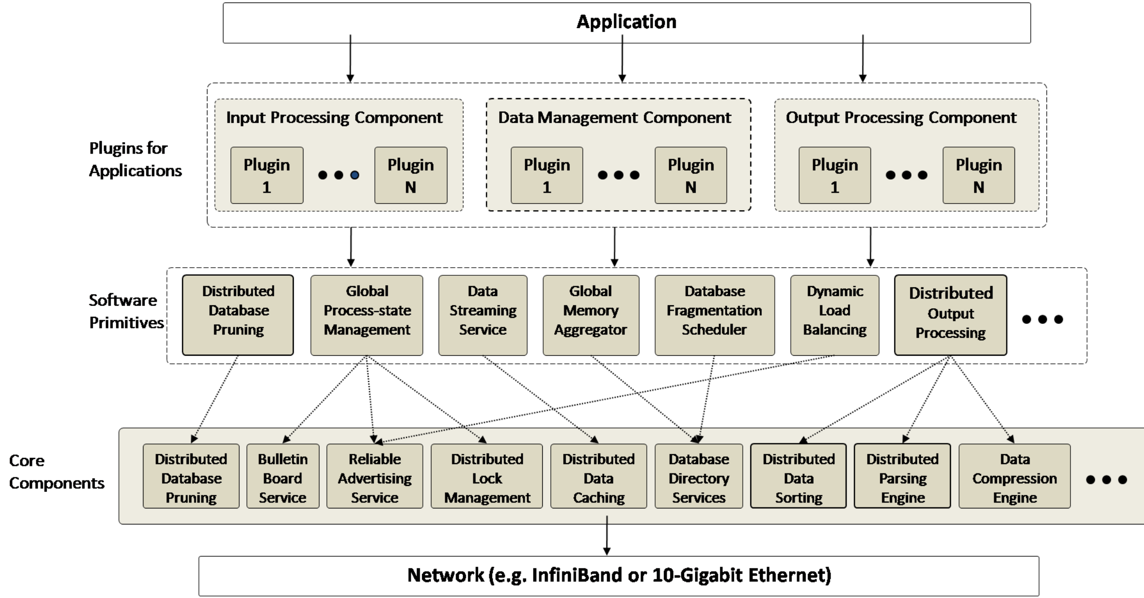


Figure 2: Accelerator Engine Framework

designed depending on application needs. Dynamic group management, for example, can dynamically partition the cluster into a number of sub-clusters. For applications that involves searching against databases, database pruning and indexing plug-in can be used to prune and index the database dynamically.

3.1.2 Application Data Management

Application data management plug-ins primarily deal with the management of the existing application data. For applications that involves databases, plug-ins related to database management such as plug-in to move the database fragments in the cluster or to create new database fragments dynamically can be designed. We developed Hot-Swap database fragment plug-in for mpiBLAST application described in section 4.2.3.

3.1.3 Output Processing

Output processing plug-ins primarily deal with the processing of the results. Plug-ins in this area can be designed, for example, to merge the data from multiple nodes in the cluster asynchronously or to perform runtime compression or uncompression of output data. We implemented both these plug-ins for mpiBLAST application described in sections 4.2.1 and sections 4.2.2.

3.2 Software Primitives

The software primitives are designed to provide generic capabilities that can be used by the various upper-level services. While a number of primitives are possible, refer Figure 2, we designed some of the more challenging ones. We describe some of them in some detail.

3.2.1 Global Memory Aggregator

Many memory intensive operations require larger memory for better performance. Since the cost of remote memory access can be lower than the cost of disk access, these components can effectively utilize the free memory available on the other nodes. The global memory aggregator allows applications to utilize the memory on the entire cluster. Specifically, this primitive presents a global address space to its upper layers and maintains the mappings of these locations on the global address space with the actual node and physical address of these locations. This primitive utilizes the services of distributed directory service core component.

3.2.2 Global Process-state Management

Managing the data processing performed by parallel applications requires sharing certain information about each node, such as whether node is idle, waiting for data, what database fragment it currently hosts, and various others, among different nodes on the system. The global process-state maintains an up-to-data and complete information about the status of the different nodes in the cluster. This primitive utilizes the services of three core components namely bulletin board service, reliable advertising service and distributed lock management described in section 3.3.

3.2.3 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. Dynamic load balancing primitive provide mechanism to balance the load on each node. This primitive uses the services of the reliable advertising service core component to keep track of availability of all the nodes in the cluster or sub-cluster. Under this technique, after a node has finished its previously assigned work, it announces its availability to a special node called the *scheduler*. The *scheduler* assigns the work to each node based on its availability. Nodes periodically query the *scheduler* to obtain the information about the work assigned to them and to other nodes in the cluster. Load balancing primitive can be utilized by applications to balance the computation load on each node and by the accelerators to balance the load among each other.

3.2.4 Data Streaming Service

This primitive is responsible to keep the search algorithm fed with data. For applications using databases, it can be used by the applications to fetch the required data beforehand from other nodes and thus avoid stalls. It can also asynchronously pre-caches the database fragment that would be required by the application for next computation task. This primitive utilizes the data caching service core component.

3.3 Core Components

Core components consists of core features that are provided by accelerator framework. These features are simple yet powerful tasks that can be utilized by many upper layers. While we found the core components shown in Figure 2 meets the requirements

for most applications, new core components can easily be added that implement specific functionality. Below we describe some of the core components very briefly..

Reliable Advertising Service: Reliable advertising service enables communication between accelerators in the cluster and is responsible for reliable delivery of messages among nodes in the cluster. It also implements multicasting.

Bulletin Board Service: Bulletin board service provide addressable memory that can be read or written to by any node in the cluster system.

Distributed Lock Management: This service allows lock-based synchronization between multiple nodes to avoid various race conditions while accessing shared resources.

Distributed Data Caching: For applications that uses databases, this task provides a caching multiple database fragments into the local memory of the node. Nodes within cluster or sub-clusters together will host complete database.

Database Directory Service: It provides an index of the databases that are present in the entire physical cluster system.

Distributed Data Sorting: This service sort and/or merge the data incrementally that is distributed across multiple nodes in parallel.

Distributed Parsing Engine: The data parsing engine provides capabilities to parse through data generated by the application and extract relevant information from it. For example this can be used to extract meta-data from the output generated by the application.

Data Compression Engine: This utility can be used to compress and uncompress the data.

4 Case Study

We used a popular bioinformatics genetic sequence search application called mpiBLAST as a case study to test our accelerator framework. mpiBLAST is one of the most widely used tool for the sequence search by the biosicentists.

4.1 mpiBLAST Overview

The parallelism of mpiBLAST search is based on database segmentation. Before the search, the raw sequence database needs to be formatted, partitioned into fragments, and stored in a shared storage space. mpiBLAST organizes parallel processes into one master and many workers. The master uses a greedy algorithm to assign un-searched fragments to workers. Then the workers copy the assigned fragments to their local disks (if available) and perform BLAST search concurrently. Upon finishing searching one fragment, a worker reports its local results to the master for centralized result merging. The above process repeats

until all the fragments have been searched. Once the master receives results from all the workers for a query sequence, it formats and print out results to an output file.

mpiBLAST like most parallel sequence search algorithms follow scatter-search-gather model. *Scatter* stage consist of query and/or database segmentation. In *Search* stage each worker searches the query against the assigned database portion. Finally, *Gather* stage consist of merging of output results from individual workers.

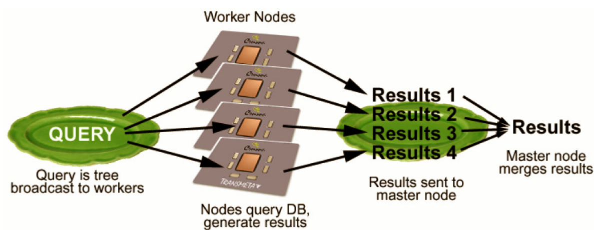


Figure 3: High level view of parallel sequence search applications

4.2 Implemention

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

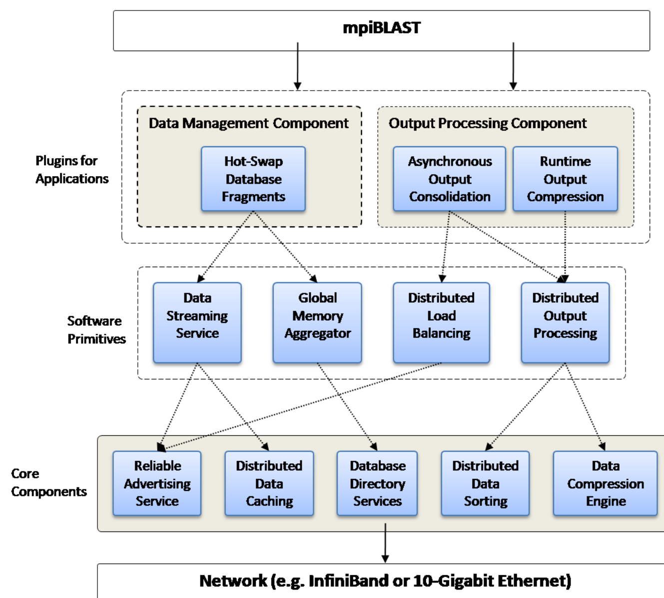


Figure 4: Accelerator Framework used by mpiBLAST

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 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.

4.3 Experimental Evaluation

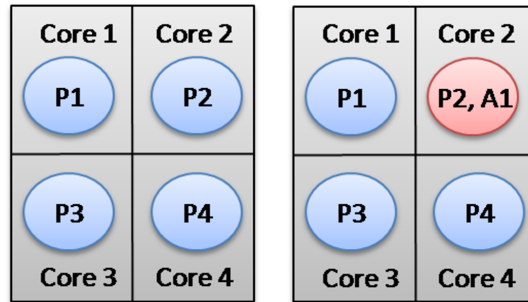
In this section, we evaluate the accelerator for mpiBLAST and do a performance analysis. For all our experiments, we used the GenBank **nr** database, a protein type repository frequently searched against by bioinformatics researchers. The size of the raw **nr** database is nearly 1 GB, consisting of 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.

4.3.1 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.



(a) Each worker process runs on a separate core

(b) Core 2 shared between worker process P2 and accelerator A1

Figure 5: Node configuration for 4.3.1

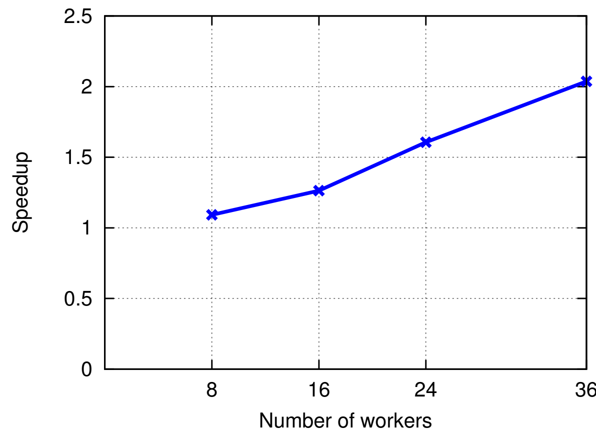


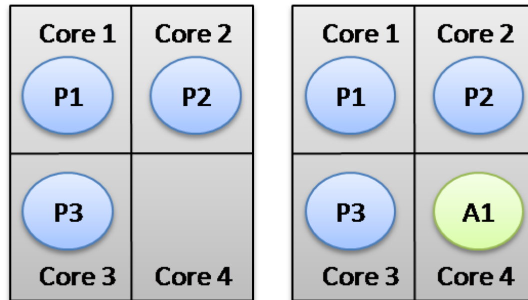
Figure 6: Speedup when accelerator run on existing core

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.

4.3.2 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.



(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 7: Node configuration for 4.3.2

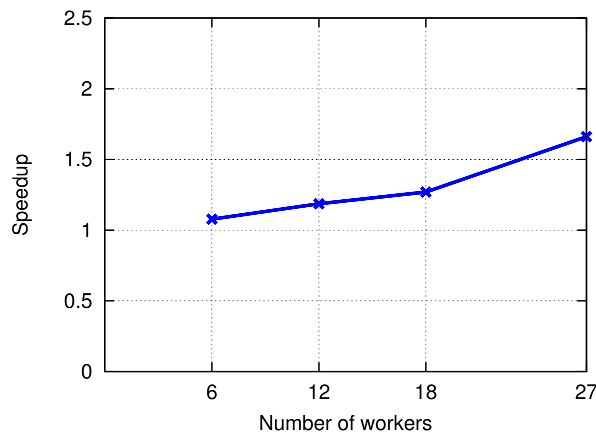
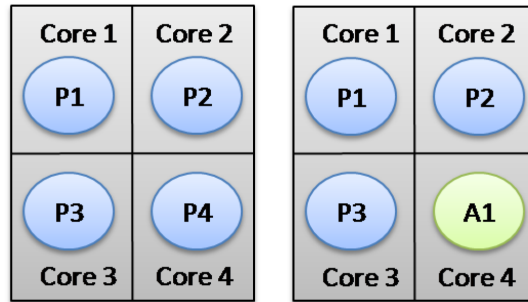


Figure 8: Speedup when accelerator runs on spare core

4.3.3 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.



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

Figure 9: Node configuration for 4.3.3

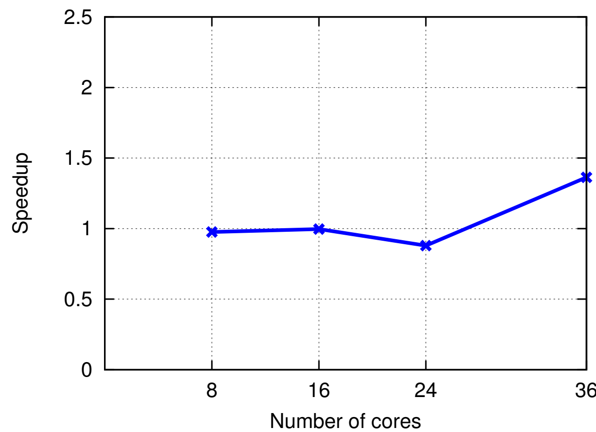


Figure 10: Speedup using accelerator with different number of workers

4.3.4 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.

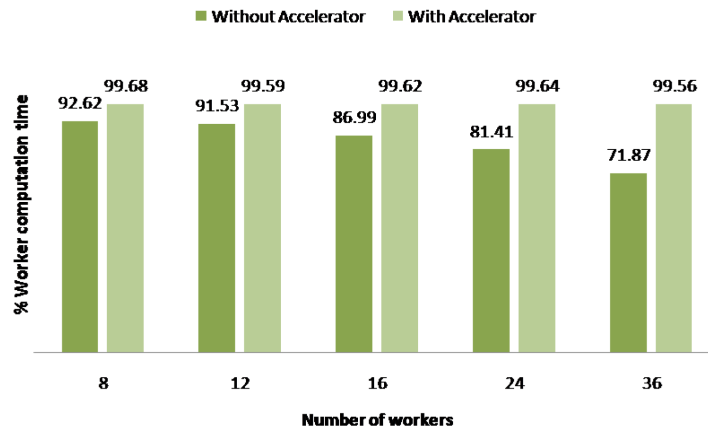


Figure 11: Worker search time as a percentage of total worker time with and without accelerator

4.3.5 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.

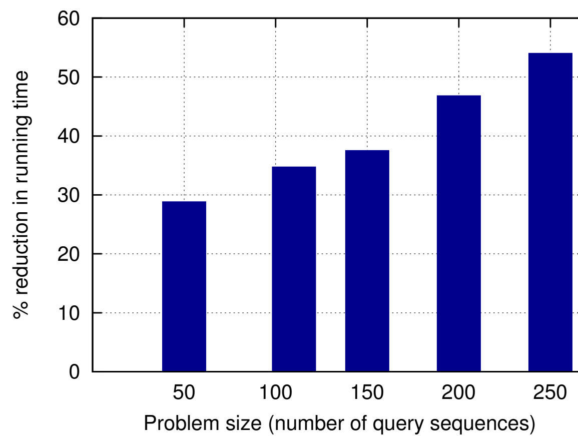


Figure 12: Reduction in running time of mpiBLAST due to asynchronous result consolidation feature

4.3.6 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 14. With highly *uneven* queries this difference could be very high.

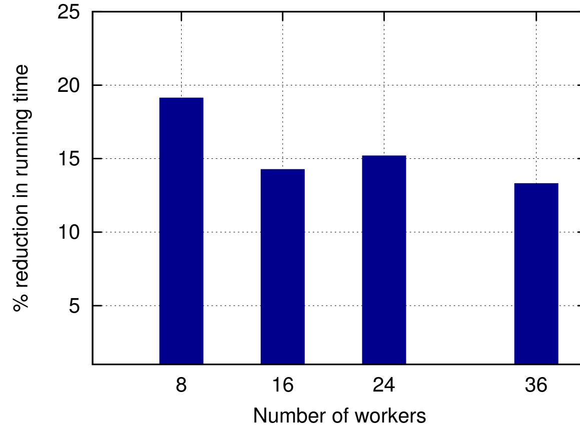


Figure 13: Reduction in running time of mpiBLAST due to dynamic load balancing

4.3.7 Runtime output compression

We tested compression functionality provided by accelerator as described in section 4.2.2. Negative values Figure 15 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 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 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.

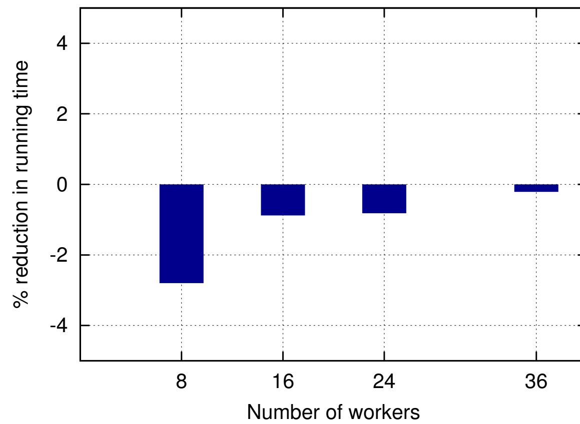


Figure 14: Effect of using compression engine to compress the output at runtime

5 Related Work

While there have been many recent research efforts both on developing hardware accelerators [14][15] and developing frameworks and libraries [7][8][9] 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 [7][9] 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 [10][11], 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 [12] authors developed a communication engine to exploit the core in multi-core systems using various multithreading 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.

6 Conclusions and Future Work

In conclusion, our experience of using our software accelerator for mpiBLAST application has been very encouraging. Here are our main contributions of the paper.

First, we have developed a 3-layered framework for software acceleration for parallel applications. We have designed and implemented many core components and software prmitives that are the building blocks of our framework. Using these, many important and innovative high-level features and services can quickly be developed. These features can be reused by the parallel applications with minimum code change in the application thereby saving the efforts required to design and develop these features from scratch. We modified at the most 25 lines of code in the mpiBLAST application to integrate it with our framework.

Second, we developed software accelerator using our framework. We have tested our accelerator for bioinformatics application called mpiBLAST. Our results show signficant speedup of 205% for 36 workers. Therefore the accelerator can be etremely useful in offloading the lightweight tasks by the application. We have shown with mpiBLAST as case study, overlapping

computation and non-computation tasks of the application can provide considerable acceleration.

Third, we have shown in section 4.3.1 that using our accelerator in multi-core systems give significant speedup even when it runs on an oversuscribed core. We believe that our accelerator could be extremely beneficial in multi-core systems where one or more core may be under-utilized. Accelerators exloits the system cores by running on a under-utilized core thereby eliminating the CPU contention with other application processes. CPU utilization of accelerator would increae with more work being offloaded to the accelerator. We have seen during our experiments that for query sequences producing large result set, the CPU utilization of accelerator can go as high as 20% or even more in some cases. With higher CPU utilization, there is a strong case for accelerator on multi-core systems.

Lastly, we developed three important plugins for mpiBLAST applications as described in section 4.3. We believe that these plugins can also be used by other sequence search algorithms like SWAT and HMMER that follow *scatter-search-gather* model followed by mpiBLAST.

Our future work would involve extending our framework to support new abstract features so that the accelerator can be used by more and more applications. These new features would be implemented and integrated with existing modules of the framework. For this paper, we have used bioinformatics application mpiBLAST as case study to test our acceleration framework. Our future work would be to use the accelerator with other parallel applications and conduct extensive performance and usability analysis.

Acknowledgements

This work was supported in part by 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.

References

[1] John R. Douceur, Jon Howell. Distributed directory service in the Farsite file system. In *proceedings of the 7th symposium on Operating systems design and implementation*, 2006

[2] William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA,1999

[3] Kishida, H. Yamazaki, H. SSDLM: architecture of a distributed lock manager with high degree of locality for clustered file systems. In *IEEE Pacific Rim Conference on Communications, Computers and signal Processing,* 2003.

[4] 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

[5] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel BLAST. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS05),* Washington, DC, USA, 2005. IEEE Computer Society.

[6] Delisle, M. Krajecki, M. Gravel, and C. Gagne. Parallel implementation of an ant colony optimization metaheuristic with openmp. In *Proceedings of the 3rd European Workshop on OpenMP (EWOMP01),(Barcelone, Espagne)*, 2001

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

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

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

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

[11] Mao Jiayin Song Bo Wu Yongwei Yang Guangwen. Overlapping Communication and Computation in MPI by Multi-threading. In Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications, 2006

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

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

[14] 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,* Research on Integrated Systems, 1993.

[15] W. Feng. Green destiny + mpiblast = bioinfomagic. In *International Conference on Parallel Computing (ParCo),* 2003

[16] K. Hokamp, D. Shields, K. Wolfe, and D. Caffrey. Wrapping up BLAST and other applications for use on Unix clusters. *Bioinformatics, 19(3), 2003.*

[17] A. Shpuntof and C. Hoover, *Personal Communication,* August 2002.