An Investigation of I/O Strategies for MPI Workloads

Sanya Attari

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science and Applications

Ali R. A. Butt, Chair
Deborah G. Tatar
Layla Nazhandali

December 2, 2010
Blacksburg, Virginia

Keywords: I/O, performance, mpi
Copyright 2010, Sanya Attari

An investigation of I/O strategies for MPI workloads

Sanya Attari

(ABSTRACT)

Different techniques could be used for improving application performance in parallel systems. Studies have been shown that I/O communication delay is the main reason for different behavior of I/O intensive applications with specific requirements for performance optimization. So, using common strategies, generally defined and effective for computationally intensive applications may not have the same effect on performance improvement for these applications. Moreover, background system configuration effects on the behavior of the application and its performance. Growing use of parallel multi-core systems is an important factor in increasing performance and speeding up the applications. Since changing multi-core systems hardware is not an efficient method in satisfying different expectations of unique application, it is application developer's responsibility to design flexible and scalable code that is compatible with different environments. On the other hand, predicting application behavior and I/O requirements for I/O intensive applications with irregular communication patterns is a complicated and time-consuming task that pushes the problem to runtime impacts.

Addressing this issue, we provided an overview on different techniques used for solving this problem. We have studied I/O bound parallel applications that use MPI as the communication method in order to define a general perspective to optimize cost performance ratio. Our designed experiments cover different setups for these applications in order to define various criteria that should be considered in design stage as well as runtime. Moreover, targeting one of the popular I/O intensive applications, we have discussed some possible solutions to speed it up on a multi-core system.

Acknowledgments

I am heartily thankful to my supervisor, professor Ali Butt, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject. Without his guidance and persistent help this thesis would not have been possible.
I gratefully acknowledge professor Deborah Tatar for her advice, supervision, and crucial contribution, which made her a backbone of this research and so to this thesis. Her involvement with her originality has triggered and nourished my intellectual maturity that I will benefit from, for a long time to come.
Many thanks to professor Leyla Nazhandali, whose valuable feedbacks and supports helped me in accomplishing this work.
I thank Department of Computer Science at Virginia Tech and the staff of the graduate program office especially Ms. King for being accommodating to my queries.
My parents deserve special mention for their inseparable support and prayers, showing me the joy of intellectual pursuit ever since I was a child and raising me with their gently love.
Last but not the least I express my appreciation to my very special friend, Babak, for all his supports and being there in rough life time to encourage me continuing my way.

Contents

1 Introduction **1**

1.1 Contribution 1

2 Background **4**

2.1 Mpi 4

2.2 MpiBLAST 5

2.3 SystemG(Green) 6

3 Related Works **8**

4 Design **17**

5 Evaluation **21**

5.1 Lessons Learned 34

6 Discussion **35**

6.1 Conclusion 35

6.2 Future Direction 36

Bibliography **37**

List of Figures

1.1 Computation-I/O Gap . . . 2
1.2 Speedup obtained for an I/O intensive application: mpiBLAST . . . 3

2.1 mpiBLAST:general view . . . 6

3.1 Possible performance bottleneck locations in a system . . . 9
3.2 cluster architecture . . . 9
3.3 MP computer with DCS nodes . . . 12
3.4 multilevel cache hierarchy . . . 12
3.5 Master-worker model in mpiBLAST . . . 14
3.6 Multiple masters architecture . . . 15

4.1 3 levels hierarchy in mpiBLAST application . . . 18

5.1 General performance model on multi-core systems . . . 22
5.2 Execution time for locally saved nt database on 14 nodes . . . 23
5.3 The effect of local data vs. nfs data on performance . . . 23
5.4 The effect of computational distribution on I/O intensive applications . . . 24
5.5 The effect of spreading cores without data correlation . . . 25
5.6 The impact of data locality on execution time . . . 26
5.7 Assigned data for different number of machines . . . 27
5.8 Centralized data impact on execution time . . . 28
5.9 Decreasing task by adding machines to application . . . 28
5.10 Reading centralized data with/without using cached fragments . . . 29

5.11 Increasing execution time using "copy-via-none" option of command 30
5.12 Decreasing workload by involving more nodes 30
5.13 Total execution time divided to 'I/O' and 'computation part' 31
5.14 Increasing cores doesn't always help 31
5.15 Parallel vs. sequential run in 4 applications 32
5.16 Overall execution time for 4 parallel/sequentially executed applications . . . 32
5.17 Parallel vs. sequential run in 8 applications 33
5.18 Overall execution time for 8 parallel/sequentially executed applications . . . 33

List of Tables

2.1 Systemg:Virginia Tech supercomputer 6

4.1 Dedicated I/O vs. Static Assignment 20

Chapter 1

Introduction

Large parallel systems have complex architectures making it a challenge for optimizing performance for these applications. Tools are needed that collect and present relevant information on application performance in a scalable manner enabling developers to identify and determine the cause of performance bottlenecks from time and hardware efficiency point of view. On the other hand, applications can be categorized in two computationally intensive and I/O intensive groups, based on the amount of work that should be done for transferring data and processing it. Recent advances in parallel computing and multi-core system designs improve computational applications performance rapidly. Although I/O bound applications can have benefits of being executed in such systems, their performance improvement rate is not as much as expected rate. One reason is computation-I/O gap, i.e., computational power is growing fast while I/O and storage devices have lower advance rate. Figure 1.1 illustrates this concept. In other word, "Simply assigning cores to an application does not scale": End-to-end application performance is not expected to grow linearly with the number of cores [41]. Massive I/O, storage hierarchy problems, and communication requirements in these applications are considered the main reasons for having similar performance after involving a threshold number of cores. Moreover, other source of overhead in the system added to this is the overhead from using complex programming techniques both in symmetric [19, 60, 56, 14, 57] and asymmetric [61, 62] multi-cores. Figure 1.2 shows the speedup achieved with increasing number of cores for mpiBLAST [16], an I/O intensive application, observed by [10]. As it has been illustrated in this figure, using increasing number of cores does not provide corresponding improvement in execution time for selected workloads.

1.1 Contribution

The contribution of this work is evaluating different I/O techniques and approaches for solving performance and utilization issues for I/O intensive applications when running on

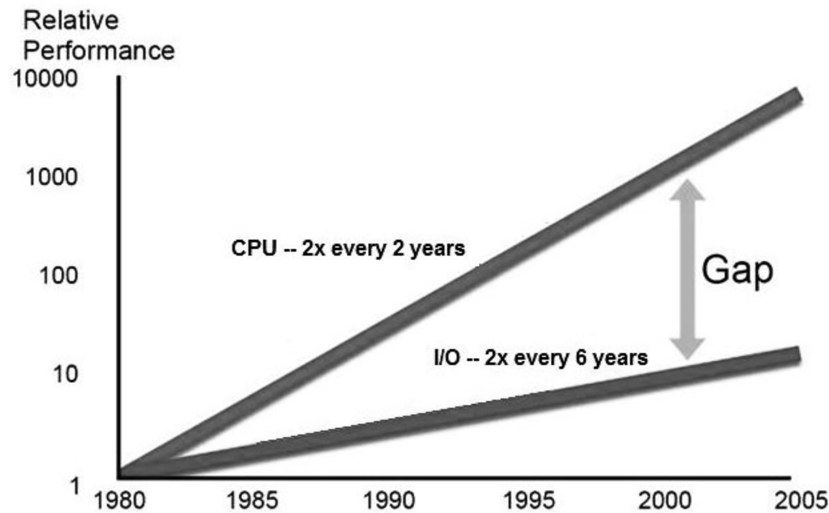

Figure 1.1: Computation-I/O Gap

multi-core systems in order to demonstrate an overall picture of these applications behavior. The main problem that this work is concerned about can be defined as: "I/O intensive applications need transferring massive amount of data in order to reach final results in a multi-core system which makes communication delay the main barrier in improving performance". Improving computational power is a method to improve performance. But this is not an efficient solution for any task on any system configuration. As an example in I/O limited applications, with less computation and more communication, there should be different plans to speedup the application. This problem is exacerbated on multi-core machines where the computation power is distributed in a system and internal network limitations decreases data exchange rate. One issue we have concerned about is that no analytical work has been done in order to find an efficient I/O distribution patterns that cause performance improvement. In other words, the best scenario to increase cost performance ratio would be defining application specific I/O patterns before any execution on different system configurations. We have investigated different I/O distribution methods and their effects on applications. The goal of this study is providing a general view about managing massive data and its decomposition techniques in order to reach better execution time and resource usage. Whether spending some times in copying entire data in a multi-core system would speed up the execution or using perfecting techniques could decrease the time. The necessity of predicting application behavior on different system configurations is a blind side of today's research which is the main goal of this work. Our experiments prove that how different could be results in running one application with different I/O distribution patterns. Moreover, we have studied different techniques on I/O intensive applications and proposed new disciplines to improve application performance. We have used general concepts of load balancing as well as dedicated I/O nodes and implemented them on our open source application. For load

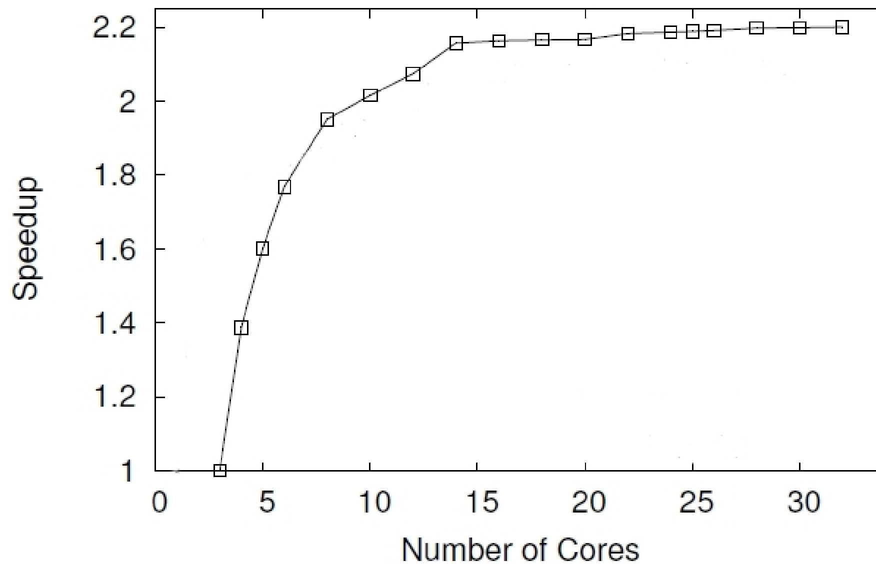

Figure 1.2: Speedup obtained for an I/O intensive application: mpiBLAST

balancing techniques considering the fair distribution of high computational power among nodes, we assigned same workloads predefined by the algorithm in the way that job scheduler has a list of tasks for each node. This technique decreases the task managing issues on system and there will be more time to gather the results and form the final output. Having centralized policy in current implementation of application to distribute task and aggregate output is a bottleneck in system and that has been degraded using proposed load balancing technique. Also, in order to facilitate I/O communication between different layers of application hierarchy we assigned a percentage of cores as dedicated nodes to do I/O tasks for others. These nodes are responsible for communicating with the central node and prepare the next assigned data for related nodes so that data will be ready to use by completing the current task on each core. This method can decrease the execution time by providing data without communication requirements on compute nodes. Although proposed schemes can solve some problems, it should be mentioned that more studies are required for improving system efficiency on various configurations.

We will discuss the main system components in Chapter 2, and a brief overview of recent approaches for solving I/O problem in Chapter 3. Different experimental designs in this work will be presented in Chapter 4 and Chapter 5 shows the evaluative results for designed experiments. We will conclude this work in Chapter 6 providing future paths in this topic.

Chapter 2

Background

Parallel computing plays an increasingly critical role in advanced scientific researches as simulation and computation are becoming widely used to augment and replace physical experiments. The gap between peak and achieved performance for scientific applications running on large parallel systems has grown considerably in recent years. Most common parallel programming paradigm for these applications is to use C with MPI message passing to implement parallel algorithms. In this chapter we will introduce the main components of this study including the I/O intensive application we used and the system we ran the application on. While the main goal in this work is discussing the problems that I/O intensive applications have from performance aspects, we selected one of the most popular sceintific applications called mpiBLAST [12, 2, 22, 58]. We studied different configurations and system settings in which we could run mpiBLAST in order to make an in depth view of its behavior. Since, mpiBLAST has been implemented on mpi and uses BLAST as the main algorithm, we will also provide an overview of these methods. Moreover, we had SystemG, a supercomputer in Virginia Tech, as the multi-core testbed to run the application. From 300 nodes on this machine, 16 were assigned to this work.

2.1 Mpi

Message Passing Interface is an API specification that has become the standard protocol for application layer communication in cluster environments [3]. Because the number of processes in an MPI computation is normally fixed, our focus is on mechanisms used for transferring data among processes. Processes can use point-to-point communication operations to send a message from one named process to another; these operations can be used to implement local and unstructured communications. A group of processes can call collective communication operations to perform commonly used global operations such as summation and broadcast. MPI's ability to probe for messages supports asynchronous communica-

tion [23].

Since MPI is now widely accepted standard for message-passing parallel computing libraries, both applications and important benchmarks are being ported from other message-passing libraries to MPI. In most cases it is possible to make a translation in a fairly straightforward way, preserving the semantics of the original program. On the other hand, MPI provides many opportunities for increasing the performance of parallel applications by the use of some of its more advanced features, and straightforward translations of existing programs might not take advantage of these features. New parallel applications are also being written in MPI, and an understanding of performance-critical issues for message-passing programs, along with an explanation of how to address these using MPI, give the application programmers the ability to provide a greater percentage of the peak performance of the hardware to their application.

2.2 MpiBLAST

BLAST is one of the most popular computational biology tools [55] providing the capability to compare all possible combinations of query and database sequences by translating sequences on the fly. MpiBLAST is bilogical software to find similarities among gene sequences and make it possible to find out any relationship between genomic stracture and different disease helping scientists to make better decisions. Extensive use of MpiBLAST has made it very useful tool in biological researches and had more than 40,000 downloads [32] in recent years and is officially supported application at high performance computing facilities such as [1] and [6]. It is a large-scale genomic sequence search implemented through an open source application with a highly irregular I/O behavior. The original design is a database segmentation approach that defines a two level hierarchy and divides cores into master and worker levels. Master node(s) is responsible to assign tasks, which is a combination of database fragment and query sequence, and workers concurrently run search to find similarities. All results are gathered as the output to file system on master node. With small or moderate number of concurrent workers the speedup has a good rate for mpiBLAST but centralized output processing approach restricts application scalability [38] when the number of cores has an incresing pattern. Open source parallelization of BLAST, mpiBLAST, fragments the database and distributes them among clusters so each node compuet specific data partitions, i.e., searching the similarities on unique part of the entire data [26]. Different databases are available in `ftp://ftp.ncbi.nih.gov/blast/db/FASTA` containing different genomic information, a database could be protein or nucleotide with different sizes. For experimental part of thos work we used nt, the second largest database, which is the nucleotide database containing GenBank, EMB L, D, and PDB sequences. The database is an unordered ASCII flat file with daily updates and the size used for this experiments is 32GB. Original implementation of this applications has been shown in Figure 2.1.

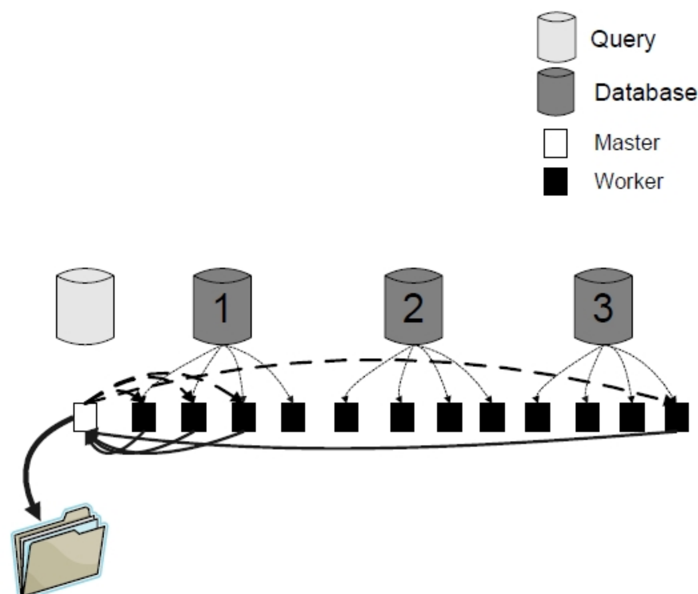

Figure 2.1: mpiBLAST:general view

2.3 SystemG(Green)

Systemg features	
System model	Apple Mac Pro Cluster
Processor	Intel EM64T Xeon E54xx(Harpertown) 2800 MHz(11.2 GFlops)
Main memory	2592 GB
Operating system	Linux
Vendor	self made
Interconnect	Infiniband
Site	Center for High-End Computer Systems, Virginia Tech

Table 2.1: Systemg:Virginia Tech supercomputer

The multi-core system used in this work is Systemg [7, 5] that is placed in Center for High-End Computer Systems, Virginia Tech. SystmG provides a research platform for the development of high-performance software tools and applications with extreme efficiency at scale. System G is a cluster supercomputer at Virginia Tech consisting of 324 Apple Mac Pro computers with a total of 2592 processing cores. It was finished in November 2008 and ranked 279 in that month's edition of TOP500, running at 16.78 teraflops and peaking at

22.94 teraflops. It now runs at a "sustained (Linpack) performance of 22.8 TFlops". It transmits data between nodes over Gigabit Ethernet and 40Gb/s Infiniband (Table 2.1). Each of the 324 Mac Pro machines contains two quad-core 2.8 GHz Xeon processors and 8 gigabytes (GB) of ram. System G's name stems from its homage to System X and to its focus on Green Computing. The cluster has thousands of power and thermal sensors to test high performance computing at low power requirements. It is the largest power-aware research system in the world. A summarized information about this system is as follow:

- 325 Mac Pro Computer nodes, each with two 4-core 2.8 Gigaherts (GHZ) Intel Xeon Processors.
- Each node has 8 GB random access memory. Each core has 6MB cache.
- Mellanox 40 Gb/s end-to-end InfiniBand adapters and switches.
- LINPACK result: 22.8 TFLOPS (trillion operations per sec)
- Over 10,000 power and thermal sensors.
- Variable power modes: DVFS control, Fan-Speed control, concurrency throttling, Dynamic system temperature control.
- Intelligent Power Distribution Unit: Dominion PX

In summary usign part of systemg, 16 nodes, and open source mpiBLAST we studied different I/O strategies for an I/O intensive application on a multi-core system. In addition to run the bassic implementation of the application, which clears blind aspects of application and system effect on that, we have implemented two new techniques in order to reach better performance and utlization. We will discuss about these in future chapters.

Chapter 3

Related Works

Advances in multi-core system development causes the growing gap of I/O and computation as applications is scaling to different cores within a single node. This situation is what makes I/O a performance bottleneck in multi-core systems. In spite of having enough power sources, cores, they may not yield expected performance benefits. There are numerous application activities, e.g., checkpointing, file reformatting, etc., in a typical application workflow that can benefit if just a few cores were exclusively allocated for I/O part. Prior work in this area has often been relegated to application-specific solutions of running a few support threads on cores with less attention to runtime solutions for this problem.

There has been many studies to increase system performance in different system configuration that are not beneficial in the case of I/O intensive applications because of their unique behavior independent from computational power. There should be an indepth study in these applications I/O pattern, different for various environment, to run the more efficiently on multi-core systems. In this chapter we will have a summarized overview on some studies and main issues discussed in previous researches.

I/O interconnect bandwidth and low throughput could be considered as storage and I/O performance bottlenecks which are not limited to large enterprise or scientific high performance computing systems. Performance bottleneck can be located in different parts of system as shown in Figure 3.1. I/O bottlenecks can result in increased response time and latency in applications. By adding more workload to a system with existing I/O issues, response time will correspondingly increase.

Gennaro in [27] modeled an analysis of performance of I/O bound parallel applications executed on clusters of workstations with centralized I/O resources. Figure 3.2 shows the general model for cluster architecture. Having buffer can improve I/O phase since it allows application to send data to the I/O channel with no need for waiting to write data on disk [27]. [27] has studied qualitative and quantitative behavior of I/O intensive applications and found a speedup surface model that can be used in finding the effect of processors and

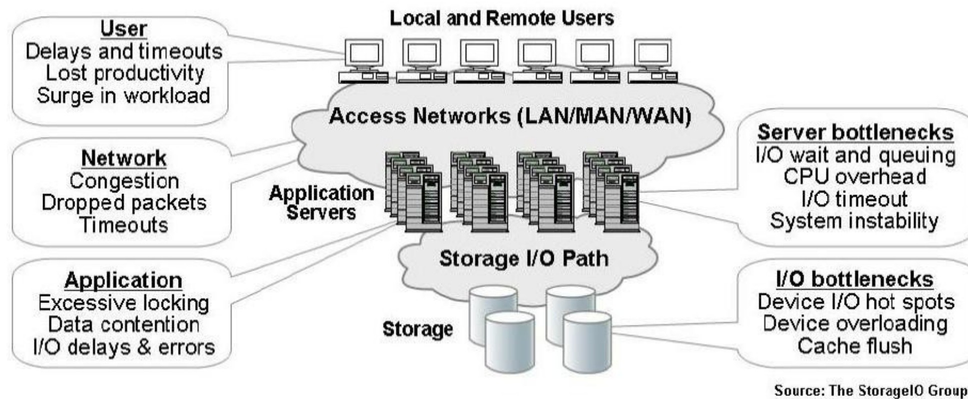

Figure 3.1: Possible performance bottleneck locations in a system

disk parallelism on the performance of applications. This model also used in defining the impact of I/O buffer and caching mechanisms on the speedup of parallel programs.

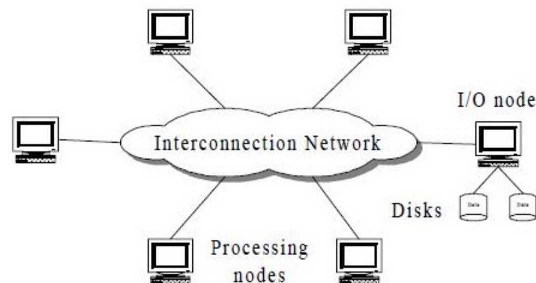

Figure 3.2: cluster architecture

As extensively replacing physical experiments by simulation and computation, parallel computing becomes a vital aspect in advanced scientific research. The problem is growing gap between gained performance and peak performance expectations for running scientific applications on parallel systems. On the other hand, due to architectural complexity of large parallel systems performance optimization is a challenging goal [44]. There should be proper tools to collect performance related data so that developers could come up with optimized solution for different applications. Providing this information would help developers in updating code but manually updating code is too tedious to be practical for large scale applications.

I/O intensive applications/environments with large, sustained workload require predicting workloads and planning for their management ahead of time. Some examples of such environments are video on demand and video services, medical sciences, and online transaction processing. One of the common characteristics for these applications is having many I/O

per second which applies to mpiBlast, as the application we have used for our experiment. Evaluation metrics differ, i.e., I/O rate per second, latency or response time, for different applications. For our experiments, we considered I/O rate and application execution time as performance metrics.

In spite of the expected high performance rate most HPC applications have a shared problem of low performance which needs attention. Involving multi-core and multi-socket cluster nodes increase high dimensionality and complexity of performance optimization. Performance optimization requires identification of code segments that are bottlenecks and determination of effective restructuring of code. The problem of code optimization is that most application developers are not familiar with the intricacies of each system as well as the fact that code should be independent of system's configuration. Burtscher et al. [42] designed PerfExpert to make performance optimization applicable for developers and users. PerfExpert is a tool that captures and uses necessary software, architecture and compiler knowledge for finding performance bottlenecks in applications. The proposed tool executes a structured sequence of performance counter measurements using existing measurement tools. By analyzing the results potential bottlenecks are recognized and a list of possible optimizations will be recommended. Overall PerfExpert is an expert system for automatically identifying and characterizing performance bottlenecks. However, PerfExpert need some optimization in order to be useful for improving overall performance of applications running of multi-core systems. Hurwitz in [31] proposed SAN as a networking environment that is fundamentally different from LAN and WAN. One difference is throughput that is not considered as a metric of performance on SAN, System Area Network. Many cluster-based applications are dependent on calculation that is limited by the time required for transferring a small amount of data among nodes. So, it is important that SAN interconnects not place too heavy a burden on the CPU. With LAN/SAN-oriented 10GbE layer-2 switches capable of port-to-port latencies of under 1micro second [31] achieves very low latency rate which warrants considerations in high end computing clusters. The advantage of 10GbE is increasing performance without requiring modifications to application code. The use of MPI over 10GbE has difficulties in achieving good performance. The level of specialized optimization required to get expected performance in MPI-base applications, including changes to the applications, threatens the benefits of running 10GbE in a SAN environment. "A large numbers of slower processors may be better than a small number of faster processors for I/O dominant applications". Simple linear sums of individual processor performances do not provide an accurate estimation of I/O performance for a parallel computer. Also, this could give a better cost estimation in selecting the number and type of the processors for massively parallel I/O applications. A distributed Cache Subsystem (DCS) technique [18] is proposed to improve the performance of running I/O dominant applications on massively parallel computers. One common use of massively parallel computing is in scientific applications. In spite of requiring very large I/O, I/O subsystems for scientific applications were considered as auxiliary devices with the main goal of loading processing units with sufficient data [43] Dedkov and Eadline in [18] had a clarifying look over the evaluation criteria for being used in evaluation I/O dominant massively parallel systems and suggest a way to satisfy enormous I/O requirements without

using high performance hardware or parallel file system. The behavior of I/O dominant applications is similar to the general model of most business applications that need to periodically read/writing a data block and processing it. In these applications performance is the number of blocks being processed in one time unit. [18] stated the new rule as for two given parallel computers with same cumulative CPU performance, the one with slower processors has better performance for I/O dominant applications. One critical fact about performance optimization methods is that the focus should be on issues under the control of the application developers because of the lack of ability for changing parallel hardware. Our experiments have been placed and could be applied on parallel applications with the following properties:

- The application processes very large databases.
- The application requires multiple accesses to some or all of the databases parts, i.e., fragment.

One way in decreasing communication delay is caching which are proprietary disk caches in each node that keep most recently used data blocks. In DCS, Distributed Cache System, part of the processing nodes serve as specialized pools with the goal of serving I/O requests from processing nodes and store most recent data blocks that have been used. Coordination among DCS nodes prevents possible data duplication in local caches of processing nodes. When there is a request to read a data block the corresponding DCS that holds the data will be determined. DCS nodes are passive servers waiting for I/O requests, sending the requested data if it is saved and reading the data fragment from I/O node in case it was not saved. In other word, DCS is an intermediate layer between processing and I/O nodes [18]. See Figure 3.3. It may appears that allocating large amounts of each processing node's memory as the local disk cache will question the benefits of DCS. As we will discuss in Design chapter this is not true. Experiments have shown that DCS is efficient for parallel I/O dominant applications that are processing very large databases on computers with large number of nodes as DCS nodes that can hold a significant part of the database. The advantage of DCS depends on individual application and characteristics of target computers. Selecting the optimal number of DCS nodes is important since small number wouldn't be efficient to save enough data and having a large number as DCS nodes would decrease the processing power by the lack of enough processors for this task. Having large enough memories in massively parallel computer nodes, there are two methods for performance improvement using DCS nodes:

- Conventional multiprogramming techniques will be deployed if parallel I/O dominant application doesn't use all nodes memory.
- In low memory consumption situation DCS processes will be running on compute nodes resulting maximal DCS and compute nodes, i.e., both on all nodes, and eliminating the determination difficulties of optimal number of DCS nodes.

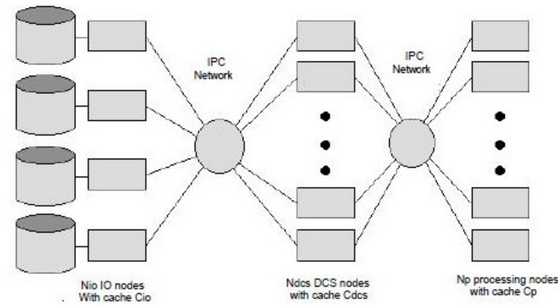

Figure 3.3: MP computer with DCS nodes

We have implemented a similar technique to improve performance in our design. Dedicating 10% of system power to I/O and adding this middle layer between master and worker nodes helps application in better managing I/O and improving performance.

One method to improve I/O performance in high performance storage systems is using multilevel cache hierarchies [30]. In a distributed I/O environment buffer caches are organized as multilevel cache hierarchies placed on multiple machines (Figure 3.4). uCache [30] is an algorithm for unified buffer cache management to improve multilevel cache performance that improve hit ratios for both low-correlated and high-correlated workloads. Speedup of uCache in high-correlated workloads is network dependent and slower networks decreases its performance.

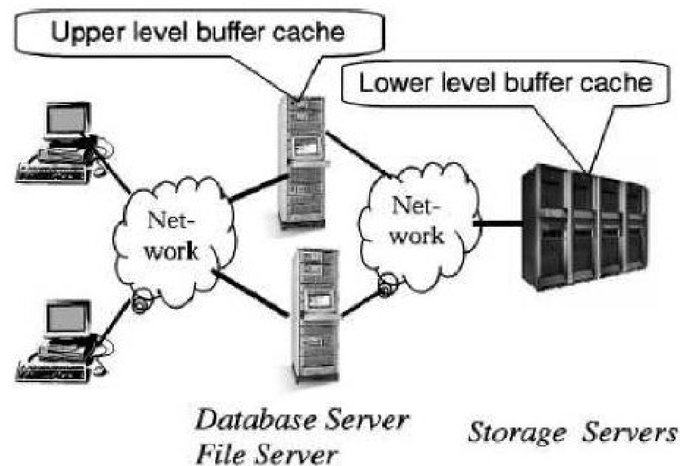

Figure 3.4: multilevel cache hierarchy

Assigning work to machines with idle or under-utilized resources at run time is a load balancing scheme that can improve performance. There have been different studies on load

balancing in a workstation-based cluster systems focusing on effective usage of global CPU and memory resources. Traditional load balancing policies can degrade system performance for I/O intensive applications. Qin et al. in [51] proposed two I/O aware load balancing methods called IOCM, load balancing for I/O-CPU-Memory, and WAL-PM, Weighted Average Load Balancing with Preemptive Migration, that can be used to improve the overall performance. These methods dynamically detect I/O load imbalance of nodes in a cluster and decide if transferring some data from overloaded nodes to others would help application. To guarantee achieving the performance level of current algorithms in well balanced systems, these techniques consider CPU and memory load sharing as well as balancing I/O load. One advantage of these techniques is sustaining high bandwidth requirements of I/O intensive applications and making clusters more flexible. IOCM uses remote I/O execution facilities to improve performance while WAL-PM utilizes a preemptive job migration strategy to increase performance. As [51] is concluded since data movement has a significant impact on the overall performance of load balancing, a predictive model for moving data without compromising the performance of applications running on local nodes in a cluster should be able to optimize performance. The single server node tends to become performance bottleneck for large-scale clusters that can be solved by different techniques such as prefetching and collective I/O. The effect of disk I/O on overall system performance is more important in I/O intensive applications executed on clusters. The growing speed gap between CPU and disk I/O makes storage devices a performance bottleneck as well. In such environments dynamic I/O balancing techniques should be I/O aware in order to achieve high performance [50]. We have implemented an I/O balancing technique in our experiments which we called Static fragment assignment. The experiments have shown that even having a balanced load in system, due to hierarchical design of the application communication is still a problem in system's performance. Qin in [49] introduced a new load balancing algorithm for clusters that maintains high resource utilization under a wide range of workload conditions. This technique can reduce the average slowdown of all parallel jobs on a cluster by balancing load in disk resources which results an effective usage of global disk resources as well as degrading response time of I/O intensive parallel jobs. It should be mentioned that this work doesn't consider network communication cost in its studies. Lee et al. in [35] studied two file assignments algorithms to balance load across all disks improving overall system performance by fully utilizing available hard drives. Studies have shown that I/O cache and buffers are useful mechanisms to optimize storage systems. Active buffering is a method implemented by Ma et al. [40] uses local idle memories and overlapping I/O with computation in order to alleviate the I/O burden caused by I/O intensive applications. Conventional tools for parallel performance analysis are not accurate due to large data volume that may be required. Performance analysis tools collect run time information to help developers in identifying performance bottlenecks and improving performance [46]. For massively parallel systems that scalability is an important issue, these tools should be scalable as well. Generally speaking two scaling techniques are used for studying the scalability for an application. Strong scaling keeps the same total workload by increasing the number of processors which makes reduced workload for each processor while in weak scaling the workload increases for

more processors.

Lin [38] suggested that the key design challenge of massively parallel sequence search tools is in their irregular computation and I/O patterns and are two-fold:

- Differences in similarity level for each case that cause different compute time and output distribution.
- Predicting the execution time is difficult because BLAST uses heuristics to improve computational efficiency.

As mentioned previously genomic databases, as one of the popular I/O intensive application, have been growing exponentially so there is a need for faster sequence search algorithms. In spite of algorithmic improvements in BLAST [9], it cannot keep up with the increasing size of databases. So, parallel implementations such as mpiBLAST designed to solve this problem [59] considering the fact that the performance will depend on algorithm, architecture, and the mapping of algorithms to the architecture. One reason for the widespread

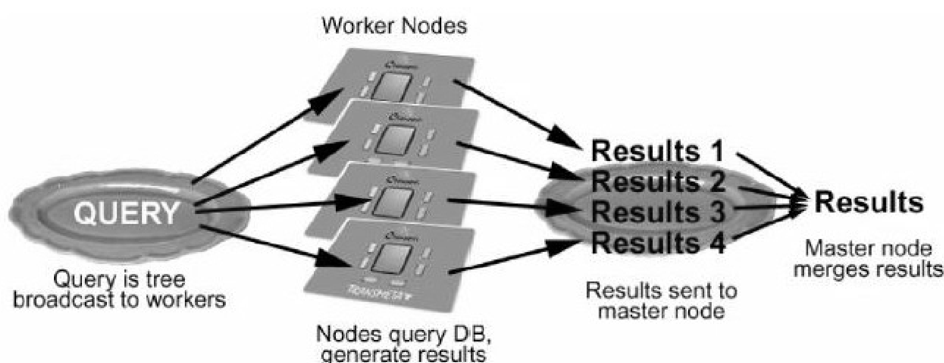

Figure 3.5: Master-worker model in mpiBLAST

popularity of mpiBLAST is its open source development model which makes it easier to improve the application and make it compatible on different system configurations [32]. To take advantage of the processing power of computer clusters, master-worker hierarchical model(Figure 3.5) is used for mpiBLAST that has three main stages:

1. Distributing query sequence among worker nodes,
2. Searching query against the specific fragments of database which are assigned to workers,
3. Merging the results and resulting one output file,

One way to improve the performance of this application is changing each one of this stages or the methodology behind them. Deciding the (fragment, query) set for each worker is called task that has a great impact on the performance. Moreover, managing the output returned by different workers and merging them could be a complicated task and different optimization can be done in this level.

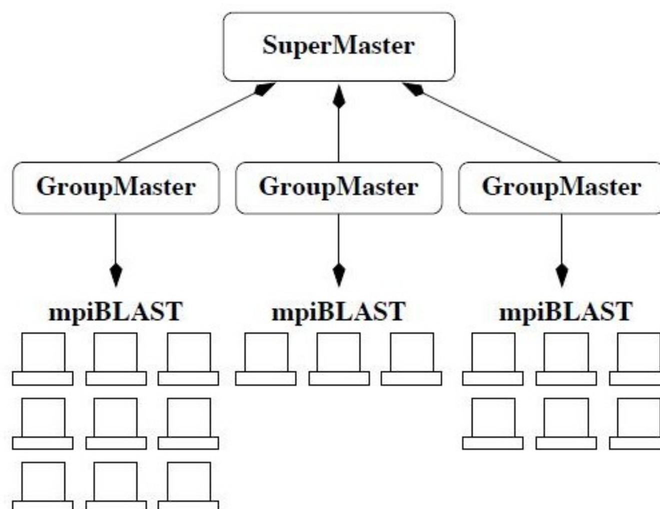

Figure 3.6: Multiple masters architecture

Database searches and sequence alignment are not only I/O intensive but also computationally demanding which would be critical by increasing size of the query and database. One example could be searching the nt databse with the same database as query. Thorsen et al. [59] proposed an approach in optimizing mpiBLAST for massively parallel systems which is based on eliminating I/O by shifting the location of database from disk to the memory of compute nodes and multiple masters scheme using grouping workers (Figure 3.6). Although using multiple masters is important in scaling application, finding the correct number of masters, i.e., the number of workers in each group, plays a key role [59]. With growing scale and complexity of scientific applications, there is more need for powerful systems from computation as well as storage resources aspect [48]. On the other hand most of the large scale supercomputers either have computer power or storage resource. This makes it difficult to find an efficient system for I/O and computation intensive applications and leads researchers to study the optimization techniques inside the application code. Altschul et al. in 1990 proposed a new algorithm for rapid comparison called Basic Local Alignment Search Tool, BLAST, which can be applied in various contexts such as DNA and protein sequence database searches, motif searches, gene identification searches, and analysis of multiple regions of similarity in DNA sequences. Simplicity and robustness of underlying concept of BLAST provides a good opportunity for different implementations and utilizations [9]. Several studies have assigned various computational tasks of an application to different cores

for homogeneous [54, 47, 11, 21] and heterogeneous [17, 29, 24, 34] systems for parallelizing applications. One solution to the compute-I/O imbalance problem is assigning some cores to absorb part of I/O tasks rather than continuing to assign more of the available cores to computation which adds more pressure on memory and causes performance decreasing. Examples of such asymmetric division of labor among cores are I/O libraries that dedicate processors [53, 39] or use separate threads [39, 45] for handling parallel I/O operations. Similarly, BlueGene/L [25] uses distinct cores for compute and networking tasks. Recent studies in [10] resulted a new method, Functinal Partitioning, for runtime environment that dedicates a subset of cores within a compute node to data processing services to help alleviate the I/O bottleneck improving the overall system resource utilization and speed up.

Chapter 4

Design

System scheduling can be done in one of three ways: centralized, hierarchical and decentralized [13]. The scheme used in mpiBlast is hierarchical model and it defines three levels of supermaster, master and workers [37] in which master level nodes are responsible to assign tasks to workers and gathering the result to make final output(Figure 4.1). Task for this application is a combination of query sequence and data partition that will be searched. MpiBLAST uses Message Passing Interface [28] for communication among distributed nodes, master and workers. It has been designed to run on clusters with job scheduling software such as PBS (Portable Batch System) [16]. In these environments it adapts to resource changes by dynamically re-distributing database fragments. Although fragment assignments are determined by the algorithm that minimizes the number of copies, there is a huge amount of repeated copies in the case of large databases. One reason for this repeatation is having multiple query sequences that should be searched in all fragments. The more query sequences were defined by user, the more redundancy in system will be happened. One example is searching a database against itself. This introduces the need for working on scheduling policy for mpiBlast in order to utilize resource use and increase performance. Huge databases could cause big fragments and copying them repeatedly among nodes will make I/O the main performance bottleneck for such a massively I/O related application. This probelm would be more complicated running application on more core in a multi-core environment, i.e., multi-core systems pose unique scheduling problems that require utilization for distributed on-chip memory well [15].

In order to examine applications' behavior in different conditions we desgined various scenarios on Systemg [7, 5]. Firstly we should examine the general performance running in a multi-core environment. So, we run the basic command using 16 nodes, 128 cores, and nt database with 30GB size. In order to follow load balancing policy we partitioned database to 128 fragments in order to be equally distributed among nores. It should be mentioned that not all 128 cores work as workers and a small fraction have master responsibility but the effect of this is neglectable. We also had two query sequences for this experiment which

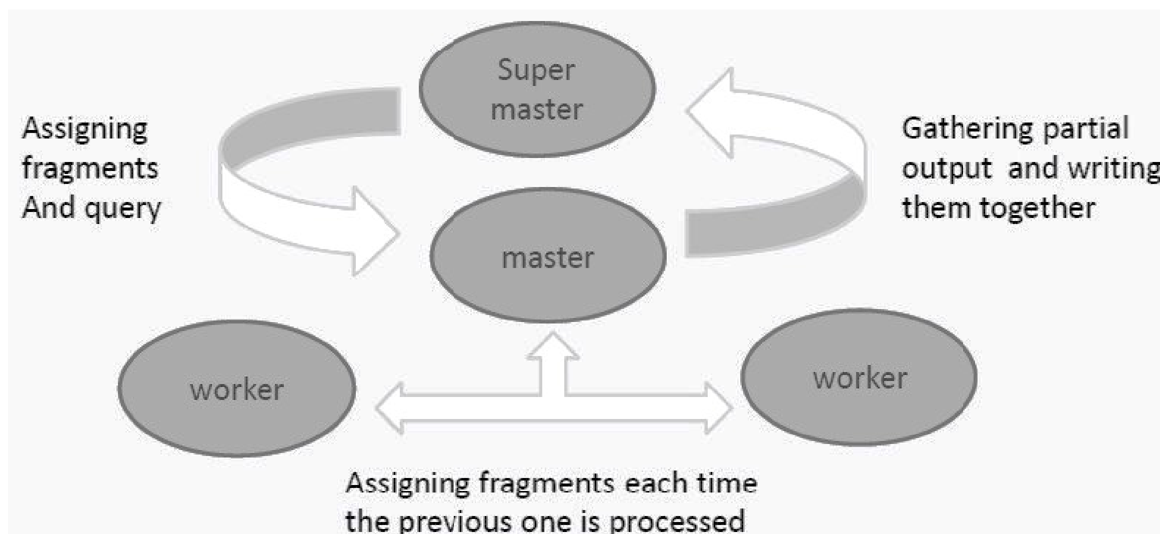

Figure 4.1: 3 levels hierarchy in mpiBLAST application

should be searched separately against all 128 fragments producing 256, 128×2, tasks. Other step was comparing parallel run vs. sequential run for this application. We examined how assigning more nodes for sequential run could make difference with parallel run on less number of cores. Data locality was also studied to define if solving storage problem could help improving performance on multi-core systems with multiple copies of data. This was mainly in proving the irregular I/O patterns which not only is unpredictable but also hasn't gotten enough attention from developers point of view [33]. The other important feature that we considered in designing our experiments was distribution of computational power on system. By using entire cores on one machine vs. part of cores from different machines we tried to demonstrate the effect of I/O communication and caused latencies in a multi-core system. In each stage of these experiments we measured number/size of the transferred data compared to computation time. Results of these experiments have been shown in next chapter but before that it should be mentioned that having the broad range of effective factors on system performance more studies need to be done in this area.

Having an open source application with the mentioned problems, we changed parts of the main code in order to improve its performance running on the multi-core system. Before explaining the changed parts we are going to have an overview on main functions of mpiBLAST that can effect performance [8, 4, 36]. Main functions are as follow:

- master() control flow of master role
- worker() control flow of worker role
- RecvFragmentLists() receive a list from worker notifying the local database fragments.

- HandleMessages()handle me s sages that are received from workers.
- getAssigment()implemented scheduling algorithm

Since, the foucs is on I/O intensive applications in which I/O is a big performance bottleneck, load balancing techniques sound promissing to solve the problem. Based on these methods I/O workload is distributed equally among comoutation power in a multi-core system. To implement this technique in the application we defined "Static Fragmentation Assignment". Initial partitioning of data makes almost equal sizes of fragments. We code an entirely new version of GetAssignment function, called GetAssignmentStatic(), which is called from the worker() method in mpiblast.cpp. This function is responsible to assign tasks to worker nodes. Predefined number of tasks, based on number of query sequences multiplied by number of database partitions, simplifies I/O management in master nodes. Also, handleMessagesStatic() function called from mpiblast_scheduler, is coded as a new version of handleMessages. This change decreases the number of messages between master node and workers which is required in job scheduling process. Although, this method degrades I/O effect on performance it has one main problem. Considering different computational power levels in a multi-core system, assigning the same amount of work to all cores increases the overall execution time to the slowest cores, i.g., similar to parallel run example with highest execution time as the overall result. While dynamically assigning task method would give more tasks to the faster nodes and moderates system executaion time among multiple cores. This is the tradeoff between computational load balancing and I/O load balancing which could be considered in different applications. Howeveer, based on the growing advances in processor design this effect can be disregarded in this context. The second technique for performance issue on I/O bound applications is implemented based on "dedicated I/O nodes concept". We defined 10% of nodes as I/O nodes for different workers. These nodes are placed between master and workers speeding I/O process. These nodes manage the I/O for their dependent workers decreasing the pressure on master node as well as increasing data distibution. Dedicated I/O cores read data from server and pass it to the workers for computation part saving compute nodes from being involved in netwrok traffic caused by single master scheduling policy. Currently we defined 10% of workers as I/O nodes but this is not an accurate amount and could have different efficinecy levels on different applications. Finding the proper percentage for I/O nodes is a critical point for this method. More I/O nodes will decrease computation power while less nodes can add more complexity to I/O management. So, there should be extended studies to define an efficient percentage of I/O dedicated nodes. The base changes for implementing this method in source code were made on GetAssignment() function in blastjob.cpp file. There are further changes in the worker() method in mpiblast.cpp, in order to change their main responsibility, and HandleMessages() method in mpiblast_scheduler adding middle layer in communication hierarchy.

Comparing these techniques we found that each can satisfy certain area while there are some points in their implementation that need more study. In static assignment since the same amount of work is assigned to all nodes with different computation power total execution

time may decrease, total execution time will be the highest execution time, while in dedicated I/O method still faster nodes can do more tasks decreasing overall execution time. In static assignment central scheduler has less work because of predefined feature of this algorithm but there is still need to managing complicated I/O from different nodes requesting tasks. This is different for dedicated I/O that there is less scheduling complexities since instead of workers, central scheduler, i.e., master nodes, are dealing with I/O nodes. Also, less I/O managing work needs to be done because of having extra layer of I/O node to application hierarchy. However, static assignment is simple, flexible and scalable to different system configurations since the main change is equally distributing tasks among all machines but in I/O dedicated approach it is difficult to find optimized percentage of I/O nodes for different configurations. Pros and cons for these techniques are summarized in Table 4.1.

Method	pros	cons
Dedicated I/O	decreased I/O traffic	difficult to define an efficient number of I/O nodes
Static Assignment	balanced I/O distribution among cores	degraded exe. time to the slowest computer power

Table 4.1: Dedicated I/O vs. Static Assignment

Chapter 5

Evaluation

Searching of sequence databases becomes one of the most I/O intensive scientific applications with exponential growing in genomic information. Although, there have been extensive studies on computation scheduling for irregular scientific applications and noncontiguous file accesses optimization, there isn't enough researches on the coordination between them. A sequence search tool compares a set of query sequences against a database of DNA or amino-acid sequences using an alignment algorithm with significant matches as the result. These similarities are useful in finding sibling species with common ancestors. Based in these specifications, we used mpiBLAST as our targeted application and the second largest database, nt with the size of 32GB, as the data resource. Different databases woth different size could be found in `ftp://ftp.ncbi.nih.gov/blast/db/FASTA`.

The main consideration for this work is that for I/O intensive applications on parallel systems there is no performance improvement after reaching to a certain point by adding more resources in system. Figure 5.1 shows the general performance model on parallel systems.

We designed several experiments which will be discussed in this section. Different factors that we changed were:

- The location of files (fragments), local disk or server
- The number of nodes running application
- The number of involved cores from each node
- The number of fragments for same applications

Since performance is the main focus in this job it should be mentioned that we have used job execution time as a performance metric in this study. Moreover, different command options have been used to evaluate different situations.

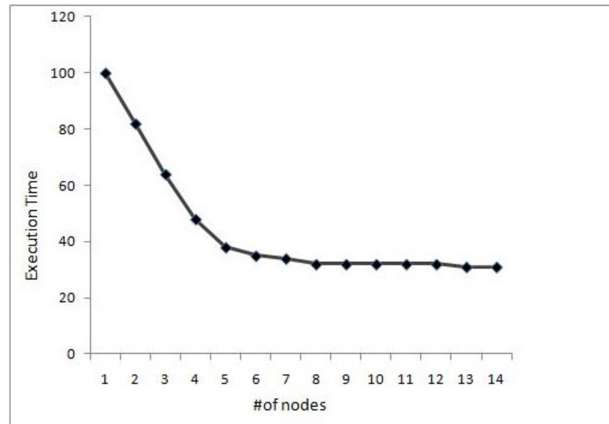

Figure 5.1: General performance model on multi-core systems

First experiment was designed in order to see the application general behavior. We saved copies of data on local memories but removed caches after each run. Having data locally gave the same performance chart as Figure 5.1. This could demonstrate the effect of output gathering policy in this application. Although, data can be read fast enough to expect better performance results, the resulted data should be managed more efficiently. So, in I/O intensive applications adding more computation resources doesn't necessarily cause higher perfromance. See Figure 5.2 as the reference for this experiment.

To study the differenece of local data and reading data from server, i.e., reading data through network, We repeated the previous design without local data. Compared to local data reading data globally could increase execution time with involving more nodes in application. It should be mentioned that the problem with locally copying data on all executing nodes is the limited disk space. In most cases data is huge and it is not efficient to load disk with repeated information. Moreover, having data in local memory doesn't limit the need to communicate with server and there is still need to find out which part of data should be used and returning results to be calculated in finalas final output which is master's responsibilty. Consistency issue is another aspect of having multiple copies of data. Updating repeated data in specific period of time would be time consuming. For this experiment we had 11 nodes each with 8 cores fully used. Note that using the same resource for computational applications would give different results because of smaller amount of time spent for communication. See Figure 5.3 as the result of this task.

We started our experiments by using entire power of nodes, i.e., 8 cores. The number of fragments was divided equally among nodes with close size range. For this purpose we saved the log files so that the IO time could be differentiated with computation time. In this way we could measure the percentage of IO for each application.

Other experiment was designed in order to find out the effect of distribution of computation

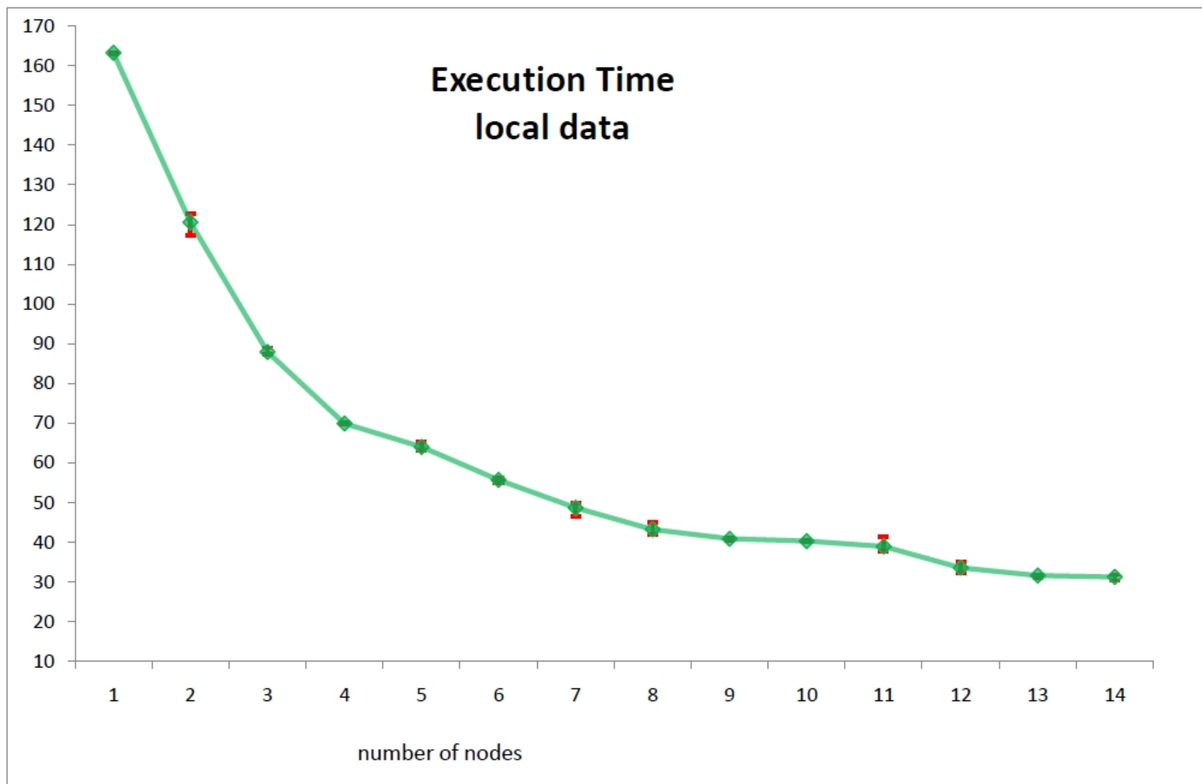

Figure 5.2: Execution time for locally saved nt database on 14 nodes

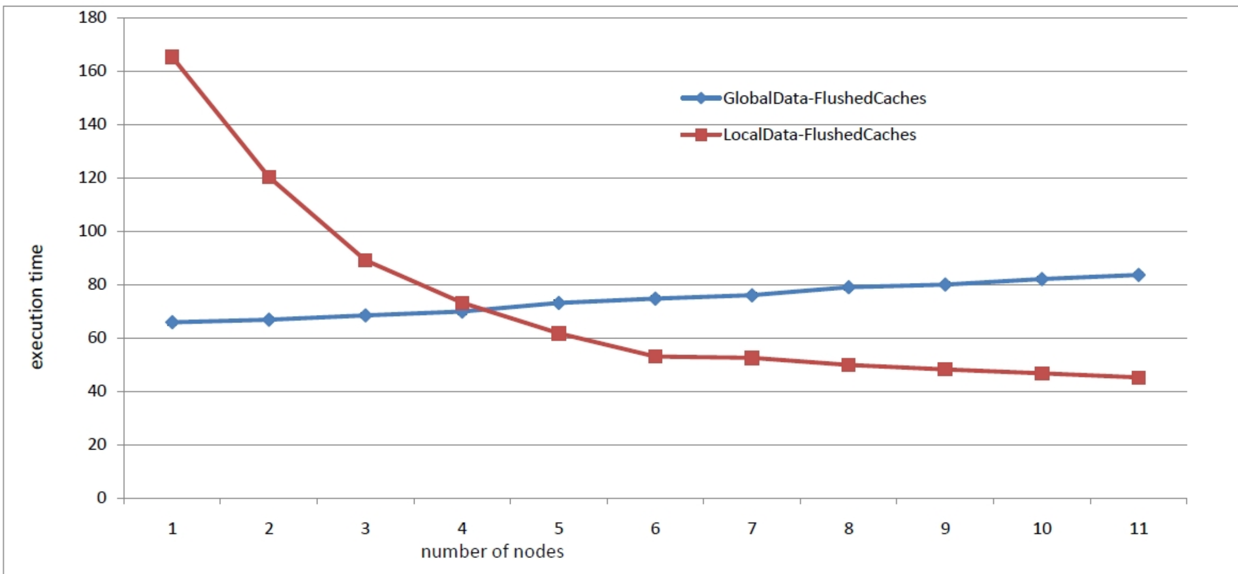

Figure 5.3: The effect of local data vs. nfs data on performance

power for I/O intensive applications. We have data on nfs file system so each node should read data through the network. This is a general case for running parallel applications considering the fact that storage is the main drawback in most multi-core systems. Based on the results shown in Figure 5.4 we can conclude the following results. Having more cores on one machine is not the best solution for performance optimization. Although increasing number of cores on one machine give slightly better results, after a certain point there is no big change in execution time. As you can see for 6 cores on 1 machine results are very similar to 8 cores. Also, same number of cores perform better on less number of machines. As an example consider 16 cores total on 2 machines comparing 16 cores on 4 machines. The final result and the most important one proves I/O affect on performance. As it has been represented in the chart after a certain number of cores used in running this experiment there isn't any significant change in performance and we get the same results as presented in Figure 5.1 as our first assumption. Reading data via network in I/O intensive applications without considering specific configuration to handle I/O bottleneck would have this result. This implementation is not an efficient one considering the same results with less resources.

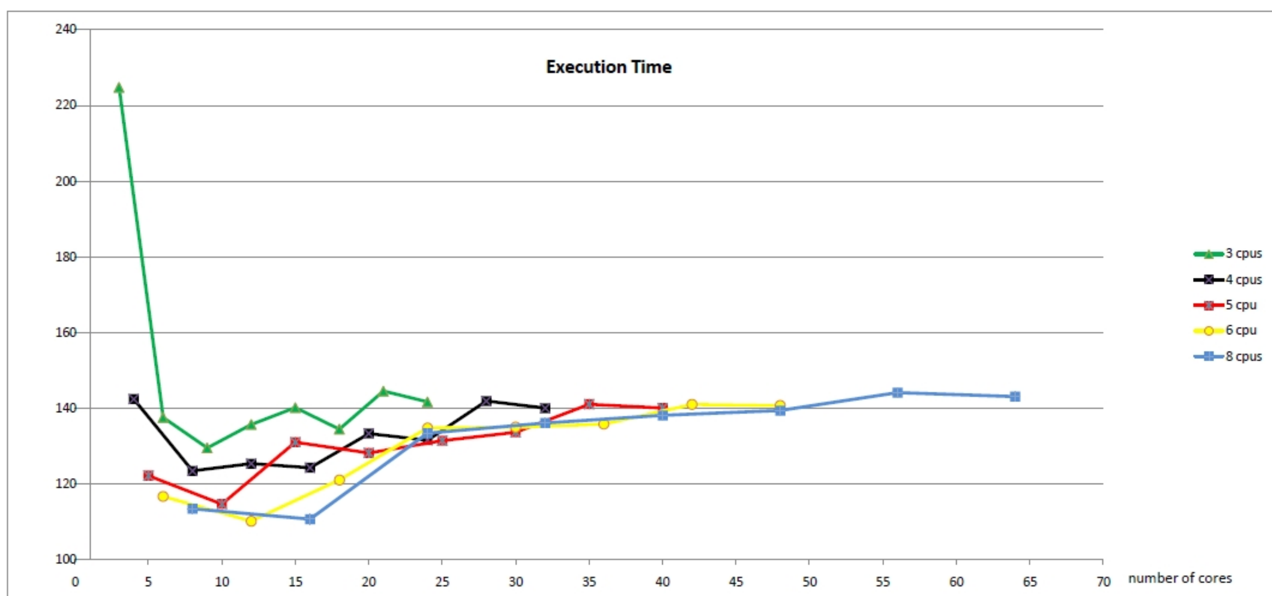

Figure 5.4: The effect of computational distribution on I/O intensive applications

For the second part of our studies we saved data locally in each node's memory so that we could study network role on the performance of this application. Knowing that saving data in local machines is not cost efficient and applicable in most cases, because of the lack of enough storage on local machines, let's have a look at the chart 5.5 and compare the results for different situations. We changed two factors for this part; the number of nodes and the number of cores on each node. For the first change, increasing involved nodes, goal is finding

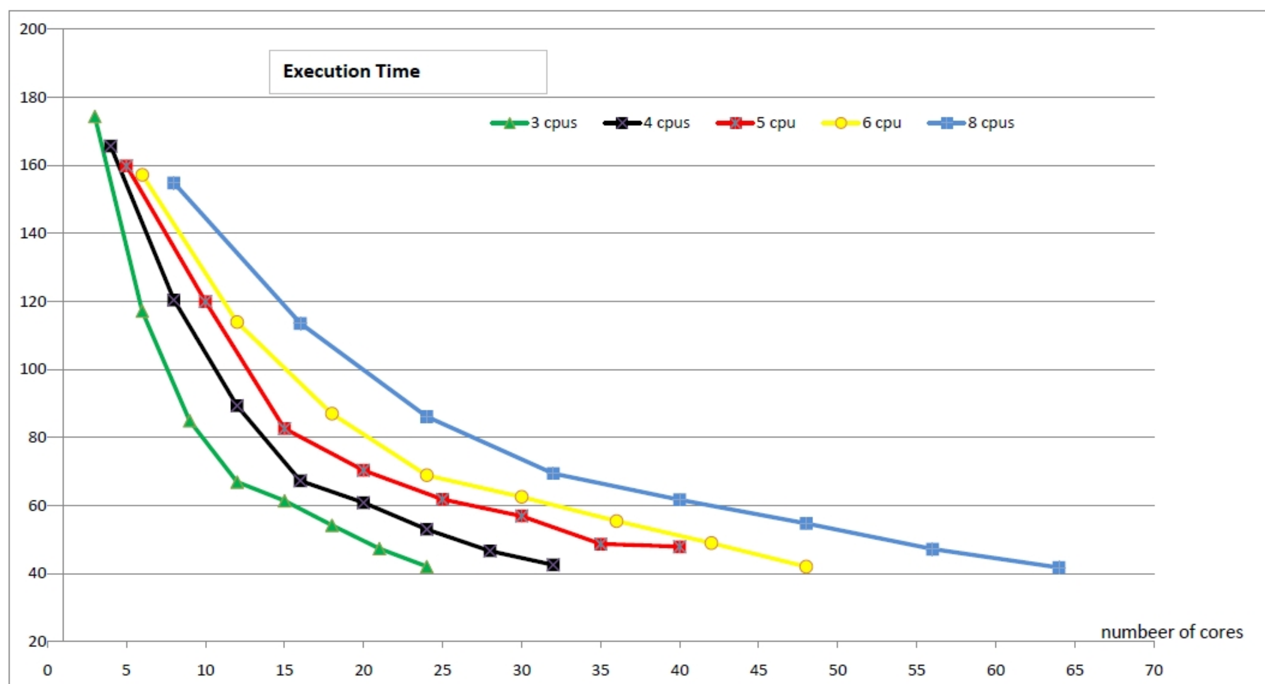

Figure 5.5: The effect of spreading cores without data correlation

the network effect on I/O intensive applications while for the second change, increasing cores, we wanted to find out how well would be I/O management using different number of resources in system. Also, relatively optimized numbers of involved cores on each machine could be determined based on this study. Starting with three cores on one machine, the minimum allowed number of cores, we repeated the same application for 3 cores from 2 machines (6 total) and 3 cores from 3 machines (total 9 cores) and so on. The second run was with 4 cores on 1 machine, 4 cores from 2 machines (8 total) and so on. The most obvious result from this study is that without data correlation spreading cores is a good idea. It means having local copies of data 8 cores on 8 machines, the biggest number of involved CPUs, gave the best performance in system. The second thought would be having more cores involved even on different machines provides better performance. Also, this experiment can demonstrate the fact that having enough storage on computationally powerful machines results optimized performance for I/O intensive applications even having irregular I/O patterns.

In Figure 5.3 we ran an experiment in which data was read globally. We flushed all the caches in order to prevent using previously read data in next runs. In default situations locality, i.e., using cached data, helps in speeding up the application. So, we designed another implementation for system to study the effect of cached data on performance. Figure 5.6 illustrated the impact of data locality. For no Flushed chart, after second run adding more nodes gave the same results while in flushed case more I/O adds more complexity to applica-

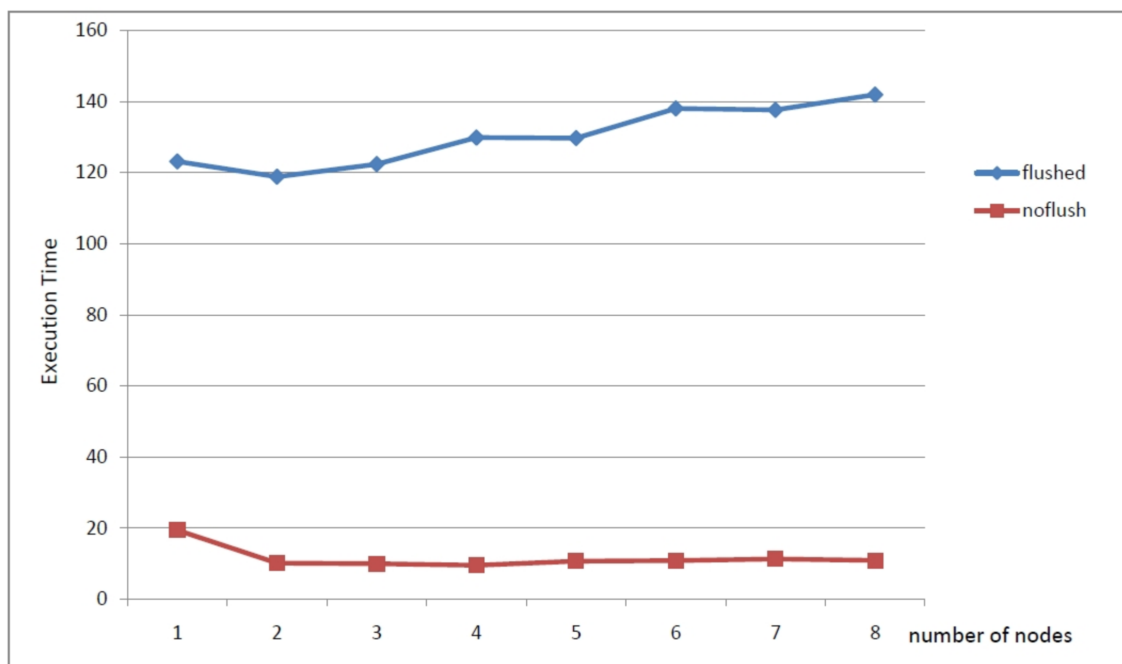

Figure 5.6: The impact of data locality on execution time

tion and increases the execution time. Although, cached data is a good way in performance improvement, e.g., in case users run the same application with same resources sequentially, such a system is not cost efficient as 2 nodes with total 16 cores gave the same results as 8 machines with total 64 CPUs involved in the same application. So, what would be the benefit of adding more resources if performance remains the same? This is the performance issue on multi-core systems that this work is addressing. Hardware changes aren't usually applicable method in parallel systems. However, it is application developer's responsibility to define utilized resources for parallel running of specific applications. Figure 5.7 is the experiment for measuring assigned fragments for two query sequences for this implementation. I/O correlation for I/O intensive applications could get worst by involving more cores in computation.

For next set of experiments data is located in nfs and all other nodes are reading it through network. Each node has 8 cores and we used all of them for this case. As shown in Figure 5.8 execution time increases by adding new machines in application. For one node the performance is much better than 8 nodes besides the cost issues. In other word, increasing the number of nodes in this system is equal to more wasting of system resources. We have measured the amount of data read in each run as well in order to check overall communication for system. As it can be seen in Figure 5.9 the number of fragments are decreasing by adding more machines to system bringing out this question that so what would be the reason for increased execution time? As discussed about mpiBLAST previously, one of the bottlenecks in this application is output gathering. Therefore, it can be said that despite decreased

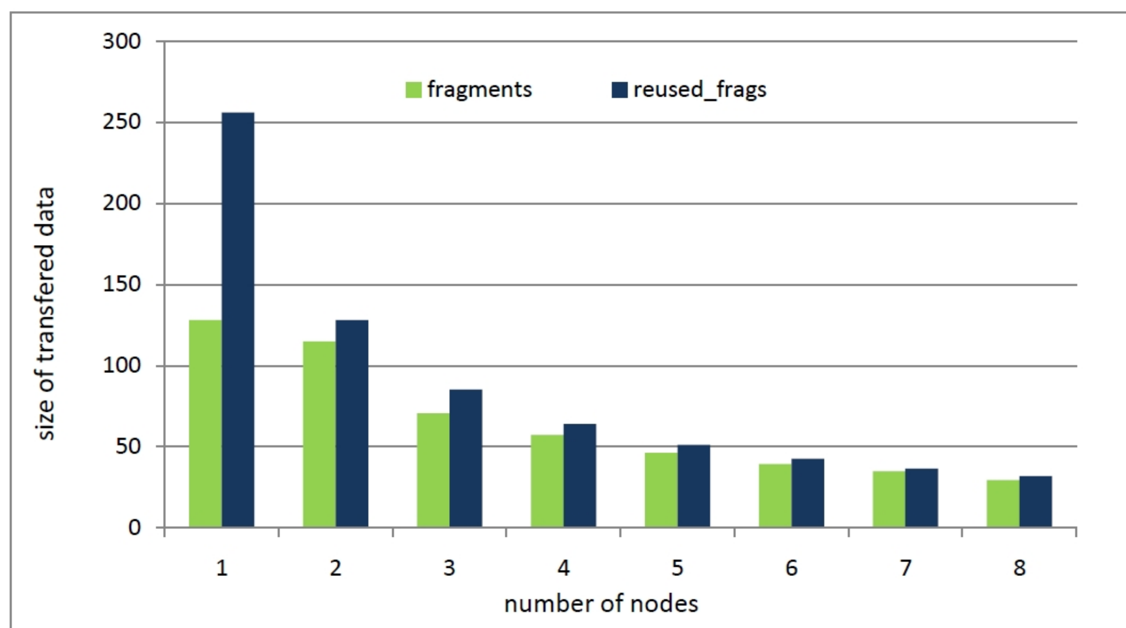

Figure 5.7: Assigned data for different number of machines

I/O, task, for each node by adding more nodes in this application, because of distributed output and larger number of output files returned from more nodes, overall execution time grows and leads to degraded system performance. This again points out the consideration of application behavior that should be given by developers. Note that the bar shown for each node in Figure 5.9, starting with 1 node and counting to 8, is the number of fragments, i.e., not data size.

We wanted to see how much application is effected when there are previously read data buffered in node's cache memory. For this reason we measured copy time, the time it takes to copy required data from server to local nodes, in two different situations. For "no flush" case we let data that was used in previous run cycle remain in cache while for "flushed cache", cache memories were flushed. As you can see for first three run copy time is almost the same but after that it is degraded by near 50%. This proved the repeated data transferring which can be used for future references. Figure 5.10 demonstartes this experiments.

"Copy-via-none" is a command option to decrease repeated versions of data partitions for mpiBLAST application [8, 4]. We used this command to study the application behavior on 8 nodes with 8 cores each (total 64 CPUs) when data is on nfs machine and all nodes are reading it via network. Figure 5.11 shows the resulted execution time which is still increasing. We also measured the number of fragments read for 2 query sequences to check amount of repeated data read (Figure 5.12) which demonstrates the decreasing data for increasing rate of CPUs. So, to find the reason for increasing execution time we measure total I/O and computation effecting execution time giving the results in Figure 5.13. Growing I/O rate

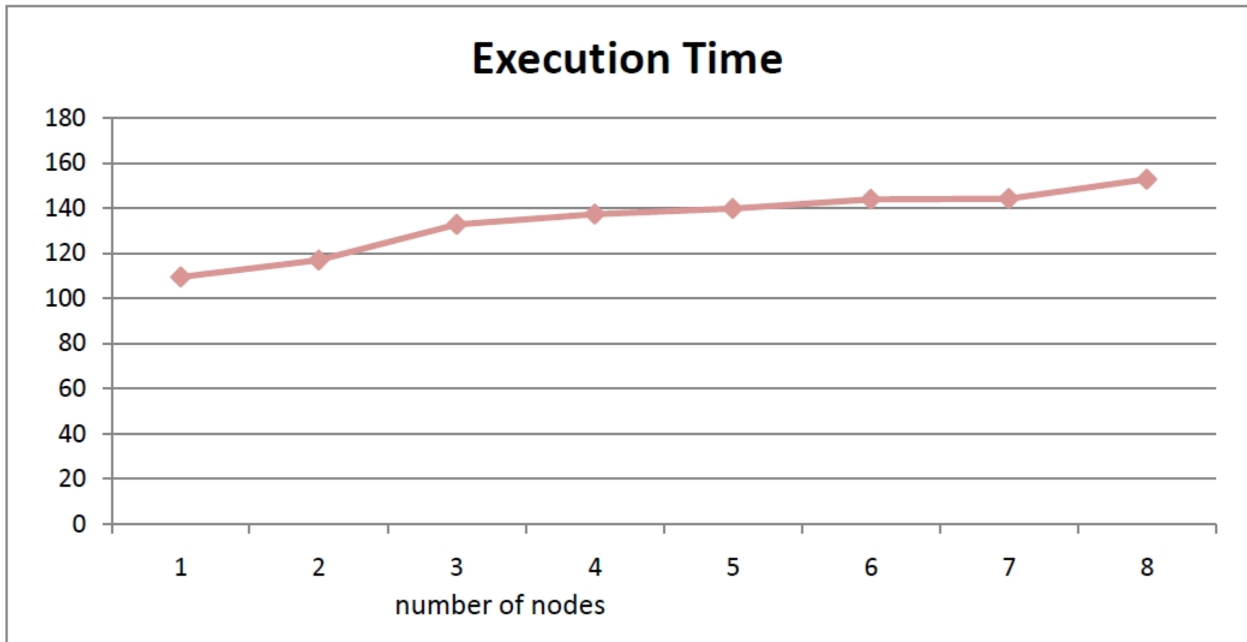

Figure 5.8: Centralized data impact on execution time

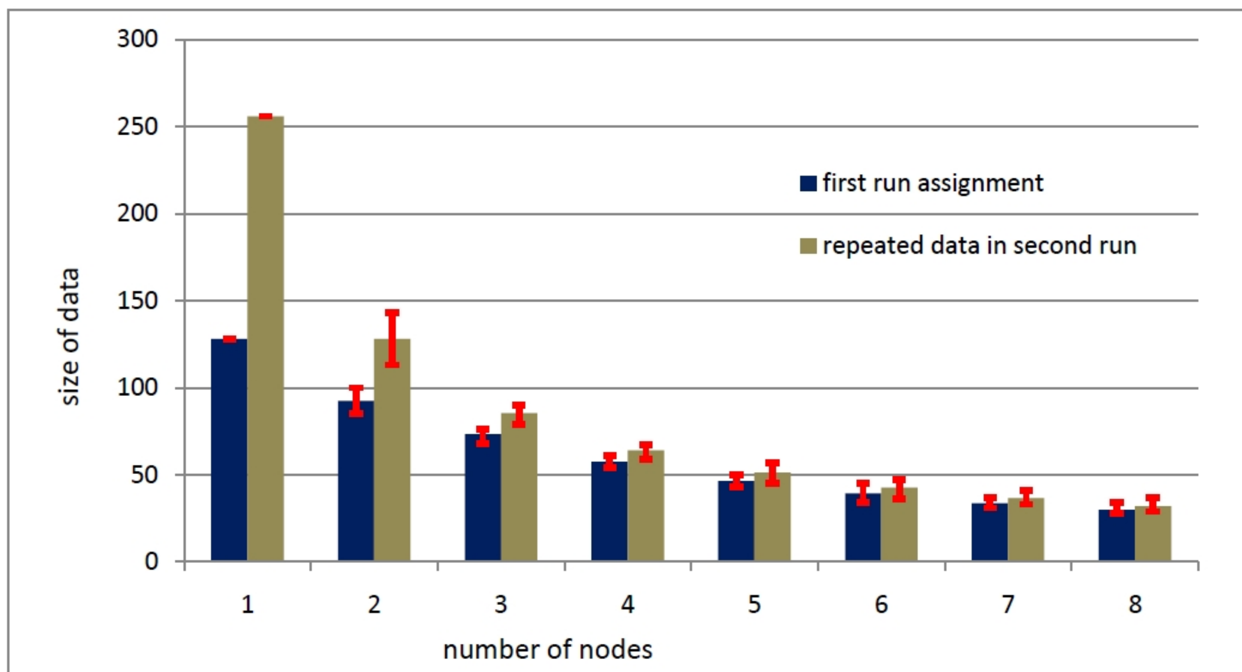

Figure 5.9: Decreasing task by adding machines to application

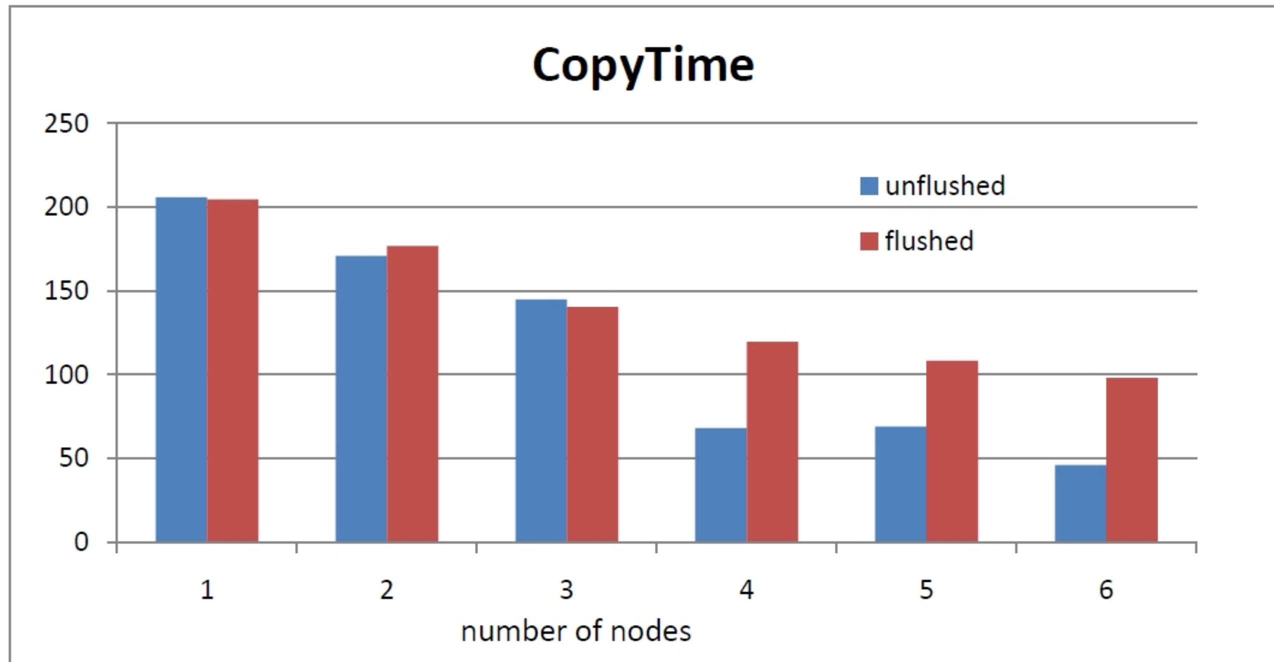

Figure 5.10: Reading centralized data with/without using cached fragments

for this application is a proof of irregular behavior for such applications. Although, the task size has been decreased by involving more resource in system, distribution of computation and dependencies of multiple output which should be merged by single node produces more I/O traffic in system.

Based on what we had so far, one way to decrease communication latency is having data locally saved on each node. Although this is not always possible due to the space efficiency and consistency issues, besides the required time to update data, we implemented next run to see the effect of saved data on application performance. We have used 6 nodes each with 8 cores, total 48 cores, starting with 3 cores on one machine. In each run we add one more core, e.g., second run was with 4 cores on node A, seventh run was with 8 cores on machine A plus one core from machine B. Note that starting point using three cores is because of the application designed feature that requires at least three cores. The experiment was on 'nt' data base and we repeated each run for three times to get a reliable execution time. Also, in each run all buffers were flushed. As Figure 5.14 shows this wasn't a helpful solution in this application case and the results are the same as performance problem described previously, i.e., increasing cores doesn't always help. From this experiment we concluded that despite source data is available in local disk, communication delay in scheduling jobs among nodes is a restriction in performance improvement. Since worker nodes need masters to decide and assign tasks, involving more nodes in application adds more complexity from IO aspect.

In order to study the impact of parallel run on a multi-core system and the differences with sequential ordering of applications we designed 4 and 8 applications for sequential and

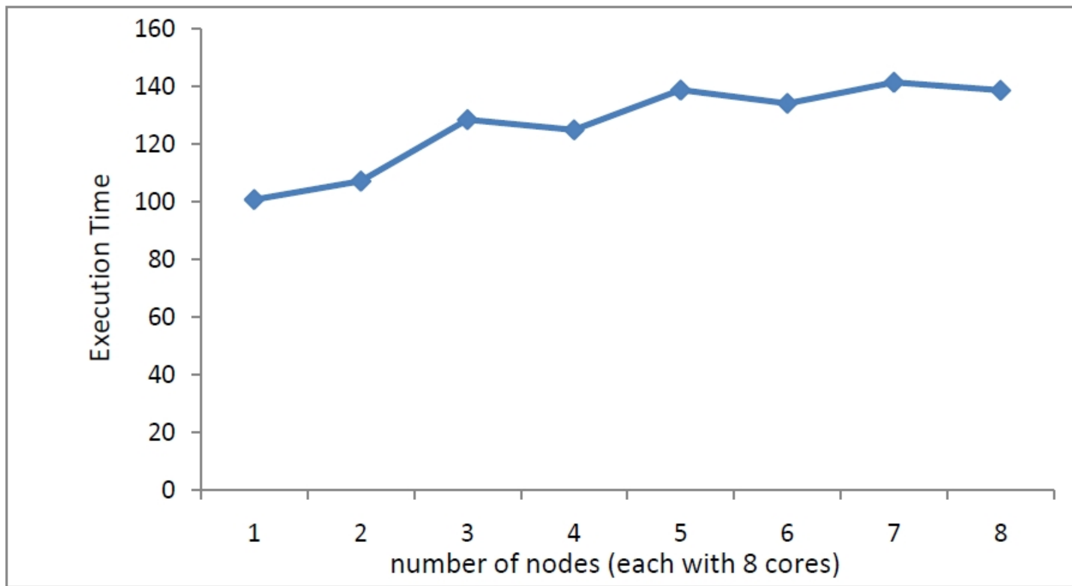

Figure 5.11: Increasing execution time using “copy-via-none” option of command

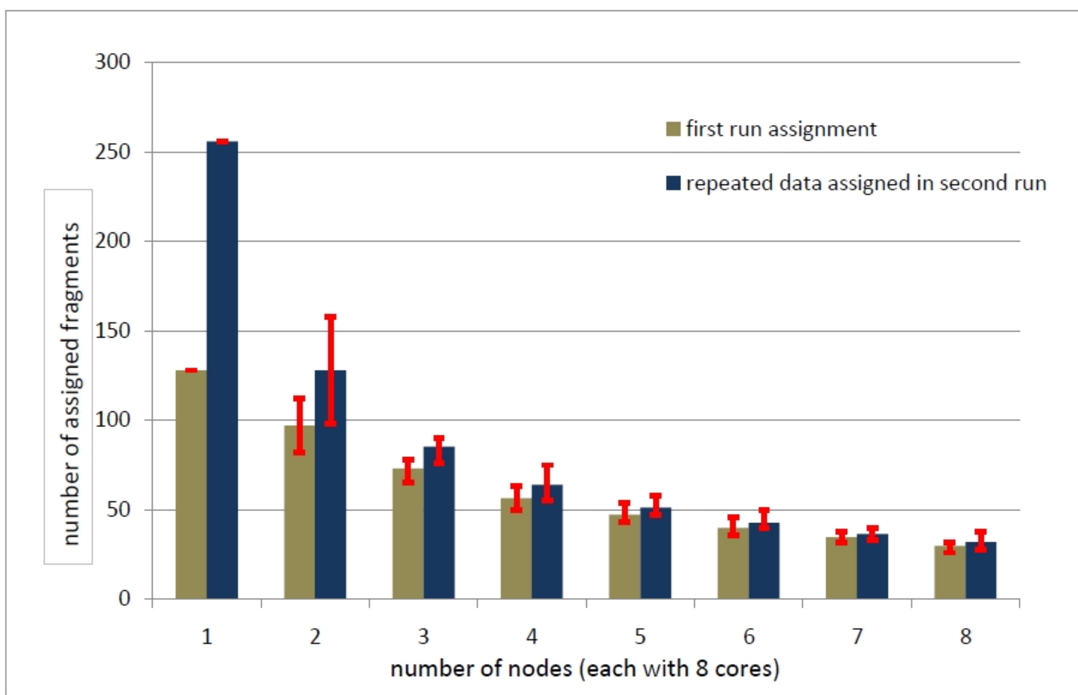

Figure 5.12: Decreasing workload by involving more nodes

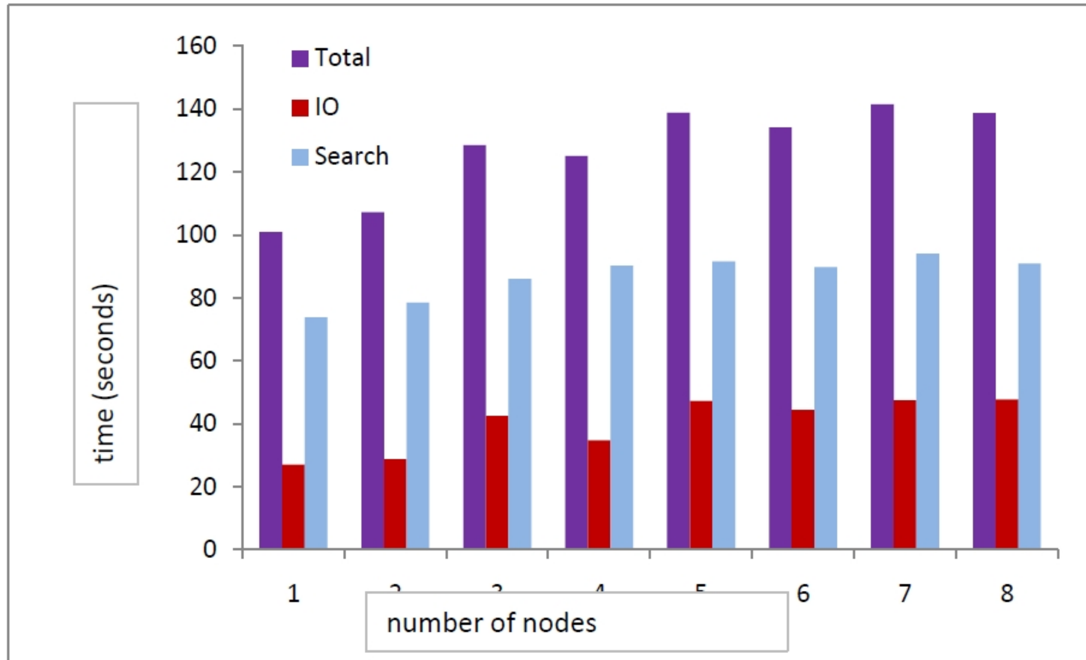

Figure 5.13: Total execution time divided to 'I/O' and 'computation part'

Figure 5.14: Increasing cores doesn't always help

parallel executions. For sequential experiment caches were flushed after each application run. We assigned 8 cores on one machine and ran 4 applications at the same time on them. The first bar, blue, in Figure 5.15 shows the execution time for 4 applications on 8 cores placed on one machine. The second bar, red, is the result for 4 applications on 16 cores from 2 nodes, the third bar, green, is 4 applications' execution time on 32 cores from 4 separate nodes and

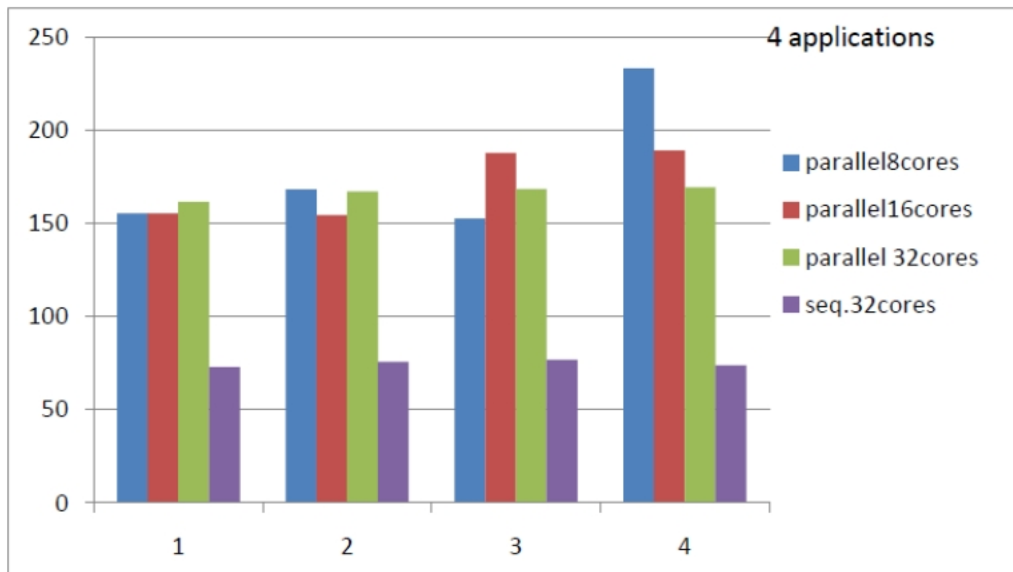

Figure 5.15: Parallel vs. sequential run in 4 applications

fourth bar, purple, is sequential time for each application resulted from sequential run of application on 32 cores from 4 nodes. To compare overall execution time, for parallel case we considered the biggest execution time as the overall runtime for 4 parallel applications which is 233 while for sequential case the sum of all was considered overall execution time which is 300. See Figure 5.16 as the reference for this comparison. We concluded following

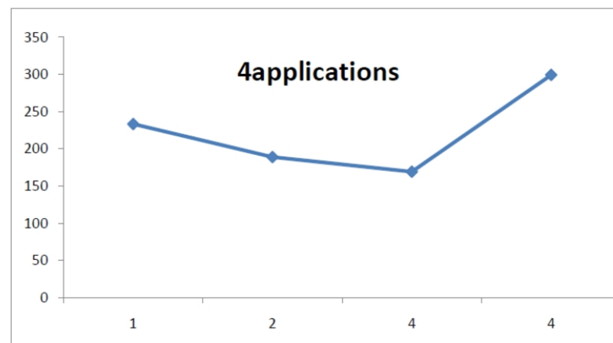

Figure 5.16: Overall execution time for 4 parallel/sequentially executed applications

results in this experiment:

- Parallel run even on smaller number of resources is better than sequential run.
- Running parallel applications on more nodes has better performance (slightly different)

which could point out that concurrent parallel applications are using cached data from concurrent applications.

The same results are gained in 8 applications case illustrated in Figure 5.17 and 5.18.

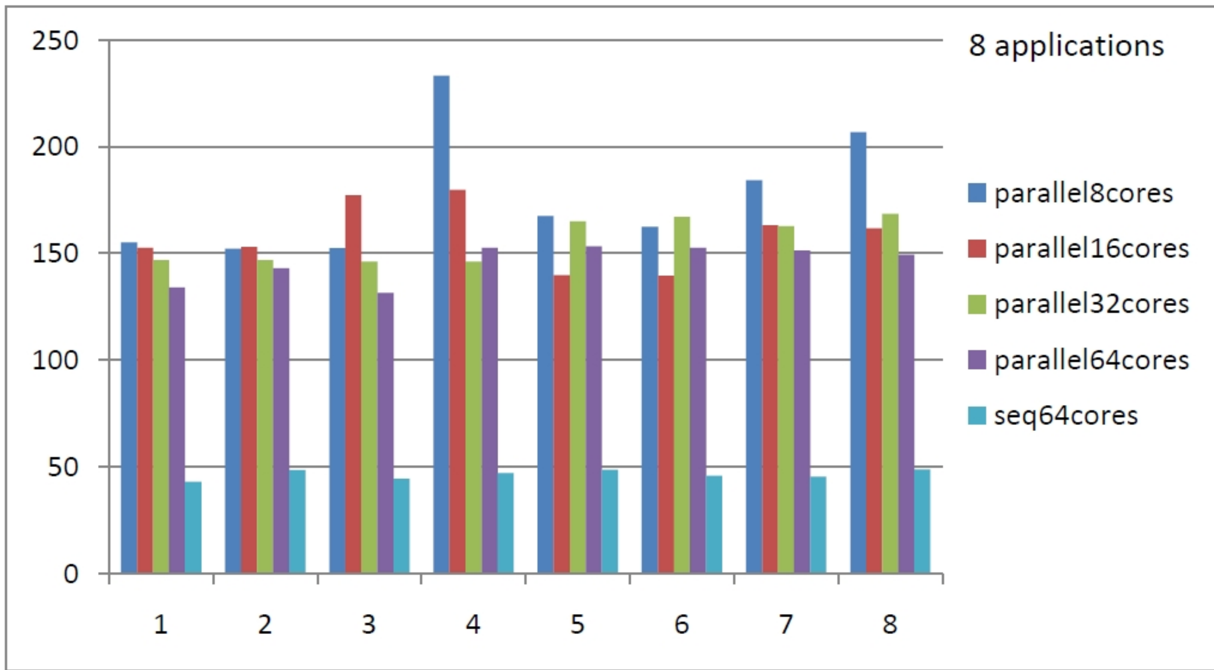

Figure 5.17: Parallel vs. sequential run in 8 applications

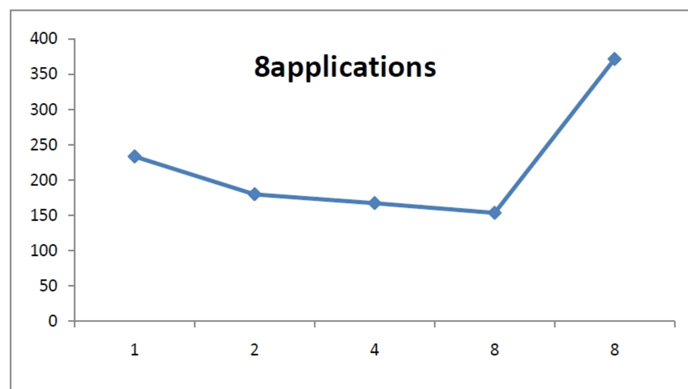

Figure 5.18: Overall execution time for 8 parallel/sequentially executed applications

5.1 Lessons Learned

We summarized the results of different experiments as follow:

- Involving more core in an I/O correlated application can increase execution time.
- Without I/O correlation spreading cores among system improves perfromance.
- Locality-aware task assignment can decrease communication delay but not necassarily execution time, because of complicated output aggregation stage.
- Parallel run on less number of cores give better results comparing to sequential run on more cores.

As mentioned before this experiments could be extended in other aspects of different applications to give more in depth view in order to solve performance issues with I/O intensive applications. However, the findings of this work could be a great check list for application developers and system designers.

Chapter 6

Discussion

We investigated I/O intensive applications behavior on multi-core systems in order to define bottlenecks in overall system performance. Some examples of I/O intensive applications are long running simulations of time-dependent phenomena that periodically generate snapshots of their state, archives of raw and processed remote sensing data, parallel video processing [11], and out of core applications [51]. The common feature shared by these applications is having extremely high I/O requirements. For our investigation we should select an I/O intensive application which was mpiBLAST for our case. Based on previous works this application not only has I/O intensive nature but also considered as an irregular behaved on parallel system. Using the open source feature of this application we implemented two algorithms in order to change its behavior and make it more compatible on parallel systems. Our experiment results bring out the fact that in order to optimize the performance of I/O intensive applications there should be in depth studies in each application by developers because of difficulties in manipulating parallel systems' configuration. Also, based on the differences in parallel applications defining a common rule is a difficult task so in order to get optimized performance in parallel systems there should be unique studies for each application.

6.1 Conclusion

I/O intensive applications behavior is not predictable on multi-core systems because of the massive communication requirements in these applications. Involving more cores means more communication complexity and new I/O management issues in system. On the other hand, having irregular I/O patterns make it difficult to predict communication patterns in these applications and their performance is different based on system configurations. Moreover, different command based options provided by these applications can have a big impact in different system implementations. We designed different experiments to study mpiBLAST, as an example of I/O intensive application, in order to define different factors

effecting performance for different system configurations. These studies lead us to find out unpredictable attributes of irregular I/O intensive applications as the main reason for unexpected performance level having more resources. We considered extensive range of system configurations in this study such as data locality, core distribution, and parallel vs. sequential configuration and concluded that there should be particular attention to I/O issue in designing application and not leaving everything to runtime. Although the multi-core and parallelism speedup the computation process, application should be designed compatible to run in these systems in order to get expected performance. We also suggested some changes in mpiBLAST implementation in order to moderating communication effect on application performance. Assigning specific percentage of machines to serve as communication ports for other node and static task assignment as a load balancing technique are new algorithms added to current mpiBLAST implementation. In spite of having some problem, we expect that these techniques could help in solving communication problem and effect performance improvement for this application.

6.2 Future Direction

The need for an improved characterization of the resource requirements for different cases to increase predictability of resource usage would result in optimized cost performance rate and resource ustilization. One step as future direction is generalizing this study to other applications with similar features, e.g., irregular I/O pattern. Example applications are FLASH [20, 52], HMMER and large-scale protein family identification. Keeping in mind that simple input and easy-to-understand output are essential for a tool to be useful to the community, application developers need to define clear steps and clarification for users explaining the optimized resource distribution for different configurations.

Bibliography

[1] http://www.arc.vt.edu/.

[2] Bio roll: Users guide. http://cbrc.musc.edu/roll-documentation/bio/4.3/mpiblast_usage.html.

[3] The message passing interface (mpi) standard. http://www-unix.mcs.anl.gov/mpi/.

[4] mpiblast: Open-source parallel blast. http://www.mpiblast.org/Docs/Guide.

[5] System g - its green and powerful. http://www.cs.vt.edu/node/4666.

[6] *Teragrid*. http://www.teragrid.org/.

[7] Top500 supercomputer sites. http://www.top500.org/.

[8] Using mpiblast on quarry. http://kb.iu.edu/data/axun.html.

[9] Basic local alignment search tool. In *Journal of Molecular Biology*, volume 215, pages 403–410, 1990.

[10] Functional partitioning to optimize end-to-end performance on many-core architectures. In *Proceedings of the 2010 IEEE International Conference on Super Computing*, SC 10, 2010.

[11] Turgay Altilar and Yakup Paker. Minimum overhead data partitioning algorithms for parallel video processing. In *Proceedings Domain Decomposition Methods Conference*, pages 25125–8, 2001.

[12] Pavan Balaji, Wu-chun Feng, Jeremy Archuleta, Heshan Lin, Rajkumar Kettimuthu, Rajeev Thakur, and Xiaosong Ma. Semantics-based distributed i/o for mpiblast. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 293–294, New York, NY, USA, 2008. ACM.

[13] Cristina Boeres, Aline P. Nascimento, Vinod E. F. Rebello, and Alexandre C. Sena. Efficient hierarchical self-scheduling for mpi applications executing in computational grids. *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, (7):1–6, January 2005.

[14] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue hua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 43–57. USENIX Association, 2008.

[15] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek. Reinventing scheduling for multicore systems. *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, May 2009.

[16] Aaron E. Darling, Lucas Carey, and Wu chun Feng. The design, implementation, and evaluation of mpiblast. In *In Proceedings of ClusterWorld 2003*, 2003.

[17] M. de Kruijf and K. Sankaralingam. Mapreduce for the cell broadband engine architecture. *IBM J. Res. Dev.*, 53:747–758, September 2009.

[18] Anatholy F. Dedkov and Douglas J. Eadline. Performance considerations for i/o-dominant applications on parallel computers. In *Proceedings of the 4th International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA '95, 1995.

[19] Jack Dongarra. The impact of multicore on math software and exploiting single precision computing to obtain double precision results. *Parallel Processing, International Conference on*, 0:xix, 2006.

[20] Anshu Dubey, Katie Antypas, Murali K. Ganapathy, Lynn B. Reid, Katherine Riley, Dan Sheeler, Andrew Siegel, and Klaus Weide. Extensible component-based architecture for flash, a massively parallel, multiphysics simulation code. *Parallel Comput.*, 35:512–522, October 2009.

[21] Robert Ennals, Richard Sharp, and Alan Mycroft. Task partitioning for multi-core network processors. In *In Compiler Construction*, pages 76–90, 2005.

[22] Zaida Luthey-Schulten Anurag Sethi Taras Pogorelov Felix Autenrieth, Barry Isralewitz. Bioinformatics and sequence alignment. In *San Francisco Workshop*, 2005.

[23] Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[24] M. Cabot-J. Chu J. Meinecke K. Oliver F.T. Hady, T. Bock and W. Talarek. Platform level support for high throughput edge applications: the twin cities prototype. In *Proceedings of the IEEE Network*, pages 22–27. IEEE, 2003.

[25] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the blue gene/l system architecture. *IBM J. Res. Dev.*, 49:195–212, March 2005.

[26] Mark K. Gardner, Wu chun Feng, Jeremy Archuleta, Heshan Lin, and Xiaosong Ma. Parallel genomic sequence-searching on an ad-hoc grid: Experiences, lessons learned, and implications. *SC Conference*, 0:22, 2006.

[27] Claudio Gennaro. Performance models for i/o bound spmd applications on clusters of workstations. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:263, 1999.

[28] William Gropp. Using mpi-portable parallel programming with the message-passing interface. *Sci. Program.*, 5(3):275–276, 1996.

[29] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[30] Li Ou Xubin Ben He, Martha J. Kosa, and Stephen L. Scott. A unified multiple-level cache for high performance storage systems. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:143–152, 2005.

[31] Justin (Gus) Hurwitz and Wu-chun Feng. Research note: Analyzing mpi performance over 10-gigabit ethernet. *J. Parallel Distrib. Comput.*, 65:1253–1260, October 2005.

[32] Eli Tilevich Jeremy Archuleta and Wu chun Feng. A maintainable software architecture for fast and modular bioinformatics sequence search. In *23rd IEEE International Conference on Software Maintenance*, 2007.

[33] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 3–14, New York, NY, USA, 2009. ACM.

[34] Sanjay Kumar, Ada Gavrilovska, Karsten Schwan, and Srikanth Sundaragopalan. C-core: Using communication cores for high performance network services. *Network Computing and Applications, IEEE International Symposium on*, 0:171–178, 2005.

[35] Lin-Wen Lee, Peter Scheuermann, and Radek Vingralek. File assignment in parallel i/o systems with minimal variance of service time. *IEEE Transactions on Computers*, 49:127–140, 2000.

[36] Heshan Lin. mpiblast 1.4 pio design document. `http://www.mpiblast.org/downloads/files/mpiBLAST-PIO-design.pdf`.

[37] Heshan Lin, Pavan Balaji, Ruth Poole, Carlos Sosa, Xiaosong Ma, and Wu chun Feng. Massively parallel genomic sequence search on the blue gene/p architecture, 2008.

[38] Heshan Lin, Xiaosong Ma, Wuchun Feng, and Nagiza F. Samatova. Coordinating computation and i/o in massively parallel sequence search. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2010.

[39] Xiaosong Ma, Jonghyun Lee, and Marianne Winslett. High-level buffering for hiding periodic output cost in scientific simulations. *IEEE Trans. Parallel Distrib. Syst.*, 17:193–204, March 2006.

[40] Xiaosong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Faster collective output through active buffering. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 151–, Washington, DC, USA, 2002. IEEE Computer Society.

[41] Ami Marowka. Parallel computing on any desktop. *Commun. ACM*, 50:74–78, September 2007.

[42] Jeff Diamond John McCalpin Lars Koesterke James Browne Martin Burtscher, Byoung-Do Kim. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. SC10. IEEE, 2010.

[43] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputing applications. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 567–576, New York, NY, USA, 1991. ACM.

[44] Shirley Moore, Felix Wolf, Jack Dongarra, and Sameer Shende Allen Malony. A scalable approach to mpi application performance analysis. In *In Proc. of the 12th European Parallel Virtual Machine and Message Passing Interface Conference (EUROPVMMPI*, pages 309–316. Springer, 2005.

[45] Sachin More, Alok Choudhary, and Ian Foster. Mtio a multi-threaded parallel i/o system. In *In Proceedings of the Eleventh International Parallel Processing Symposium*, pages 368–373, 1997.

[46] null I-Hsin Chung, R.E. Walkup, null Hui-Fang Wen, and null Hao Yu. A study of mpi performance analysis tools on blue gene/l. *Parallel and Distributed Processing Symposium, International*, 0:442, 2006.

[47] S. Arash Ostadzadeh, Roel J. Meeuws, Kamana Sigdel, and Koen Bertels. A multi-purpose clustering algorithm for task partitioning in multicore reconfigurable systems. *Complex, Intelligent and Software Intensive Systems, International Conference*, 0:663–668, 2009.

[48] Wuchun Feng Pavan Balaji and Heshan Lin. Semantic-based distributed i/o with the paramedic framework. In *17th ACM/IEEE International Symposium on High-Performance Computing*, HPDC'08, 2008.

[49] Xiao Qin. Performance comparisons of load balancing algorithms for i/o-intensive workloads on clusters. *J. Netw. Comput. Appl.*, 31:32–46, January 2008.

[50] Xiao Qin, Hong Jiang, Yifeng Zhu, and David R. Swanson. Dynamic load balancing for i/o-intensive tasks on heterogeneous clusters. In *In Proc. the 10th Intl Conf. High Performance Computing*, pages 300–309. Springer-Verlag, 2003.

[51] Xiao Qin, Hong Jiang, Yifeng Zhu, and David R. Swanson. Improving the performance of i/o-intensive applications on clusters of workstations. *Cluster Computing*, 9:297–311, July 2006.

[52] Robert Rosner, Alan Calder, Jonathan Dursi, Bruce Fryxell, Donald Q. Lamb, Jens C. Niemeyer, Kevin Olson, Paul Ricker, Frank X. Timmes, James W. Truran, Henry Tufo, Yuan-Nan Young, Michael Zingale, Ewing Lusk, and Rick Stevens. Flash code: Studying astrophysical thermonuclear flashes. *Computing in Science and Engineering*, 2:33–41, 2000.

[53] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective i/o in panda. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[54] Kue-Hwan Sihn, Hyunki Baik, Jong-Tae Kim, Sehyun Bae, and Hyo Jung Song. Novel approaches to parallel h.264 decoder on symmetric multicore systems. In *Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP '09, pages 2017–2020, Washington, DC, USA, 2009. IEEE Computer Society.

[55] Daniel Xavier Sousa, Sergio Lifschitz, and Patrick Valduriez. Blast distributed execution on partitioned databases with primary fragments. *High Performance Computing for Computational Science - VECPAR 2008: 8th International Conference, Toulouse, France, June 24-27, 2008. Revised Selected Papers*, 99(8), June 2008.

[56] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[57] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3:54–62, September 2005.

[58] Mikita Suyama, David Torrents, and Peer Bork. Blast2gene: a comprehensive conversion of blast output into independent genes and gene fragments. *Bioinformatics*, 20:1968–1970, August 2004.

[59] Oystein Thorsen, Brian Smith, Carlos P. Sosa, Karl Jiang, Heshan Lin, Amanda Peters, and Wu-chun Feng. Parallel genomic sequence-search on a massively parallel system. In *Proceedings of the 4th international conference on Computing frontiers*, CF '07, pages 59–68, New York, NY, USA, 2007. ACM.

[60] Nalini Vasudevan and Stephen A. Edwards. Celling shim: Compiling deterministic concurrency to a heterogeneous multicore, 2009.

[61] Duc Vianney, Gad Haber, Andre Heilper, and Marcel Zalmanovici. Performance analysis and visualization tools for cell/b.e. multicore environment. In *Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*, IFMT '08, pages 7:1–7:12, New York, NY, USA, 2008. ACM.

[62] Sain zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen mei W. Hwu. W.m.w.: Cuda-lite: Reducing gpu programming complexity. In *In: LCPC08. Volume 5335 of LNCS*, pages 1–15. Springer, 2008.