2011-01-01

Development of Load Balancing Algorithm Based on Analysis of Multi-core Architecture on Beowulf Cluster

Damian Valles
University of Texas at El Paso, utepgeek@gmail.com

DEVELOPMENT OF LOAD BALANCING ALGORITHM BASED ON ANALYSIS OF MULTI-CORE ARCHITECTURE ON A BEOWULF CLUSTER

DAMIAN VALLES

Department of Electrical and Computer Engineering

APPROVED:

David H. Williams, Ph.D., Chair

Patricia A. Nava, Ph.D.

Michael McGarry, Ph.D.

Patricia J. Teller, Ph.D.

Benjamin C. Flores, Ph.D.
Dean of the Graduate School

Dedication

I dedicate this dissertation work to my loving parents Eduardo Valles and Adriana Valles. Also to my caring sisters Ruth and Nereida Valles, and in loving memory to my grandfather Adan Valles and my former manager Gerald (Jerry) West who both passed away during my doctorate studies.

DEVELOPMENT OF LOAD BALANCING ALGORTHM BASED ON ANALYSIS OF MULTI-CORE ARCHITECTURE ON A BEOWULF CLUSTER

by

DAMIAN VALLES, M.S.C.E, B.S.E.E

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering

THE UNIVERSITY OF TEXAS AT EL PASO

December 2011

Acknowledgements

I first would like to thank my advisor and committee chair Dr. David Williams. For the many years I have learned in the classroom from him, his undivided attention and valuable advice. I want to also thank him for his help in molding the ideas for this dissertation work, the many discussions and revisions of the chapters. It was a pleasure working with someone who has challenged me during my doctorate years and help me strive to understand the fundamentals before tackling higher challenges.

Also I want to thank Dr. Patricia Nava for being part of my doctorate committee. I want to thank her for her trust in me over the many years working for the department, and giving me the opportunities to grow with valuable experience personally and professionally.

I want to thank Dr. Michael McGarry for being part of the committee in such a short notice. I want to thank him for the time in serving, revising and input to this work. I know that Dr. McGarry will be a great asset to the department over the years to come.

I want to highly acknowledge is Dr. Patricia Teller for her participation, input and expertise to this dissertation work. I appreciate her kindness of accepting being part of this committee back in 2008. Above all, I want to thank her time in revising this work.

To Nito Gumataotao, I want to thank him for his friendships over the many years working together in the labs and in the department. It was amazing learning the many things that I know now about system administration. Thank you for teaching me in handling anything to do with the cluster and the valuable advice in my work for this dissertation.

I want to give many thanks to all of the Faculty and Staff in the ECE department which has been part of my career path and helping me to reach this point of my career.

Abstract

In this work, analysis, and modeling were employed to improve the Linux Scheduler for HPC use. The performance throughput of a single compute-node of the 23 node Beowulf cluster, Virgo 2.0, was analyzed to find bottlenecks and limitations that affected performance in the processing hardware where each compute-node consisted of two quad-core processors with eight gigabytes of memory. The analysis was performed using the High Performance Linpack (HPL) benchmark.

In addition, the processing hardware of the compute-node was modeled using an Instruction per Cycle (IPC) metric that was estimated using linear regression. Modeling data was obtained by using the Tuning CacheEdge program, which is part of the ATLAS libraries, and collected using the PerfMonitor program. The model presented a peak IPC throughput and higher Level 2 (L2)-cache memory hit rate with a five thread concurrency for the eight processing cores.

Modifications were made to the Linux Scheduler in order to improve the performance throughput using the results obtained from the hardware analysis and model which indicated potential bottlenecks at the processor Front-Side Busses (FSBes), Memory Controller Hub (MCH), and L2-caches. The modifications included: changing policy of tasks, grouping runnable tasks, load balancing with affinity assignment of the task groups, and control of process termination and feedback.

The results showed that this approach helped to improve performance throughput since the load balancing approach created a higher L2-cache awareness, with increased hit rate, while reducing the number of times processes accessed the FSB and MCH during execution. Performance throughput peaked with block sizes of 64 and 128 for different matrix size and problem sizes, however as problem and block sizes increased, the performance throughput decreased due to hardware contentions found in the FSBes and MCH. The peak was due to the matching of the block sizes with the data width of the FSBes and MCH.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 High Performance Computing (HPC) Beowulf cluster

High Performance Computing (HPC) clusters are now an economical approach to high computational needs for engineering and scientific applications. Such applications are in the fields of flight simulations, weather forecasting, earthquake simulations, medical visualization, biology, and biomedical models, etc. As the research fields continue to grow in complexity and analysis of data, clusters will be at a higher demand for processing power. However, as the processing power increases, there must be awareness of task management in order to efficiently execute applications.

Beowulf clusters are scalable performance clusters constructed with off-the-shelf hardware components, communicating on a private system network, with an open source software infrastructure [Beo04]. A Beowulf cluster consists of a front-end machine that communicates with the outside world, and manages and distributes jobs to identical compute-nodes. The compute-nodes are connected to the front-end via a private network that normally uses a commonly available communication protocol such as Ethernet or InfiniBand for communication. Together, the compute-nodes and associated private network form a homogenous environment that make-up the structure of the cluster. The compute nodes are servers that are always listening for incoming requests and provide the necessary processing computational power to the assigned tasks coming from the front-end machine.

Virgo 2.0 is a Beowulf cluster that consists of a front-end machine, twenty-one dedicated compute-nodes and two memory-nodes. All compute-nodes consist of homogenous architecture parts which include two Intel Quad-core Xeon processors without hyper-threading capabilities. In effect, the cluster can have up to one hundred and eighty-four processing cores working on a single application not including the front-end machine. Therefore, it gives the opportunity to explore the possibilities of

improving the scheduling of tasks on Virgo 2.0 through reprogramming of the Operating System (OS) kernel to improve performance and execution time.

Rocks version 5.0 is an open-source OS for Beowulf clusters that help to manage the distribution of tasks from the front-end machine to the compute-nodes. The Rocks OS consist of open source code based on the Linux 2.6 OS kernel. It is a management software tool that helps to deploy, upgrade, manage, and scale clusters, and provides mechanisms to control the complexity of the cluster installation and expansion process [Roc08]. Once the cluster is formed, Rocks helps the front-end to identify the compute-nodes and their communication channels to establish an Ethernet network so that the front-end can apply parallel or distributed executions as can be seen in Figure 1.1.

The Optional Specialized Network (OSN) in Figure 1.1 is a possible additional network connected to the main Ethernet switch for the cluster system. The OSN can consist of alternative network topologies and differ in the number of compute-nodes connected to this network. The main reason to consider an OSN is to improve network bandwidth and present different paths to different destinations for network packets. This can only be achieved only if the compute-nodes have multiple network ports to support OSN configurations. This and other details of Rocks 5.0 are fully described in Chapter 4.

1.1 INTRODUCTION TO THE PROBLEM

The Linux Scheduler algorithm is largely determined by its target market and vice-versa

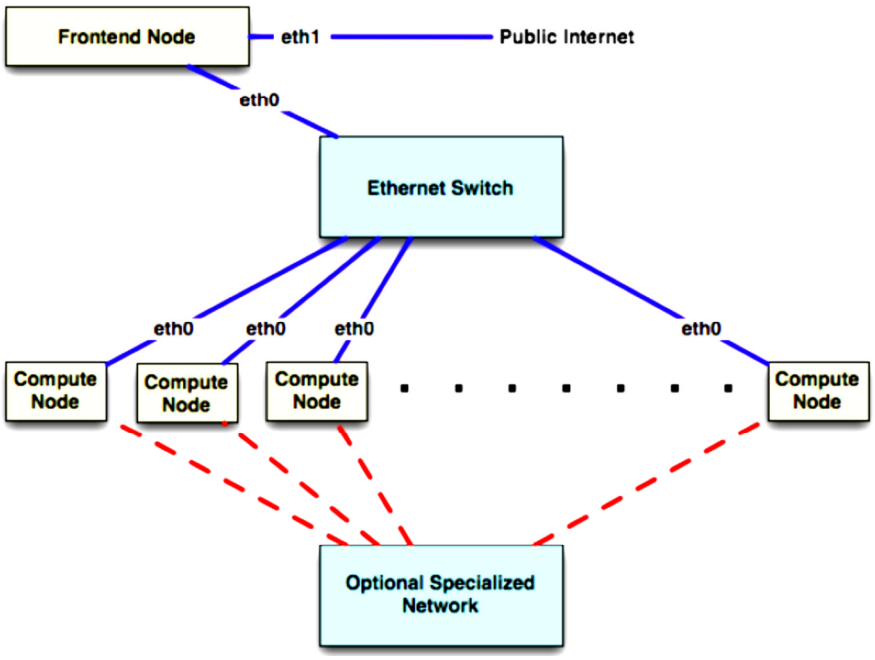


Figure 1.1: Rocks Cluster Architecture Configuration [Roc08].

[Aas05]. This market is mostly defined for desktop and server categories. And since desktop users are much more numerous than server installations, the default Scheduler appears to be oriented to be responsive to interactive requests and be fair to users of desktop systems. In other words, quick context switching between tasks is given priority over efficient task execution and resource utilization. Although these scheduling mechanisms work well for desktop use, it is not desirable for a HPC system where the efficiency of computational processing is affected by the interactivity and fairness of tasks. Therefore, it is significant to examine how the Scheduler is managing tasks and taking advantage of all processing hardware resources so that they can be efficiently and concurrently employed in a HPC system.

It is expected that all the processing capabilities must be used in order to fully use all resources in executing large data-intensive tasks. As all processing units execute tasks, the hardware can reach saturation of resources and bandwidth as it tries to efficiently employ all processing cores at full

potential. Therefore, it is important to observe the effects of the scheduling algorithm to the hardware performance when large data-intensive tasks are executing.

An important goal is to have each of Virgo 2.0's compute-nodes to reach high efficiency in order to provide low processing time in its hardware. However, as the Linux Scheduler context switches tasks, the context switching process becomes computationally expensive. The Scheduler code is run quite often, thus the code making scheduling decisions should run as quickly and efficiently as possible. Efficiency suffers for the sake of other goals such as interactivity, because interactivity essentially means having more frequent context switches [Aas05]. Consequently, we explore modifications that can be made to the Linux Scheduler to context switch less in order to increase efficiency in execution performance.

Another important goal for the compute-nodes in Virgo 2.0 is in how the hardware manages resources in order to execute the scheduled task. When tasks are scheduled to run, the Scheduler sets affinity or assignment to a particular processing core in a Multi-core Architecture (MCA) environment. Preliminary work found that the Linux Scheduler loads one processing core first, and then off-loads work to other cores as it becomes busy. Other parts of the hardware where bottleneck and/or limitations that have been found include moving data through the Front-side Bus (FSB) of the processors and the single channel Memory Controller Hub (MCH) that provides the main data paths from the processing cores to the main memory. Thus, we need to recognize the effects when tasks encounter the bottleneck in the FSB and MCH, and find a better solution than the existing load balancing performed in the Linux Scheduler. For hardware details of Virgo 2.0 see Chapter 2.

1.3 DISSERTATION STATEMENT

The hypothesis of this dissertation can be stated as follows:

Assume that the Linux Scheduler is applied to all compute-nodes in the cluster system, it is developed for its specific instruction set for the hardware, and it is the only mechanism that schedules all of the tasks that are executed for all users and applications. We hypothesize that the scheduler can be modified to further take advantage of the hardware configuration, in being more aware of the pre-existing limitations to the load balancing to specific hardware limitations, and redefined the decision-making when performing load balancing of tasks in order to obtain an average of double-digit percentage improvement in performance and execution time for large computational jobs executed on a HPC system.

1.4 IMPROVEMENT OF LOAD BALANCING

Load balancing consists of managing the utilization time of the cores in a MCA in order to avoid idle processing time on a system. It is important that tasks execute on only one core for the most part. This is for cache hotness and memory bank proximity reasons, and especially since cache sizes for each core in a MCA keep increasing at each level of memory. When load balancing is invoked by the Linux Scheduler, the load balancing function looks for the busiest group in the domain[1], and if there is no busiest group it exits [Aas05]. The improvement that we are proposing is to have the load balancing applied at the beginning of the scheduling of tasks in order to share the processing load between all cores within the compute-node to increase effectiveness in the use of resources.

The reason to consider a load balancing solution is because it can be modified to be more effective for HPC systems. The changes must regard how processes assign affinity (i.e. tasks) to processing cores, reduce the number of context switches since it can reduce performance of execution, and develop a better load balancing mechanism that is more aware of the hardware limitations and possible performance bottlenecks that can occur when executing applications. The methodology will

[1] On a SMP system, each physical processing core would be a group. Groups are maintained as a circular linked list, and the union of all groups is equivalent to the domain. No processing core can be in multiple groups.

consist in having the Scheduler be more consistent with HPC processing efficiency and not too complex for the Scheduler to take too much time in executing decision-making for tasks.

Each application task needs to be scheduled to fully use the processing resources for as much time as possible. For tasks to execute for long periods of CPU time, priority queues must be changed from Round Robin to First-In First-Out (FIFO) format in order to eliminate time slice restrictions and reduce the number of context switches. Then the assignment of affinity processing cores for each task must be decided taking into account pre-existing bottlenecks and limitation of the hardware. Further discussion of these modifications is in Chapter 5.

In order to explore how the Linux Scheduler will be able to have a more efficient task-management, the hardware in the compute-nodes needs to be analyzed to create a better understanding in how resources are managed. Once we understand how the hardware resources are managed, the behavior can be modeled to minimize the limitation the hardware presents when the Linux Scheduler load balances processes when executing applications. We believe that once the hardware model is developed and implemented in such a way that the Linux Scheduler is more aware of the limitations and is able to make better decisions when load balancing tasks, then the overall performance of all (large) tasks should improve in each of the compute-nodes in Virgo 2.0.

1.5 DISSERTATION CONTRIBUTIONS

There are four main contributions provided by this dissertation:

1. Develop an approach that aims to improve performance and execution time that not just only focuses on a specific processing hardware, but contribute a general Multi-core HPC solution.

2. Modeling the existing hardware in Virgo 2.0 cluster's compute-node to better understand the limitations and bottlenecks that causes a decrease of performance, both for Virgo and HPCs in general.
3. The methodology approach that helps to modify the Scheduler to generate decision making more favorable to the hardware configuration and avoid the pre-existing bottlenecks.
4. Qualitative discussion of the results obtained as a solution to improve performance to Virgo 2.0 system without having to spend resources for new equipment.

1.6 DISSERTATION OUTLINE

Chapter 2 of this dissertation gives background information of the overall description of the Virgo 2.0 system, the processing architecture that resides in all system nodes and a review of previous related research. Chapter 3 provides a full description of Linux Processes and an overview of the Linux Scheduler's tasks managements, policies, priorities and load balancing. Chapter 4 describes the experimentation procedures performed to the cluster's compute-node in order to obtain the performance and execution time of the High Performance Linpack (HPL) benchmark, detection of the bottlenecks and limitations in the hardware. Chapter 5 explains the modeling approach to a single compute-node, modifications made to the Linux Scheduler and the experimental setup which test the modified Scheduler. Chapter 6 contains the experimental results obtained from the compute-node analysis with conclusions and future work.

Chapter 2: Background

When trying to improve the computational performance of a server, desktop computer or compute-node in a cluster, it's important to understand how the hardware works and its limitations and design, and be able to model the hardware. This chapter describes the overall specifications of the cluster under study, i.e., the Virgo 2.0 cluster, and gives a description of its main components primarily used during execution of large data-intensive tasks. In addition, previous related work is discussed, providing methods for modeling HPC hardware, which are used to develop a representative model of Virgo2.0 compute-node hardware.

2.1 OVERALL VIRGO 2.0 HARDWARE

Virgo 2.0 is a Beowulf HPC cluster that consists of a front-end machine, twenty-one compute-nodes, and two memory-nodes. Virgo 2.0 was built in 2007 to replace the old Virgo system that was built four years earlier. Since then, Virgo 2.0 continues to provide a high-performance computational platform for researchers in UTEP's Colleges of Engineering and Science.

Virgo 2.0 is a cluster system in which a user has access to all compute-nodes once he/she accesses the front-end machine. The user is able to launch individual and/or concurrent applications on each compute-node. If distributed applications are needed, the user must employ a communication protocol to spawn the processing to all needed compute-nodes. Such protocols include sockets; Remote Procedure Calls (RPCs) and the Message Passing Interface (MPI). The HPL benchmark uses MPI; further discussion on MPI and the associated MPICH Libraries can be found in Chapter 4. All scheduling and job creation is interactive. This cluster system does not contain an internal batch queue program that launches and distributes jobs to its compute-nodes.

Virgo 2.0's front-end machine is the node that manages the overall operation of the cluster. This node is:

- the cluster's login node, i.e., where users connect to the cluster;
- the node from which users access main disk storage;
- the designated node for application installations; and
- the only node that can be accessed .

In addition, once users connect to the front-end, they can install any tools that are necessary to setup the correct environments for application execution and for saving resultant data files. The front-end node contains enough hardware to provide computational services, but ideally it is used as the main connection and data saving node.

Table 2.1 lists the front-end node hardware components.

Table 2.1: Front-End Node Hardware Configuration

Front-End Node Hardware Configuration	
Component	**Model**
Processor	Two Intel 5430 Quad-core Xeon
Motherboard	Tyan Tempest i5400 Series
Memory	Kingston 8GB FB-DIMM DDR2
Hard Drive	Four 500MB SATA2 Seagate
	One 250MB SATA2 Seagate
Cards	Single Channel SCSI Card
	PNY Quadro FX1700 Graphics Card
Drives	Lite-on DVD

Virgo 2.0's compute-nodes are server units; their only purpose is to provide computational resources to user programs. The compute-nodes have a homogenous hardware configuration that is very similar to that of the front-end node with the exception of disk space and graphics and communication capabilities. This hardware configuration gives flexibility to the user that s/he can run applications on any available compute-node and expect the same performance. Also, the compute-nodes only have network communication with the front-end machine; this way, all the compute-nodes deal only with intra-network communication.

Each compute-node consists of two Intel 5430 Quad-core Xeon processors. Although the processors are not hyper-threaded, as a whole, they provide a reasonably high number of processing cores for execution of parallel distributed applications. Each of the processors has a Front-Side Bus (FSB) that is the gateway for data and instruction bus lines connected to the rest of the hardware. Each FSB operates at a high-bandwidth rate in order to provide data and instructions to one or all processing cores. Each node has 8GB of RAM to take advantage of the memory Double Data Rate (DDR2) channels through the Memory Controller Hub (MCH) and serve large data-intensive tasks. Table 2.2 lists the compute-node hardware components.

Table 2.2: Compute-Node Hardware Configuration

Compute-Node Hardware Configuration	
Component	**Model**
Processor	Two Intel 5430 Quad-core Xeon
Motherboard	Tyan Tempest i5400 Series
Memory	Kingston 8GB FB-DIMM DDR2
Hard Drive	One 250MB SATA2 Seagate
Cards	Gigabyte GeForce 7200GS Graphics Card
Drives	Lite-on DVD

Table 2.3: Memory-Node Hardware Configuration

Memory-Node Hardware Configuration	
Component	**Model**
Processor	Two Intel 5430 Quad-core Xeon
Motherboard	Tyan Tempest i5400 Series
Memory	Kingston 64GB FB-DIMM DDR2
Hard Drive	One 250MB SATA2 Seagate
Cards	Gigabyte GeForce 7200GS Graphics Card
Drives	Lite-on DVD

Virgo 2.0 memory-nodes are dedicated nodes and are like the compute-nodes with the exception of the memory that they provide; instead of 8GB of RAM, each memory-node provides 64 GB of RAM. Table 2.3 summarizes a memory-node's configuration. Memory-nodes are used for high data-intensive tasks that would exhaust all of the RAM memory in a compute-node.

2.2 INTEL'S HARPERTOWN PROCESSOR

Intel's Harpertown Quad-core Xeon processor is a 45nm (nanometer) server/workstation processing solution that is employed in the Virgo 2.0 cluster. The Harpertown processor family was launched in November 2007. The Virgo 2.0 cluster uses two Quad-core Intel Xeon processors for the front-end, in each compute-node and in each memory-node. Each of the two processors has eight processing cores, providing a total of eight cores. Theoretically, theses can deliver in excess of 100GFLOPS (floating-point operations per second) for 64-bit applications [Int07]. Figure 2.1 shows a block diagram of the Harpertown processors as employed in Virgo's 2.0 compute-nodes.

As shown in Figure 2.1, a dual Intel Quad-core Xeon has two processors, is labeled with a physical identification number, with four cores each. The physical identification given to the processor helps the system to identify the processing cores with even numbers assigned to processor physical id zero and odd numbers assigned to processor physical id one. The Harpertown processor is design to have two pairs of two processing cores each. Each of the processing cores has their own Level 1 (L1) cache memory and each pair shares a 6MB Level 2 (L2) cache memory. The pair of processing cores also shares a Front Side Bus (FSB) which provides the communication path for data and address buses.

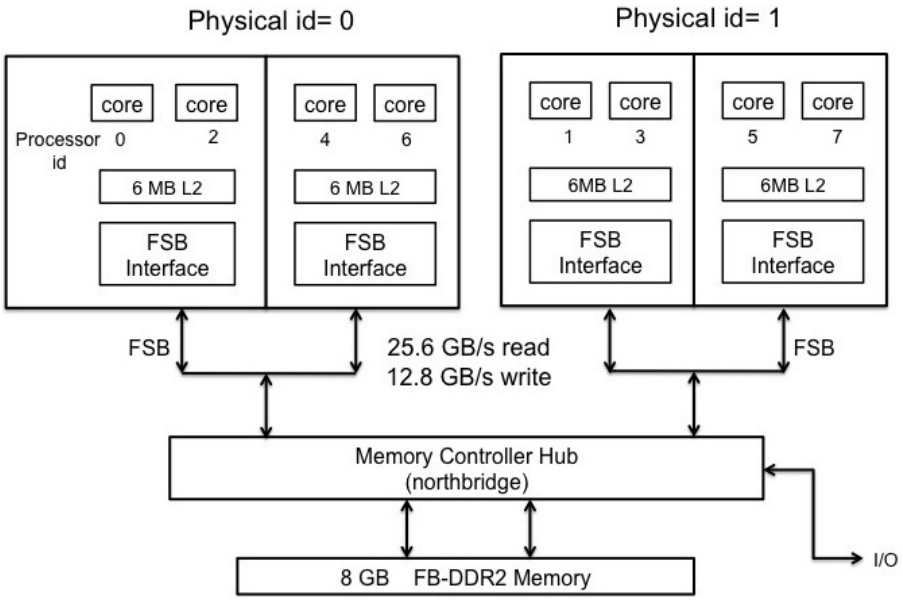


Figure 2.1: Harpertown Block Diagram Configuration in Compute-Node

Each processor combines the signals from the two FSBs into a single dedicated FSB bidirectional interface to the Memory Controller Hub (MCH). Each of the FSBs has a 38-bit address bus, a 64-bit data bus, and associated control signals. The setup time for requesting an address/data bus is four FBS-clock cycles and a bus request can last up to a maximum of 20 FBS-clock cycles [Inm07] if the four clock cycles are not enough for the task to complete before another request is issued.

As we try to improve the performance and management of tasks within the compute-node, we consider the functionality and design of the FSBs. The merging of the two FSBs within each Harpertown and the merging of the FSB interfaces from the two processors within the MCH is considered as tasks normally access the addresses and data from memory many, many times. What we can expect is that the larger the tasks become and the more memory accesses needed, bottlenecks will start to occur as more and more data is moved through the FSBs and the MCH to each pair of processing cores within the Harpertown processor. Also, it can be assumed that execution time will start to slow down as more writing to memory will be needed to be performed.

2.2.1 The Penryn Microarchitecture

The Harpertown employs the Penryn microarchitecture design for each processing core. This microarchitecture design was developed by Intel to deliver increased performance and performance-per-watt, thus increasing overall energy efficiency [Wec06]. One innovation of this microarchitecture is that it takes less time to execute instructions via the use of a 14-stage pipeline, an increased number of buffers, better decoding of instructions and enhanced ALUs (Arithmetic-Logical Units). The Penryn allows each processing core to simultaneously fetch, dispatch, execute, and return up to four instructions per cycle [Wec06]. This means that the Harpertown can concurrently execute up to 16 instructions per cycle when executing on all four processing cores of one processor or 32 instructions per cycle when the whole compute-node uses all processing resources. Figure 2.2 shows the Penryn architecture design that is implemented by the cores of the Harpertown.

Figure 2.2 shows the stages of instruction execution in each processing core and the operation that is carried out within each stage. At the beginning of execution, each core begins pipelining the instructions that need to be carried out for a task. At this time the core fetches instructions from memory, normally storing them in the L1-cache, and pre-decodes them for the following stages. The Deeper Buffers are used to queue up instructions. Branch-prediction, part of the 4-Wide Decode and Execute, is then performed to speed up the pipeline. The Rename and Allocations stage performs renaming of instructions that keep being issued in other instructions and allocate registers for the next instruction to use. The Micro and Macro Fusion stage manages the out-of-order instructions by sending the instructions that are ready to execute to the scheduler and moving the other instructions, which are not ready to execute to the end of the scheduler. Once the instructions are called by the scheduler, they are ready to use the ALU stage for execution.

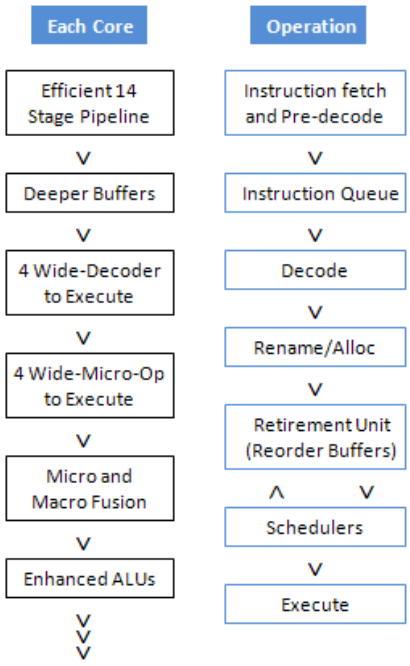


Figure 2.2: The Penryn Microarchitecture Design in the Harpertown [Wec06].

Another innovation of the Penryn is the new way processing cores share the L2-cache, which is called the Advanced Smart Cache. In previous MCAs from Intel, each of the processing cores had its own L2 instruction and data caches. This method limited each core to a certain amount of L2-cache. However, by sharing the L2-cache among a processor's cores, the Advanced Smart Cache allows each core to dynamically utilize up to 100% of the available L2 cache [Wec06] as long as the other processing cores are using negligible amount of the L2-cache.

With these advantages of the core design, we understand that many tasks can be assigned to a single processing core. However as the tasks increase in data-intensity, contention for the L2-cache increases and, thus, the L2-cache hit rate may increase, stalling tasks executing on a processing core and, thus, preventing the core from fully utilizing all of its resources. Therefore, we intend to explore over-saturation of a processing core, i.e., the scheduling of a core with too many tasks and measure the effect of this on the performance of the overall compute-node.

2.3 MEMORY CONTROLLER HUB (MCH)

In a dual Harpertown processor architecture, the Memory Controller Hub (MCH) is the main chipset that forms a bridge between the two processors; it is also the pathway to the I/O Controller Hub. The MCH provides two FSB processor interfaces, four fully-buffered DIMM (Dual In-line Memory Module) memory channels, nine x4 PCIe (Peripheral Component Interconnect Express) bus interfaces configurable with x8 or x16 ports, an Enterprise South Bridge Interface (ESI), six SM Bus interfaces for system management, and DIMM Serial Presence Detect (SPD) [inm07]. Figure 2.3 shows a block diagram of the MCH with all of its interfaces.

Within a compute-node, the main role of the MCH is the management of data moving from and to DRAM. When an application is launched in any of the compute-nodes, the MCH manages all data that needs to flow from and to memory and handle destination points as data needs to flow to different destinations. The MCH also has other modules to control such as the PCIe channels, the FSB management modules, the Direct Memory Access (DMA) module and the interface with the South Bridge. Figure 2.4 show a block diagram of the inner modules of the MCH.

The MCH consists of nine PCIe ports, which can be used as the video channel, Graphical Processing Unit (GPU) channels, and/or fiber optic communication channels. In the case of the compute-nodes, the PCIe channels are not fully used and only one of the PCIe modules is used to provide video whenever the channel is enabled. Therefore, when executing applications on the compute-node, it can be safely assumed that no data traffic will be generated in the MCH by the PCIe modules.

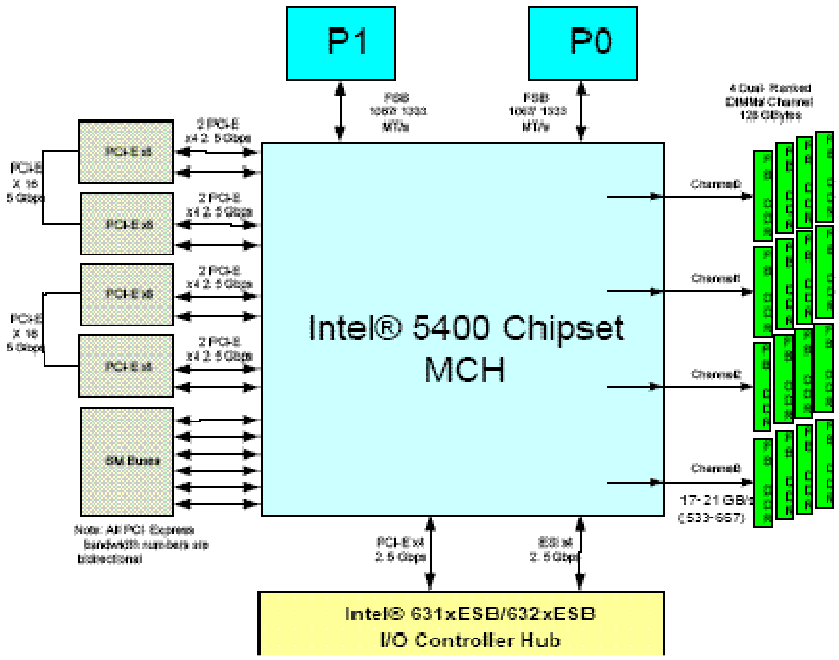


Figure 2.3: Intel's 5400 Chipset Memory Controller Hub Block Diagram [Inm07].

Another function of the MCH is to manage the modules for the FSB. The MCH has two interfaces, one for each of the two processors, which are connected to same bus. The FSB modules provide registers to control the two interfaces and handle the destination of the processor interface that owns the bus. The management of the FSBs becomes important when considering the task performance in the compute-node, along with how the MCH grants ownership to any of the tasks that are executing, the time of ownership, and the number of destinations that it needs to address.

The most highly utilized modules of the MCH are those associated with memory mapping, control, DMA, and memory channel modules. When a compute-node is executing any type of application, the task needs to access either data or instructions that reside in RAM. To do so, each task needs to go through the FSB and the MCH to assemble necessary information from memory. As seen in Figure 2.4, there are four fully-buffered memory channels that are managed by two memory channel

control modules (devices 21 and 22) in the MCH. This can be considered as the total path length that each task needs to go through in order to read from and write to.

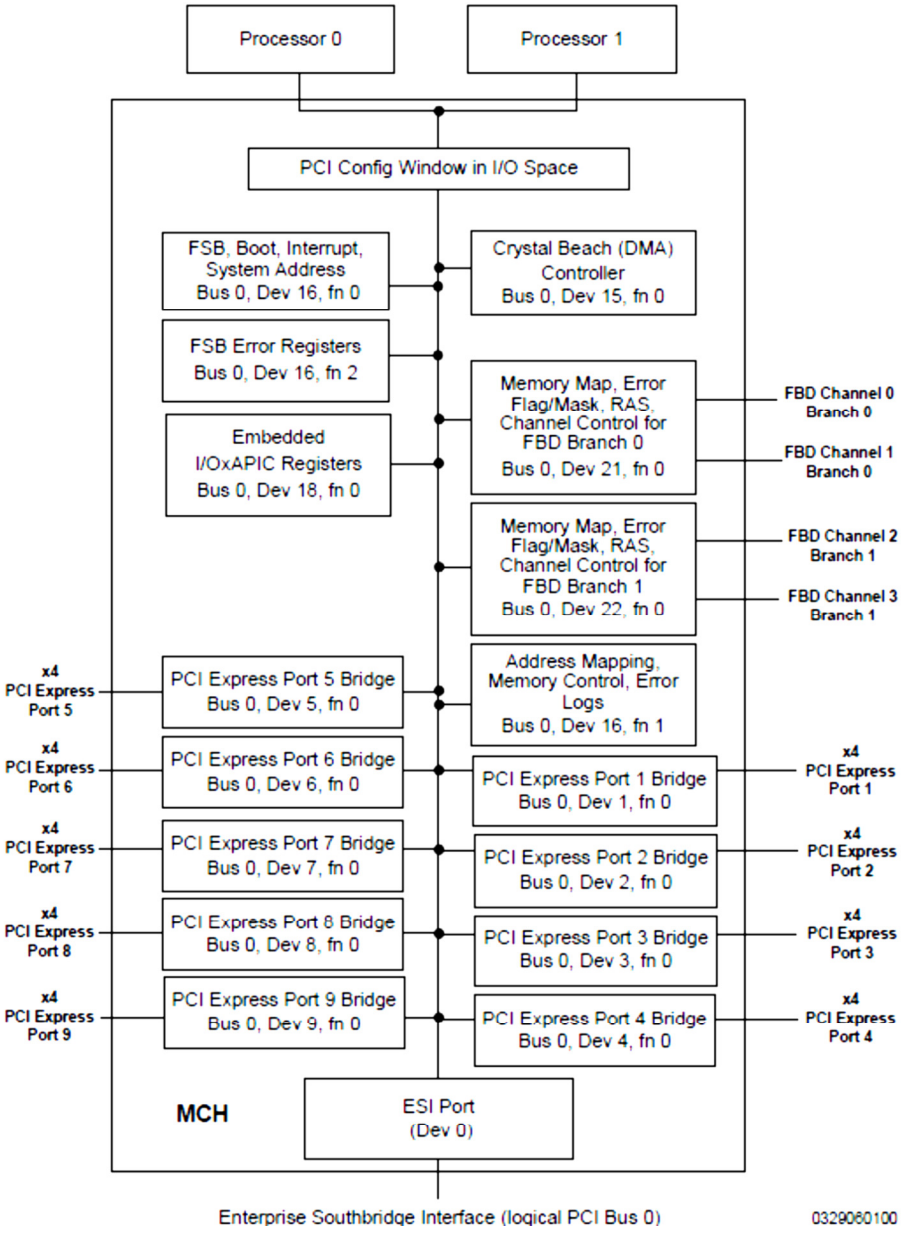


Figure 2.4: Intel's 5400 MCH Inner Module Block Diagram [Inm07].

The MCH can play a big role when developing an improved scheduler for the cluster. One of the main decisions of the scheduler is to know when it is appropriate to migrate tasks from one core to

another. It may not be a problem to move a task to a neighboring core within the same die since all of the resources of the task are located in the same area. However, such a move can be expensive if the migration is from a core on one processor to a core on another. This is because to move the task and all of its resources requires multiple requests, which forces a context switch. Notes, also, that the intended destination for a task migration may not have sufficient space to accommodate the task's resources. Accordingly, there must be an MCH policy with respect to the migration of tasks between two dies.

2.4 PREVIOUS RESEARCH

There is an abundance of previous research that investigates the performance of scheduler. The work of Fedorova et al. [Fed06] the performance improvement achieved by a scheduler that avoids saturation of hardware resources by decreasing the number of cores allocated to the execution of an application. The benchmarks that were used by [Fed06] were executed in simulated hardware that was constructed by using models that describe the characteristics of the desired hardware. We adopted some of these modeling ideas to model Virgo 2.0 in order to understand the limitations of the hardware and to evaluation the performance of our modifications to the scheduler. Our approach in modeling our system hardware is described in Chapter 5.

Rajagopalan et al. [Raj07] presents Intel research on thread-scheduling for multi-core environments. The research consists investigates the scheduling framework for managing hardware threads, the placement of threads to accomplish load balancing, and the design of the scheduling mechanics in a multi-core environment. The approaches that are considered are the fine-grain approach of thread grouping, placing and mapping. We adopted some of these general ideas to implement a scheduling framework that can help our system to improve the performance and timing of executing tasks within the compute-node. The design of our scheduling framework is presented in Chapter 5.

Fedorova et al. [Fed05] researched performance of multi-threaded on multi-core processors and the implications on the OS design. The focus of the work was to make the OS more L2-cache conscience when scheduling and executing tasks. Their results demonstrated a reduction in L2-cache miss rate by at least 25% and improvement in the processing throughput of at least 27%. We made an effort to achieve similar results with our modification of the load balancing in the Virgo 2.0 scheduler. Details of our load balancing efforts are discussed in Chapter 5 and results are presented in Chapter 6.

Ziemba et al. [Zie08] researched the effectiveness of multi-core scheduling by using performance counters. Performance counters are extensions of the Linux scheduler which analyzes the performance of each individual processing core and each individual task. The study was done on Apache servers that run HTTP and PHP threads. Their results indicated that by using performance counter and thread-specific information, the way to improve performance per processing core was to segregate and group threads to avoid idle processing time. We borrowed the idea of grouping tasks to avoid processing idle time and to more efficiently assign affinity to each task to better utilize resources and avoid hardware contention.

Klug, et al. [Klug08] researched how to develop a framework for automated *thread-to-core* binding in multi-core systems. The automated *thread-to-core* binding is based on decisions using timestamps of each thread and measured by performance counters like the ones used in [Zie08]. It was analyzed in different processor architectures with the SPEC OMP benchmark suite which consists of 11 benchmarks. Their results showed that their automated *thread-to-core* tool was able to detect optimal binding for nearly all benchmarks and improved runtime performance by up to 66%. Certain ideas from their work were implemented in this research in order to obtain higher performance through task load balancing. The difference in our approach is that our target environment is an actual cluster system, whereas theirs was a processor. Our approach to *thread-to-core* binding is similar to theirs, however, we also incorporate the grouping of *thread-to-core* in load balancing decision making.

Josh Aas [Aas05] documented the changes that were performed to create the Linux 2.6.8 scheduler in order to give an in-depth view of that part of the operating system. His work describes the main parts of the scheduler, the decision associated with task scheduling, the differences between processes and threads, and context switching. It contrasts the main goals of the *O*(1) scheduling algorithm: efficiency, interactivity, fairness, and prevention of CPU starvation. The properties of real-time threads and the scheduling mechanisms used in Symmetric Multi-Processor (SMP), Symmetric Multi-Threading (SMT) and Non-Uniform Memory Access (NUMA) architectures also are explained. This reference greatly helped in understanding the source code in the *sched.c* and *sched.h* files of the Linux kernel code.

Michael Pinedo [Pin08] presentes general theories and algorithms that deal with scheduling for a wide variety of systems those are not just limited to computers. Deterministic models and stochastic models are initially introduced with preliminary single machine models, parallel machine models, and shops. The author also discusses scheduling in practice with general purpose procedures for scheduling and solving scheduling problems from generic to application-specific systems. Finally, he discusses several approaches for designing and implementing scheduling systems including systems architectures, reconfigurable systems, web-based scheduling and many others.

Chapter 3: Linux Processes and Scheduler

This chapter focuses in providing a full description in how the Linux OS defines, constructs, maintains and ends a process. Also in having the understanding in how the Linux process functions, this chapter additionally explains the structure and mechanics of the Scheduler in how processes are managed for execution. The Scheduler consists of many components such as tasks policy, priority, scheduling algorithm and load balancing of tasks. All these parts of the Scheduler give an insight of the obstacles and possible modifications for the Scheduler when trying to improve the performance of Virgo 2.0 and other supercomputer clusters which use Linux.

3.1 THE LINUX PROCESS

A process is an instance of a program that is executing [Bac90]. A process is created at the moment a user launches a program in order to execute instructions or commands. Some of the main attributes of a process in the OS are:

1. Handles all resources for opening files and pending signals
2. Internal kernel data
3. Process state
4. Memory address space with one or more memory mappings
5. Data section containing global variables
6. It may contain one or more threads of execution.

The basic interaction with a process from the user's perspective:

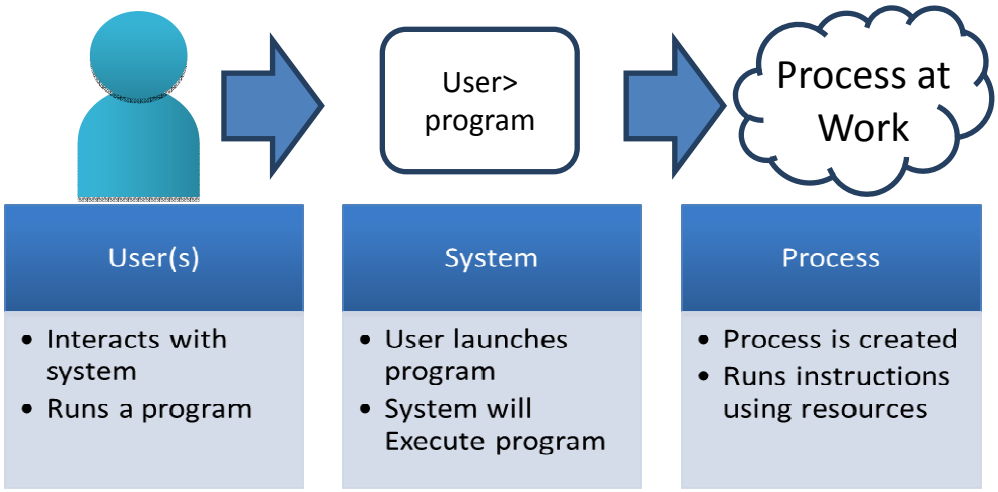


Figure 3.1: Flow of Process Creation

When a user interacts with a system (or cluster system), the user issues a command that launches a program to execute a task as shown in Figure 3.1. The system will then interpret the command and start executing the program by creating a process. The process will then execute the instructions of the program by using resources from the system and hardware [Vah96]. Once a program is executed on the system, a process is created by the program and the system will then assign:

1. A unique identification number or a Process ID (PID), which is assigned to each process when created.
2. The system will calculate its time slice that indicates the amount of time the process will execute in the CPU.
3. The system then calculates the process' priority that indicates the importance of execution (compared to other processes) and placement on the run-queue.
4. Finally the system assigns CPU affinity when more than one processing core exists, that is, assigns of a core for execution.

It is most likely more than one process exists in the system. The system itself creates processes that carry different instructions in order to provide services and environments for the user(s). Also, the user may concurrently execute more than one program or more than one instance of the same program at the same time. Each instance of the program will create its own unique process and have its own properties for that process. When more than one instance of the program exists, the system will then:

1. Calculate priorities and time slices for the processes that were created.
2. Assign the same CPU affinity since all instances of the program will have common instructions and identical data.
3. The processes will also be given the same affinity assignment in order to share cache-memory and similar hardware resources.

This can present an issue in trying to improve performance of a compute-node since the affinity assignments can hurt the overall performance of all programs, because processes will tend to be scheduled on one core. The main reason the system will share processing core affinity is to have as many threads that share resources in one place. As the previous research indicates is that it is desirable for tasks to have the same affinity in order to produce higher performance and processing core utilization. However as the number of processes increase either by different instances from one user or different users, these tasks would need to be grouped together in order to improve the processing cores utilization by grouping affinity to different applications. Therefore, nonrelated processes on different processing cores will have to be spread out the work load in both processors in the compute-node.

3.1.1 Process Descriptors and Task Structures

The kernel stores the list of processes in a circular doubly linked list called the task list as showed in Figure 3.1. The task list becomes very useful when the system needs to create connections between processes and namespaces, connection of process IDs (PIDs), and assigned group IDs (GIDs) to threads

and processes alike. Each element of the task list contains the process descriptor for each process that is created in the system. Process descriptor is of data type `struct task_struct`. The process descriptor contains the data that describes the task that is executing. The data can be information such as open files that it's using, address space, priority, process state, etc.

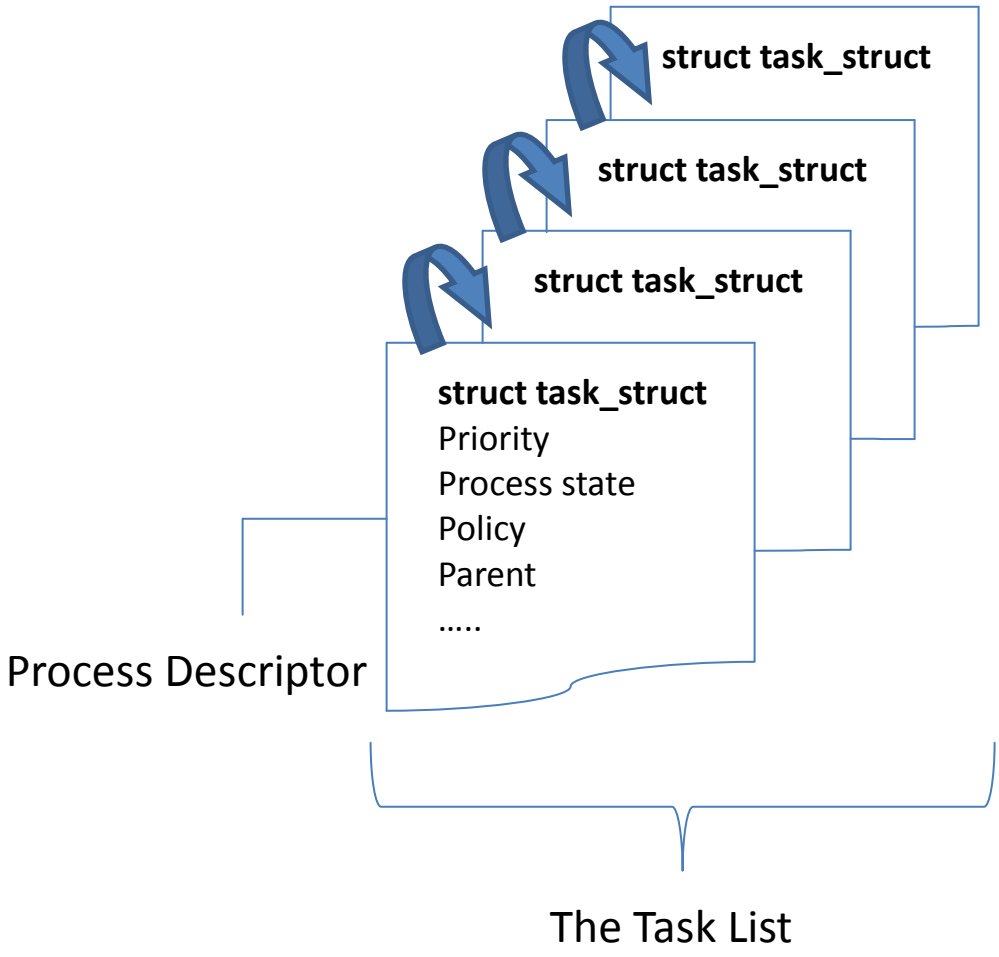


Figure 3.2: Double-linked Process Structures [Lov10].

Part of the process descriptor is the information that is stored in the `struct task_struct` for each process. The Linux kernel creates a `task_struct` for each process that is created in the system. The `task_struct` holds all the necessary information of an individual process that aids the Scheduler in how to manage each process. Some of `task_struct` elements are:

1. The priority information of the process. There are four types of priority values a process can have: normal priority value, a static priority value, real-time priority value and dynamic priority value which changes during execution.
2. Time slice value. This is the calculated value that the Scheduler assigns to each process for the time allocation it has in the CPU.
3. Policy value. This value helps the Scheduler in deciding how to treat the task or tasks in different methods
4. The `cpu_allow` variable is a bit field used on multiprocessor systems to restrict the CPUs on which a process may run [Mau08].
5. The `run_list` is a list head used to place the process on a run list.

In older Linux versions, `struct task_struct` was placed at the bottom of the Stack of every process. It was necessary back then since the x86 architectures did not have registers to spare for process operations and by placing the `task_struct` at the bottom of the Stack there was no need to save a Stack Pointer to any register. In current Linux versions, the `task_struct` is now dynamically allocated by the "slab allocator" which provides code reuse. A new structure is now allocated at the bottom of the Stack called the `struct thread_info` as seen in Figure 3.3 [Lov10].

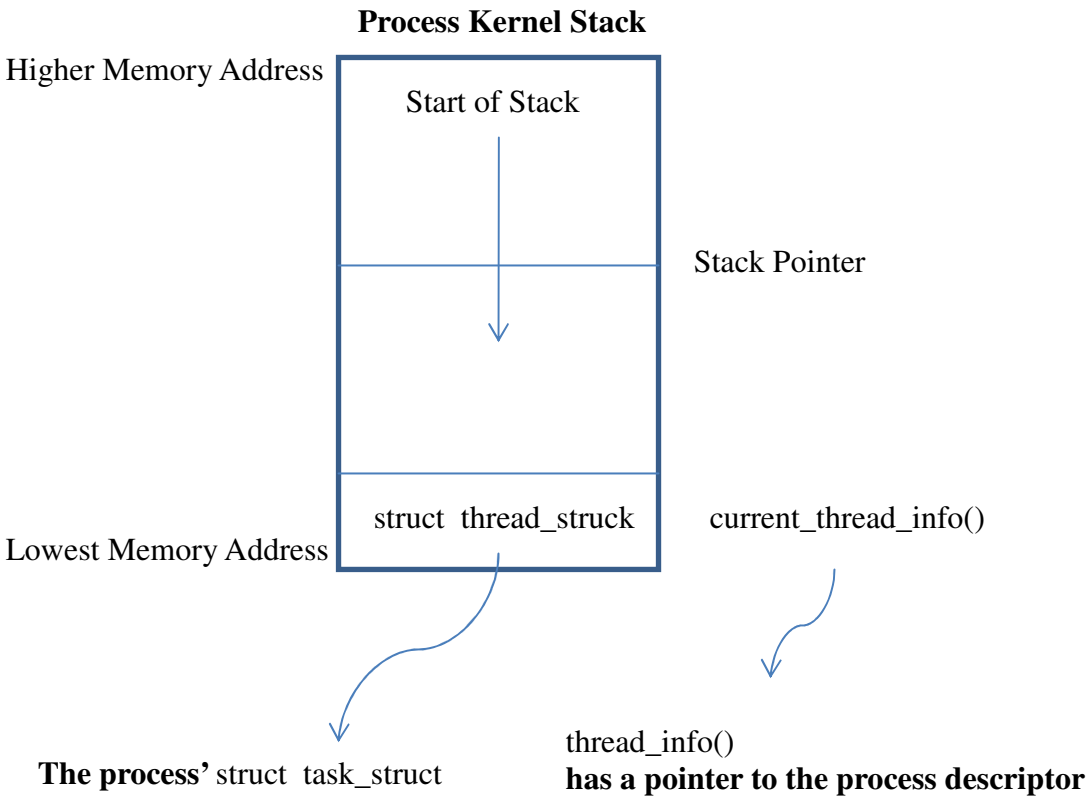


Figure 3.3: Process Kernel Stack and `thread_struct` Placement Diagram [Lov10].

3.1.2 Process State

Each process in the Linux kernel is always in one unique state at any instance of time. When a process is being created, it is getting ready for execution and most tasks will be set to the TASK_RUNNING - *Running* state as seen in Figure 3.4. If the process is not ready to execute, the system will set the process to the TASK_INTERRUPTABLE/ TASK_UNINTERRUPTABLE state in order to make room for processes ready to execute. If a processing is in the TASK_INTERRUPTABLE/TASK_UNINTERRUPTABLE state, the process cannot be set to a TASK_RUNNING - *Running* state directly but it is assigned to the TASK_RUNNING – *Ready but waiting* state first. The reason for this is because even though the process may be ready for execution there might not be any processing resources available at that moment as they may be servicing other

processes' execution. Once processing resources are available, the process will then change from TASK_RUNNING – *Ready but waiting* to TASK_RUNNING - *Running* for execution. Finally once a process is finished executing, the process will then be changed to a TASK_STOPPED state in where the process' resources are released and the process is terminated.

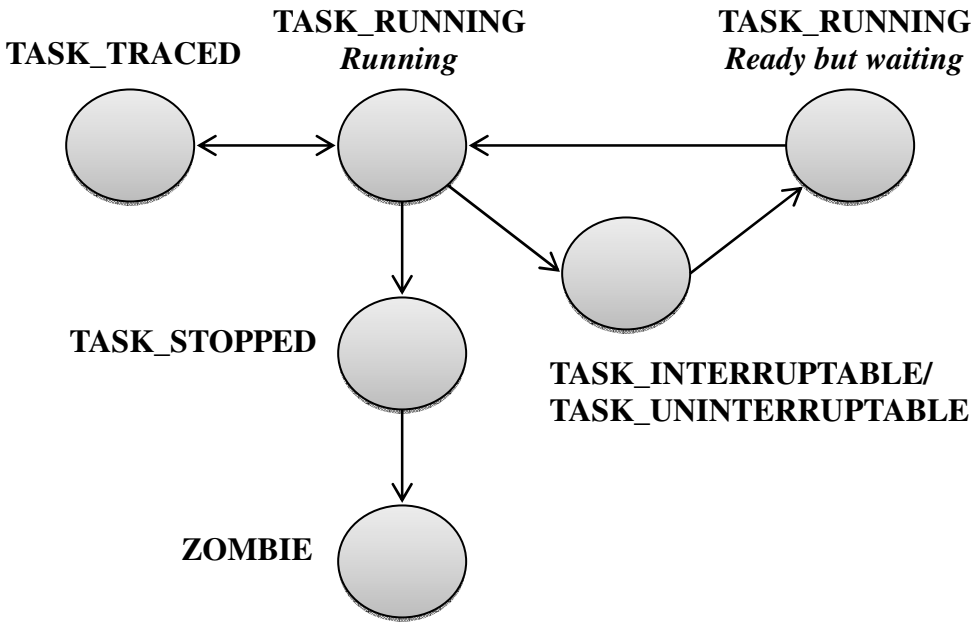


Figure 3.4: Transitions between Process States.

In the scheduler there are five states that a process that exists in the `task_struct` for each process that is created. The TASK_RUNNING state indicates that the process is runnable. The task is either TASK_RUNNING - *Running* or TASK_RUNNING – *Ready but waiting* for execution. This is the only possible state for a process executing in user-space or kernel-space [Lov10].

The TASK_INTERRUPTIBLE is assigned when the process is a sleeping state but can be interrupted by a signal. At this point the task is waiting for a condition to occur in order to exit from the TASK_INTERRUPTIBLE state. When the process receives the proper signal indicating that the condition exists, the state of the process is not directly changed to TASK_RUNNABLE – *Running* for

execution but instead it will be placed in the TASK_RUNNABLE - *Ready but waiting* state in order to check if processing resources available for execution. The TASK_UNINTERRUPTIBLE state is similar to TASK_INTERRUPTIBLE except the process does not wake up and become runnable if it receives a signal. It's used when a thread needs to wait without interruptions and only the kernel has permission to interrupt such tasks [Mau08].

The TASK_TRACED state is when the thread is being tracked by another thread. This state usually exists when debugging of a task is involved. The TASK_STOPPED state indicates that the process execution has stopped or finished execution. The process at this point will not run and it is not eligible to run anymore. This state happens when a thread receives a: SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal [Lov10]

3.1.3 Process Creation and Termination in Linux

A process is created by the `fork()` system call which creates a child process that is a copy of the current task. All resources of the original process are copied in a suitable way so that after the system call there are two independent instances of the original process [Mau08]. Traditionally when invoking `fork()`, the parent process is duplicated and the copy is handed to the child process. This approach of the `fork()` call is inefficient in that it copies a lot of data that might otherwise be shared. In Linux, `fork()` is implemented through the use of the "copy-on-write" (COW) technique. The purpose of COW is to delay or not copy data. Rather than duplicate the process address space, the parent and child can share a single copy. However, if the copy issues a write, then duplicate copies of the address space needs to be created and implemented [Lov10].

In Linux, `fork()` is called through the system call `clone()`. The `clone()` function takes a series of flags which specify which resources the parent and child processes should share. Then,

`clone()` calls the `do_fork()` system call which then calls `copy_process()` which has the following properties [Lov10]:

1. Creates a kernel stack, `thread_info` structure, and `task_struct` for a new process.
2. At this point, the parent and child processes have identical process descriptors.
3. It then checks that the new process will not exceed the resource limits for the number of processes allowed in the system.
4. Then child process obtains its own PID. The Child process is set to TASK_UNINTERRUPTIBLE state so it won't run yet.
5. The `copy_process()` calls `copy_flag()` to update flags in `task_struct`
6. Depending on flags, the `copy_process()` either duplicates or shares open files, filesystem information, signal handlers, process address space and namespace.
7. The `copy_process()` function cleans up and returns control.
8. Back in `do_fork()`, the new child is woken up and it's set to the Running state.

Once a process finishes executing its assigned workload, it changes state from TASK_RUNNING - *Running* to TASK_STOPPED as indicated in Figure 3.4. In this state, the processes begins the progression of termination by calling the `exit()` system call which calls the `do_exit()` function that performs the following steps[Lov10]:

1. It sets the exit flag in the processes' `task_struct`.
2. It removes kernel timers that manage time of execution for the process.
3. Then calls `exit_mm()` which releases any memory mapping that the process was using during execution. If no other process is using the address space (assuming that the thread was sharing address space) the kernel will destroy it.
4. Then calls `exit_sem()`, if the process was in a semaphore queue, the kernel will take it out of the queue.

5. Then calls `exit_files()` and `exit_fs()` which lets go of objects related to the file descriptor and filesystem data.
6. Then calls `exit_notify()` to send signals to the parent and sets the thread to `exit_state` in the `task_struct` to ZOMBIE state.
7. The `do_exit()` calls `schedule()` to context switch to another process in the TASK_RUNNING state.

The ZOMBIE state in Figure 3.4 occurs when the process terminates execution, but the parent process from which the process originates did not invoked the `wait()` or `wait4()` system call which acknowledges the termination of the child process. In order for the process to exit from the ZOMBIE state, either the parent process terminates execution and releases all information and resources in which any child process had inherited, or the ZOMBIE process is inherited by another process, usually the INIT process in Linux, which will invoke the `wait()` or `wait4()` system call for termination. If a SIGKILL signal is sent to a ZOMBIE process, the ZOMBIE process may not respond since at this state it is not a process anymore. The way the Linux kernel terminates a ZOMBIE process either invoked by the parent or INIT process is:

1. The `wait()` system call invokes `release_task()` system call.
2. The `release_task()` performs:
    a. Calls the `detach_pid()` to remove the process from the task list
    b. `__exit_signal()` releases any remaining resources of the dead process
    c. Calls `put_task_struct()` to free the pages containing the process' kernel stack and `thread_info`.

3.1.4 Linux implementation of Threads

The Linux OS has a unique implementation of threads in which it does not differentiate between threads and processes [Lov10] [Aas05][Mau08]. The purpose of this approach is to accomplish the illusion that each process owns the entire system and all of its resources when it is executing. Threads are just standard processes that shares resources with other processes from the same Thread Group (TG). Thus, each thread has its own `task_struct` like all other processes.

Threads are created through the `clone()` system call [Lov10][Mau08]. The clone system call passes flags or parameters to specify what kind of thread needs to be created. One flag that is essential to the Scheduler is the CLONE_THREAD flag. The CLONE_THREAD flag helps the Scheduler to identify which processes belong together in the same TG. This flag is important to the modified Scheduler because it is employed in the decision making for affinity assignment before sending the processes to the run-queue. Description of some of other `clone()` flags that can be used in the system call are shown in Table 3.1.

Table 3.1: Description of Flags for System Call `clone()`

Clone Flags	Description
CLONE_FILES	Share open files
CLONE_FS	Share filesystem information
CLONE_IDLETASK	Set PID to zero
CLONE_PARENT	Child process is to have the same parent process as its parent
CLONE_STOP	Start Process in the TASK_STOPPED state
CLONE_THREAD	Parent and child are in the same thread group
CLONE_VM	Parent and child share address space
CLONE_PTRACE	Continue tracing child
CLONE_SIGHAND	Parent and child share signal handlers and blocked signals
CLONE_VFORK	`vfork()` is called and the parent will sleep until the child wakes it

3.2 OVERVIEW OF THE SCHEDULER

The Linux kernel must have a form to have processes share CPU resources in order to have all processes efficiently accomplish their work. The Scheduler is the part of the kernel that manages processes that are ready to execute on the CPU (i.e. in the Running state) in a manner that creates the illusion that all tasks are executing concurrently. In order to accomplish a proper scheduling of tasks in the OS many other factors must work together in order to have a proper organization for processes to use the CPU. Factors such as scheduling policy, process priority, fairness and load balancing are important issues for the Scheduler as it handles processes for execution. The Scheduler that is used for this dissertation work is that in Linux kernel version 2.6.18 base, which is the version installed in the Virgo 2.0 cluster.

In Linux kernel 2.6.18, the system's scheduler employs run-queues and priority arrays to manage the tasks that are assigned to each core processor in a MCH environment. An illustration of the interaction is shown below in Figure 3.5.

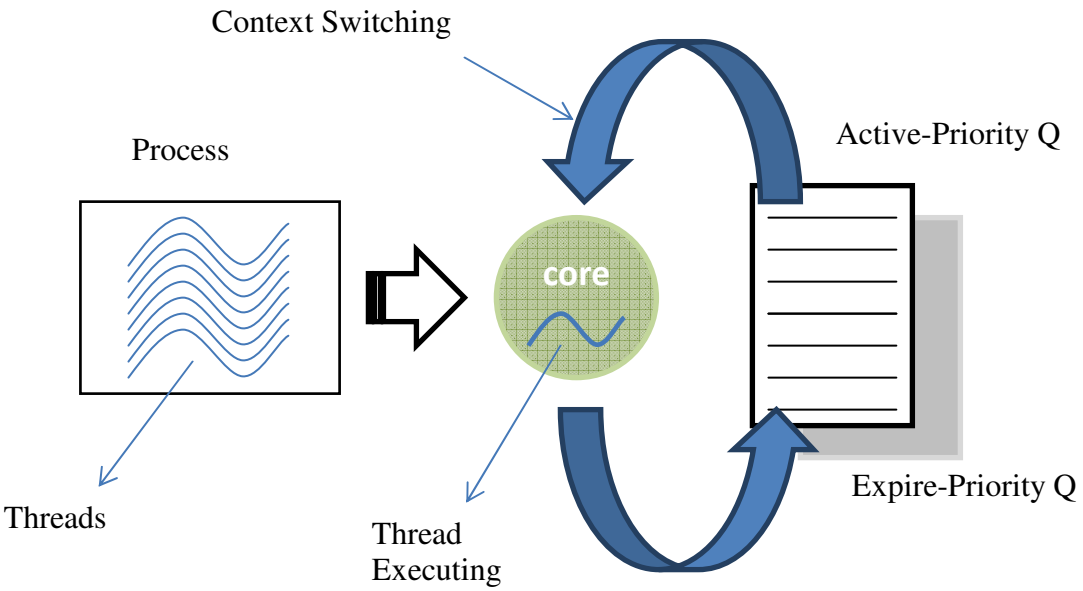


Figure 3.5: Flow of Thread Execution Block Diagram

From Figure 3.5, a process will have one or more threads that need to be executed. A thread that is TASK_RUNNABLE is initially placed on the Active-Priority Q in priority order. Threads with equal priority are placed on the Q in Round Robin order, oldest first. The thread will wait for the current executing thread to be preempted or context switch or run out of its time slice. The thread that was executing on the processing core will then be moved to the expired-priority queue array upon preemption. Then, the next high priority thread on the Active-priority array will be context switched to the processing core.

Runqueues are the main data structures that make up the *O*(1) Scheduler. A runqueue keeps track of all runnable tasks assigned to a particular CPU or processing core. As such, one runqueue is created and maintained for each processing core in a MCA [Aas05]. Each of the runqueues has two priority arrays as shown in Figure 3.5.

The runqueue has two main jobs on the scheduler: it has to keep track of thread information and manages the priority arrays. The understanding of the runqueues becomes essential when creating a modified *O*(1) algorithm since most of the management of the threading can be found in its data structure. The information of the threads can be found in the process descriptor as described in section 3.1.1.

Priority arrays allow the Scheduler's algorithm to find the task with the highest priority in constant time [*O*(1)] [Aas05]. Each of the priority arrays is broken down to two different arrays each with 140 priority levels: active array and expired array. The Priority arrays consist of an array of linked list in which each pertains to a priority level. Therefore, each task that is scheduled as a TASK_RUNNABLE is given one of the possible 140 priority levels and placed on the active array. When multiple tasks are given same priority value/level, the Scheduler will then execute all tasks in a round robin format. Once one of the tasks is finished running, the task will be placed at the bottom of

that priority level's list. And even though multiple tasks were assigned the same priority values, the scheduler will continue to execute in $O(1)$ constant time.

Once each task completes its timeslice, the task will then be removed from the active array and placed to the expired array. During the move from active array to the expired array, the Scheduler will calculate a new timeslice for that task. Once all tasks on the active array have ran out of timeslices, then pointers for the active and expired arrays swap places and reverse roles. Because timeslices are recalculated when they ran out, there is no point at which all tasks need new timeslices calculated for them; that is, many small constant time operations are performed instead of iterating all of the tasks, and calculating timeslices for all of them (which would be undesirable) [Aas05].

This Scheduler is implemented using $O(1)$ algorithm. This means that context switching time is constant, irrespective of the number of threads/processes waiting for execution. (See section 3.5 for additional details.) In a cluster system, we can safely assume that all user threads are CPU-Bound. This aids in simplifying the analysis of the scheduler's affinity for each processing core since efficient use of the cores is more important than providing fast context switching response (fairness) to interactive processes as with a desktop environment. Since the Scheduler can only have one thread executing on a processing core at a time, the rest of the threads that are not executing, both from the same process and other processes, are placed on the Active-priority run-queue for the assigned processing core.

For the $O(1)$ Scheduler, for processes with equal priority, only the wait time of a process is considered, that is, how long it has been sitting around in the run-queue ready to be executed. The task with the greatest need for CPU time is scheduled. Every time the current scheduler is called, it picks the task with the highest priority, and highest waiting time within that priority run-queue, and gives the CPU to it. If this happens often enough, no large unfairness will accumulate for tasks, and the unfairness will be evenly distributed among all tasks in the system [Mau08]. This type of scheduling is called Completely Fair Scheduling (CFS). Although it is very important to have fairness of tasks in the

scheduler for execution, it is not a desirable concept to apply to large, CPU-bound tasks that run in large systems such as clusters. In that case, it is preferable that a task have as much CPU time as possible in order to complete its workload and then move to the next large task for execution. This type of scheduling can be accomplished by modifying the current *O*(1) Scheduling algorithm.

3.3 PRIORITY AND POLICY HANDLING

The goal for priority of tasks in the Scheduler is to rank processes based on their worth and need for CPU time [Lov10]. The tasks with high priority will be scheduled to run first, before tasks with lower priorities. Also, tasks with higher priorities will have longer time slices to execute on a CPU. All priorities for all tasks are calculated in the Scheduler.

There are two different sets of priority calculations that are made in the Scheduler. One set of priority calculations are known as the nice values. The nice values range from -20 to +19 with a default of 0. These values are given to "normal" processes scheduled to execute. A normal process is described as a task that has no specific time constraints but can still be classified as more important or less important by assigning a nice value to it [Mau08].

The other set of priority calculations are known as real-time priority. The real-time priority values range from 0 to 99 with 0 being the highest priority. These priority values are assigned to tasks that are classified as real-time processes, that is, processes that require execution within a limited, fixed time period. Consequently, real-time processes are tasks that are assigned higher priorities than normal processes and have different priority values. Using real-time priorities for compute-node processes is attractive because they are assigned higher execution priorities than normal processes. More on real-time processes and its policies are given in the following section.

Policy is the behavior of the Scheduler that determines what runs when. A scheduler's policy often determines the overall feel of a system and is responsible for optimally utilizing processor time

[Lov10]. The Linux version 2.6.18 supports four possible scheduling policies that can be applied to processes. The SCHED_NORMAL is the scheduling policy for normal processes. This policy is mostly applied for the CFS scheduler. The SCHED_BATCH policy is dedicated policy for CPU-intensive batch processes that are not interactive. This policy will not preempt another process handled by the CFS scheduler and will therefore not disturb interactive tasks.

The SCHED_RR policy is used for real-time processes. This policy implements the round-robin method when assigning CPU time to tasks of equal priority. Since the Linux kernel does not have CFS scheduling, the SCHED_RR policy is applied to all created tasks within the system. Tasks that are assigned SCHED_RR will have real-time priorities and are placed on the Active priority array depending of the calculated priority value by the Scheduler. The SCHED_FIFO policy is also used in real-time processes. This is the First-in First-out method when allocating CPU time to all tasks. When tasks are assigned SCHED_FIFO policy, this means that the tasks have real-time priorities which are placed on the Active priority array in a FIFO order. As they are placed in a FIFO order in the priority array, the Scheduler will still calculate and assign priority values to all FIFO tasks. When this policy is used, the time slice for all tasks is not calculated and lets the first task have the high priority and unlimited CPU time.

3.4 REAL-TIME SCHEDULING POLICIES

As mentioned earlier, Linux provides two real-time policies SCHED_RR and SCHED_FIFO. The real-time processes are not handled by the CFS [Lov10]. Once a SCHED_FIFO task is running, it continues to execute until it blocks or intentionally gives up the processing core. SCHED_FIFO tasks do not have time slices and they may run indefinitely. This scenario is desirable for the cluster since by avoiding preemption, performance increases with increased processing core utilization. A

SCHED_FIFO task can only be interrupted or preempted by other FIFO tasks that have higher priority or by system interrupts [Aas05].

SCHED_RR is similar in operation to SCHED_FIFO but with time slices. A task using SCHED_RR will only run until it exhausts its time slice. Once the time slice ends, the Scheduler implements a context switch to begin executing the next highest priority SCHED_RR task. This policy is better when the system needs to have more fairness in execution of all tasks in dealing with real-time processes. However as part of the modifications of the Scheduler, we want to utilize all processing cores as much as possible, and it seems that the proper approach is to have the each compute-node Scheduler employ the SCHED_FIFO policy for all user tasks. This approach will have each task execute until it finishes the workload, yields the processing core, or is interrupted by the system, and tasks will not have to waste time and resources in context switching and calculating time slices, except where needed.

3.5 FAIR SCHEDULING VS. O(1) SCHEDULING

Completely Fair Scheduler (CFS) is a scheduling algorithm that will run each process for some amount of time in round-robin order, then context switch, selecting next the process for execution that has run the least. Rather than assign each process a timeslice, CFS calculates how long a process should run as a function of the total number of runnable processes. And instead of using the nice value to calculate a time slice, CFS uses the nice value to weight the proportion of processor time a process is to receive: Higher valued (lower priority) processes receive a fractional weight relative to the default nice value, whereas lower valued (high priority) processes receive a larger weight [Lov10].

This scheduler is designed to increase fairness for all processes in the Running state including interactive processes in the system. However, this is a concern because for a cluster, as the number of tasks increases, the fairness of the CFS can affect the performance of the overall system as it tries to be fair to all scheduled tasks and the system will waste time with excessive context switching. (Context

switching is pure system overhead.) The CFS scheduler was implemented in Linux kernel version 2.6.23 which is not supported on the Linux kernel version in the Virgo 2.0 cluster.

The *O*(1) Scheduling algorithm is the current scheduler on the Linux 2.6.18 kernel. The big-*O* notation is used to denote the growth rate of an algorithm's execution time based on the amount of input [Aas05]. Previous schedulers contained *O*(*n*) algorithms which indicated that the algorithm increases linearly or more as the input grows in size *n*. For example - the running time of $O(n^2)$ grows quadratically. Therefore, an *O*(1) algorithm is one that guarantees to operate at a constant time independently of the load size coming in to the input. The Linux 2.6.18 *O*(1) Scheduler includes per-core structures called run-queues, which contain the set of runnable threads assigned to each core in the MCA environment [Zie08].

This becomes ideal for the Virgo 2.0 cluster system because as the number of tasks increases for execution, the scheduler will be able to handle more tasks to be scheduled without additional system overhead. The idea in implementing a CFS Scheduler for the kernel would be unnecessary since none of the compute-nodes have any interaction processes to execute at any time. Therefore in combining the *O*(1) Scheduler algorithm and the SCHED_FIFO policy of each of the tasks executing on each compute-node, will help to improve system performance.

3.6 SCHEDULER'S LOAD BALANCING

Another consideration that requires attention as well is balancing the distribution of work to all the processing cores in the system. When using the *O*(1) Scheduler, spawned threads for a process will be assigned an affinity that will place them all in the same processing core in order to localize all resources for execution. However once a processing core is overloaded, the *O*(1) Scheduler performs load balancing to efficiently utilize all available cores [Aas05] by migrating some of the tasks to other processing cores that are available for execution. Some tasks cannot be migrated, either because they

are not runnable on all cores, or because they have recently run and are still considered having useful data in the cache at their core (known as cache hot status). However, threads can be forced to move regardless of cache hot behavior if load balancing fails a sufficient number of times. Further, a core invokes idle balancing at any time before going idle, thus trying to find work for it to perform even though its own queues are empty. Like load balancing, idle balancing attempts to steal tasks from the currently busiest core to bring the run queues into a rough balance [Zie08].

The approach in modifying the Scheduler is to modify the load balancing processing that is implemented by the *O*(1) Scheduler. The idea is to stop the 200ms intervals for the Scheduler to perform load balancing only when needed and instead, place load balancing at the beginning of the Scheduler algorithm for tasks that are ready for execution. We hope that this approach would more equally distribute tasks across all cores and reduce the number of times the Scheduler would need to perform load balancing processing. Further discussion of the Scheduler modifications details are found in Chapter 5.

Chapter 4: Performance Analysis on a Compute-Node

This chapter focuses on previous work that aimed in finding bottlenecks and limitations that exist in the hardware of a compute-node in the Virgo 2.0 cluster system. The work entailed using the High Performance Linpack (HPL) benchmark, the Automatically Tuned Linear Algebra Software (ATLAS) that provides the Basic Linear Algebra Subprograms (BLAS) requirement for HPL, and MPICH, open source software which provides the Message Passing Interface (MPI) standard for communication between processing cores. This work was presented at the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA09) [Val09].

4.1 ROCKS 5.0

Rocks version 5.0 is an open-source OS for Beowulf clusters that manages distribution of tasks from the front-end machine to the compute-nodes. The Rocks OS runs over the Linux 2.6.18 OS kernel in all machines of the cluster. Rocks also provides other programs to help users to administer and manage the all the compute-nodes, the front-end node, with a variety of different tools for the system. The programs that are provided by Rocks are:

1. Area51 – This program uses a combination of two other programs, Tripwire and Chkrootkit, which provide administrative information about the processes and the */proc* filesystem, and monitoring and alerting about specific files over the whole system.
2. Bio Roll - Contains a suite of Bio-informatics applications, most commonly in use in the bio-informatics fields, and uses many other applications such as MPI libraries, BLAS libraries, perl language compilers, etc.
3. Condor - Is a specialized batch system for managing compute-intensive jobs. It provides a queuing mechanism, scheduling policy, priority scheme, and resource classification.

Users submit their compute jobs to Condor; Condor puts the jobs in a queue, runs them, and then informs the user as to the result [Con11].

4. Ganglia – Is a scalable distributed monitoring system for HPC systems.
5. HPC – Is a program that tests and uses OpenMPI and MPICH programs to test the routines that communicate with other nodes in the system.
6. Programming language support – Provides compiling environments for languages such as C, java, python and perl.
7. Grid Engine – Now owned by Oracle, is distributed resource management software that allows the resources within the cluster (CPU time, software, licenses, etc) to be utilized effectively [Cal11].
8. Xen – Is an open source industry standard for virtualization that offers a powerful, efficient, and secure feature set for virtualization of most CPU architectures. This program can be used to create a virtual cluster or virtual nodes within the system.

The most used ROCKS application for this dissertation is the Ganglia software. It helps to monitor each of the compute-nodes that are running tasks and provides hardware performance numbers for each of the compute-nodes. This tool was very handy when executing large problems in order to understand how the hardware on each compute-node was reacting during execution.

4.2 ATLAS LIBRARIES

The ATLAS (Automatically Tuned Linear Algebra Software) project is an ongoing research effort focused on applying empirical techniques in order to provide portable, high performance linear algebra software. Linear algebra routines are widely used in the computational sciences in general, and in scientific modeling in particular. In many of these applications, the performance of the linear algebra operations are the main constraint preventing the scientist from modeling more complex problems,

which would then more closely match reality. This then dictates an ongoing need for highly efficient routines; and as more compute power becomes available, the scientist typically increases the complexity and accuracy of the model until the limits of the computational power are reached [Cli07].

ATLAS also contains many other programs that are used during the configuration, build and installation of the program in order to analyze the architecture for performance tuning purposes. One of the tools that can be found is the Tuning CacheEdge program which can help improve performance up to 15% by benchmarking the cache memory hierarchy [Cli07]. There are three levels of BLAS in ATLAS:

- Level 1 - Vector-Vector operations
- Level 2 - Matrix-Vector operations
- Level 3 - Matrix-Matrix operations

Lack of hierarchical memory would, at worst, turn some of ATLAS' blocking and register usage into overheads. BLAS has a subroutine called GEMM (General Matrix Multiply), which is the main performance kernel at Level 3. GEMM performs a general matrix operation:

$$C \leftarrow \alpha * op(A) * op(B) + \beta * C$$

, where

$$op(A) = A^T or\ A$$

Matrix *C* is *MxN* matrix, and *op(A)* and *op(B)* are matrices of size *MxK* and *KxN* respectively. Inside the GEMM umbrella there are many other programs that help in tuning the performance of the ATLAS program and provide subroutine functionality to GEMM.

Overall, the ATLAS libraries provide the necessary algebra subroutines in order for the HPL benchmark to conduct proper calculations. ATLAS was configured and installed in one of the compute-nodes in order to test the performance with the HPL benchmark. Also, the Tuning CacheEdge program

was used in order to correctly measure the high performance hit-and-miss ratio for the L1 cache of the processors for better performance of the benchmark.

4.3 MPICH LIBRARIES

MPICH was developed during the MPI standards process to provide feedback to the Message Passing Interface (MPI) Forum on implementation and usability issues. With the release of the MPI standard, MPICH was designed to provide a free, open source implementation of the MPI standard that could replace proprietary message-passing systems on massively parallel computers [Mpi11]. MPICH libraries are utilized by the HPL benchmark in order to provide communication with other processing cores in the system. For this work, MPICH establishes desired communication to specific processing cores, as needed, to determine the limitations of the hardware.

4.4 HIGH PERFORMANCE LINPACK (HPL) BENCHMARK

Benchmarking is necessary to measure the timing and performance of the system in order to understand the architecture. For our work, High Performance Linpack (HPL) 2.0 was utilized in order to benchmark the performance of a single compute-node and to spawn a variable number of processes to each core. The release of HPL that we used was developed by Innovative Computing Laboratory (ICL) from the University of Tennessee [Hpl08].

HPL is a portable implementation of the High Performance Linpack benchmark for distributed memory computers. HPL uses a two-dimensional blockcyclic data distribution algorithm that employs LU factorization with pivot search and column broadcast in order to distribute the tasks throughout the MCA [Hpl08]. MPI is employed to perform communication between the cores. This is made possible by the MPICH install libraries. Also, HPL requires either the BLAS or the Vector Signal Image Processing Libraries (VSIPL) for proper operation. We used the ATLAS BLAS libraries for this work.

HPL solves a linear system of order n: A x = b by first computing the LU factorization with row partial pivoting of the n-by-n+1 coefficient matrix [A b] = [[L,U] y]. The lower triangular matrix L is applied to b as the factorization progresses and the solution x is obtained by solving the upper triangular system U x = y. The lower triangular matrix L is left unpivoted and the array of pivots is not returned. The data is distributed onto a two-dimensional P-by-Q grid of processes according to the block-cyclic scheme to ensure "good" load balance as well as the scalability of the algorithm. The n-by-n+1 coefficient matrix is first logically partitioned into Nb-by-Nb blocks, that are cyclically "dealt" onto the P-by-Q process grid. This is done in both dimensions of the matrix [Hpl08]. In order to execute the HPL benchmark, therefore, the following variables must be defined in the HPL.dat file:

- N – The number of problems sizes that will be used for execution of the benchmark.
- Ns – This specifies the exact value(s) of the problem size. The significant value read in this line depends on the specified valued for N.
- NBs – It specifies the values of each block size to be executed for each problem size defined in Ns.
- Ps – Specifies the number of processes row value
- Qs – Specifies the number of processes column value.
- Most of the other parameters in the HPL.dat file are left on default values.

The HPL benchmark was chosen for this work because of its data-intensive and memory-intensive properties. It is the standard benchmark that is employed to determine the speed of high speed computer systems. For the TOP500, Linpack is chosen because it is widely used and performance numbers are available for almost all relevant systems [Top11]. By assigning large problem sizes, large block sizes and by changing the number of processes, it gives a good representation of large applications that are executed in cluster systems. Also, HPL can help to determine the effects that exist in both of the

FSBuses of each processor and see the effects of the MCH as each process accesses memory during execution.

4.5 EXPERIMENTAL SETUP

Once all software was properly installed, the first trials consisting of running only one process to compute the HPL benchmark were executed. However, this did not convey the relationship of an executing core in respect to the other cores. Therefore, an additional set of runs which spawned additional processes to execute the HPL benchmark were employed. Here, the primary core, processing core 0, in the processor became the master core and assigned the second process to any of the other cores within the same processor as shown in Figure 4.1.

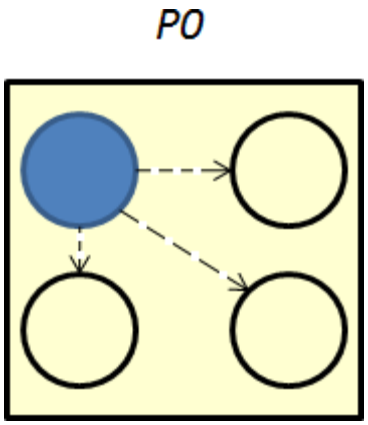


Figure 4.1: Processing Cores within First Processor

The communication between the master processing core and another core within the same processor is the simplest form for running two processes in separate cores. In addition to employing two cores, different combinations of problem, block and grid sizes of the HPL benchmark were launched which employed processes executing on up to all eight cores of the compute-node. This gave us the opportunity to discover the relationships of the master core to the other cores in the processor and to the other four cores in the second processor. Figure 4.2 shows the spawning of seven processes throughout

the compute-node in relationship to the master core, where, the eighth process is executed locally in the master core. In order for the master core to communicate with the four cores in the second processor, it must follow the path to the MCH via its FSB. It will then have to request access to the FSB of the second processor and be able to have the processes arrive successfully at the appropriate cores in the second processor.

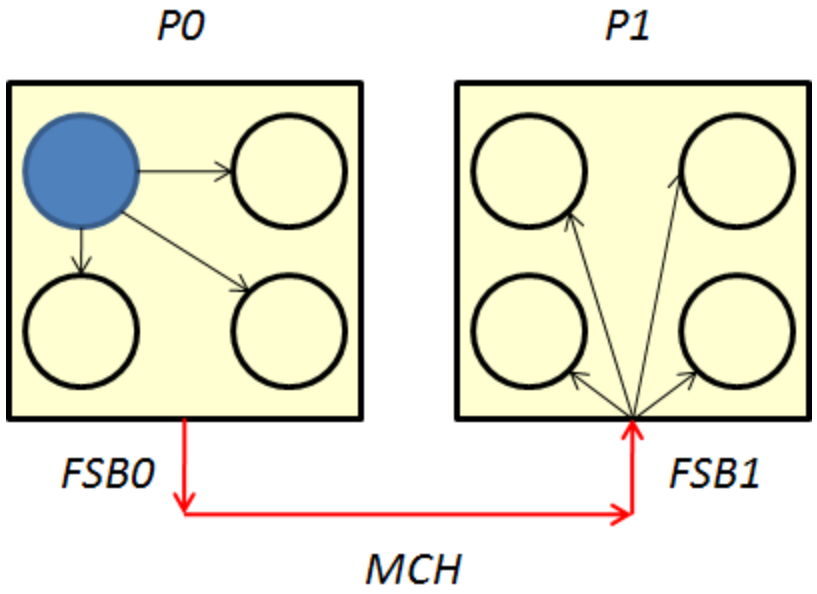


Figure 4.2: Spawn of Eight Processes of HPL through in Two Quad Processors

4.6 TEST RESULTS

Figure 4.3 shows the average execution times for the two processes spawned within one compute-node. Four different problem sizes Ns = {1000, 2500, 5000, 10000} were used with matrix dimensions of Ps = 2 and Qs = 1 and vice versa, and executed each problem size with nine different block sizes NBs = {8 – 2048}. These results show that when executing the benchmark with only two processes, the time to execute the benchmark is the greatest for any problem size at small block sizes. This is due to the program passing data in smaller sizes than can efficiently sent to their destinations.

As the block sizes reach the size of data buses in the architecture, execution time is the smallest with block sizes of 32 to 512 for all problem sizes. This is more noticeable at problem size 10000. At these block sizes, the data is passed through the hardware taking advantage of the bus widths in the architecture and the bandwidth speeds. This minimizes the duration that a process has to own data paths on either the FSB or MCH and therefore reduces the time of execution. Problem sizes 1000, 2500 and 5000 are too small to exhibit the execution time changes as the block size increases.

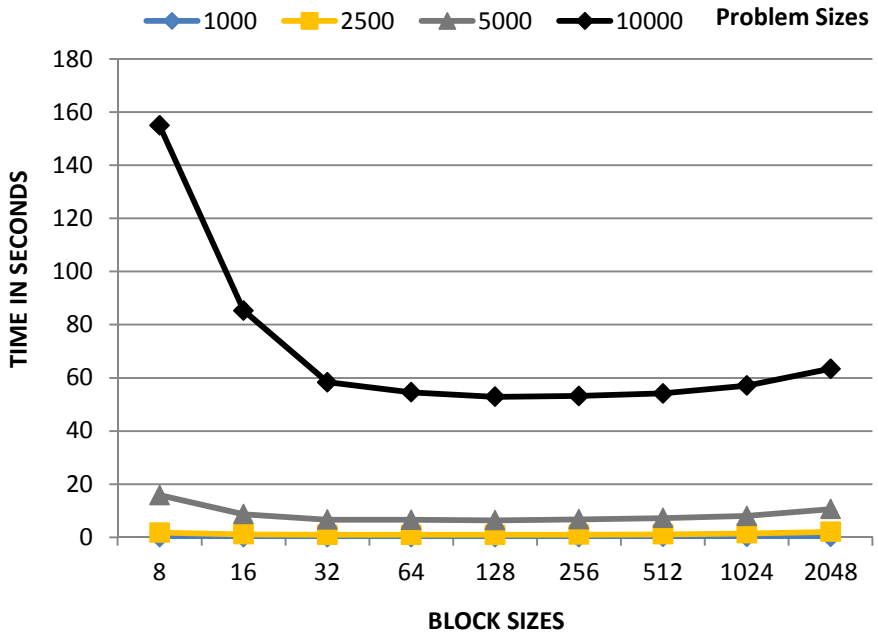


Figure 4.3: Average Execution Time of Two Processes Spawned in Same Processor [Val09].

Once the block sizes go over 512, the execution time begins to increase again as data is taking longer once again in reaching different destinations. At this point data is passed in larger blocks that are much wider than the bus widths. Each process will then have to lock and own buses in the FSB and MCH for longer period of times. This causes a bottleneck as the two processes try to pass data through between the processing core and read/write to memory through the MCH in the same FSB line.

Figure 4.4 shows results of other problem sizes for the same two process scenario that are not included in Figure 4.3. As the problem sizes increases over 10000, execution time is the highest for the smaller values of block sizes. This again is an indication again of a process having

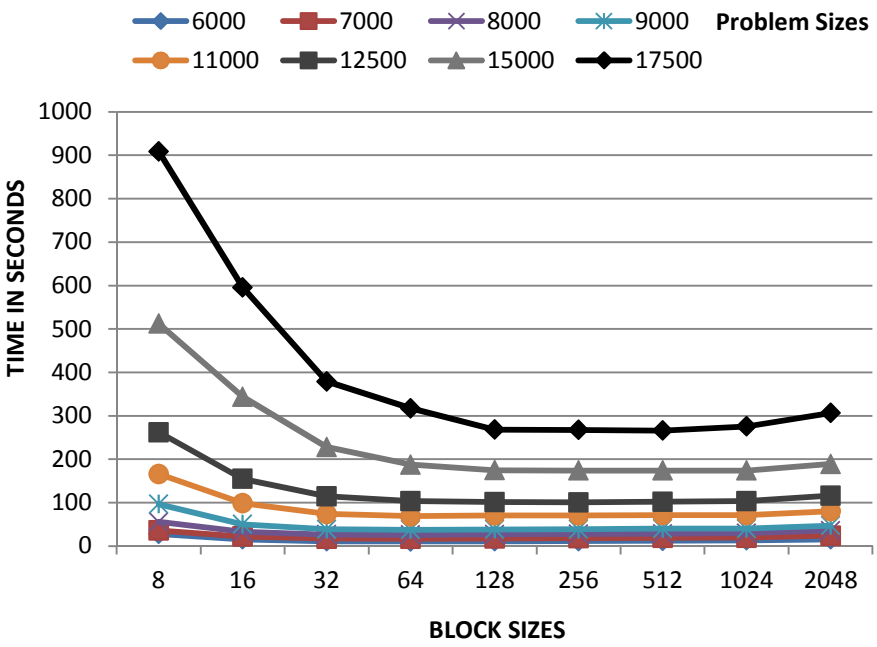


Figure 4.4: Average Execution Time of Two Processes Spawned with Other Problem Sizes.

to lock data paths in the FSB and MCH more times than necessary because of the mismatch of the block size to the data path sizes. This generates contention when the other process needs to access the same data paths to execute. And again, execution times tend to be minimal for block sizes of approximately 32 to 512 because of the compatibility with the data bus widths of the FSBs and MCH, and although only two processes are executing in the compute-node, they can still saturate all memory and data paths.

Figure 4.5 shows the average execution times for eight processes spawned within one compute-node. The same problem sizes of Ns = {1000, 2500, 5000, 10000} were used with matrix dimensions of Ps = 8 and Qs = 1 and vice versa, and executed each problem size with nine different block sizes NBs = {8 – 2048}. These results show that when executing the benchmark with eight processes it takes longer

to execute the benchmark for all problem sizes at small block sizes. However, there is a difference when executing problem size 10,000 at small block sizes. When increasing the problem size from 8,000 to 10,000, there is a factor 8X at a block size of 8. As before, this is because the contention between the processes occurs more often as the data is passed in small increments requiring more iterations of locking the FSB and MCH, etc., and thus taking more time to transfer data to and from memory.

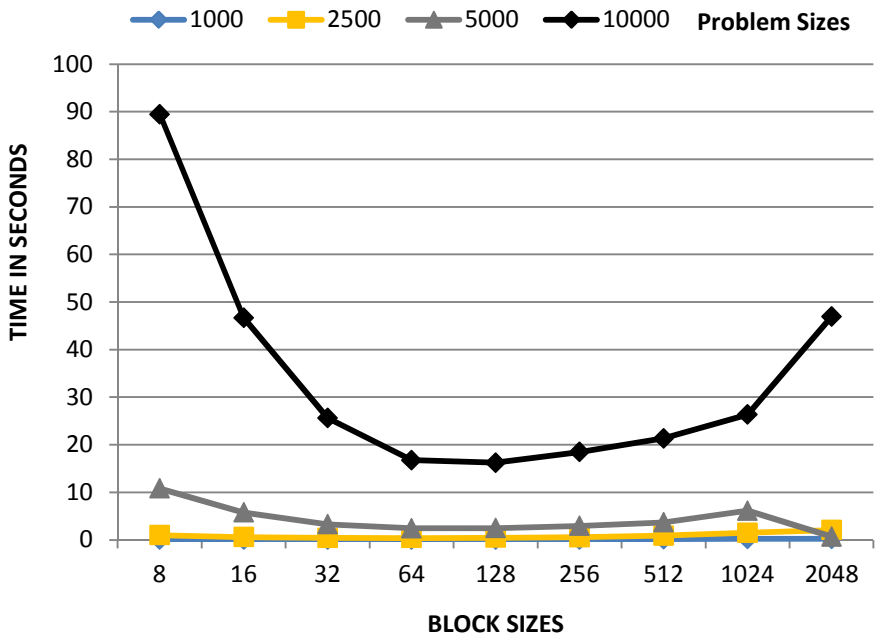


Figure 4.5: Execution Time of Eight Processes Spawned in Same Compute-node [Val09].

As the block sizes reach the size of data buses in the architecture, the execution time improves for block sizes of 64 and 128 for all problem sizes. This is more noticeable at problem size 10,000. This minimizes the duration that a process has to own data paths for each block size in either the FSB or MCH, therefore, reducing the execution time.

Once the block sizes go over 512, execution time begins to increase as now data is once again taking longer to reach different destinations. In Figure 4.5, the contention is more noticeable for large block sizes (e.g. 10,000) as the increase in execution time is much larger. In the case of eight processes,

heavier contention occurs than for the two process scenario as each process attempts to obtain ownership of its FSB and to ultimately obtain ownership of the MCH. Figures 4.6 shows the results for problem sizes not included in Figure 4.5.

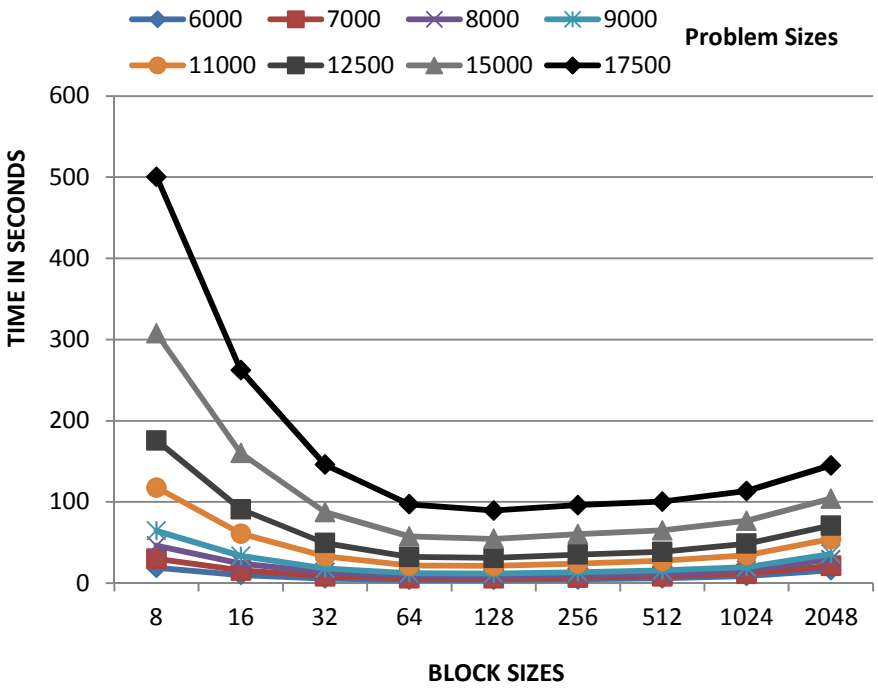


Figure 4.6: Average Execution Time of Eight Processes Spawned with Other Problem Sizes.

In comparison of Figure 4.3 and Figure 4.5, there is a large improvement in execution time with the increase in the number of processes. This is due to the fact the each problem is delegated to more processing cores and each processor can share more of its resources among all of its processing cores. This allows higher hit rates at all cache memory levels and less contention for the FSB and MCH. The largest improvements can be detected for block sizes between 32 to 128.

Figure 4.7 shows the average performance in Giga Floating-Point Operations per Second (GFLOPS) for the two processes spawned within one compute-node. The same four problem sizes and nine block sizes were launched for the HPL benchmark as are shown in Figure 4.3. Figure 4.7 shows that when executing the benchmark with only two processes, it peaks in performance at different block

sizes. With smaller problem sizes, the benchmark does not reach the total potential of the hardware; therefore, the peak performance for small problem sizes only reaches low GFLOPS values. This is due to the fact that the program is passing data at smaller sizes which takes longer to transmit to their destinations, thus keeps the processing cores waiting some of the time.

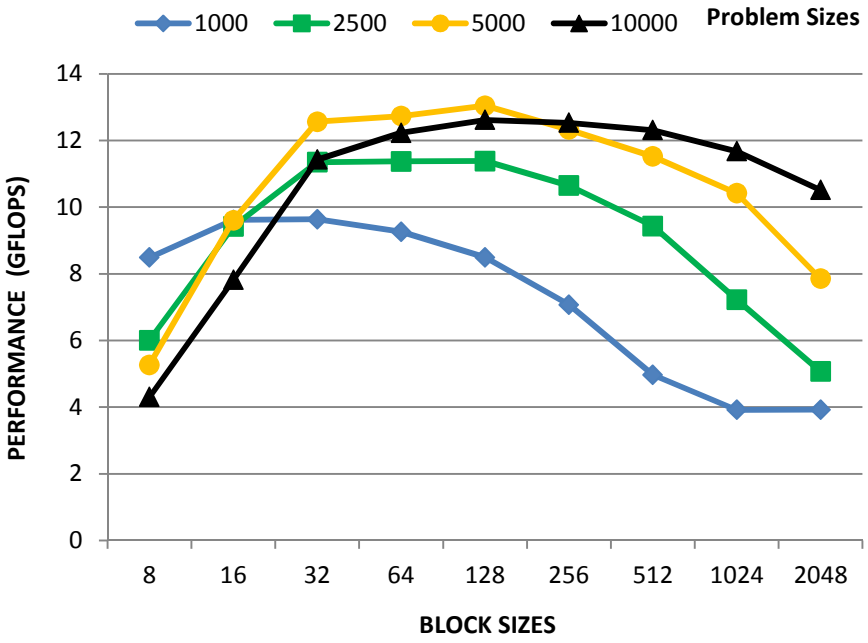


Figure 4.7: Average GFLOPS of Two Processes Spawned in Same Processor [Val09].

As the block sizes increase for most of the problem sizes, the performance of the compute-node starts to increase. Within the range of block sizes of 32 to 128, the performance peaks as the block sizes begins to match bus size and begins in taking advantage of the full bandwidth speeds. This gets the data to different destinations much faster, including the processing cores, which are then kept busy executing the benchmark code. For this scenario, the GFLOPS shown in Figure 4.7 are low since not all processing cores are employed.

As the block sizes continue to increase, as shown in Figure 4.7, the performance of the compute-node begins to decline starting around block size 256. With larger block sizes, the data buses take longer

to transmit information to the processing cores, and although part of the data is arriving, more bus cycles are required for a complete transfer of each data block and the processing core utilization slows down in executing the benchmark.

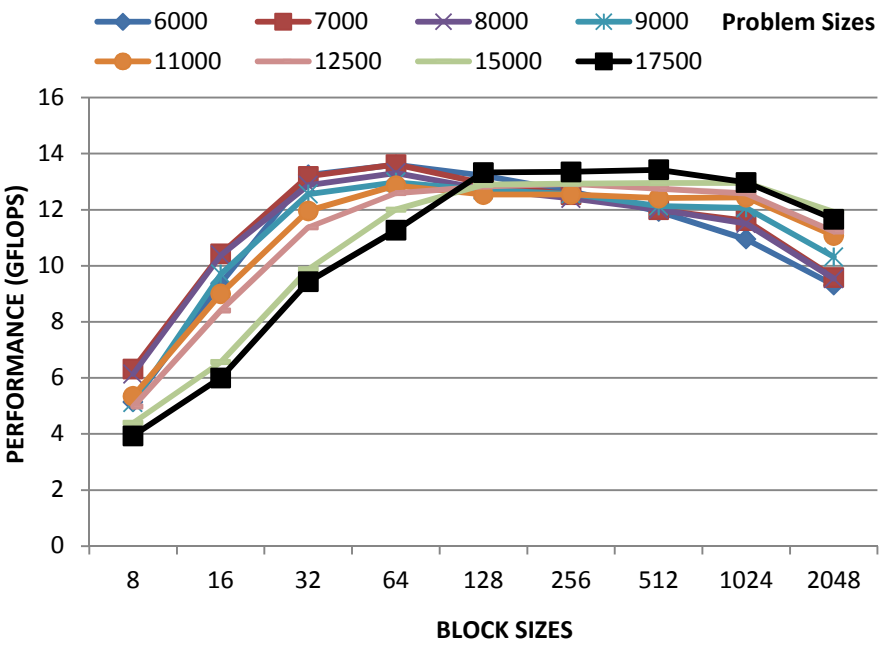


Figure 4.8: Average GFLOPS of Two Processes Spawned with Other Problem Sizes.

Figure 4.8 shows the addition problem sizes, identical to those illustrated in Figure 4.4, for executing the HPL benchmark with two processing cores. As the problem size increases above 10,000, the performance of the compute-node begins to struggle to reach peak performance as the block sizes increases. In this scenario, the performance values essentially match the performance from problem size 10,000 (almost 14 GFLOPS) shown in Figure 4.7. Therefore, it can be concluded that the highest performance of two processing cores was reached in this scenario.

Figure 4.9 shows the average performance when eight processes are spawned within a compute-node using the same four problem sizes and nine block sizes that are shown in Figure 4.5. Figure 4.9 shows a more uniformed peak performance for all problem sizes for block sizes between 64 and 128.

However there is still a lack of performance for small problem sizes. This is due again to the fact of the small block sizes cannot be transferred as fast as larger ones, keeping the processing core waiting for information to completely arrive.

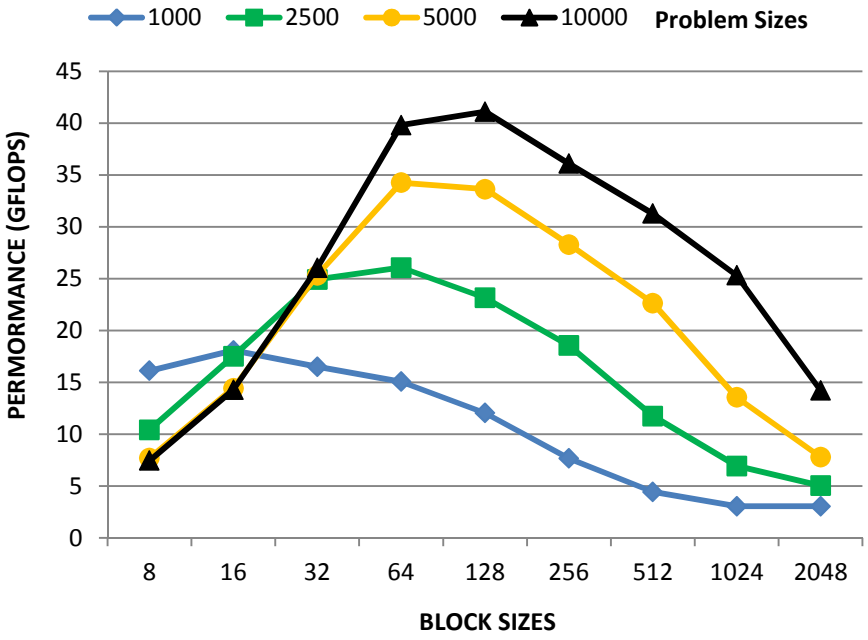


Figure 4.9: Average GFLOPS of Eight Processes Spawned in Same Processor [Val09].

As the block sizes increase for most of the problem sizes, the performance of the compute-node increases and as before, maximum utilization occurs in the middle of the block size range with values that most closely match the size of the data buses. For large block size values, however, performance drops drastically to a possible point of convergence. This again indicates bottlenecks created by contention by the eight processes for the FSB and MCH are slowing down the processors.

Figure 4.10 shows additional problem sizes to those shown in Figure 4.9 for eight processes executing the HPL benchmark. The same block sizes and problem sizes are employed as those shown in Figure 4.6. As the problem size increases above 10,000, the performance of the compute-node seems to converge to the same performance peak. As the block sizes increases, the performance of each of the

problem sizes also show a drastic drop in performance as processes continue to show large contention in the hardware.

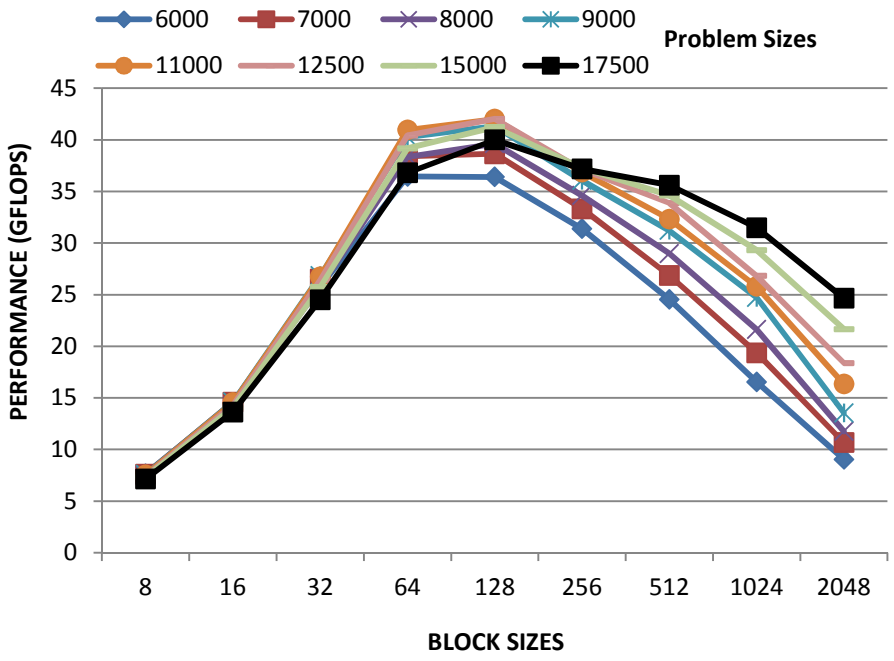


Figure 4.10: Average GFLOPS of Eight Processes Spawned with Other Problem Sizes.

The results for execution time and performance show that for smaller problem sizes (1000, 2500, and 5000), execution times were consistent irrespective of the block size for two processes of execution. Using all eight cores resulted of a speed up of approximately two-to-one in comparison to using two cores for the same computations. Again this is felt to result from contention by communication traffic in the FSB, MCH and memory [Val09].

This draws concern for the performance and timing of the architecture in the compute-node. In order to maximize performance in the MCH architecture block sizes (i.e. memory accesses) should be limited to sizes similar to the FSB and MCH data bus size. However not all programs that will run on

the compute-node will have these types of characteristics. Therefore, it is important to have a well-developed load balancing of tasks in order to maximize performance and minimize data contention.

Chapter 5: Hardware Model, Scheduler Modifications and Experimental Setup

This chapter focuses on the approaches taken in modeling the hardware in a compute-node of the Virgo 2.0 cluster, and the modifications that were performed in the Linux Scheduler in order to achieve higher throughput and lower execution times. The hardware model was obtained by a different analysis in order to obtain an overall understanding of the behavior and limitations found in the hardware during execution. Then, different parts of the Scheduler were modified, while applying the decision making that were derived from the hardware model analysis, in order for the Scheduler to make decisions that would help processes achieve more efficient performance. Finally, it explains how the experimental setup was configured in order to test the modifications made to the Scheduler by using the HPL benchmark.

5.1 APPROACH IN MODELING MULTIPROCESSOR HARDWARE

To find a model that could help explain the Harpertown behavior, we looked at previous research that had been done in modeling hardware. Fedorova et al. [Fed06] provided a useful approach that partial explained how to obtain some of the modeling functions and numerical linear approaches that were used to determine compute-node related functions.

Fedorova et al [Fed06] used an approach to improve the Scheduler by relating Instructions per Cycle (IPC) for both single and multiple processors to the L2 cache miss rate. By evaluating process memory requirements, a processing unit was assigned (or not) to match the memory requirements. This approach helped the process to execute faster since matching memory requirements obtained a high hit ratio in the L2 cache. Their model simulated hardware with eight virtual processors that had hyper-threading capabilities. Table 5.1 shows the differences in hardware between [Fed06] and a compute-node in Virgo 2.0, and the hardware specifications that were used by [Fed06].

Table 5.1: Differences in Hardware Specifications

Fedorova's Simulation Model		Virgo 2.0 Compute-Node
Number of Virtual Processors		Number of Physical Cores
DRAM Latency (DIMMs)		DRAM Latency (DIMMs)
L2_size (Varies)		L2_size (Constant)
Memory bus bandwidth		Memory bus bandwidth
		Memory Controller Delays
		Front-side Buses Delays

The general approach of their research is based on the number of Instructions per Cycle (IPC) defined in the equation below. The model consists of a function of three variables when executing threads in the virtual processors:

- The number of concurrent threads, N,
- The Miss Rate at L2 cache while executing *N* concurrent threads, *L2_MR(N)*, and
- The number of perfect Cycles per Instruction (CPI) that hit L2 cache, with no misses, from *N* of concurrent threads, *perf_cache_CPI(N)*.

$$\mathrm{IPC} = f(\mathrm{N}, \mathrm{L2\_MR(N)}, \mathrm{perf\_cache\_CPI(N)})$$

The IPC function determines the number of instruction per cycle which describes the performance of the hardware for each task executing in a single virtual processor. Since the model only describes a single thread executing in a virtual processor, it does not accurately model a multi-core compute-node executing many processes. For this situation, [Fed06] describes a multithreaded modeling approach that considers probability of virtual processors being busy or stalled and a linear regression model obtained through benchmark runs. The multithreaded model is described as:

$$\mathrm{IPC} = \sum_{\mathrm{N}=0}^{\mathrm{V}} P(\mathrm{N}) * R\_IPC(\mathrm{N})$$

, where $P(\mathrm{N})$ is the probability of a processor is in the N-state. N-states are defined from N=0 (all virtual processors are stalled), N=1 (one virtual processor is busy and V-1 are stalled)…to N=V (all virtual processor are busy). The P(N) function is defined as:

$$P(\mathrm{N}) = \binom{\mathrm{V}}{\mathrm{N}} * \mathrm{prob\_busy}^{\mathrm{N}} * \mathrm{prob\_stall}^{\mathrm{V-N}}$$

The linear regression function is a function of *N-number* of concurrent threads and the perfect-L2-cache IPC achieved by V number of threads. The linear regression function is represented as:

$$R\_IPC(\mathrm{N}) = f(\mathrm{N}, \mathrm{perf\_cache\_IPC(N)})$$

The model evaluation consists of determining the highest degree of concurrency in IPC units. We used this approach to model the Virgo 2.0 compute-node IPC execution values. Knowledge of the IPC model can help understand the compute-node better and help implement better load balancing in the Scheduler.

5.2 MODELING VIRGO 2.0'S COMPUTE-NODE

To begin modeling a compute-node from Virgo 2.0, the PerfMonitor tool was used in order to determine IPCs values for the regression linear model. The PerfMonitor tool allows the user to track up to four different parameters in a set model-specific list [Cpu09]. PerfMonitor collects data at a hardware-level as the system runs an application or benchmark. For this, the Tuning CacheEdge software was used as the benchmark program that performs a level 3 GEMM as described in Chapter 4.

For the Tuning CacheEdge settings, the program uses three parameters set by the user in order to execute the matrix multiplication: N-number of rows, M-number of columns and K-block size. The single thread scenario was launched first in order to find the optimal matrix dimensions and K-block values that gave the highest performance in MFLOPs. Table 5.2 summarizes the numbers obtained for different matrix dimensions and K-block values where outputs include MFLOPS, the output in Mega-Floating Point Operations per Second, Time, the execution time in minutes, and L1-KBs, the size of the

L1 (Level 1) cache size in Kilo-bytes (KBs) which yielded the best MFLOPS and Time results for the indicated matrix dimensions. It can be seen that matrix dimensions of 7000 give the highest throughput of MFLOPs, although execution time is one of the higher values too.

In order to obtain higher MFLOP throughput, the matrix dimension of 7000 was further explored to fit the best K-block value for the Tuning CacheEdge program. Table 5.3 shows the program values at matrix dimensions of 7000. The numbers show that the K-block value of 4096 gives the optimal highest throughput MFLOP value. Therefore, the Tuning CacheEdge program will be set to use matrix dimensions of 7000 with a 4096 K-block value when using the PerfMonitor tool. The values should give a good representation of data-intensive computation applications that requires full hardware utilization of the FSB and MCH as the number of threads increase while executing the program.

Table 5.2: Single Thread Execution Number Using Tuning CacheEdge Program

Single Thread Execution					
M	N	K	MFLOPS	Time	L1-KBs
1000	1000	1000	7547.17	0.265	512
2000	2000	2000	7532.96	2.124	256
3000	3000	3000	7529.28	7.172	384
4000	4000	4000	7571.28	16.906	512
5000	5000	5000	7554.47	33.093	768
6000	6000	6000	7579.08	56.999	1024
7000	7000	7000	7610.3	90.141	512
8000	4000	8000	7524.21	68.093	2560
9000	4500	9000	7570.33	96.297	1536
10000	5000	10000	7509.09	133.172	2048
6250	3125	12500	7513.87	64.984	2048
3750	3750	15000	7527.16	56.39	512
4375	2187	17500	7478.27	44.781	3072
5000	2500	20000	7478.42	66.859	1536
2812	2812	22500	7533.2	44.235	512

The Tuning CacheEdge was launched with from two to eight instances of the program that represented the *N-number* of concurrent threads running in the compute-node. The settings for the program were

configured to obtain the PCI, CPI, L2 Misses per Second and L2 Hit Rate. Each of the processing cores had a dedicated Tuning CacheEdge program reading and obtaining data. The data was collected as time (in minutes) from the start of executing the benchmark. After the execution of the program, the average MFLOPS of each processing core was then calculated in order to obtain a general representation of each processing core for each of the configured settings. Figure 5.1 shows the IPC linear regression model obtained for a compute-node.

Table 5.3: Matrix Dimension of 7000 Using Tuning CacheEdge Program

7000's					
M	N	K	MFLOPS	Time	L1-KBs
7000	7000	32	6922.74	0.453	0
7000	7000	64	6915.1	0.907	512
7000	7000	128	7168	1.75	128
7000	7000	256	7574.88	3.312	8192
7000	7000	512	7610.5	6.593	8192
7000	7000	1024	7600.7	13.203	512
7000	7000	2048	7600.41	26.407	0
7000	7000	4096	7613.96	52.72	512
7000	7000	8192	7609.63	105.5	512

Figure 5.1 displays the IPC values of each of the processing cores, individually executing the Tuning CacheEdge program using different numbers of concurrent threads. The regression model shows a linear model as a function of *N-number* of concurrent threads expressed as:

$$IPC(N) = 0.0705N + 0.6529$$

Because the IPC values are peaked around N = five threads, however, the linear model does not describe the best possible IPC regression function since it does not accurately represent the highest IPC throughput at N = five. The high IPC values indicate a high utilization of processing cores and therefore

the linear model in Figure 5.1 returns high error values for all processing cores at five concurrent threads.

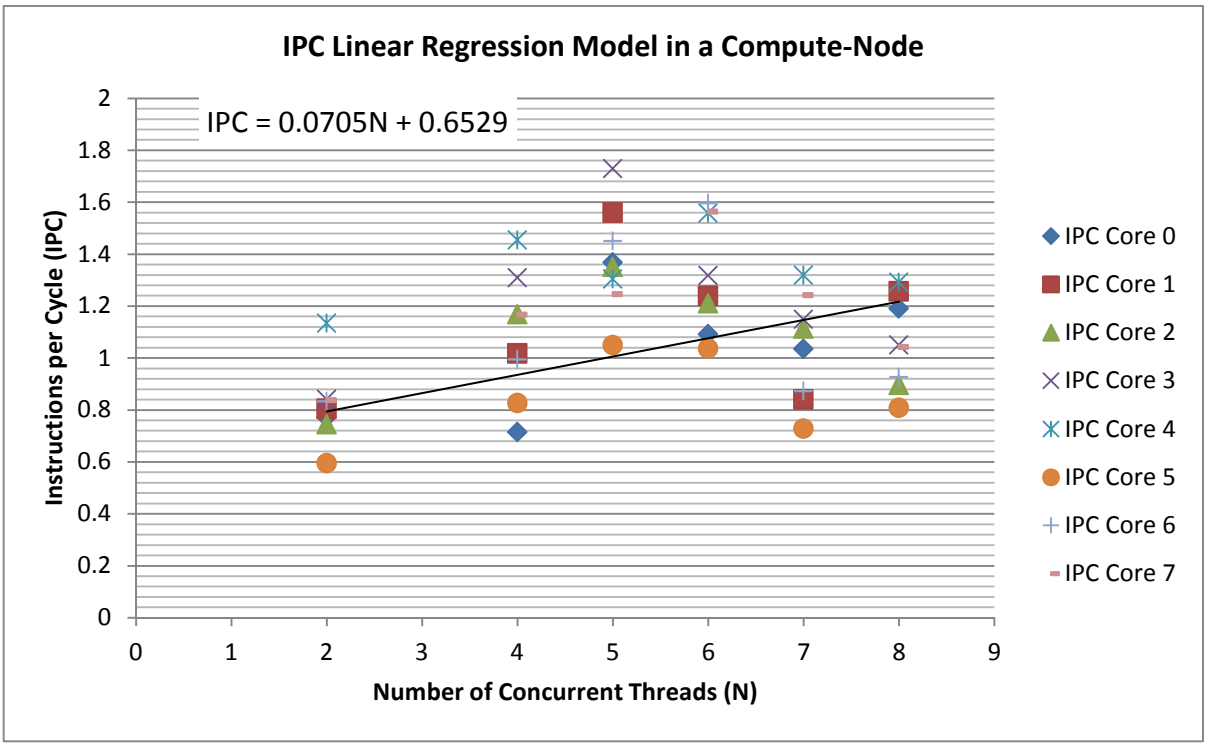


Figure 5.1: IPC Linear Regression Model of a Compute-Node

The solution was to more accurately map the peak at N = five by breaking the linear regression into two parts, $2 \leq N \leq 5$ and $5 \leq N \leq 8$. The linear regression model as a function of *N-number* of two to five concurrent threads can be expressed as:

$$IPC(N) = 0.2036N + 0.3414$$

And the linear regression model for *N-number* of five to eight concurrent threads becomes:

$$IPC(N) = 0.0705N + 0.6529$$

This model shows a more accurate representation for both a lower number and a higher number of concurrent threads executing in the compute-node. Therefore, we can conclude that the IPC linear regression model for a compute-node is represented as:

$$R\_IPC(N) = \begin{cases} 0.0705N + 0.6529 & 8 \geq N \geq 5 \\ 0.2036N + 0.3414, & 5 \geq \mathrm{N} \geq 2 \end{cases}$$

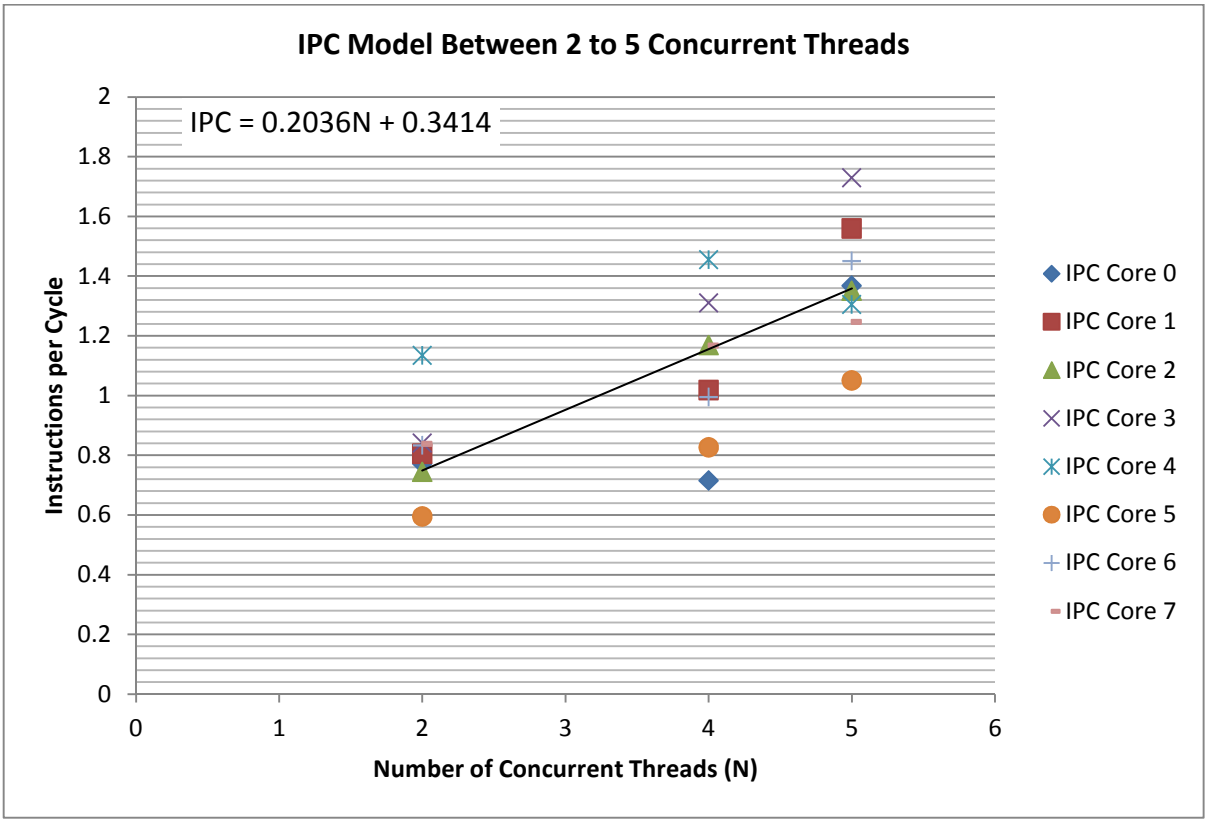


Figure 5.2: IPC Linear Regression Model Between 2 to 5 Concurrent Threads

Figure 5.2 shows the revised linear regression model for two to five concurrent threads. It represents the IPC values much more accurately than the single linear regression model shown in Figure 5.1.

For the L2 hit numbers, it is best to obtain the worst case scenario in which the Tuning CacheEdge program begins to break down. The scenario chosen to obtain the L2 figures is the eight

threaded execution where all processing cores are executing. Figure 5.3 shows the L2 hit rate of the data obtained from the PerfMonitor when executing the Tuning CacheEdge program.

The plot in Figure 5.3 displays the L2 hit rate when running eight concurrent threads in the compute-node. Throughout the execution of the program, the hit rate varies from a high of 96% to a low 87%. We can use the average of 92% hit rate that can represent an overall indication of the compute-node when using all processing cores. However in analyzing the IPC performance in Figure 5.2, the overall hit rate when running five concurrent threads should yield a higher hit rate than those in Figure 5.3.

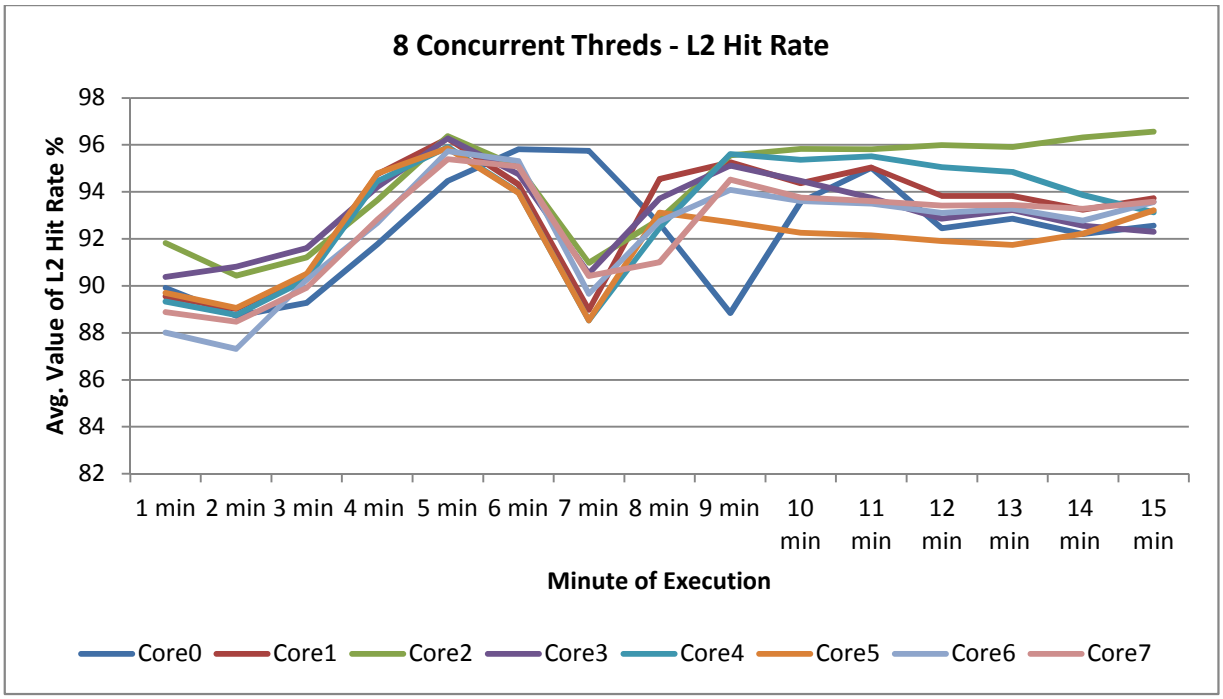


Figure 5.3: L2 Hit Rate from PerfMonitor Running Eight Concurrent Threads

Figure 5.4 shows the L2 hit rate of five concurrent threads in the compute-node. The figure indicates that with the higher IPC throughput from Figure 5.2, the five concurrent threads scenario also provide higher L2 hits. From the figure, the high peaks are just below 97% and low hit rate just above 91% in where it can be assume an overall average of 94% hit rate. This concurs with the IPC

throughput since more instructions are residing in the cache levels of the executing processing cores and there is less saturation of traffic in the FSB and MCH since not all processing cores are executing.

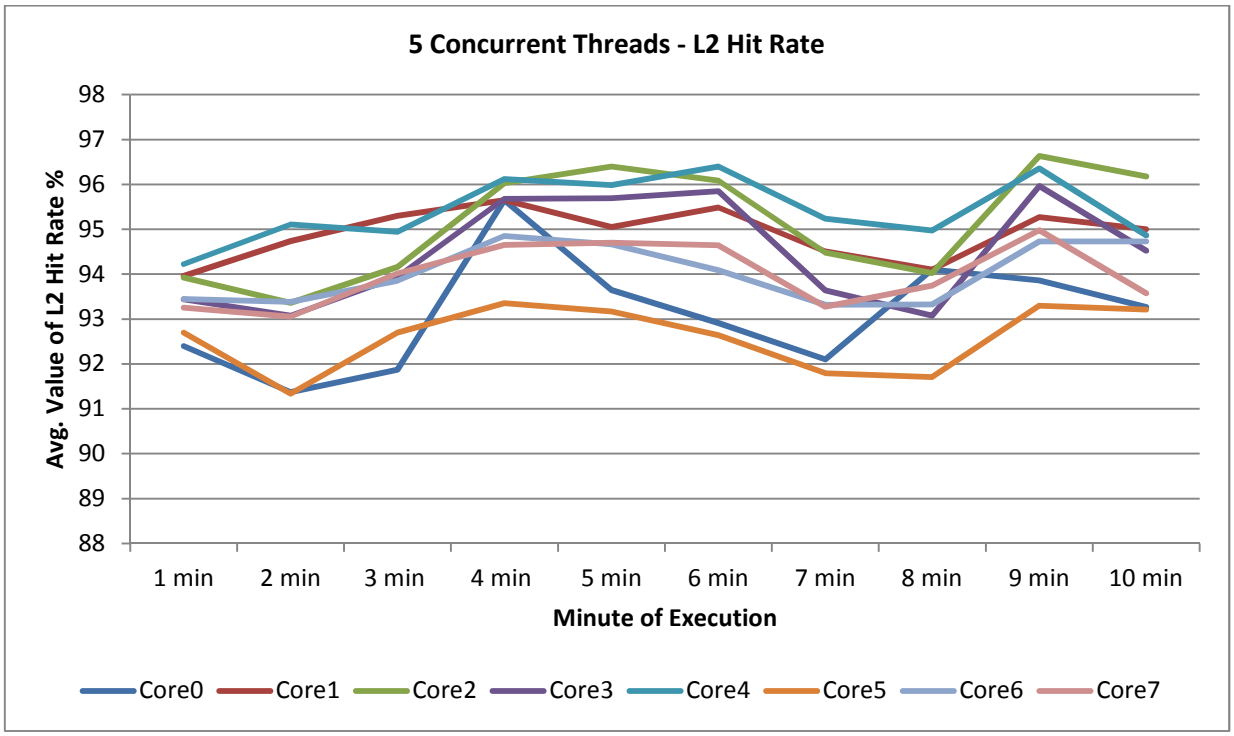


Figure 5.4: L2 Hit Rate from PerfMonitor Running Five Concurrent Threads

Figure 5.5 shows the L2 hit rate of six concurrent threads in the compute-node. From the figure, the high and low peaks of the plots have similar values from Figure 5.4 with slightly higher overall average with this number of threads.

Modeling of the hardware of the compute-node indicates that it is not necessary more efficient to execute tasks using all of the processing cores. With less processing cores executing, higher IPC throughput and L2 hits rates per processing core can occur because of less contention in the FSBs, MCHs and the L2 cache. This indicates how modification of the Scheduler should take place to increase the performance and execution time of tasks in the compute-node.

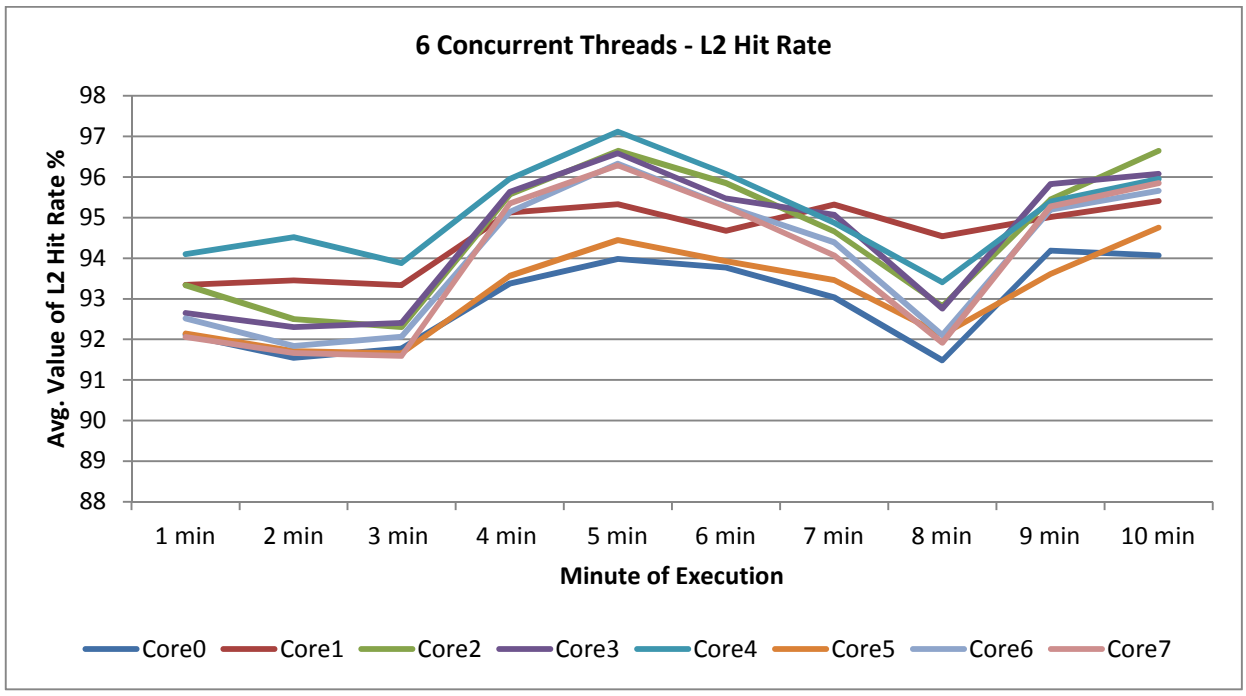


Figure 5.5: L2 Hit Rate from PerfMonitor Running Six Concurrent Threads

5.3 MODIFICATIONS IN THE LINUX SCHEDULER

This section explains the modifications that were made in the Linux Scheduler to make it more hardware conscious and more custom made for cluster performance. The goal of the modified Scheduler is to increase the performance of each task in order to improve the overall performance of the compute-node. The modifications are based on policies of the tasks that are ready for execution, management of tasks that can share resources, work balancing throughout the processing core, and process management after tasks finish executing.

5.3.1 Modification: Process Policy

The first modification that was performed in the Scheduler was the change of policy for all tasks executing in the compute-node. The modification consisted in having each task that is ready to execute

to change its policy to a FIFO policy. As described in Chapter 3, the FIFO policy in the Scheduler eliminates the timeslice and assigns higher priority to the task with such policy. By changing the policy, the Scheduler will not have to use any clock cycles in calculating the timeslices for each task which leads to reducing the number of context switches between tasks. The following steps are needed in order to safely change the policy in the Scheduler:

1. When a process is created and set to state of TASK_RUNNING, the Scheduler creates the `task_struct` for the process in the `sched.h` file.
2. Inside of the `task_struct`, the call to the `sched_setscheduler(*p, policy, *param)` function is placed in order to safely change the task's policy, in where:
    - `*p` – the pointer to the task in question.
    - `policy` – the new unsigned long policy value. In this case it would be SCHED_FIFO
    - `*param` – the structure containing the new real-time priority. In this case it is preferable to keep the same priority value of the task.

    A typical implementation is as follows:

```
/* FIFO Policy in task_struct */
unsigned long policy;
policy = (long) sched_setscheduler(*p, SCHED_FIFO, *param);
```

3. The call to the function `sched_setscheduler` is necessary because it is important that the task is able to safely lock the runqueue in which it will execute. The function has the proper declarations to lock the runqueue.
4. It is necessary to cast the policy definition to a type-`long` since the `sched_setscheduler` returns a type-`int` value.

By making this modification inside of the `task_struct`, it will guarantee that the processes that are created by an application will retain the policy information to be in SCHED_FIFO mode. Also in trying to make the policy modification during execution may bring a decrease of performance of tasks since it must stop execution in order to properly service the policy changes. Therefore each time a `task_struct` is created for each process and linked in the `task_list`, the policy change is made immediately.

5.3.2 Modification: Process Identification Grouping

This modification consists of managing the process grouping of processes that are executing in the compute-node. The reason is to gain control of process grouping to facilitate the management of load balancing for all processes. This modification is performed inside of the `schedule()` function. The `schedule()` function is the part of the Scheduler where tasks are scheduled in Active queue arrays.

Figure 5.6 shows how processes that have child processes are grouped together in order to group them for load balancing tasks. Part of this modification is to obtain the PID of the parent of the process that is being scheduled. Once the parent process is identified, the modification checks to determine if the parent process is the `group_leader`. This is part of an insanity check that revises the information of the `task_struct` of the process to insure that it correlates to the process in question at this time. After the identification of the parent process, the PID is saved in the array and it then becomes necessary to know of a child process is scheduled to execute as well.

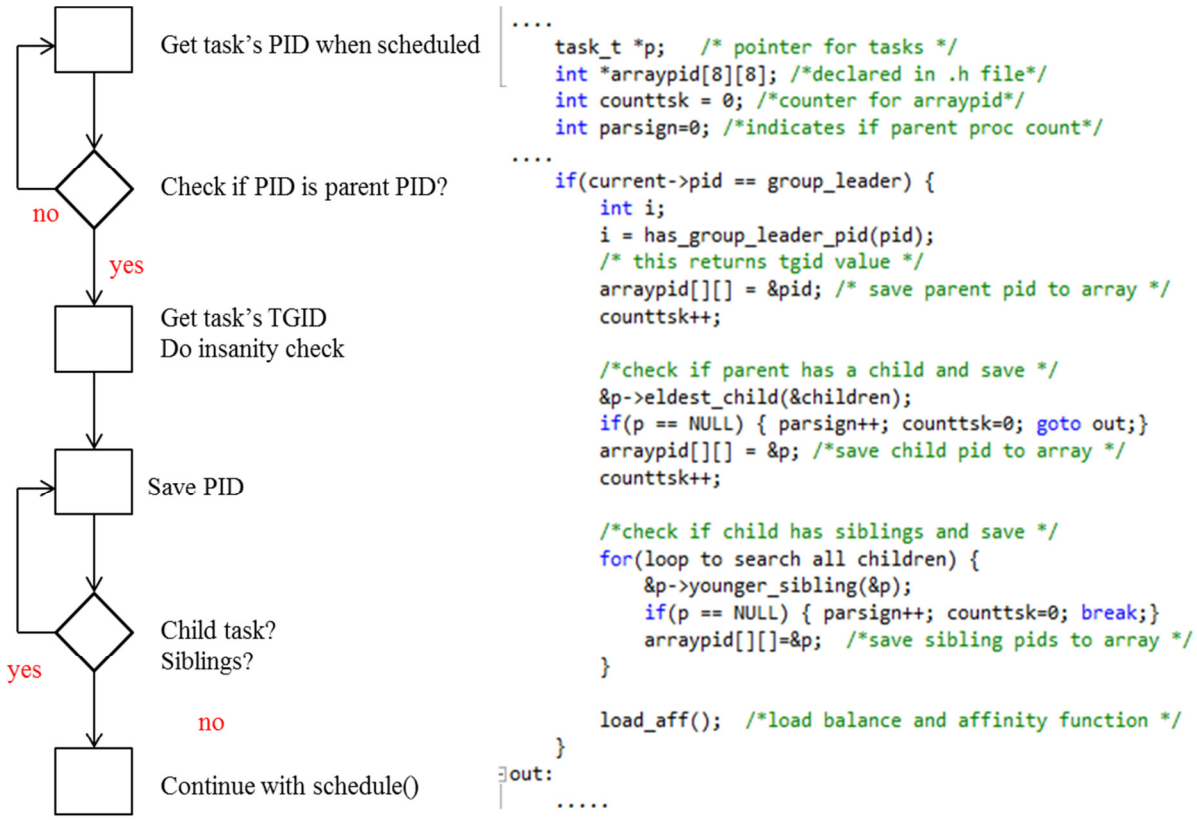


Figure 5.6: Process Grouping Modification Flowchart and Pseudo-Code

The child process can be identified by the `eldest_child()` function. The only parameter that it takes is a pointer to the child PID variable called `children`. The `children` pointer exists in the `task_struct` of the parent process. If the function returns a NULL pointer, it means that the child is, in fact, the parent process and there are no child processes created by the parent process. If the function does not return a NULL pointer, then it is necessary to save the child PID into the array that was returned from the function `eldest_child()`.

Finally, the modification determines if the child process has sibling processes that are also scheduled for execution. This goes into a loop for six more child process that the modification will add to the group. Any process created after the child process is known as a younger sibling process; therefore, the modification uses the `younger_sibling()` function to identify any other child processes created by the parent process. If the function returns a NULL pointer, it indicates that there

are no sibling processes and it will break from the loop. If it does return a pointer address, it is saved in the group array and continues to search for other sibling processes that might exist.

For each application that is launched in the compute-node, the process of grouping is reinitialized for the upcoming task set. After grouping the tasks or task, the main part of the modification is then performed to load balance and assign processing core affinities to each of the tasks that are scheduled to execute.

5.3.3 Modification: Load Balancing and Affinity Assignment

This modification of the Scheduler deals with how the processes will be balanced and affinity assignments be made to match tasks with the processing cores. Once the previous modification finishes, assigning affinity of processing cores to each of the tasks immediately starts. The manner in which each task is assigned to a processing core is implemented using the `load_aff()` function.

The `load_aff()` function is a modification that load balances the tasks that are ready to execute and assigns the task's affinity to a processing core. The `load_aff()` function is as follows:

- It first determines if the previous modification was able to group single or multiple tasks for execution, if not, the function returns control as shown in the code below.

```
/* no process is ready to run */
if(arraypid[0][0]==NULL) {
      return;  }
```

- Next, the `load_aff()` function performs an availability test of the processing cores through the `idle_cpu(int cpu)` function. The processing cores that are idle and ready for execution are saved in a vector array in order to use the list during the load balancing part of the function.

```
/*check which processing cores are idle */
for(go through the processing cores) {
```

```
        ready[] = idle_cpu(order[]);
    }
```

- The `ready[]` array keeps the information of the available processing cores that are available for execution.
- The `order[]` array is a defined array which holds the sequence of processing core identification numbers. The processing core identification number can be seen in Figure 2.1.
- The `order[]` array is ordered by pairs of processing cores that share the L2 cache memory. (i.e. Core0 and Core2 share L2 space - therefore, their identification number are placed adjacent to each other in the array).

After initial checks, the `load_aff()` function begins load balancing the tasks that were grouped in the previous modification and begins assigning affinity to processing cores. The simplest form of load balancing is when the number of tasks equals or is greater than the number of processing cores available for execution. In this case the `load_aff()` function will:

- Check that the `order[]` array and the `ready[]` array have the same number of elements and value of each element in order.
- For each element, the function begins to assign affinity to each task by their PID and the order of the processing cores.
- The pseudo-code below demonstrates in how tasks are assigned in this scenario.

```
if(check value of readyarray ==  orderarray) {
   for(arraypid[][]) {
      if(tskcore=sched_setaffinity(p->arraypid[][], cpumask_of_cpu(ready[])) == 0)
          if(ready[indx++]!= NULL) {indx++;}
           else return;
      else return -1;          }
```

}

- `tskcore` is an int-type variable which checks the return value of the `sched_setaffinity()` function.
- The `sched_setaffinity()` function is a defined function in the Scheduler which is used to set any task to a CPU. Each time the `sched_setaffinity()` function is called, it must also have the two parameters: the PID of the task and the mask-type value of the CPU.
- Each PID of tasks in the group is obtained from the `arraypid[][]` array used in the grouping of tasks modification.
- For the second parameter, each processing core ID that is in the `ready[]` array must be changed to `cpumask_t` type.
- The function `cpumask_of_cpu(int cpu)` is used to do this conversion correctly. The `cpumask_of_cpu(int cpu)` is a defined function in the Scheduler. The int-cpu value is obtained from the `ready[]` array.
- If the `sched_setaffinity()` function returns a zero, the task affinity will be set to the indicated processing core.
- Also, it needs to keep track of the indexing of the `ready[]` array. It will check if it is at the end of the array.
- If errors occur, the function returns a negative one value and exits the function.

This keeps repeating, setting affinity to each task, until all are assigned. However, many other scenarios of processing core availability can be presented when trying to execute a program with eight tasks. The load balancing would need to decide where the tasks in the group need to execute and assign affinity to all. Therefore, the load balancing must take into account the number of processes each

instance of the HPL benchmark is asked to execute and decide the best affinity assignment accordantly when the number of tasks is greater than the available processing cores.

As mentioned in the previous section, the compute-node has a better IPC performance and a higher L2 hit average rate, with between five and six concurrent threads executing. Once the number of concurrent threads continues to increase, the IPC performance and the L2 hit rate begin to decrease. Therefore, the `load_aff()` function must take consideration of performance decreases that can occur by assigning affinity to all the processing cores.

The way the load balancing is modified is by setting affinity in pairs of two processing core that share L2 cache memory. Figure 5.7 shows in how the pair grouping of processing cores is configured. The main reason to pair the processing cores in this manner is to take advantage of the L2 cache share capabilities. Secondly if any process communication is needed, the tasks will have minimal latency. Finally, as the processes share L2 cache memory to execute the same instance of the benchmark, it reduces the number of L2 miss rates which in effect reduces the number of main memory accesses and creates less contention in the FSB and MCH.

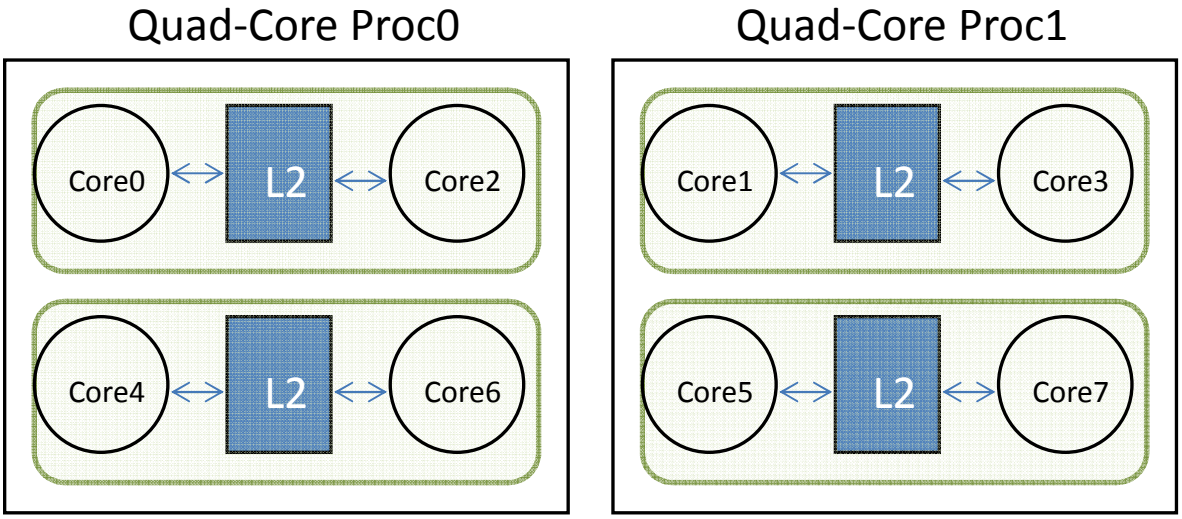


Figure 5.7: Processing Core Pair Grouping

The load balancing decision making is shown in Figure 5.8. The decision making begins with finding the number of available processing cores and servicing the number of tasks that are ready to execute. If the number of processing cores is greater than or equal to the number of tasks, the function will set affinity for all tasks and then continue scheduling. However if the number of processing cores are less than the number of tasks that are ready to execute, then the function must check the number of available processing cores and begin deciding the course of action that can be taken as follows:

- Once the number of available of processing cores is less than the number of tasks, the function checks if the number of available processing cores is less than or equal to two.
- If the number of processing cores available is less than or equal to two, the decision is to not set affinity to the available processing cores in order to:
    - Avoid using all of the processing cores which causes a decrease of performance IPC.
    - Lowers the overall L2 hit rate for all processing cores.
    - Avoid increase of traffic through the FSBuses and MCH.
- Instead, the function will set affinity to processing cores that are currently busy in order to schedule them to be executed next.
- This keeps the available processing core free and next group of tasks out of the way of the currently execution of tasks.

When checking that the number of available of processing core exceeds two, then the number can be either an odd or even number of available processing cores. The decision of the load balancing will need to consider two points:

1. Availability of processing cores in pairs and
2. If no pair exits, then continue to wait for processing cores to become available.

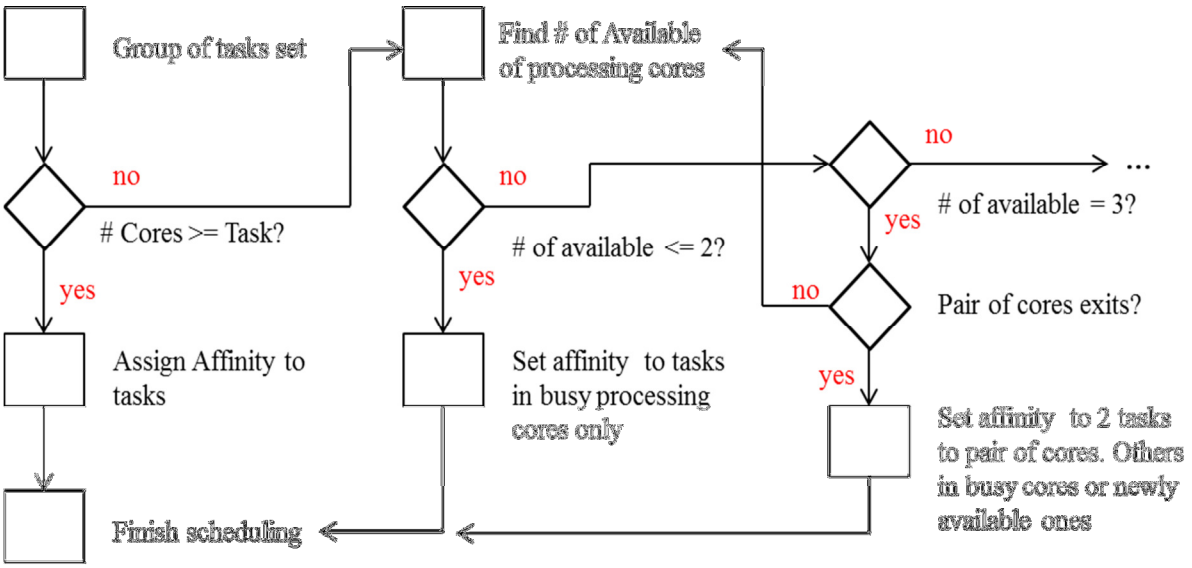


Figure 5.8: Load Balancing Flowchart

5.3.4 Modification: Process Termination and Feedback

This part of the modification supervises the process termination and feedbacks information about the availability of processing cores in order to allow the next set of tasks to execute. This modification checks on two different aspects in order to correctly assure that the processing cores are available for execution: it checks if the processing core is idle, and if the parent process is dead. It is important to check more than the state of the processing core since an idle state of a CPU can occur at different stages of execution. Therefore, this modification checks that the parent process of a group of tasks is done that indicates all are done executing in their assigned processing cores. Figure 5.9 shows the flow of this modification.

As seen in Figure 5.9, the modification starts by checking the processing cores to see if they are idle by calling the function `idle_cpu(int cpu)`. At this stage, it must check all processing cores until it detects an idle one. Once a processing core become idle, it then checks the PIDs array `pid[PIDTYPE_MAX]` from the `list_head` from the `task_list` in order to find the finished PID.

It then checks for the `group_leader` PID that has the same PID as the parent process. This will assure that the task is finished in the allocated processing core and that the idle processing core is available for execution. If the parent process was not the task that finished executing in the processing core, the code then checks for other idle processing cores

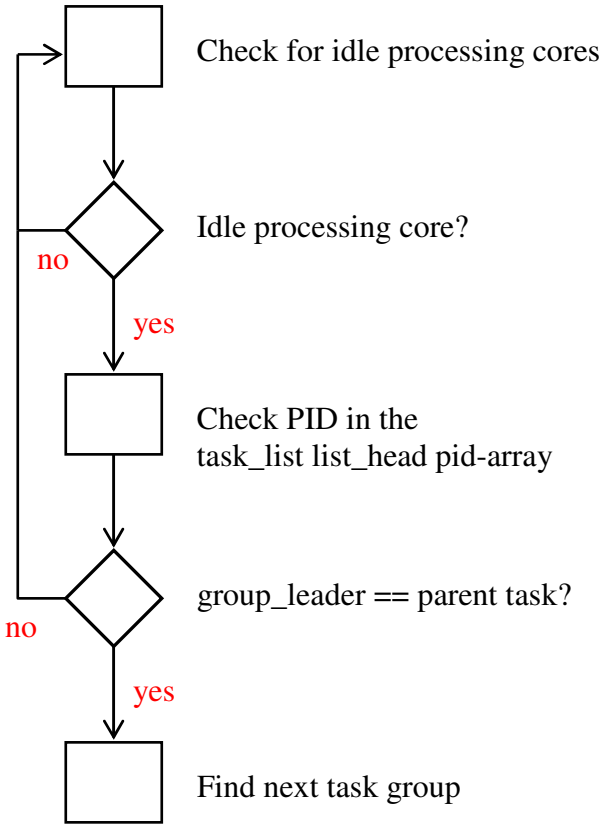


Figure 5.9: Process and Processing Core Availability

to execute any other task group unless the task group is just a single task. This will avoid over saturating all of the processing cores by alleviating traffic to the FSB and MCH. However once the parent or `group_leader` has finished execution, the last step is to go back to assign affinity to the next group of tasks that are ready to execute.

These modifications were implemented and recompiled in the Linux kernel. Once all of the errors were fixed and checked out, the process of recompiling the ATLAS libraries, MPICH libraries and the HPL benchmark followed.

5.4 EXPERIMENTAL SETUP

The experimental setup was configured to test the modifications that were done to the Linux Scheduler. The experimental setup that was used for the hardware analysis was designed to test the FSB and MCH limits, and it was not used for the Scheduler modifications. In order to test the modifications, the HPL benchmark was used with similar configurations as before but with more than one instance of the benchmark running within the same compute-node.

The configurations of the HPL benchmark were similar to the previous work, as described in Chapter 4, but each run contained a heavier load. Eight different problem sizes Ns = {6000, 7000, 8000, 9000, 11000, 12500, 15000, 17500} were used for each instance of the benchmark. From the previous work, it could be seen that the benchmark provided better data with problem sizes higher than 5000. The matrix dimensions were varied for different Ps and Qs values and also depended on the scenario that needed to be tested (i.e. all eight cores, seven cores, down to just two cores). Block sizes NBs = {64, 128, 256, 512, 1024, 2048} were used for each of the problem sizes.

Different instances of the HPL benchmark are necessary since it introduces an environment that tests the modifications made to the Scheduler. Each instance of the benchmark can spawn other processes that will test the grouping of the processes with the parent process. All processes have a FIFO policy defined in each `task_struct` in order to avoid time slices and context switches. With the grouping management set, load balancing and processing core affinity were then tested. After each instance of the benchmark finished execution, process termination and the dead state of the parent

process was tested to determine if revising the state of the processing core was possible in order to assign affinity to the next group of tasks that are ready to execute.

Chapter 6: Results and Conclusions

This chapter presents the results and conclusions from using HPL for the experiments conducted to evaluate our Linux scheduler modifications in terms of execution time and GFLOPS improvement. We look at the analysis from two extremes from low to high number of concurrency of processes. Evaluations are made for single process improvements and for overall average improvements using different parameters of the HPL benchmark (i.e. matrix dimensions, block sizes, and problem sizes). The experimental runs aimed at testing the modifications to the Scheduler to improve the performance and execution time. Observations of the results center on the issues of the behavior of the modifications made in the Scheduler, as well as, the effects of the hardware's FSB and MCH for different scenarios. The chapter concludes with future work.

6.1 LOW CONCURRENCY RESULTS

The experimental setup for running at low concurrency involved running different instances of the benchmark with different number of processes, as long as the total number of processes were at most four, executing on at most four processing cores. (The number of processes equaled the number of cores.) Medium concurrency is defined to be the case where five or six concurrent processes are executing on either five or six processing cores. High concurrency is the case where seven or eight concurrent processes are executing in either seven or eight processing cores.

The first low concurrency case is of an individual benchmark instance running three processes in an execution environment that allows a maximum of four processes. The purpose of this analysis is to measure improvements in performance and execution time in an environment of multiple instances of the benchmark at low concurrencies. Figure 6.1 shows the execution time and performance throughput

of three processes with benchmark properties of: Problem size – 175000, PxQ – 1x3, and block size – 2048.

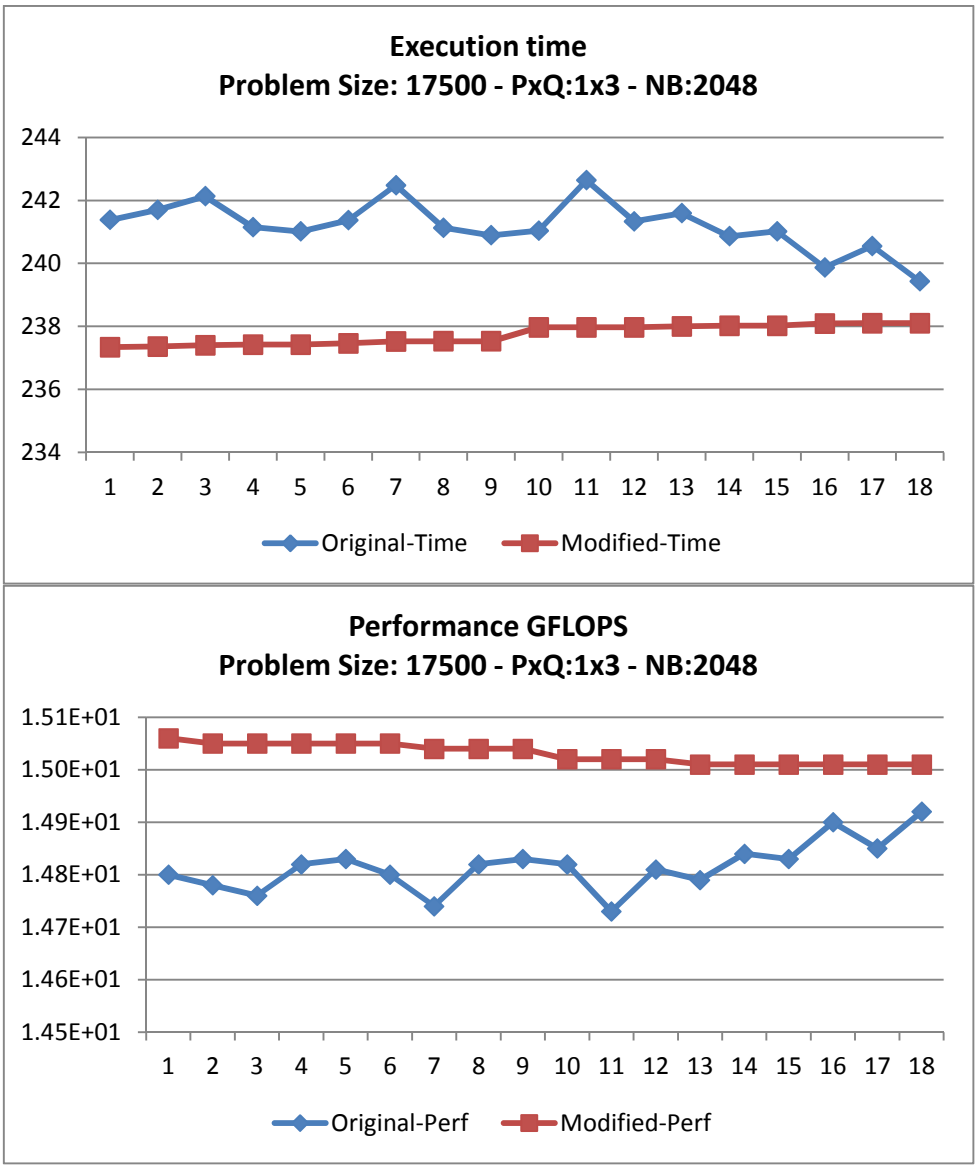


Figure 6.1: Execution Time and GFLOPS of a Three Process Instance of HPL

Figure 6.1 shows the improvements of the three process instance and display the execution time and performance improvements of the modified Scheduler versus the same three process instance when executed in the original Scheduler. The data from Figure 6.1 was obtained when another instance of the

benchmark was concurrently executing as a single process. Thus a total of three plus one processes were executing, total in the node. This scenario tested the ability of the modified Scheduler to group the tasks and assign affinity to each process. The improvements in this case were obtained from the affinity decisions made in accordance with the L2 cache processing pairings and distributing the processes evenly between the two processors. This helped to alleviate traffic flowing through the FSB and MCH a little bit and the improvement of the overall execution time is 1.4349% and performance throughput is 1.4562%.

The low concurrency case was expanded to include up to and including four processes. This qualifies as a low concurrency case because it provides enough concurrent processes to create small contentions in the FSB and MCH and test the Scheduling modifications. This number of concurrent processes also illustrates higher number of HPL benchmark instances that were launched to test low concurrency scenarios. Figure 6.2 summarize the percentage improvements of execution time for this scenario for problem sizes 6000 (smallest problem size) and 175000 (largest problem size). The improvements are listed by different matrix dimensions and different block sizes during execution as described in section 5.4. The remainders of the improvement plots for other problem sizes of this concurrency are shown in Appendix A.

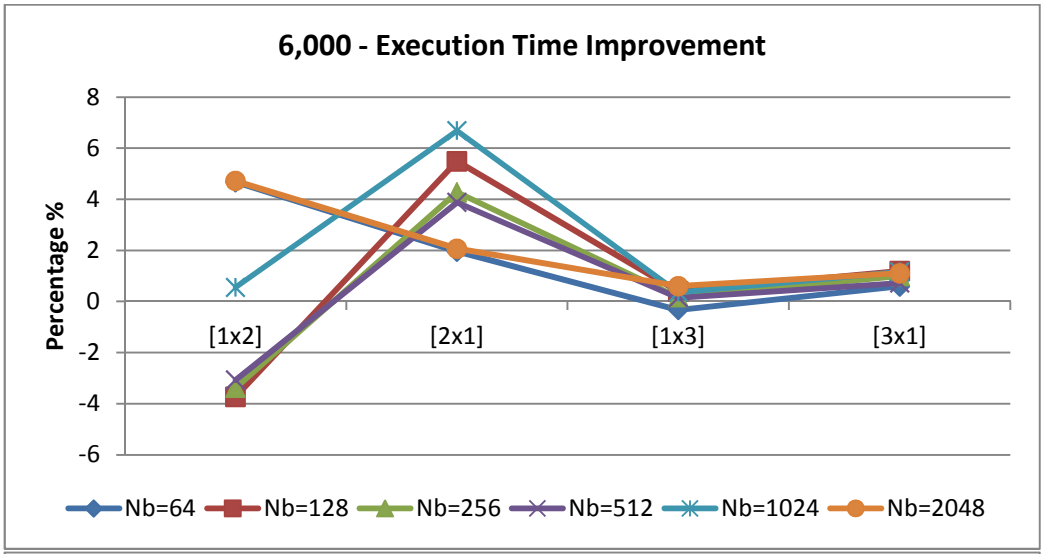
Figure 6.3 shows the improvement percentage of performance throughput in GFLOPS of the same problem sizes shown in Figure 6.2. The remainder of the of performance improvement plots for other problem sizes of this concurrency are listed in Appendix A.

The percentage improvements from Figures 6.2 and 6.3 indicate it is hard to improve the performance and execution time when the processing core executes small loads. This is due to several factors:

1- When the computational load does not employ all of the available processing cores, the performance of the overall compute-node does not reach its maximum performance. This is true even with customized load balancing and affinity task allocation.

2- The small improvements that were attained are due to the load balancing modification, which provided task affinity, which allowed the processing cores to share L2-more efficiently. This is achieved by the decision making of having tasks allocated with processing cores by:

- Spreading out the tasks between the two processors, which alleviates traffic in both FSBs; and
- Each task has an increase amount of L2-cache since not all cores are in use.



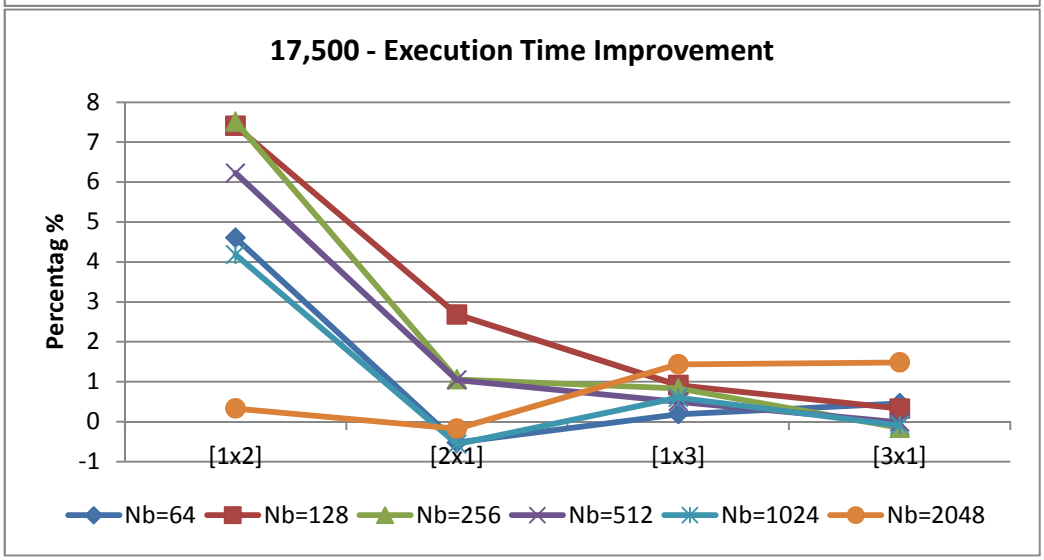


Figure 6.2: Execution Time Percentage Improvement for Problem Sizes 6,000 and 17,500 in Four Process Environment

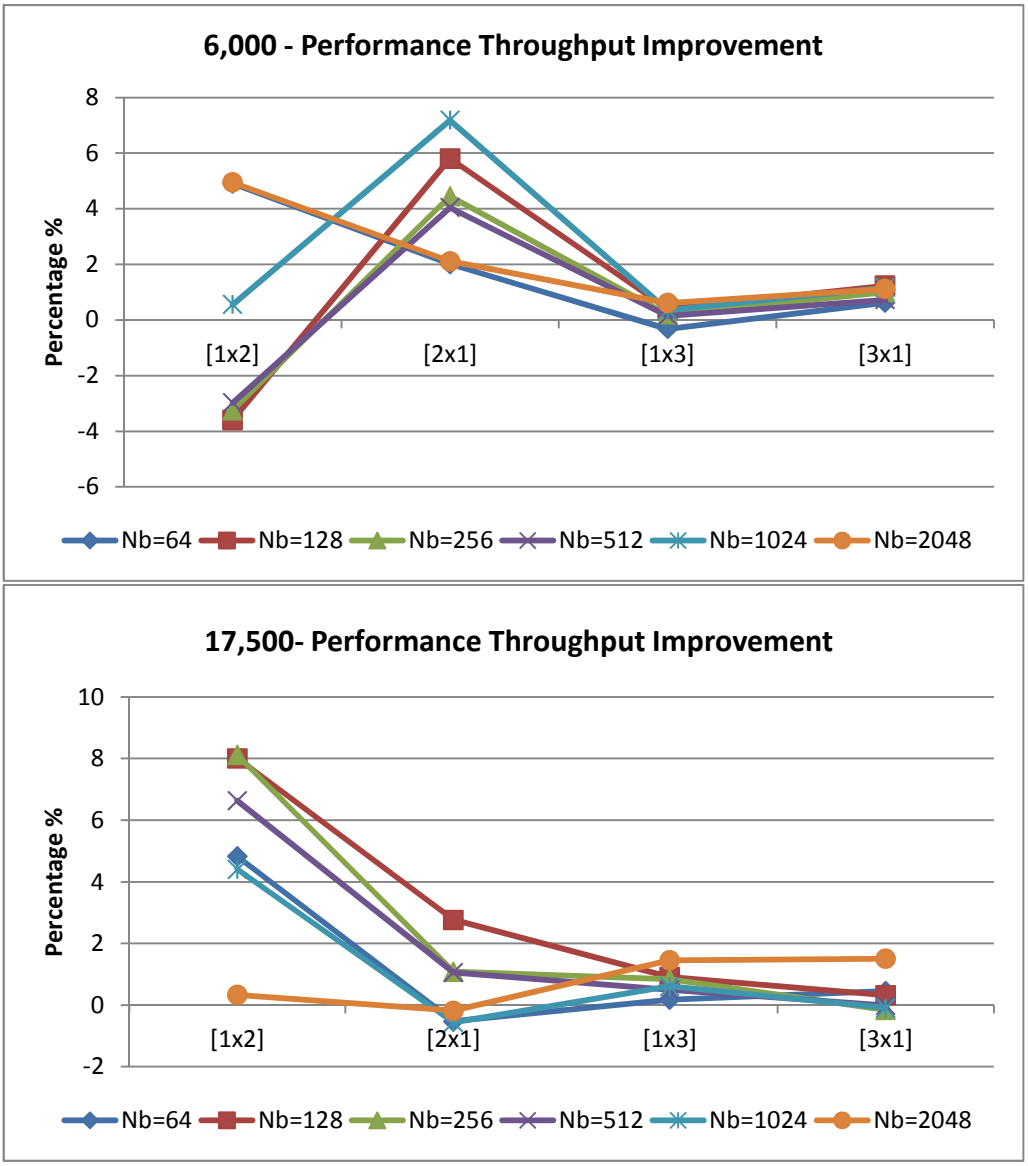


Figure 6.3: Performance Throughput Percentage Improvement for Problem Sizes 6,000 and 17,500 in Four Process Environment

Overall, the results for low concurrency show that:

- The improvements in both GFLOPS and execution times were less than 10%.
- Regardless of the block sizes, the small improvements are due to the affinity assignment which help increase the L2 cache hit rates.
- However, the affinity assignments can sometimes create contention to the FSB when load balancing other instances of the benchmark which can be seen at the negative percentage numbers at different problem sizes in Appendix A and in Figure 6.2 and 6.3.
- With larger numbers of processes (dimensions) performance cannot be improved since FSB and MCH contention increases; in fact, the GFLOPS decreases and execution times increases.

6.2 MEDIUM AND HIGH CONCURRENCY RESULTS

The experimental setup for running at high concurrency consisted of executing different instances of the benchmark with a different number of processes as long as the total number of processes equals the number of processing cores in the compute-node. As mentioned in the previous section, medium concurrency results is defined as five or six concurrent processes that are executing on either five or six processing cores. In the following cases more instances of the benchmark will be shown from the low concurrency case since the number of processes increases. The first case is of individual processes running in a high concurrency execution environment. The purpose of this analysis is to demonstrate that each process was able to improve in performance and execution time. Figure 6.4 shows the execution time and performance throughput of a seven process benchmark instance with properties of: Problem size – 6000, PxQ – 1x7, and block size – 128. The seven process instance is one of the two instances that was executing at this experimentation run. This instance of the benchmark was running with matrix dimensions defined as [1x7] and problem sizes from 6000 to 17500 and block sizes from 64 to 2048. Figure 6.4 shows a specific execution of the matrix with properties of problem size of

6000 and block size of 128 and it is compared with the same specific benchmark run with the original Scheduler in the compute-node.

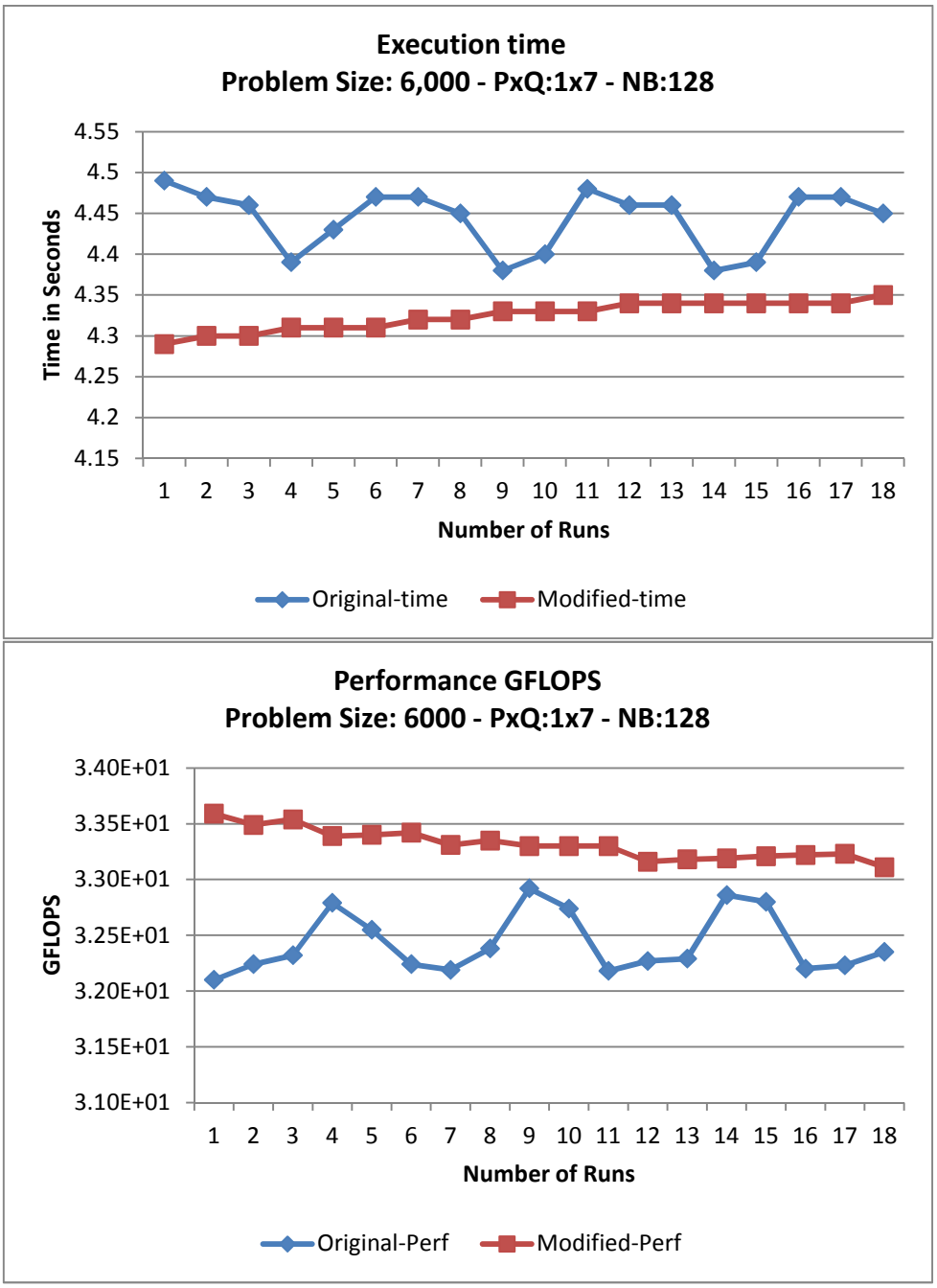


Figure 6.4: Execution Time and GFLOPS of a 7 Process Instance of HPL

Figure 6.4 shows the improvements of the seven process instance of the benchmark made during an eight concurrency process execution. Both plots display the execution time and performance throughput improvements of the modified Scheduler versus the same eight process instance when executed in the original Scheduler. The data from Figure 6.4 was obtained when another instance of the benchmark was concurrently executing as a single process. This scenario tested the ability of the modified Scheduler to group the tasks for each of the instances and assign affinity to each process. The improvements in this case were from the affinity decisions made in accordance to the L2 cache processing pairings. This helped to smooth traffic flowing through the FSB and MCH and here the improvement of the overall execution time is 2.6561% and performance throughput is 2.7557%.

With the same scenario of eight concurrency processes executing, Figure 6.5 shows the execution time and performance throughput of process a five process benchmark instance with properties of: Problem size – 6000, PxQ – 1x5, and block size – 128. In this case, the five process instance is running with other instances of the benchmark in which the total number of processes is eight. Each process that is generated by each instance of the HPL benchmark creates then number of processes depending on the matrix dimensions set for each instance. Each process that is created does not create threads of execution; however, the process that were spawned from each instance of the benchmark can be considered as a thread since each process is delegated part of the execution that needs to be carried to a processing core. Figure 6.5 shows the improvements of the five process instance of the benchmark made during an eight concurrency process execution. Both plots display the execution time and performance throughput improvements of the modified Scheduler versus the same five process instance when executed in the original Scheduler.

In the case of Figure 6.5, the data was obtained when other instances of the benchmark were executing in the remaining processing cores. The reason for showing the results of the five process instance of the benchmark was to analyze the numbers it produced since hardware modeling (see

Chapter Five) showed that the IPC throughput is at its highest with five tasks. The execution time and performance throughput in shown in Figure 6.4 are higher than those in Figure 6.5, and the overall execution time improvements with the modifications in the Scheduler were 10.204% and performance throughput is 11.454%. This demonstrates that each process continues to run at high IPC throughput, the tasks execute more efficiently, and the load balancing modification improved L2 cache awareness that also yields higher IPC throughput.

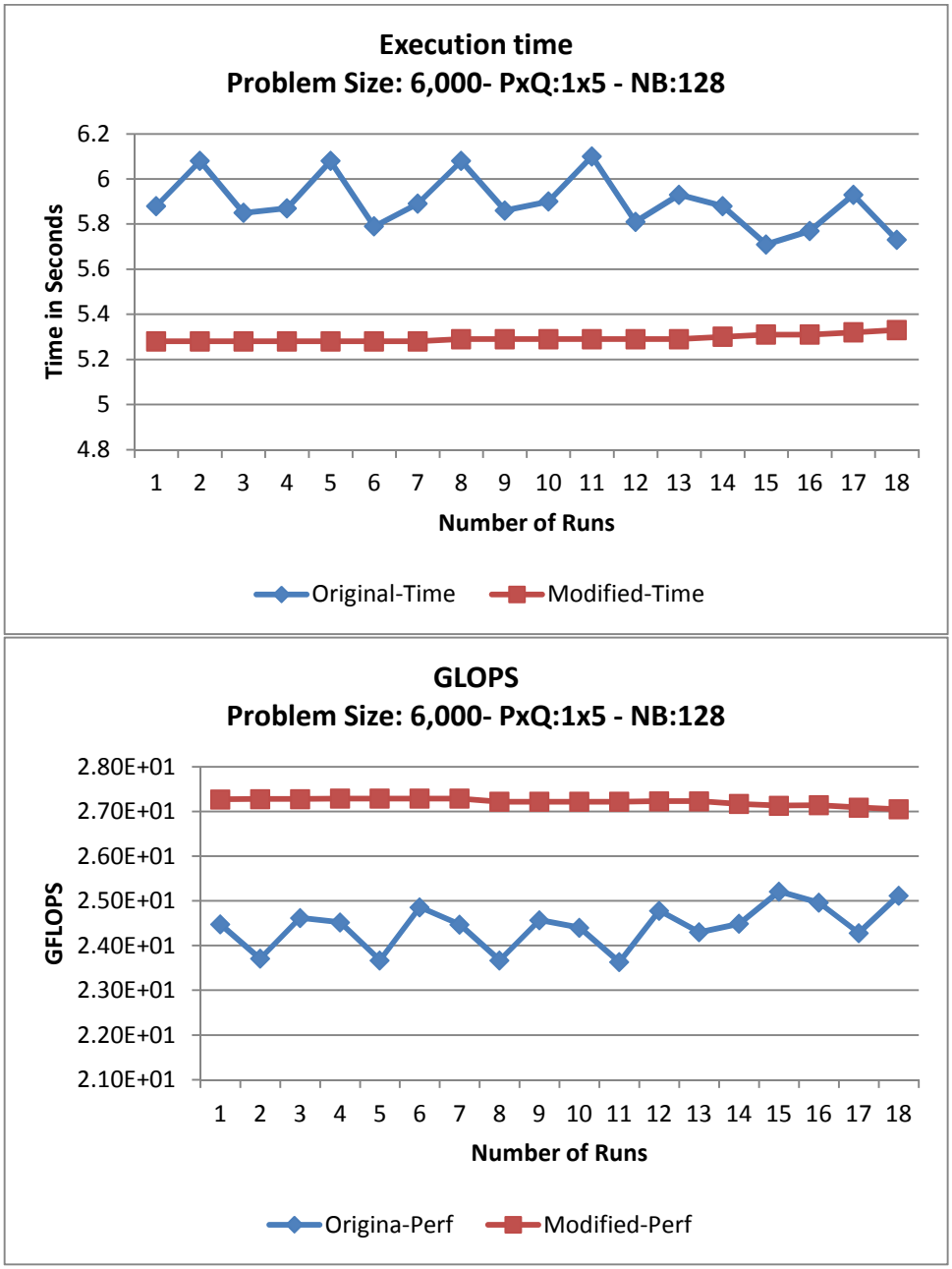


Figure 6.5: Execution Time and GFLOPS of a 5 Process Instance of HPL

The following case is of high concurrency with eight concurrent processes occupying all the processing cores. The configuration consists of having two or more instances of the HPL benchmark running in which the total of processes equal to the number of all the processing cores. The number of instances of the benchmark varies from eight single processes to an instance with at most seven

processes with a single process executing. More than one instance of the benchmark is needed to make the Scheduler group and load balance all processes. In order to summarize the percentage improvements of the eight concurrency execution scenario, Figure 6.6 shows the improvement in execution time of problem sizes 6000 (smallest problem size) and 175000 (largest problem size). The improvements are showed by different Nb block sizes, each using different PxQ matrix dimensions during execution. The remainder of the of improvement execution time plots of other problem sizes of this concurrency can be seen in Appendix A.

Figure 6.7 shows the improvement percentage of performance throughput, in GFLOPs, for the same problem sizes shown in Figure 6.6. The improvements are showed by different matrix dimensions each through different block sizes during execution. The rest of the of improvement performance throughput plots of other problem sizes of this concurrency are shown in Appendix A.

The percentage improvements from Figures 6.6 and 6.7 indicate that for low PxQ matrix dimensions, there is greater improvement in performance and execution time. However, it is much harder to improve the performance and execution time when executing with all of the processing cores. This is due to several factors:

1- The other instances of the benchmark concurrently executing increases the traffic in the FSB and MCH.

2- As the problem size increases, the limitations of the hardware (bus sizes, cache sizes) begin to decrease processing core utilization.

3- As the bottlenecks and limitations become more apparent during execution, the modifications made to the Scheduler converge to very low to no improvement regardless of the affinity set to all tasks.

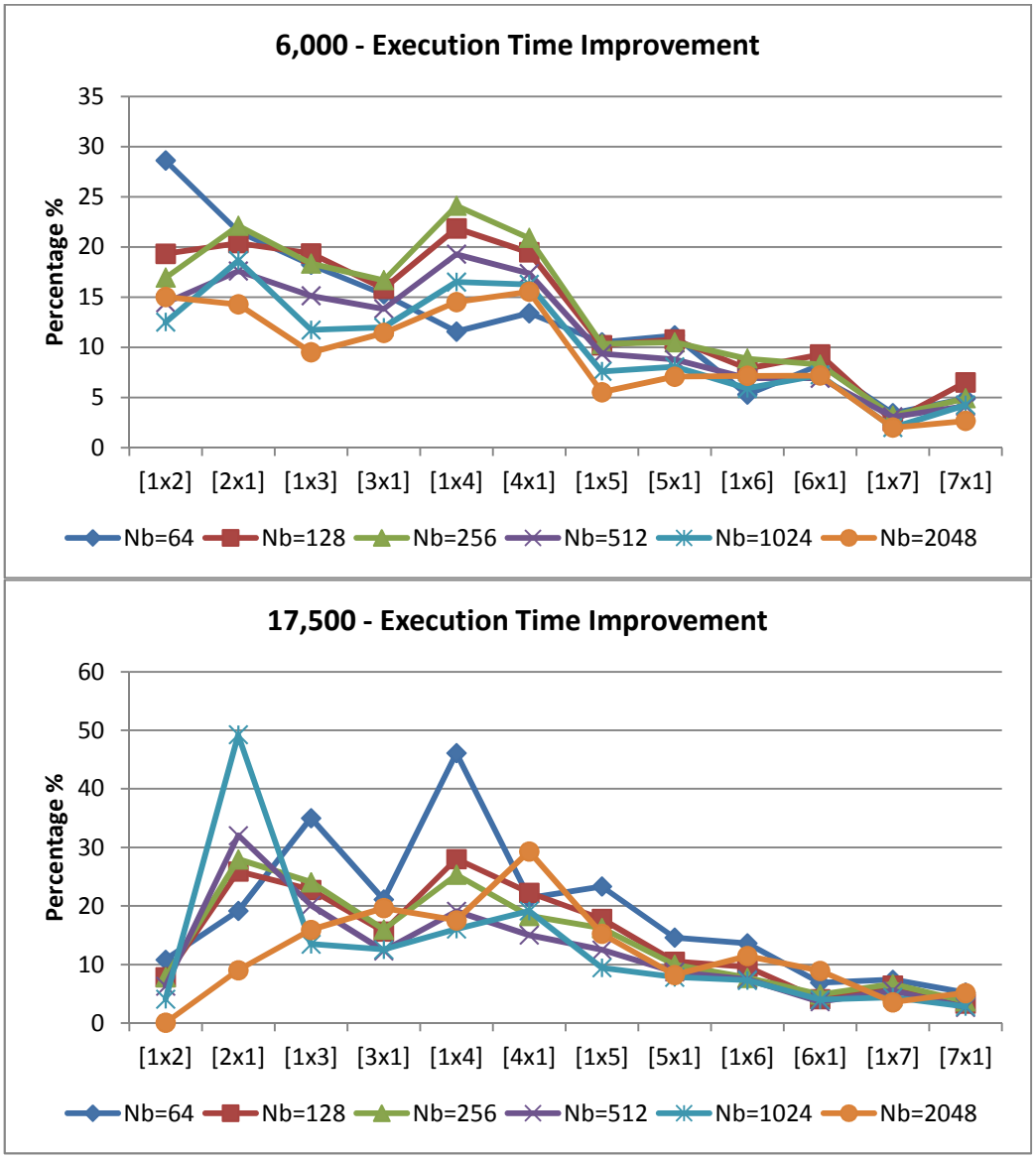


Figure 6.6: Execution Time Percentage Improvement for Problem Sizes 6,000 and 17,500 in Eight Process Environment

Overall the results for medium to high concurrency show that the larger improvements were obtained by these common factors:

- Small block sizes of 64-128 show the highest improvements than other block sizes due to the width of the data line that exists in the hardware.

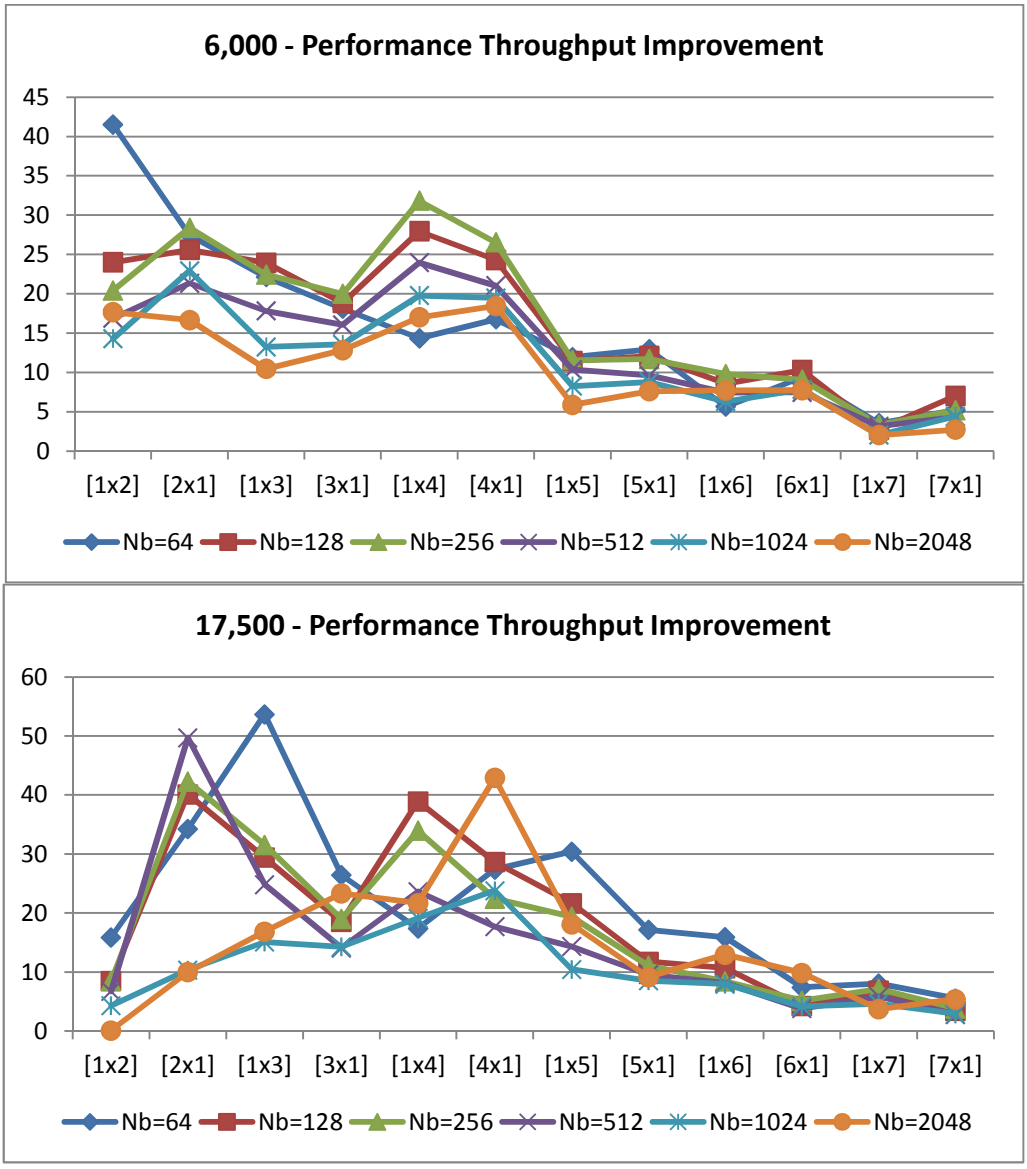


Figure 6.7: Performance Throughput Percentage Improvement for Problem Sizes 6,000 and 17,500 in Eight Process Environment

- The larger the block sizes, it is harder to improve GFLOPS and execution time due to the time to transmit information; although improvements were still achieved with smaller percentages from the smaller block sizes.

- Small numbers of processes (dimensions) are easier to allocate affinity to processing core pairs. In this case the load balance modification helps increase the processing core utilization which yields to higher improvement of GFLOPS.
- Larger number of processes (dimensions) difficult to improve as contention on the FSB and MCH increases; therefore, the GFLOPS decrease and execution time increases.
- All of the benchmark runs and instances were executed in row-major format. This means that the elements of the matrix are placed in a contiguous form in memory. Therefore, single row dimensions of the benchmark show higher peaks in GFLOPS and execution time.
- From the Figures 6.6, 6.7 and the Appendix A figures, the GFLOPS and execution time peaks reoccur at single row dimensions due to the row-major execution form of the HPL benchmark.
- In few cases of large peaks that indicate high percentage improvements in GFLOPS and execution time, in such case as in bottom figure in Figure 6.7 in where matrix size dimensions [1x3] at block size 64 or similar dimensions, is due to:
    - Non-modified run under-performed in this instance of the benchmark which causes the modified run to show higher increases in percentage in GFLOPS and execution times.
    - Dimensions are favorable in the row-major execution of the matrix.
    - Small block size that is able to use the full bandwidth of data lines
    - L2 cache awareness which helped improved hit rates for the instance of the benchmark.

- Other cases of large peaks that indicate high percentage improvements in GFLOPS and execution time, in such case as in bottom figure in Figure 6.7 in where matrix size dimensions [4x1] at block size 2048 or similar dimensions, is due to:
    - Non-modified run under-performed in this instance of the benchmark which causes the modified run to show higher increases in percentage in GFLOPS and execution times.
    - Dimensions not favorable for a row-major execution of the matrix.
    - The modifications help increase the percentages due to the affinity assignments of the instance of the benchmark.

6.3 FUTURE WORK

Part of the future work is to continue to implement and test the modified Scheduler with other compute-nodes during execution. This will test the modifications in multiple compute-node execution in a distributed application. Also, the modifications of the Scheduler were only performed in a specific hardware environment. Part of the future work can involve applying these modifications in system environments with different hardware configurations and processing architectures. This can better indicate that a general solution for all types of HPC systems or workstations has been found.

Part of the future work is to continue to improve the modifications of the Scheduler in order to have a more successful execution times and performance throughput values. Some of the results in Appendix A show that in some scenarios, different instances of the benchmark did not show any improvement while in other scenarios with the same workload improvement was shown. The results indicate that to improve the load balancing modifications in the decision making when assigning affinity to runnable tasks can help improve those scenarios where the compute-node did not improve in execution time and performance. A form of improvement is to provide a more general theory of

understanding in how the modified Scheduler works. The general theory of scheduling can help reach a theoretical model of the modified Scheduler and improve the modifications.

Another part of the future work is to use other benchmark programs that have identical and different algorithms from the HPL benchmark. Different benchmarks can test the modified Scheduler in the ability to efficiently load balance tasks and provide L2 cache awareness during different executed algorithms.

References

[Aas05] Josh Aas, “Understanding the Linux 2.6.8.1 CPU Scheduler,” Silicon Graphics Inc., February 2005.

[Bac90] Maurice J. Bach, “The Design of the UNIX Operating System” Prentice Hall Software Series, Upper Saddle river, NJ, 1990.

[Beo04] “Beowulf Cluster Overview,” 2008 [Online] http://www.beowulf.org/, [Accessed: September 2008].

[Cal11] University of California and Scalable Systems, “Grid Engine User Guide: 5.4.3 Edition," *University of California*, August 2011.

[Cli07] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra, “Automated Empirical Optimization of Software and the ATLAS Project," *University of Tennessee*, 2007.

[Cpu09] “What is PerfMonitor” 2009 [Online] http://www.cpuid.com/softwares/perfmonitor.html#whatisperfmonitor. [Accessed: Septermber 2009].

[Con11] Condor Team, “Condor Version 7.6.4 Manual," *University of Wisconsin-Madison*, October 2011.

[Fed05] Alexandra Fedorova, Margo Seltzer, et. al., “Performance Of Multithreaded Chip Multiprocessors And Implications For Operating System Design,” In *Proceedings of USENIX 2005 Annual Technical Conference* Anaheim, CA, April 2005.

[Fed06] Alexandra Fedorova, Margo Seltzer and Michael D. Smith, “A Non-Work-Conserving Operating System Scheduler for SMT Processors,” In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, in conjunction with ISCA-33, June 2006.

[Int07] Intel Corporation., “Quad-Core Intel Xeon Processor 5400 Series," *Product Brief*, p.5, 2007.

[Inm07] Intel Corporation, “Intel 5400 Chipset Memory Controller Hub (MCH),” *Datasheet*, Document Number: 318610, November 2007.

[Klu08] Tobias Klug, Michael Ott, et. al., “Autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems.” In *Transactions on High-Performance Embedded Architectures and Compilers (Transactions on HiPEAC)*, volume 3 (4), pages 219-235, 2008.

[Lov10] Robert Love, “Linux Kernel Development.” Third Edition. Addison-Wesley, Crawfordsville, IN, 2010.

[Mau08] Wolfgang Mauerer, “Linux Kernel Architecture.” Wiley Publishing, Inc., Indianapolis, IN, 2008.

[Mpi11] “About MPICH Overview,” [Online] http://www.mcs.anl.gov/research/projects/mpich2/, [Accessed: October 2011].

[Hpl08] HPL website, “A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers”, 2008. [Online]. http://www.netlib.org/benchmark/hpl/ [Accessed: September 15, 2008].

[Pin08] Michael L. Pinedo, “Scheduling: Theory, Algorithms and System,” Springer Science-Business Media, New York, NY, 2008.

[Raj07] Mohan Rajagopalan, Brian T. Lewis and Todd A. Anderson, “Thread Scheduling of Multi-Core Platforms,” In *Usenix11th Workshop on Hot Topics in Operating Systems*, San Diego, CA. May 2007.

[Roc08] Official Rocks Clusters website, "Rocks Cluster Distribution: Users Guide," 2008. [Online]. http://www.rocksclusters.org/rocks-documentation/4.3/ [Accessed: June 30, 2008].

[Top11] "The Linpack Benchmark" 2011 [Online] http://www.top500.org/project/linpack [Accessed: October 2011].

[Vah96] Uresh Vahalia, "UNIX Internals: The New Frontiers" Prentice Hall, Upper Saddle river, NJ, 1996.

[Val09] Damian Valles, David Williams and Patricia Nava, "Performance and Timing Measurements in a Multi-core Beowulf Cluster Compute-Node," In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV. July 2009.

[Wec06] Wechsler, Ofri., "Inside Intel Core Microarchitecture: Setting New Standards for Energy-Efficient Performance," *White Paper*. 2006.

[Zie08] Stephen Ziemba, Gautam Upadhyaya and Vijay Pai, "Analyzing the Effectiveness of Multicore Scheduling Using Performance Counters," In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, in conjunction with ISCA-35, June 2008.

Appendix A

A.1 IMPROVEMENT PERCENTAGE OF EXECUTION TIME: EIGHT PROCESS CONCURRENCY

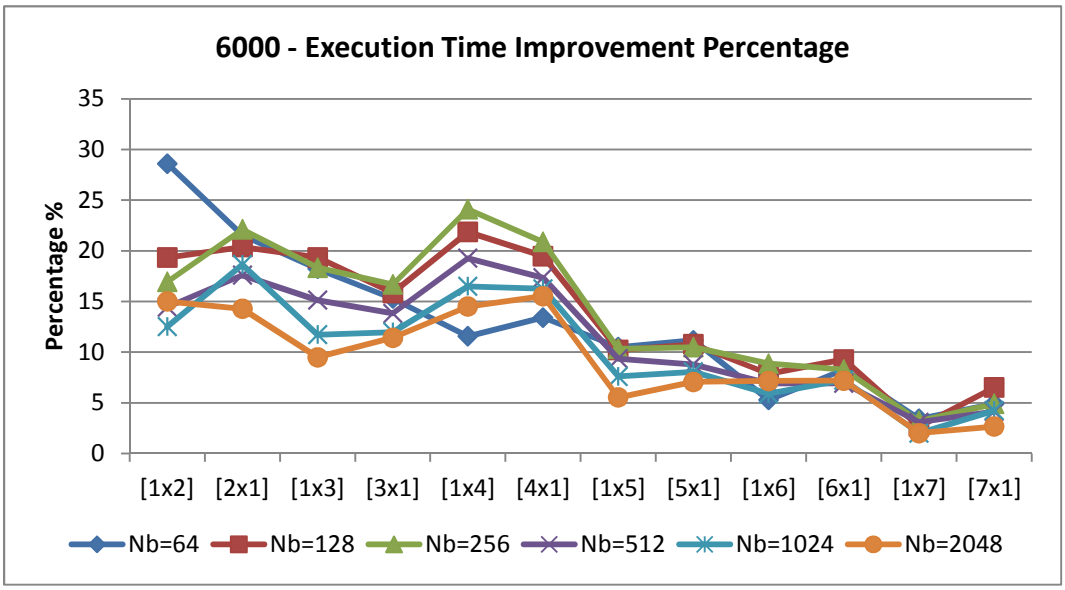


Figure A.1: Execution Time Improvement of HPL with Problem Size 6000 – Eight Concurrency.

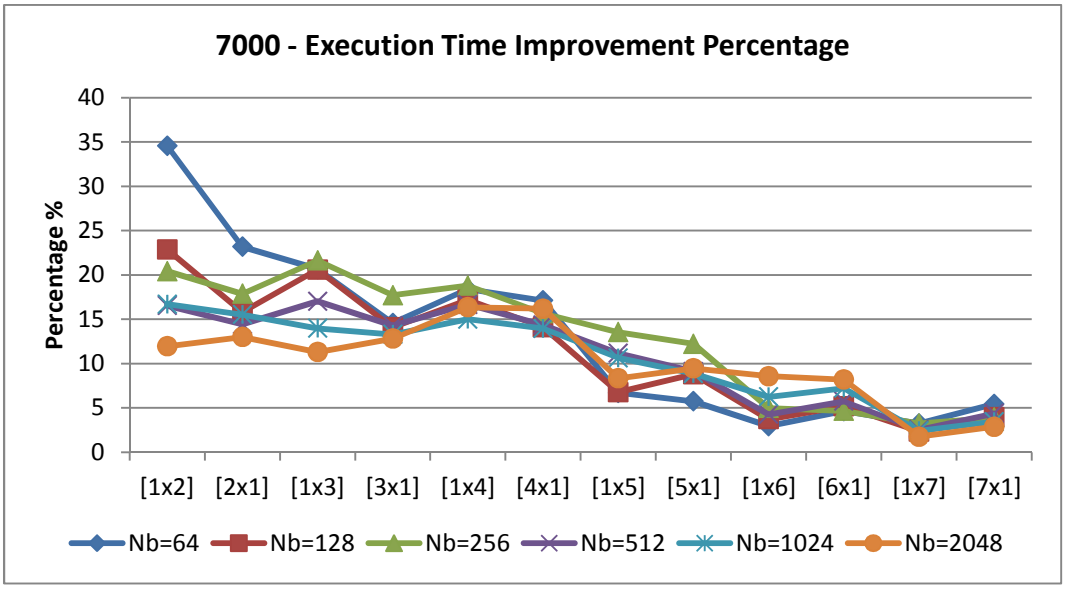


Figure A.2: Execution Time Improvement of HPL with Problem Size 7000 – Eight Concurrency.

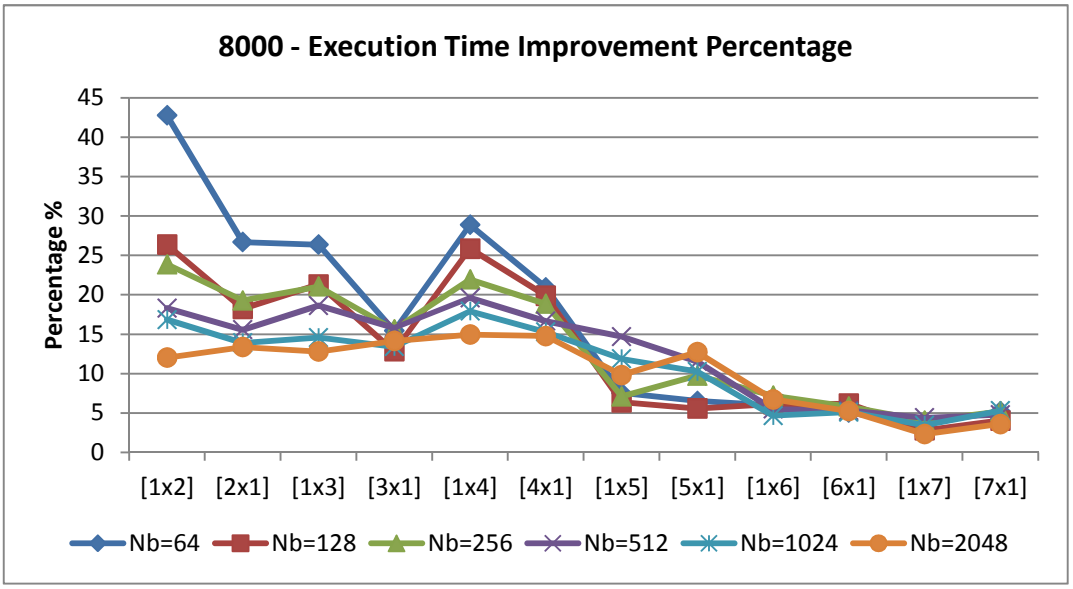


Figure A.3: Execution Time Improvement of HPL with Problem Size 8000 – Eight Concurrency.

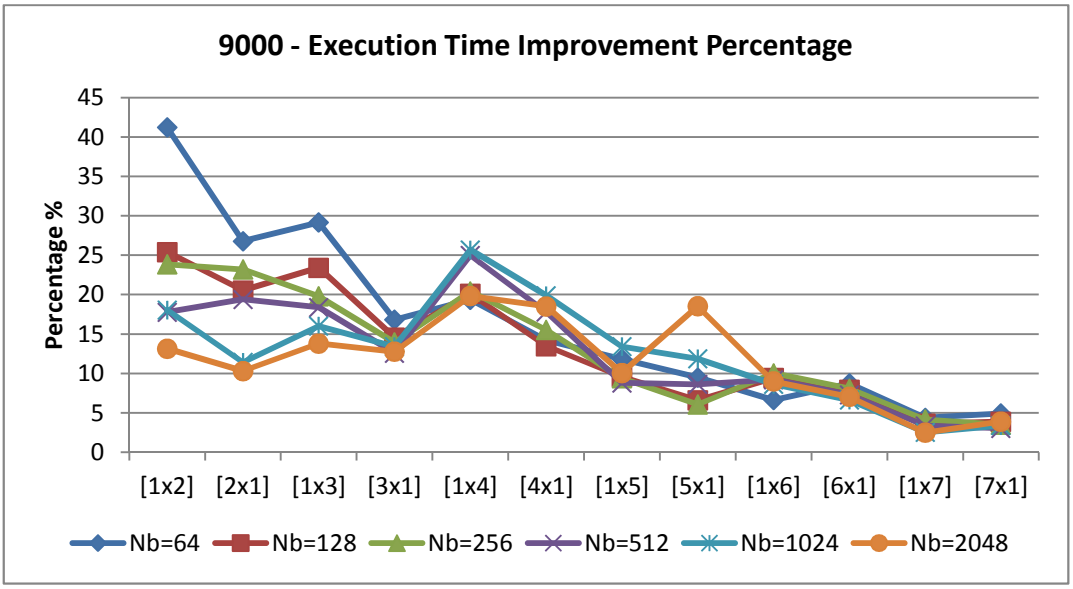


Figure A.4: Execution Time Improvement of HPL with Problem Size 9000 – Eight Concurrency.

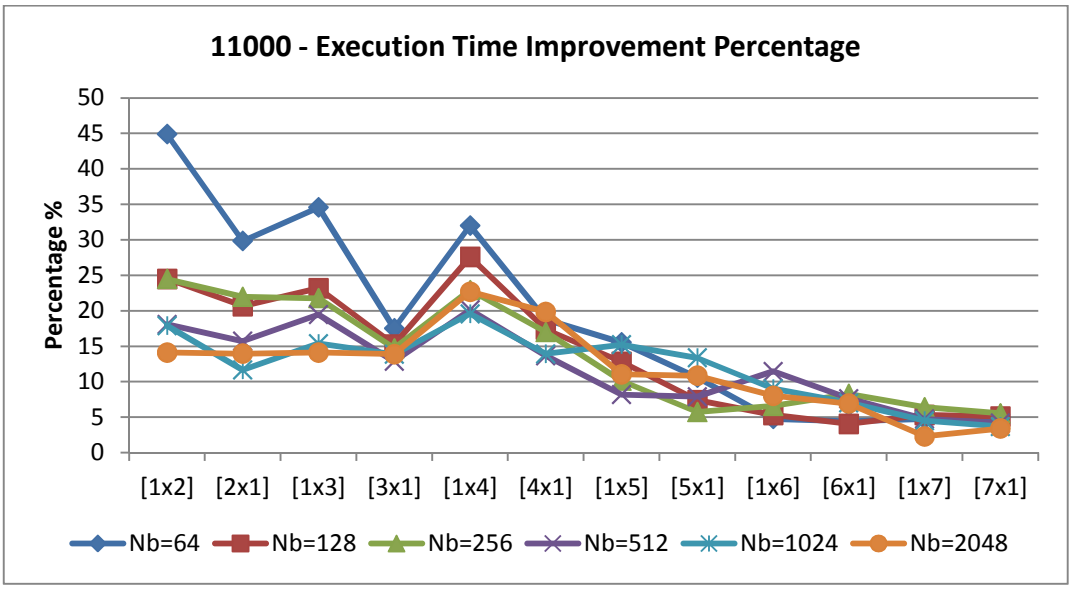


Figure A.5: Execution Time Improvement of HPL with Problem Size 11000 – Eight Concurrency.

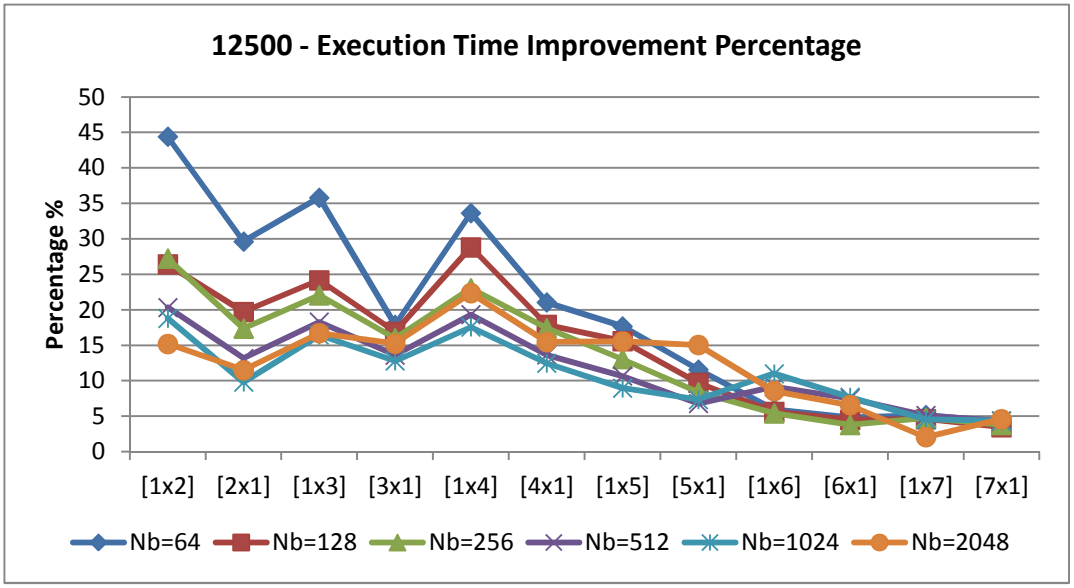


Figure A.6: Execution Time Improvement of HPL with Problem Size 12500 – Eight Concurrency.

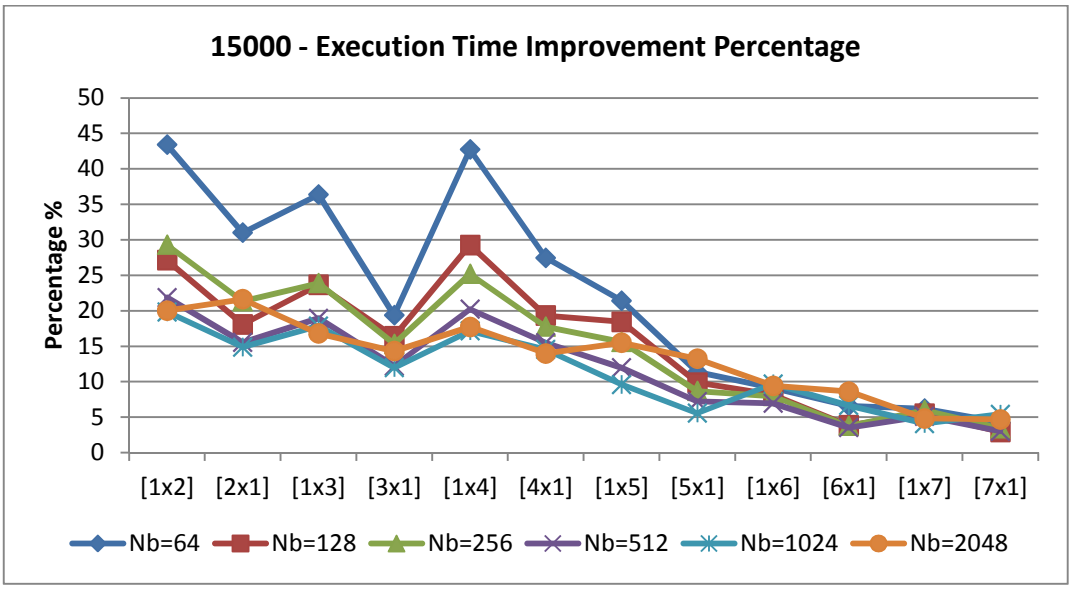


Figure A.7: Execution Time Improvement of HPL with Problem Size 15000 – Eight Concurrency.

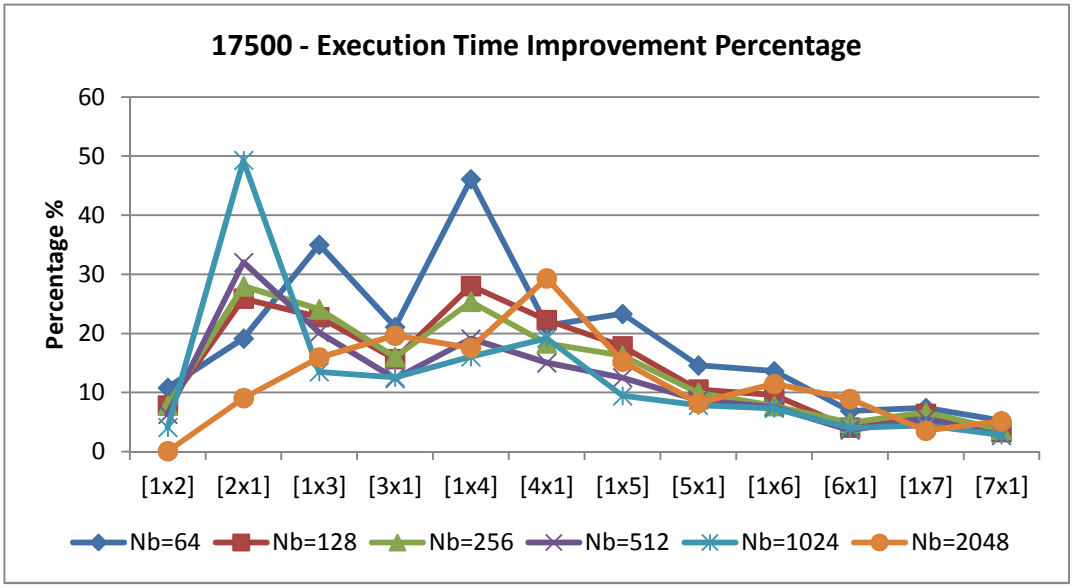


Figure A.8: Execution Time Improvement of HPL with Problem Size 17500 – Eight Concurrency.

A.2 IMPROVEMENT PERCENTAGE OF EXECUTION TIME: SEVEN PROCESS CONCURRENCY

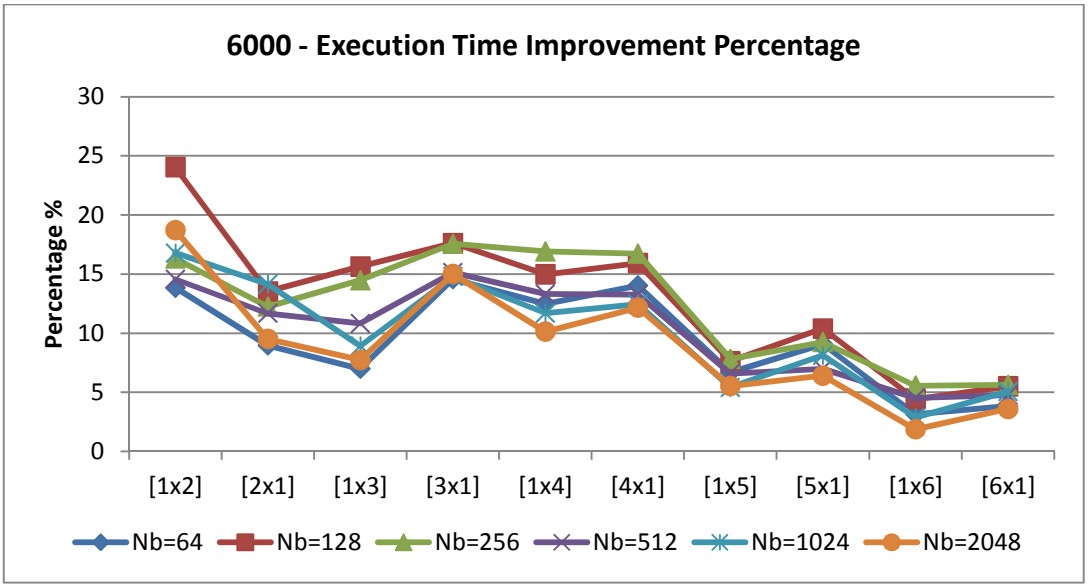


Figure A.9: Execution Time Improvement of HPL with Problem Size 6000 – Seven Concurrency.

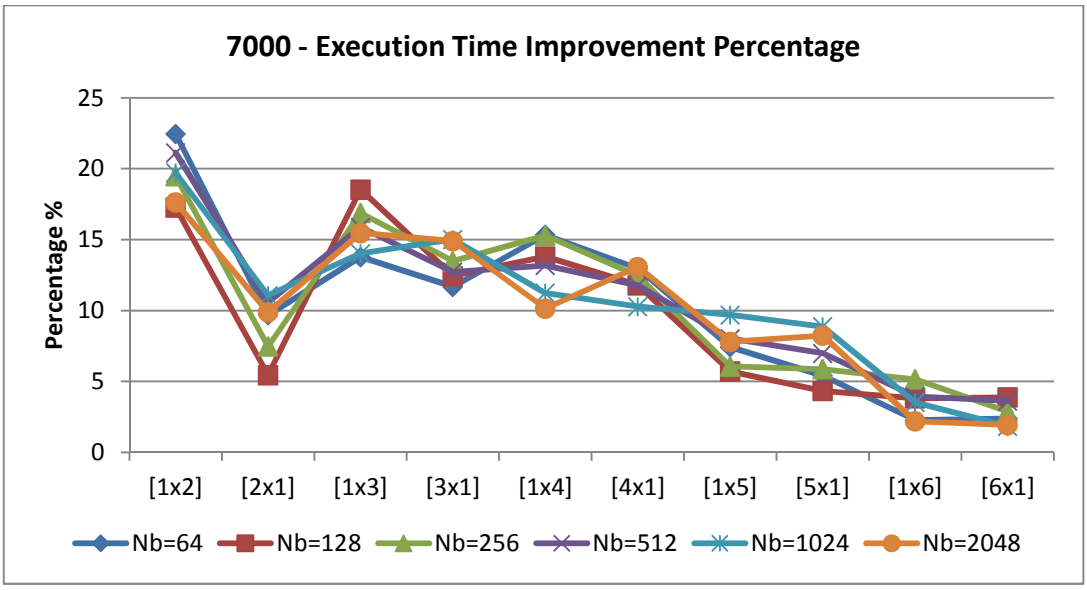


Figure A.10: Execution Time Improvement of HPL with Problem Size 7000 – Seven Concurrency.

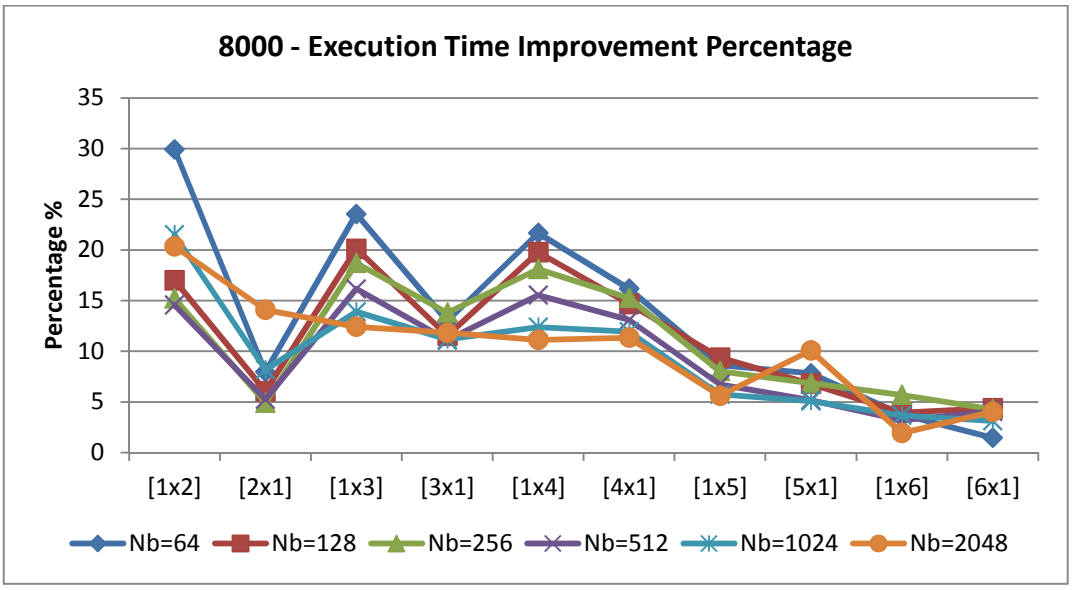


Figure A.11: Execution Time Improvement of HPL with Problem Size 8000 – Seven Concurrency.

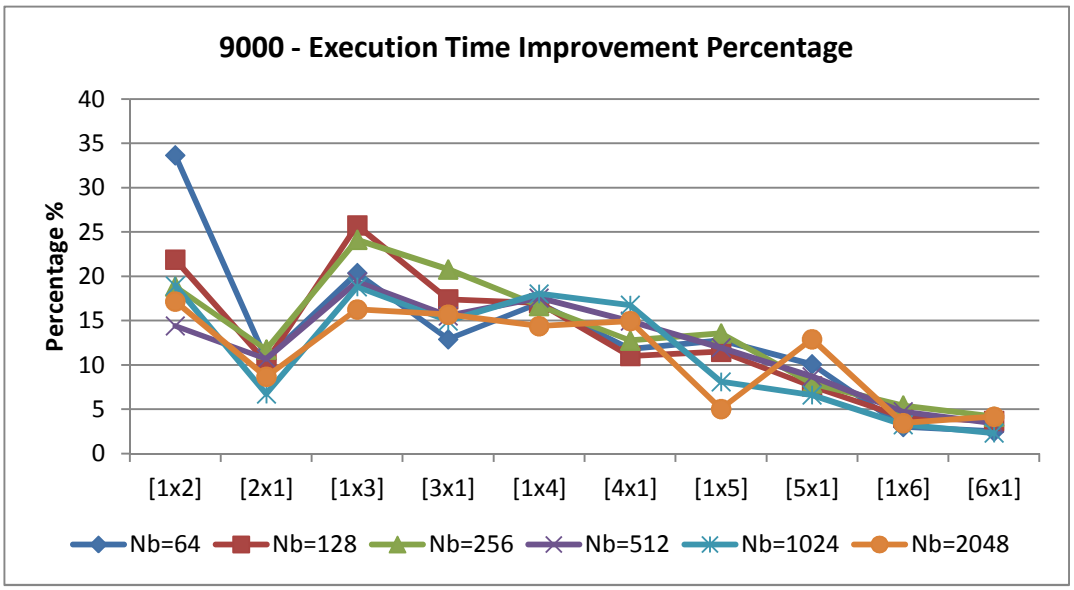


Figure A.12: Execution Time Improvement of HPL with Problem Size 9000 – Seven Concurrency.

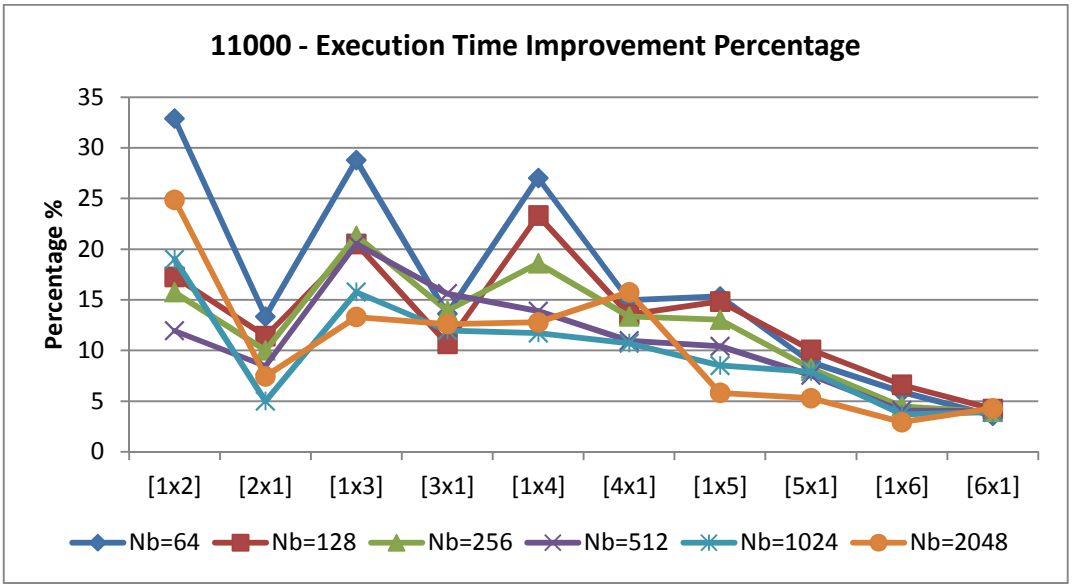


Figure A.13: Execution Time Improvement of HPL with Problem Size 11000 – Seven Concurrency.

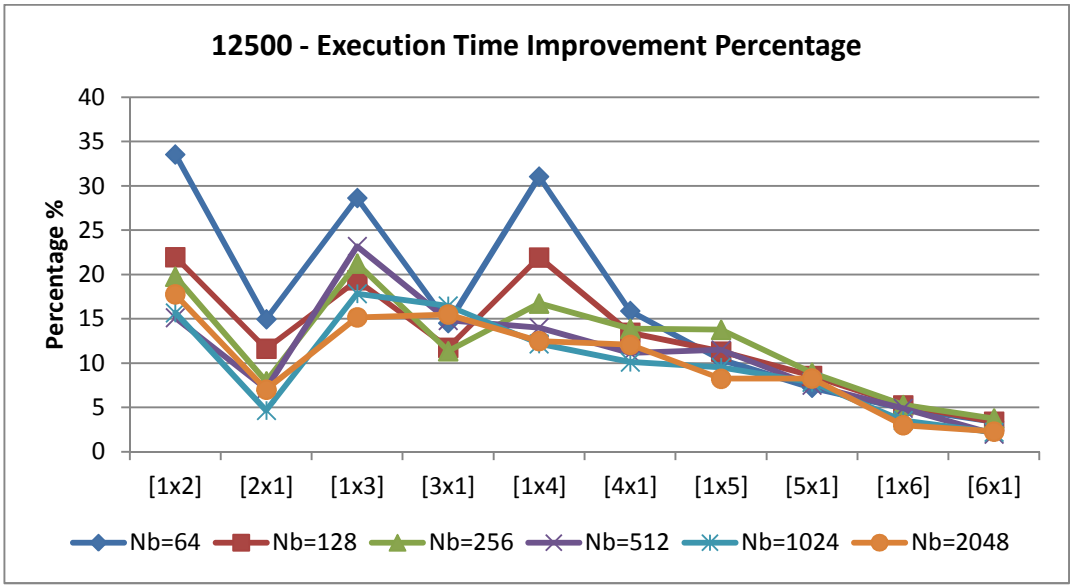


Figure A.14: Execution Time Improvement of HPL with Problem Size 12500 – Seven Concurrency.

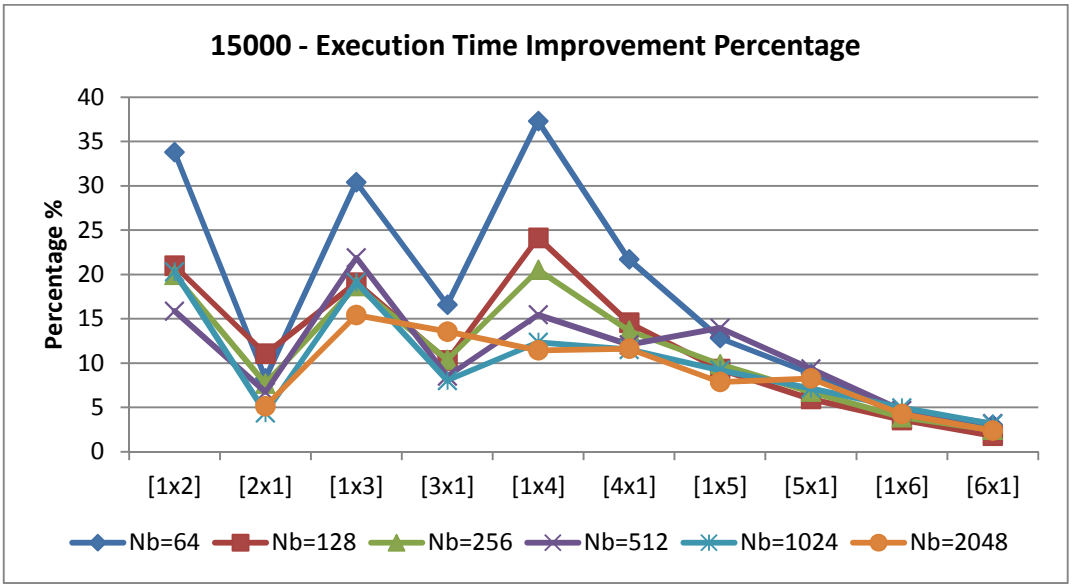


Figure A.15: Execution Time Improvement of HPL with Problem Size 15000 – Seven Concurrency.

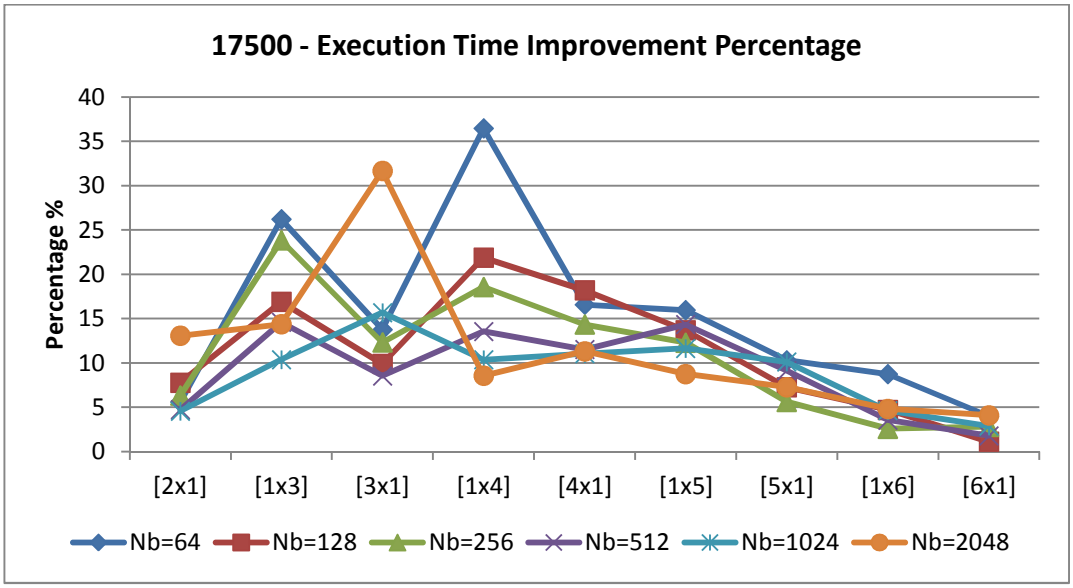


Figure A.16: Execution Time Improvement of HPL with Problem Size 17500 – Seven Concurrency.

A.3 IMPROVEMENT PERCENTAGE OF EXECUTION TIME: SIX PROCESS CONCURRENCY

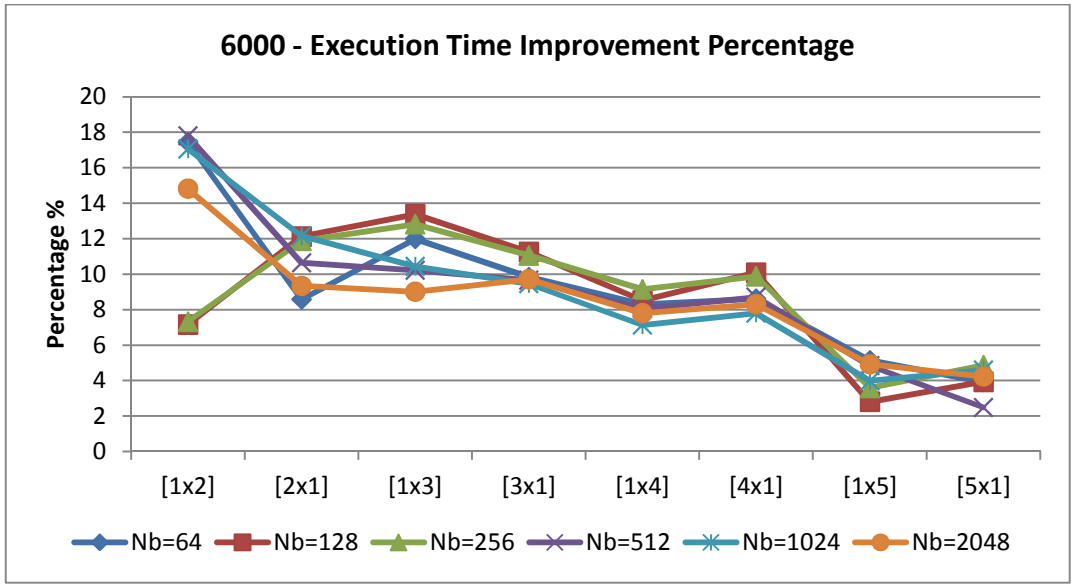


Figure A.17: Execution Time Improvement of HPL with Problem Size 6000 – Six Concurrency.

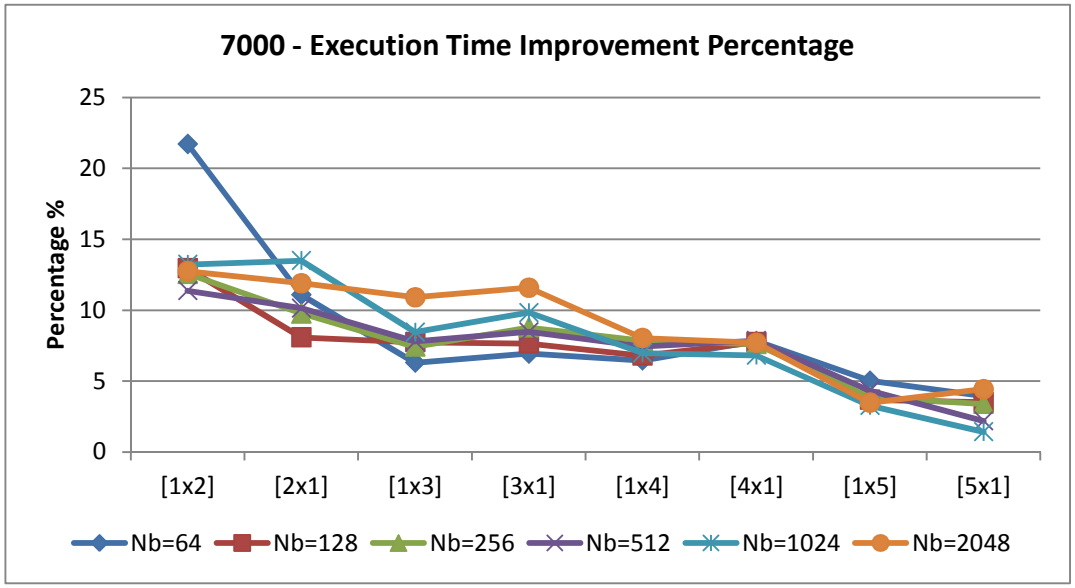


Figure A.18: Execution Time Improvement of HPL with Problem Size 7000 – Six Concurrency.

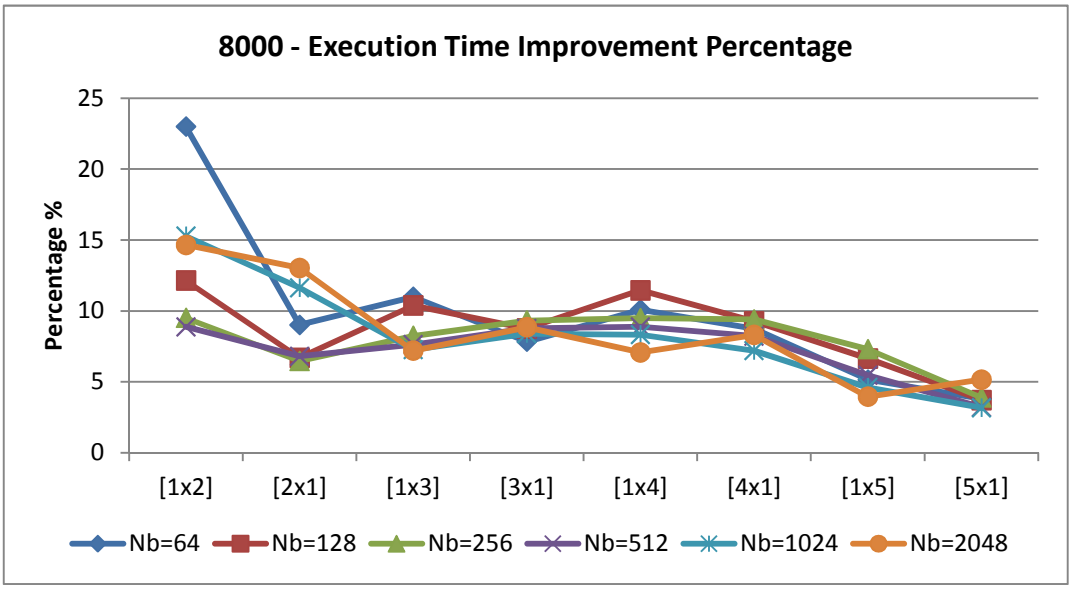


Figure A.19: Execution Time Improvement of HPL with Problem Size 8000 – Six Concurrency.

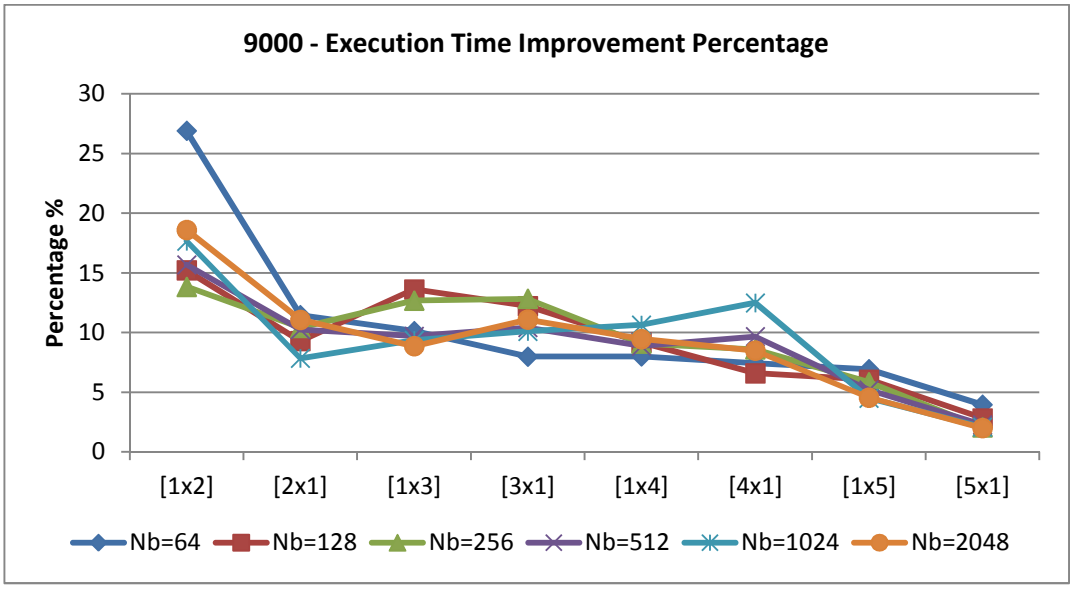


Figure A.20: Execution Time Improvement of HPL with Problem Size 9000 – Six Concurrency.

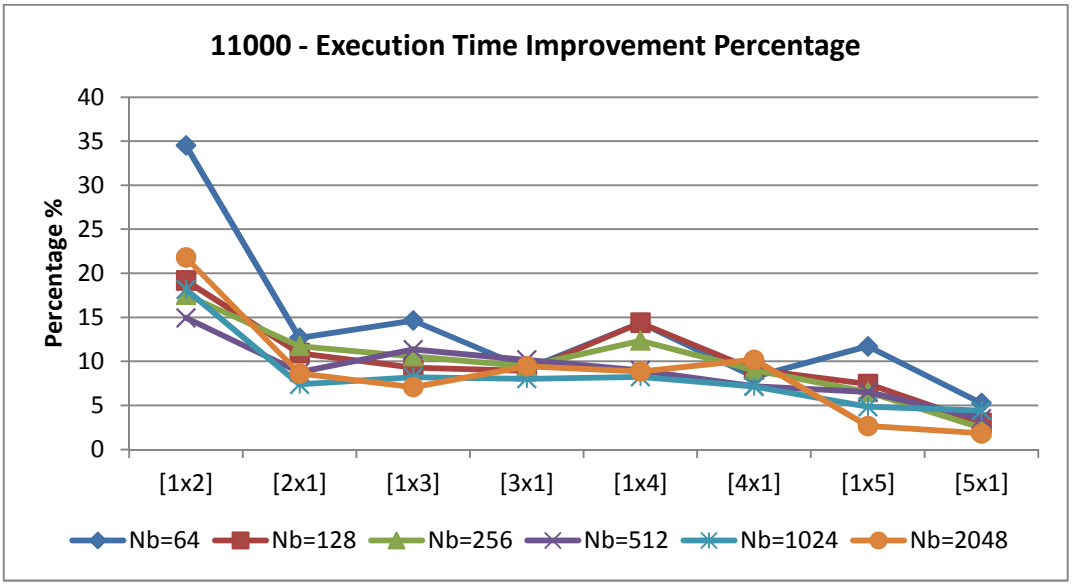


Figure A.21: Execution Time Improvement of HPL with Problem Size 11000 – Six Concurrency.

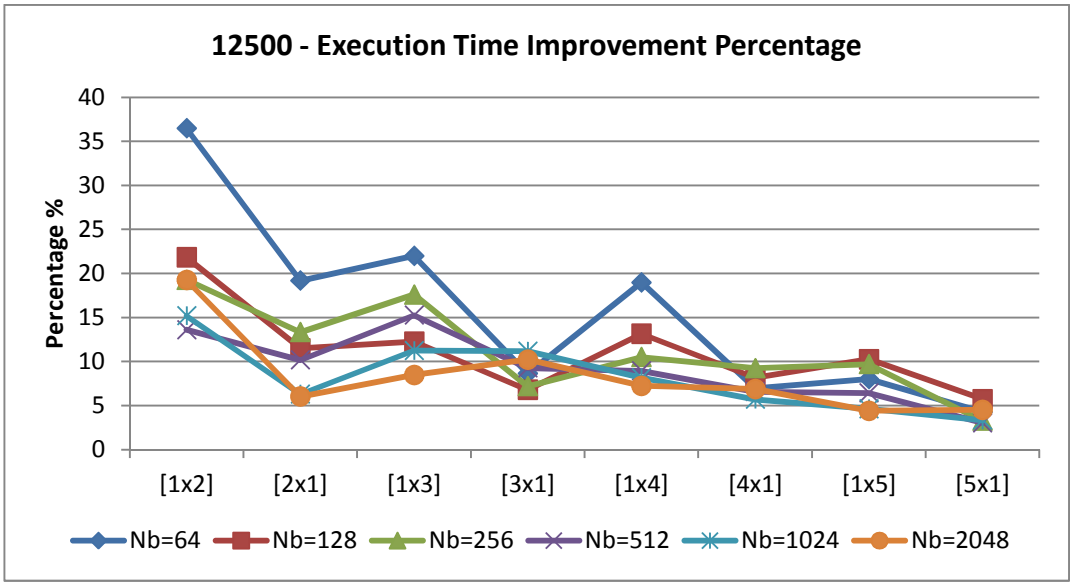


Figure A.22: Execution Time Improvement of HPL with Problem Size 12500 – Six Concurrency.

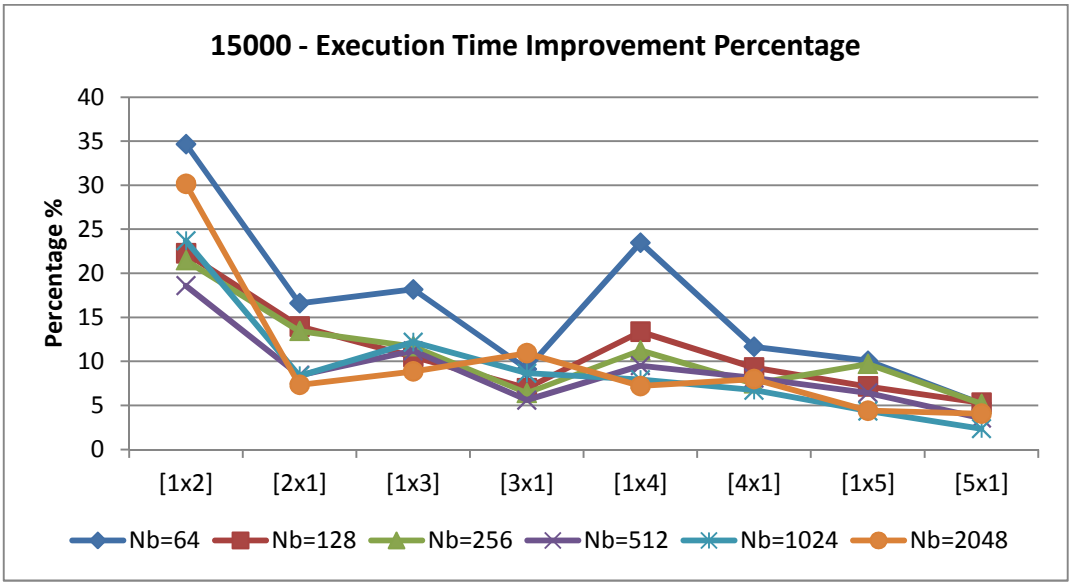


Figure A.23: Execution Time Improvement of HPL with Problem Size 15000 – Six Concurrency.

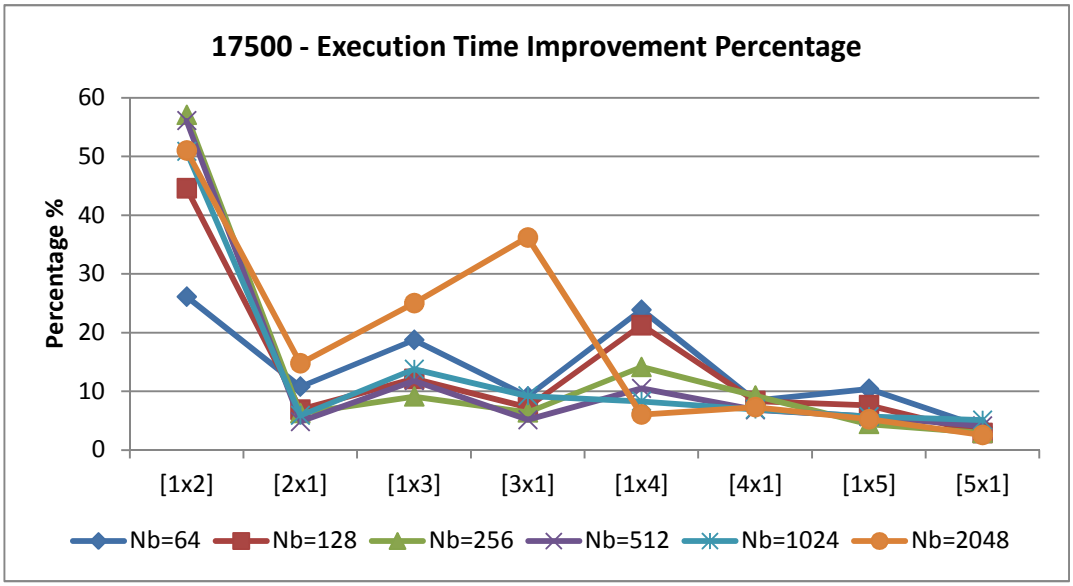


Figure A.24: Execution Time Improvement of HPL with Problem Size 17500 – Six Concurrency.

A.4 IMPROVEMENT PERCENTAGE OF EXECUTION TIME: FIVE PROCESS CONCURRENCY

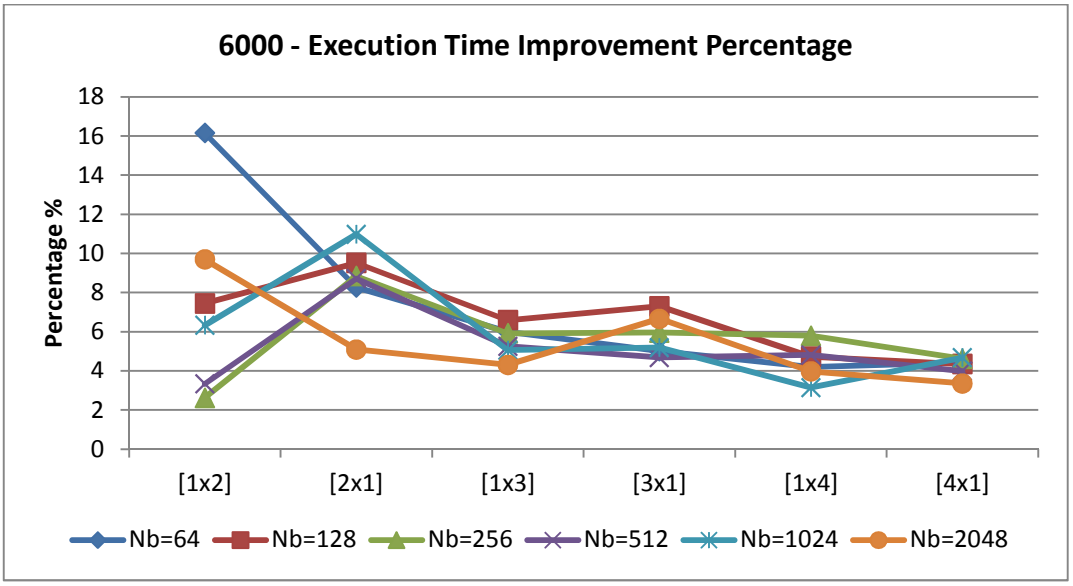


Figure A.25: Execution Time Improvement of HPL with Problem Size 6000 – Five Concurrency.

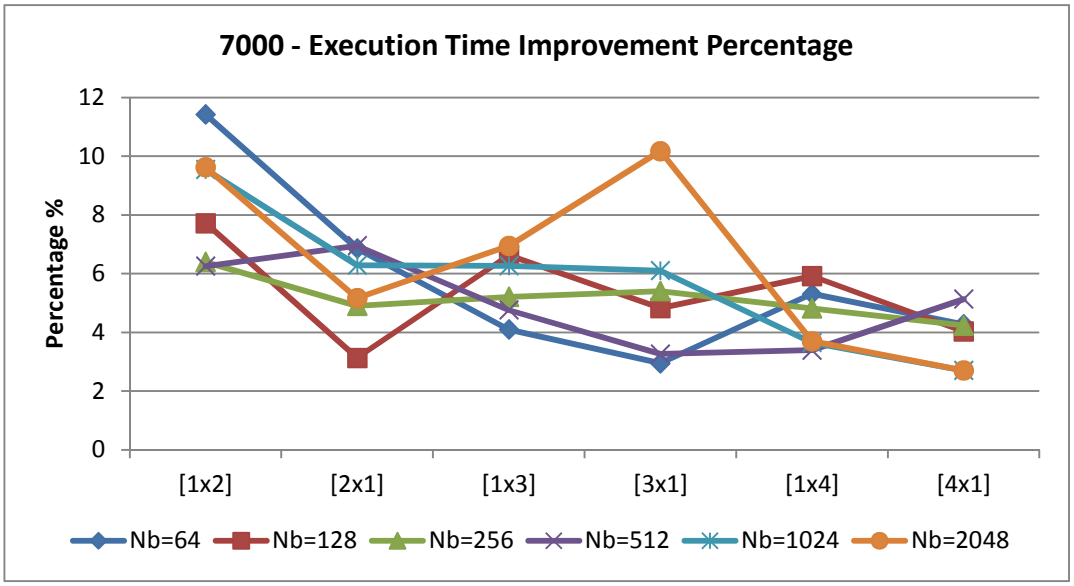


Figure A.26: Execution Time Improvement of HPL with Problem Size 7000 – Five Concurrency.

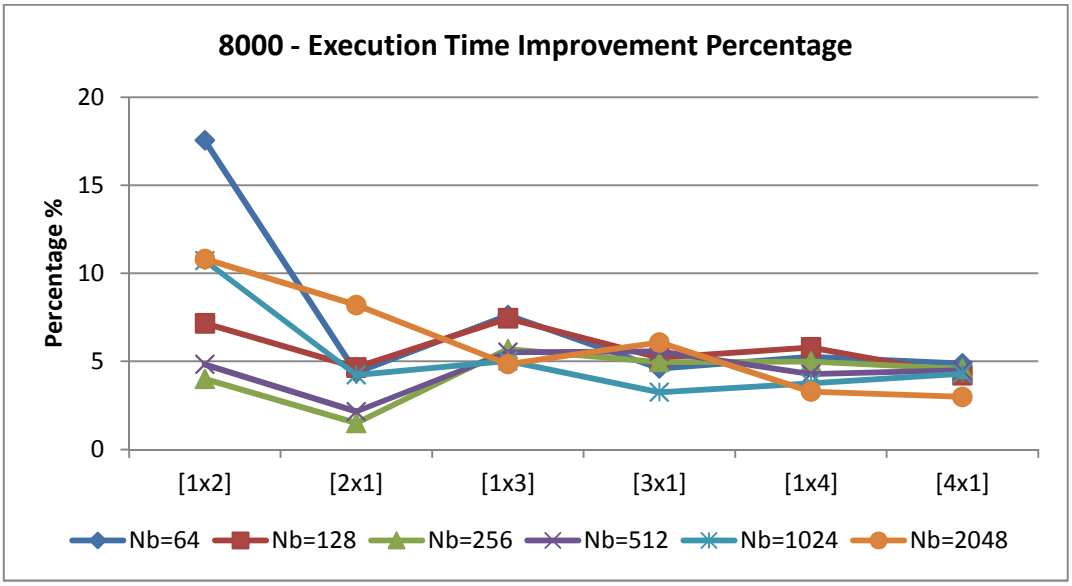


Figure A.27: Execution Time Improvement of HPL with Problem Size 8000 – Five Concurrency.

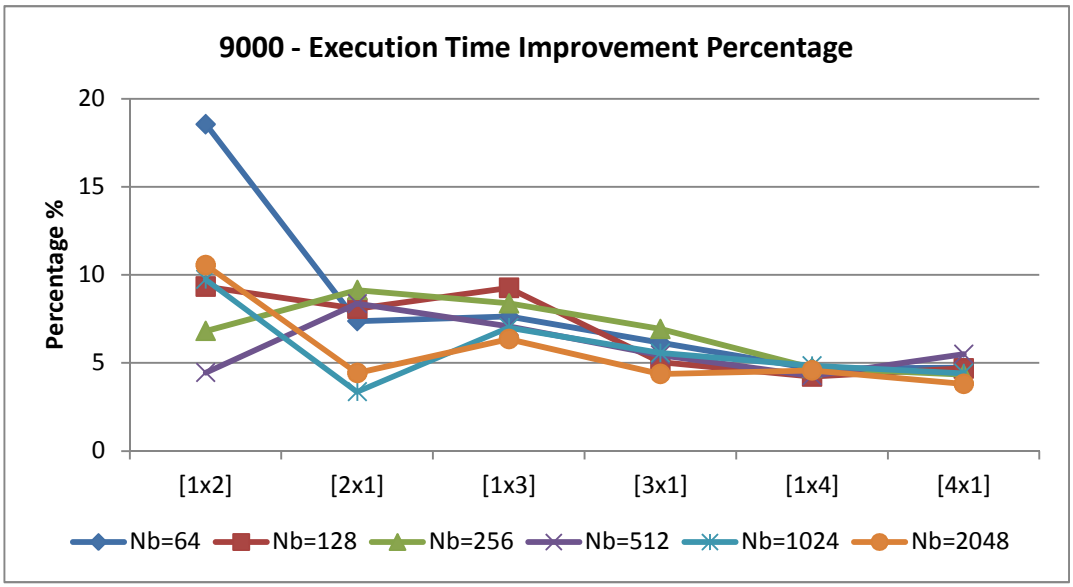


Figure A.28: Execution Time Improvement of HPL with Problem Size 9000 – Five Concurrency.

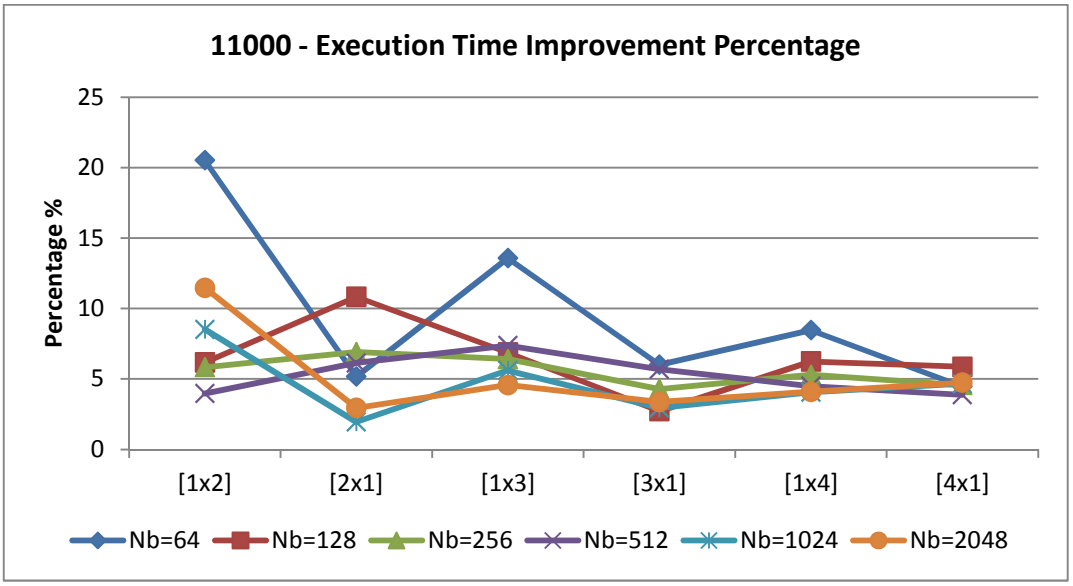


Figure A.29: Execution Time Improvement of HPL with Problem Size 11000 – Five Concurrency.

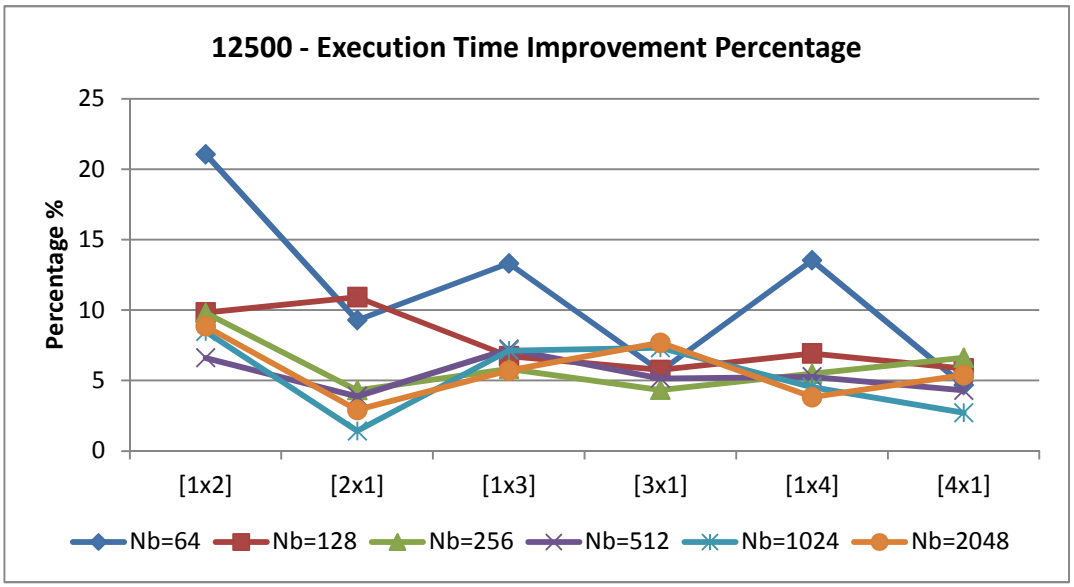


Figure A.30: Execution Time Improvement of HPL with Problem Size 12500 – Five Concurrency.

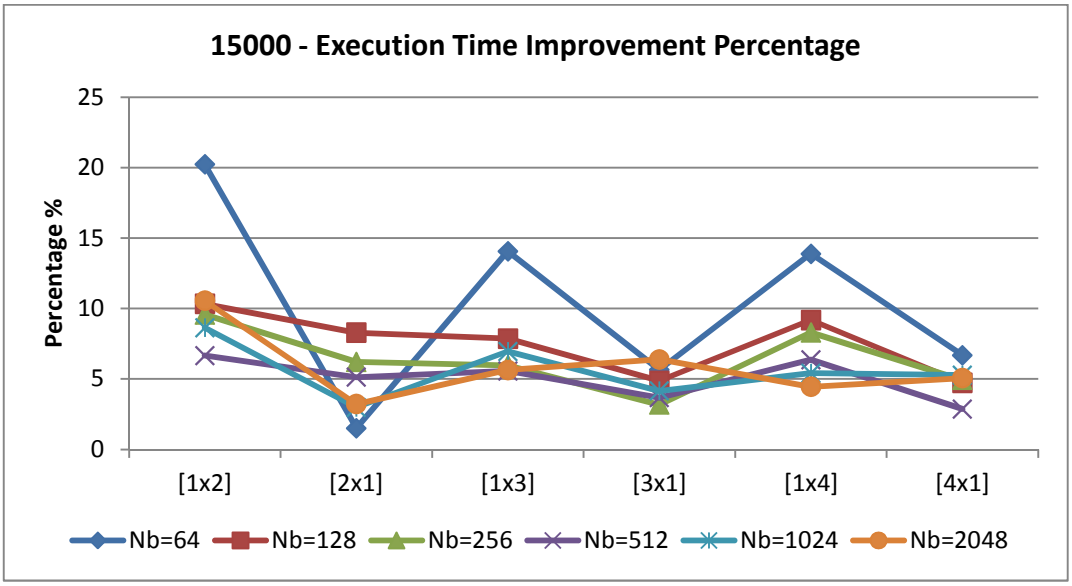


Figure A.31: Execution Time Improvement of HPL with Problem Size 15000 – Five Concurrency.

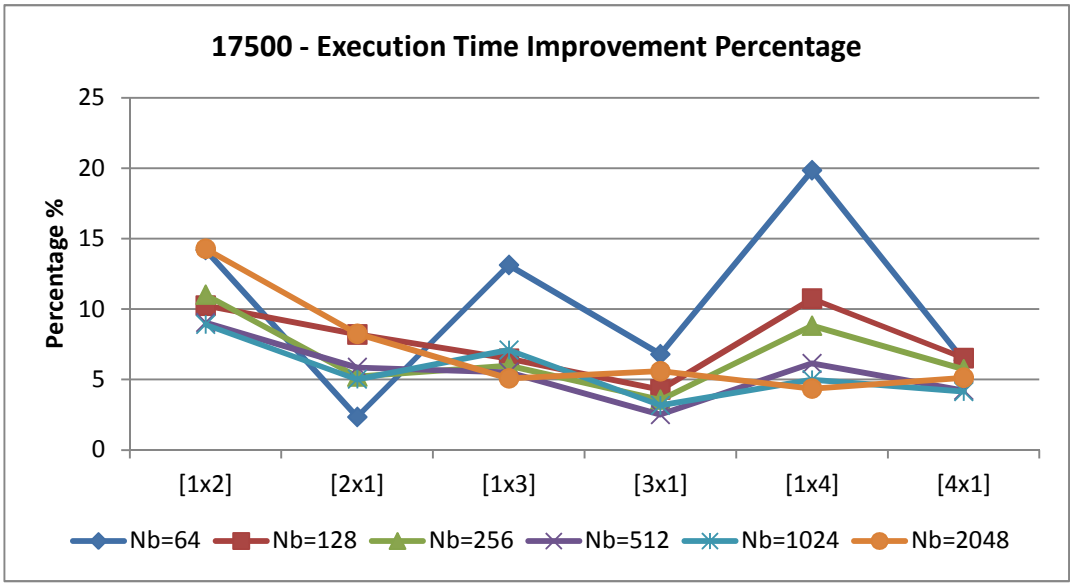


Figure A.32: Execution Time Improvement of HPL with Problem Size 17500 – Five Concurrency.

A.5 IMPROVEMENT PERCENTAGE OF EXECUTION TIME: FOUR PROCESS CONCURRENCY

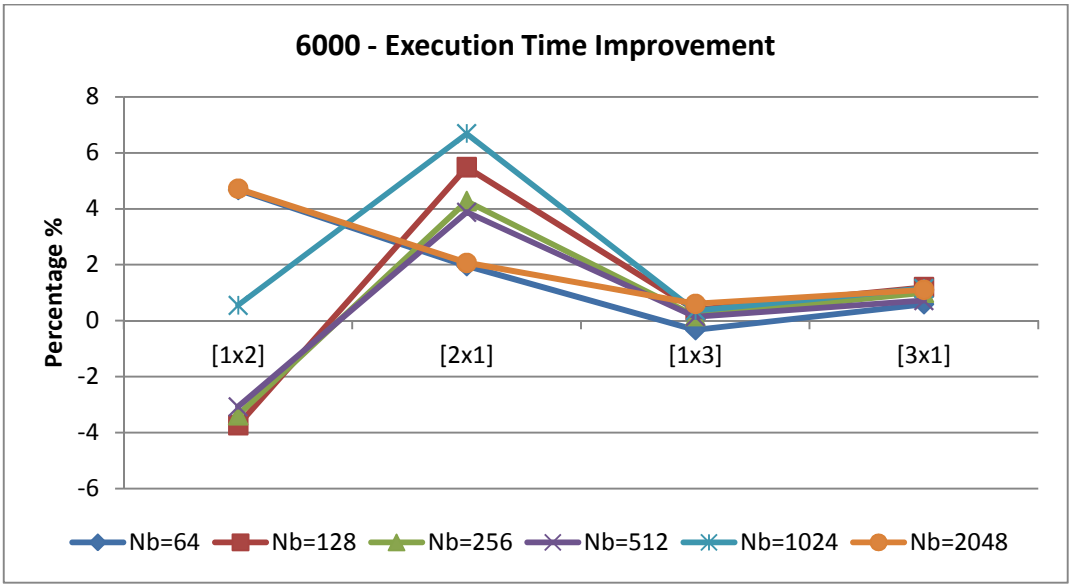


Figure A.33: Execution Time Improvement of HPL with Problem Size 6000 – Four Concurrency.

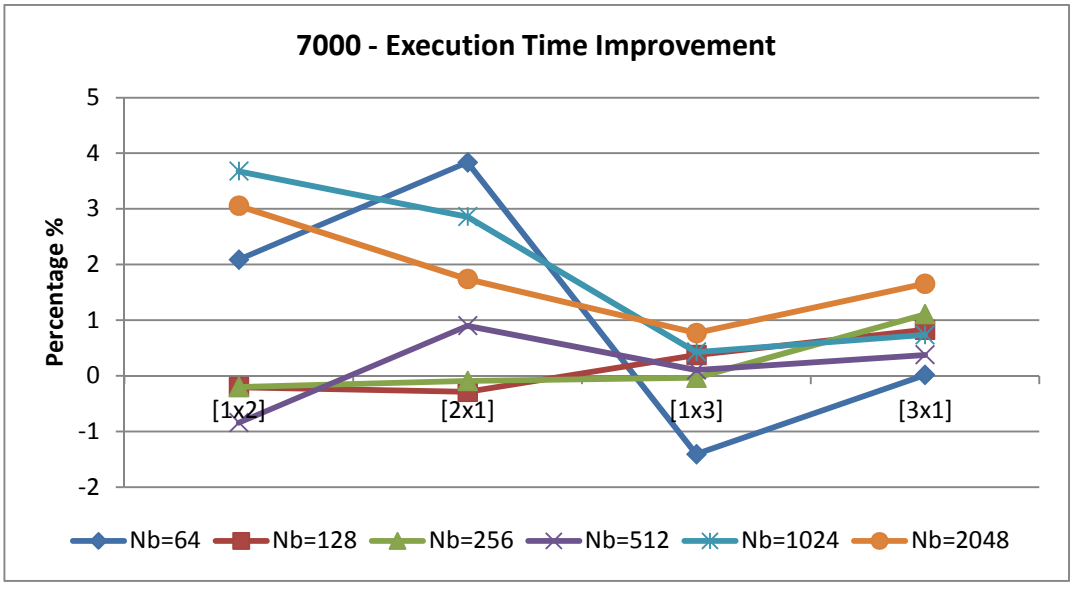


Figure A.34: Execution Time Improvement of HPL with Problem Size 7000 – Four Concurrency.

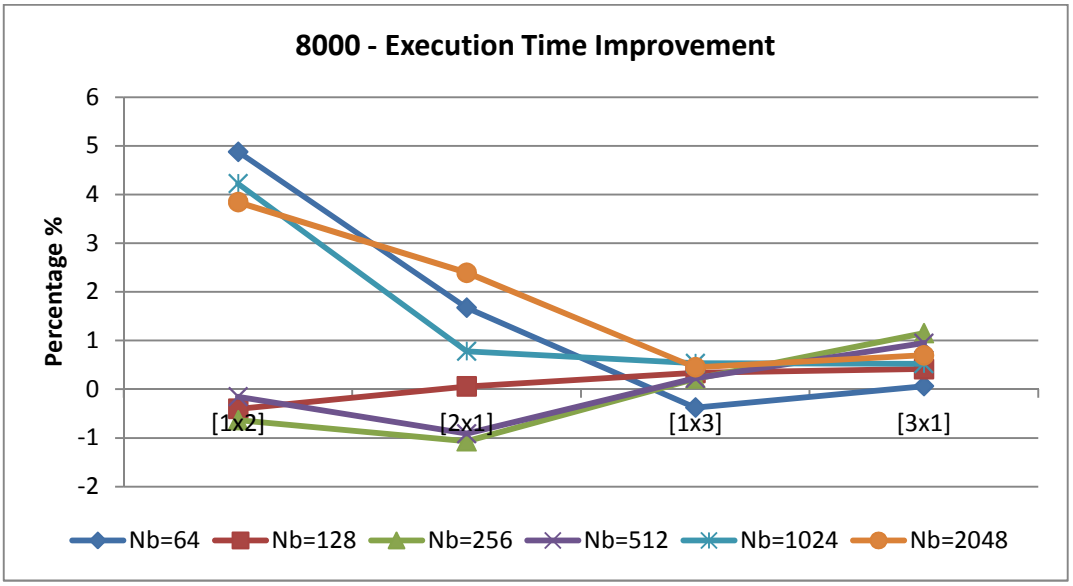


Figure A.35: Execution Time Improvement of HPL with Problem Size 8000 – Four Concurrency.

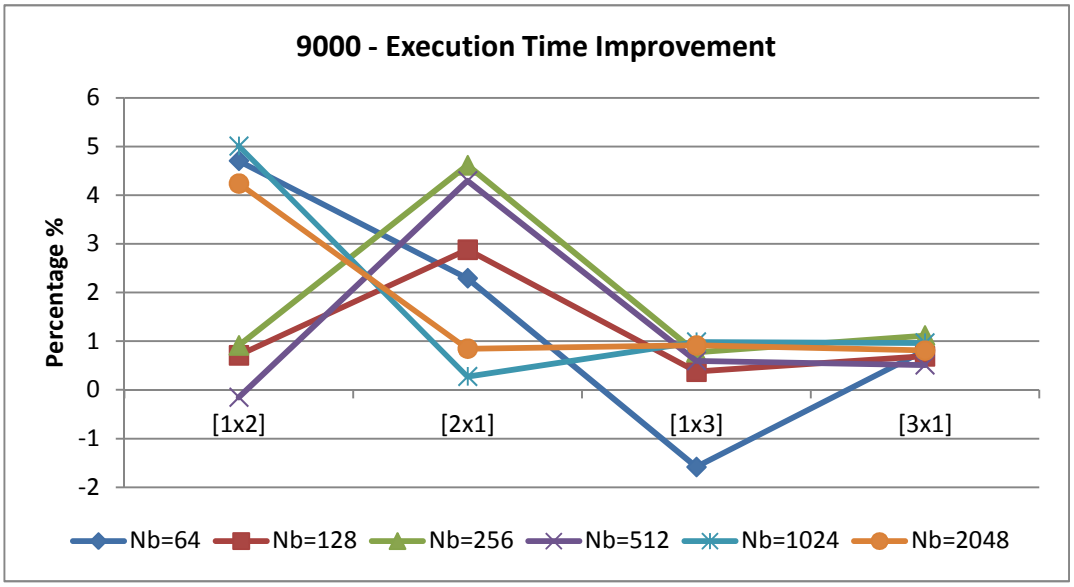


Figure A.36: Execution Time Improvement of HPL with Problem Size 9000 – Four Concurrency.

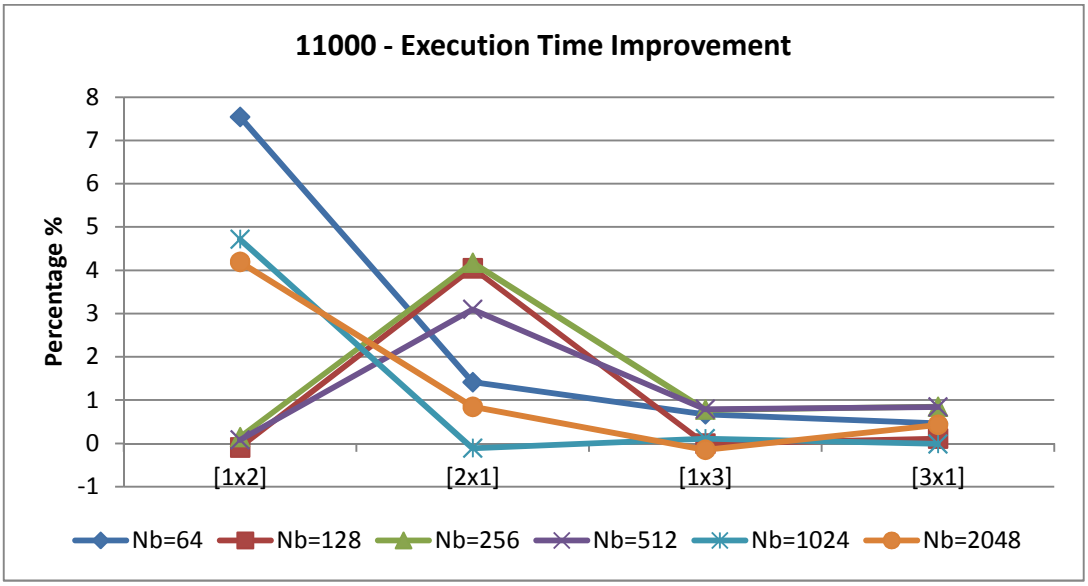


Figure A.37: Execution Time Improvement of HPL with Problem Size 11000 – Four Concurrency.

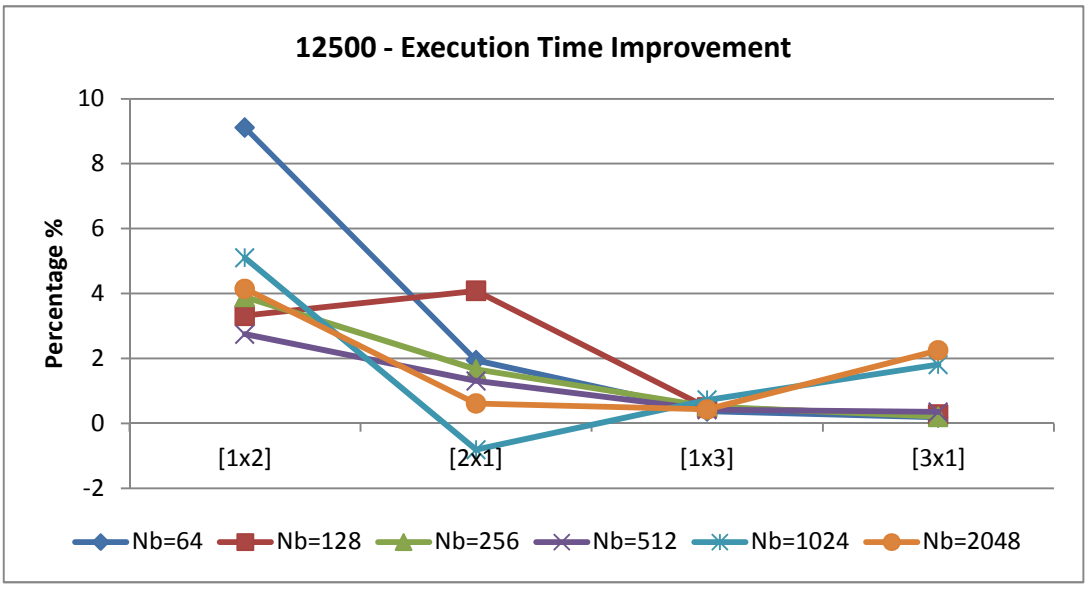


Figure A.38: Execution Time Improvement of HPL with Problem Size 12500 – Four Concurrency.

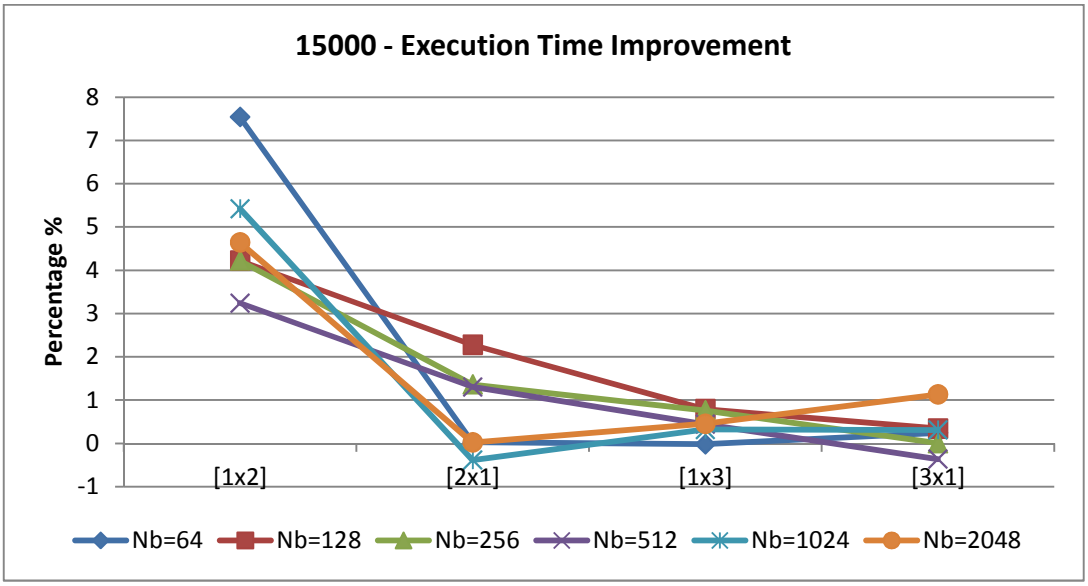


Figure A.39: Execution Time Improvement of HPL with Problem Size 15000 – Four Concurrency.

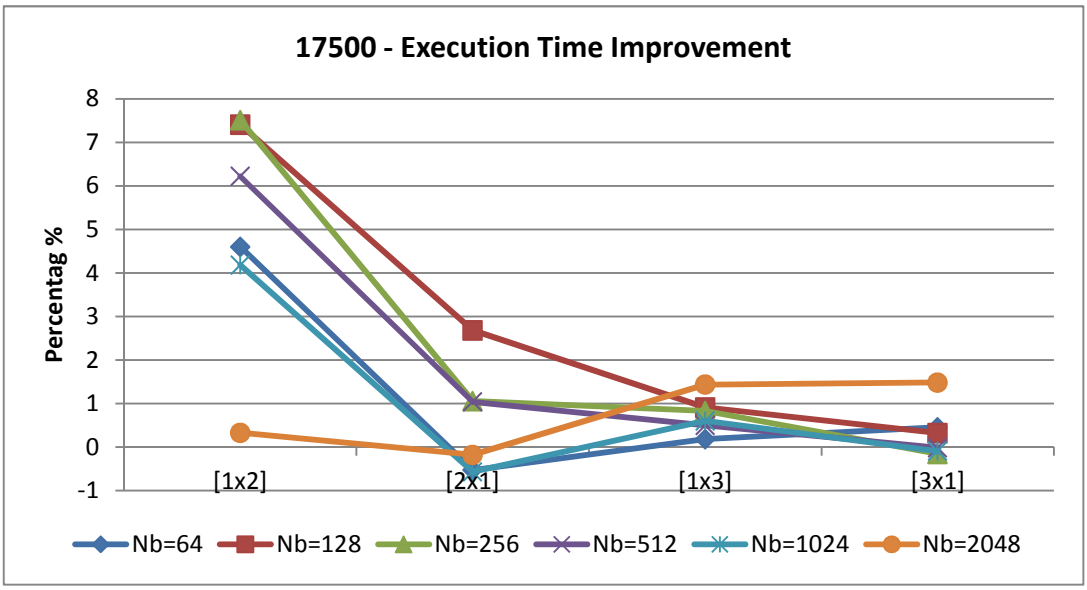


Figure A.40: Execution Time Improvement of HPL with Problem Size 17500 – Four Concurrency.

A.6 IMPROVEMENT PERCENTAGE OF GLOPS: EIGHT PROCESS CONCURRENCY

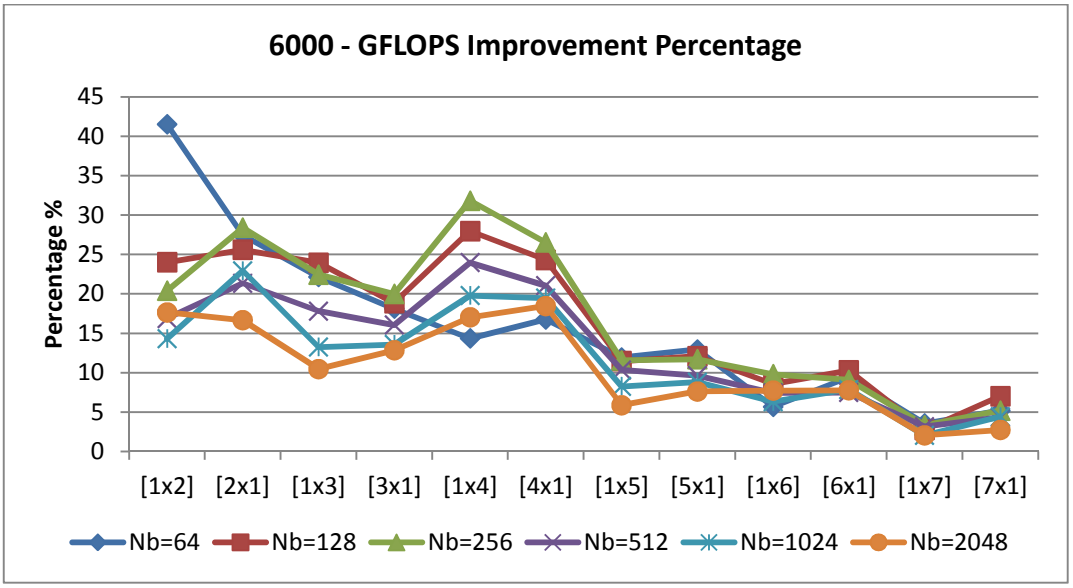


Figure A.41: GFLOPS Improvement of HPL with Problem Size 6000 – Eight Concurrency.

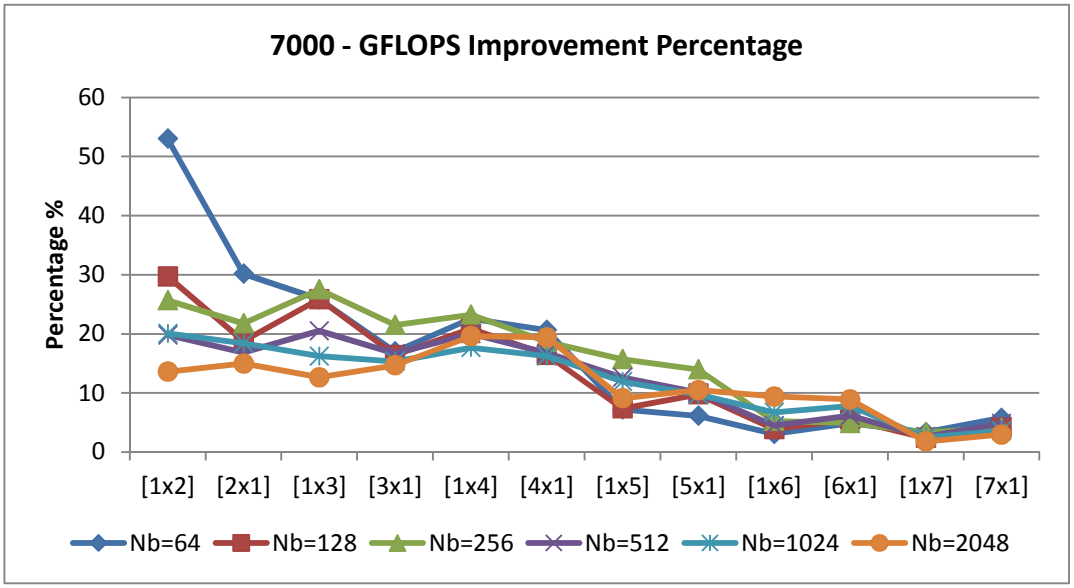


Figure A.42: GFLOPS Improvement of HPL with Problem Size 7000 – Eight Concurrency.

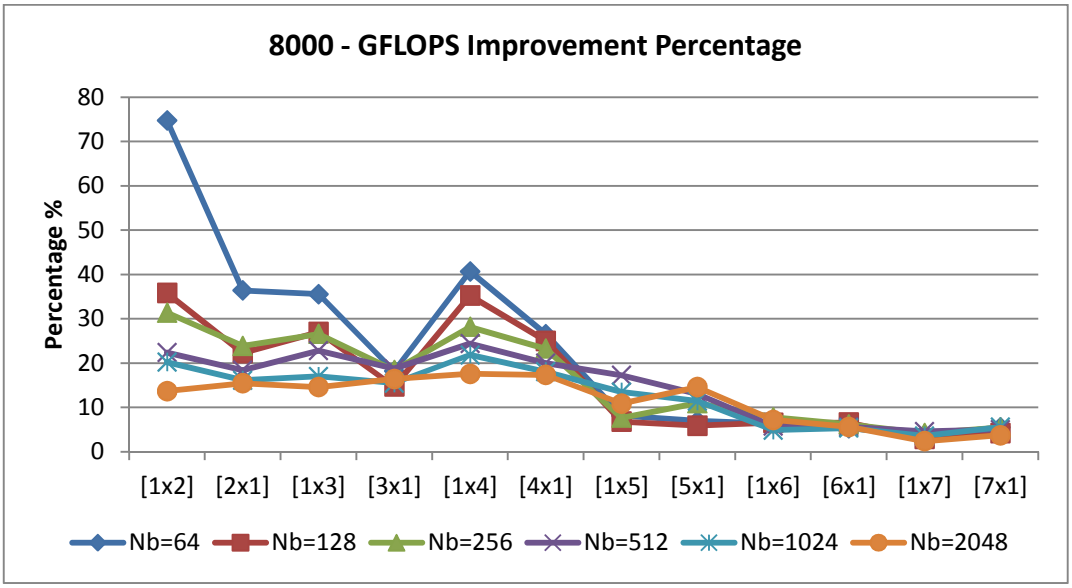


Figure A.43: GFLOPS Improvement of HPL with Problem Size 8000 – Eight Concurrency.

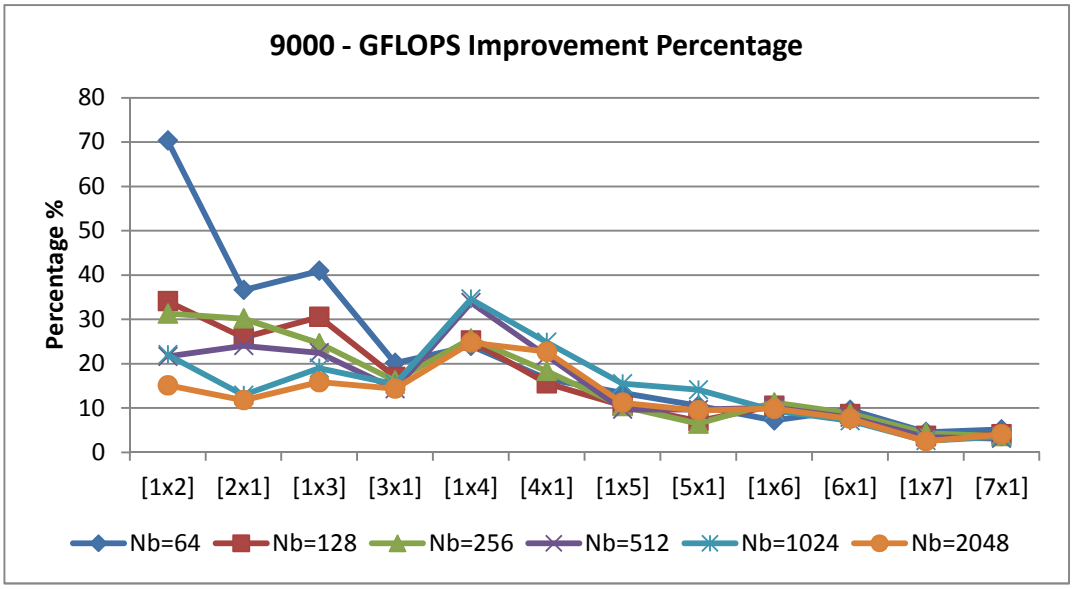


Figure A.44: GFLOPS Improvement of HPL with Problem Size 9000 – Eight Concurrency.

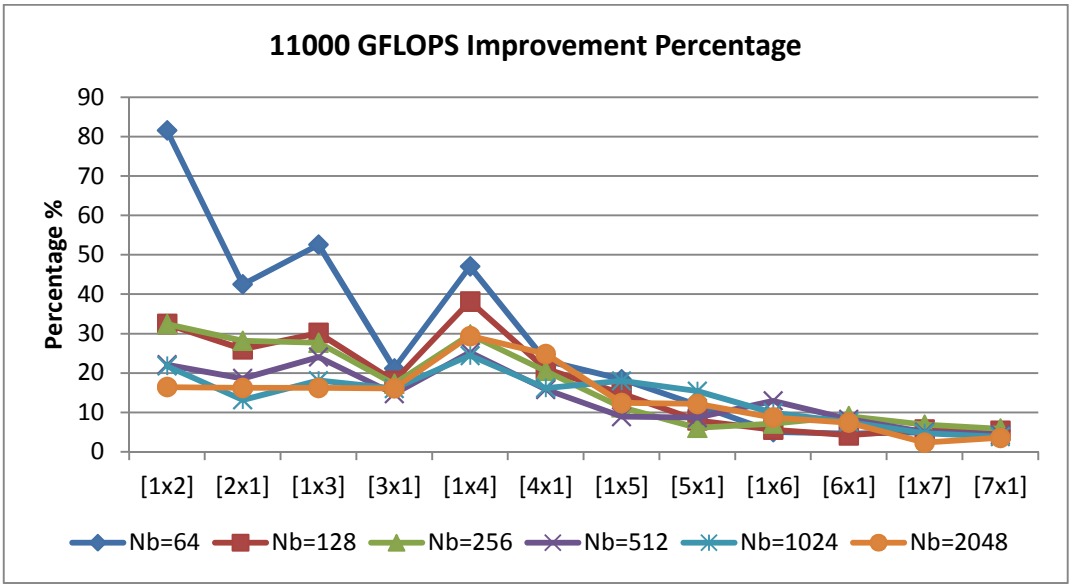


Figure A.45: GFLOPS Improvement of HPL with Problem Size 11000 – Eight Concurrency.

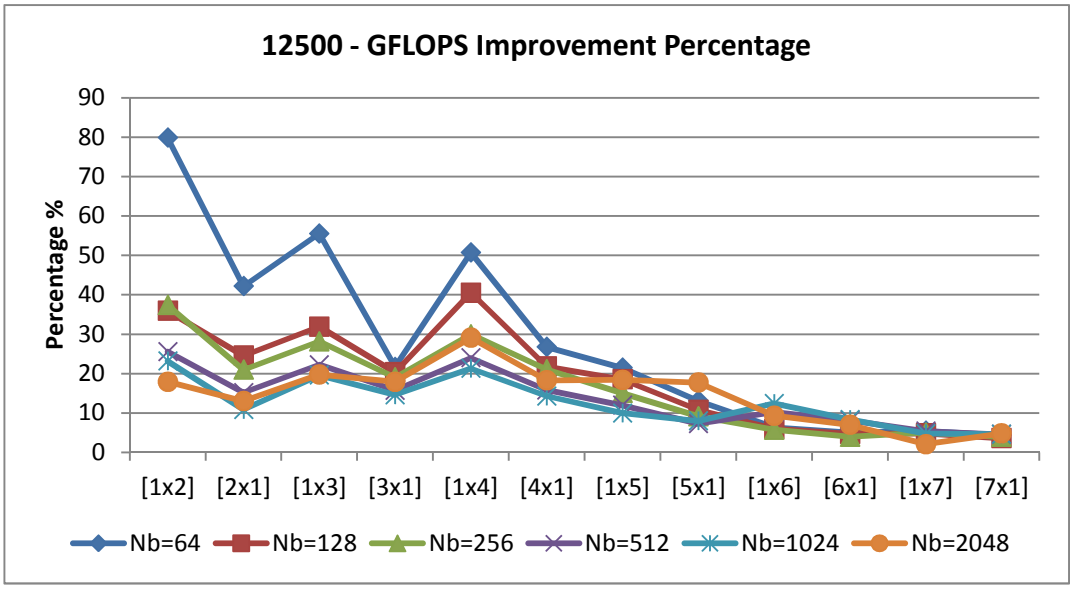


Figure A.46: GFLOPS Improvement of HPL with Problem Size 12500 – Eight Concurrency.

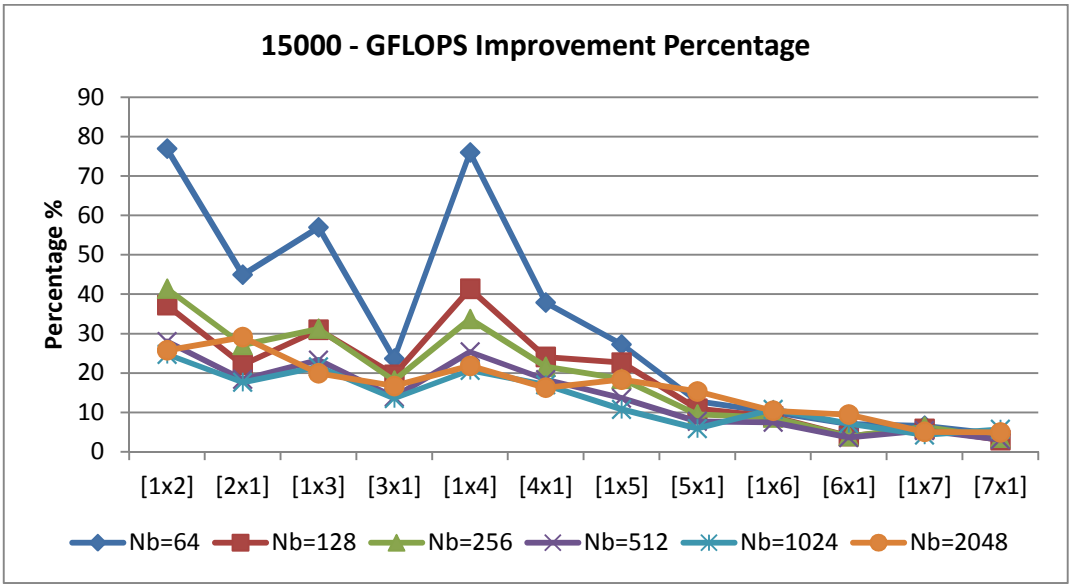


Figure A.47: GFLOPS Improvement of HPL with Problem Size 15000 – Eight Concurrency.

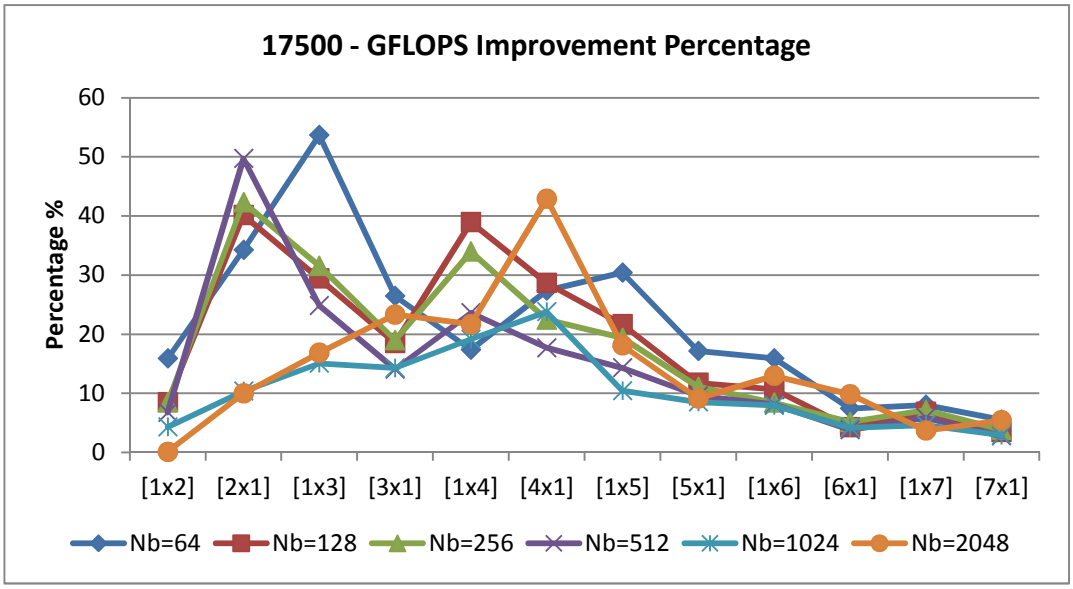


Figure A.48: GFLOPS Improvement of HPL with Problem Size 17500 – Eight Concurrency.

A.7 IMPROVEMENT PERCENTAGE OF GLOPS: SEVEN PROCESS CONCURRENCY

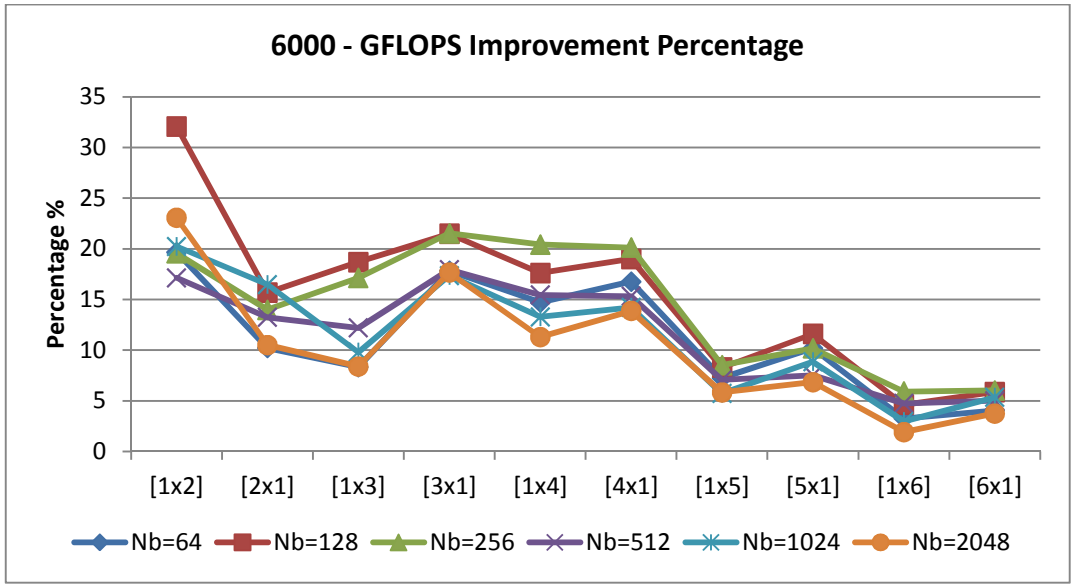


Figure A.49: GFLOPS Improvement of HPL with Problem Size 6000 – Seven Concurrency.

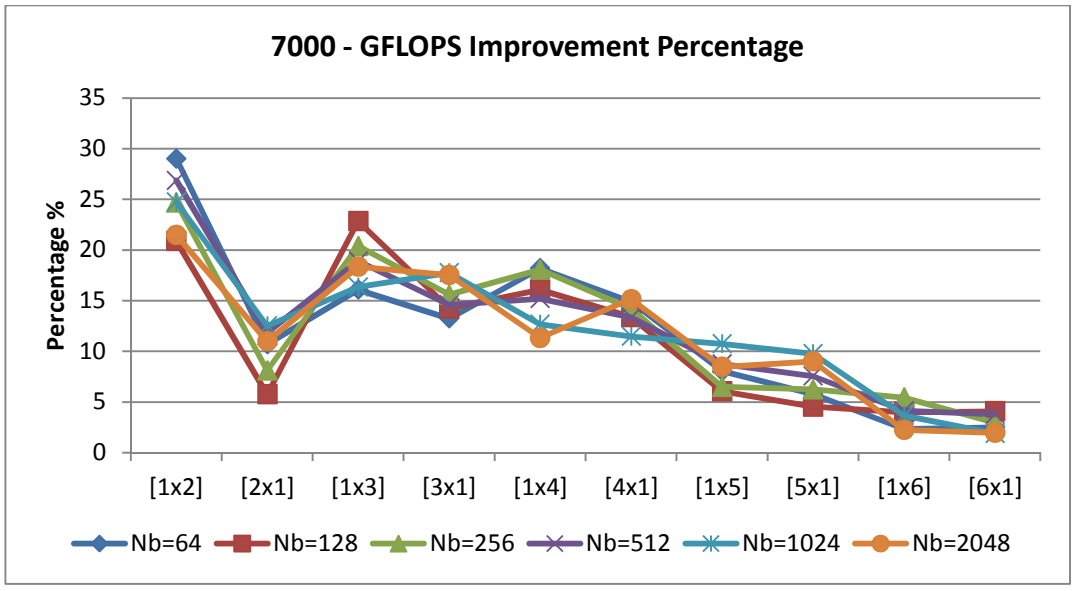


Figure A.50: GFLOPS Improvement of HPL with Problem Size 7000 – Seven Concurrency.

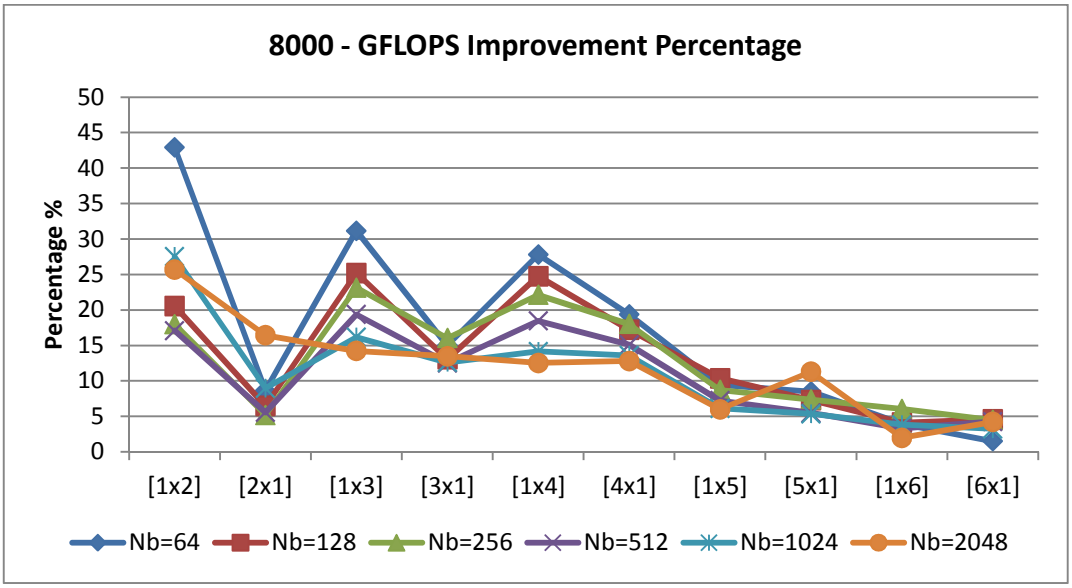


Figure A.51: GFLOPS Improvement of HPL with Problem Size 8000 – Seven Concurrency.

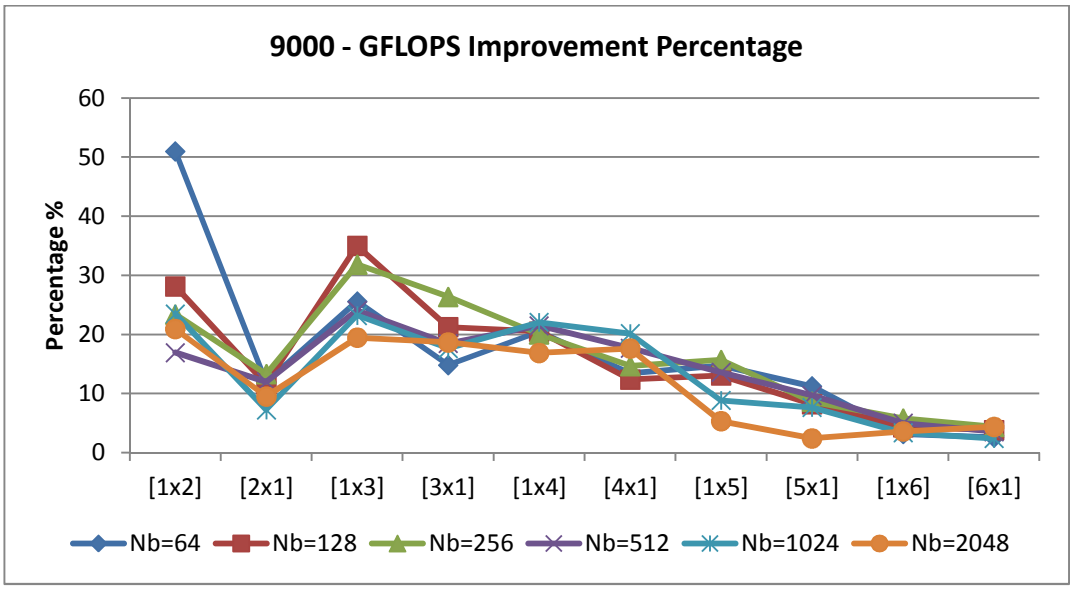


Figure A.52: GFLOPS Improvement of HPL with Problem Size 9000 – Seven Concurrency.

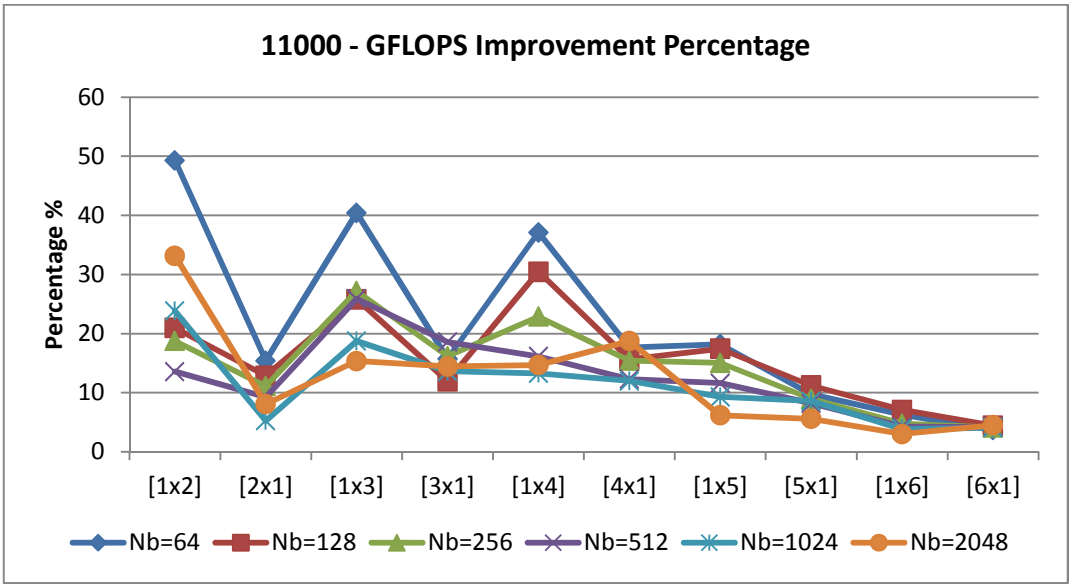


Figure A.53: GFLOPS Improvement of HPL with Problem Size 11000 – Seven Concurrency.

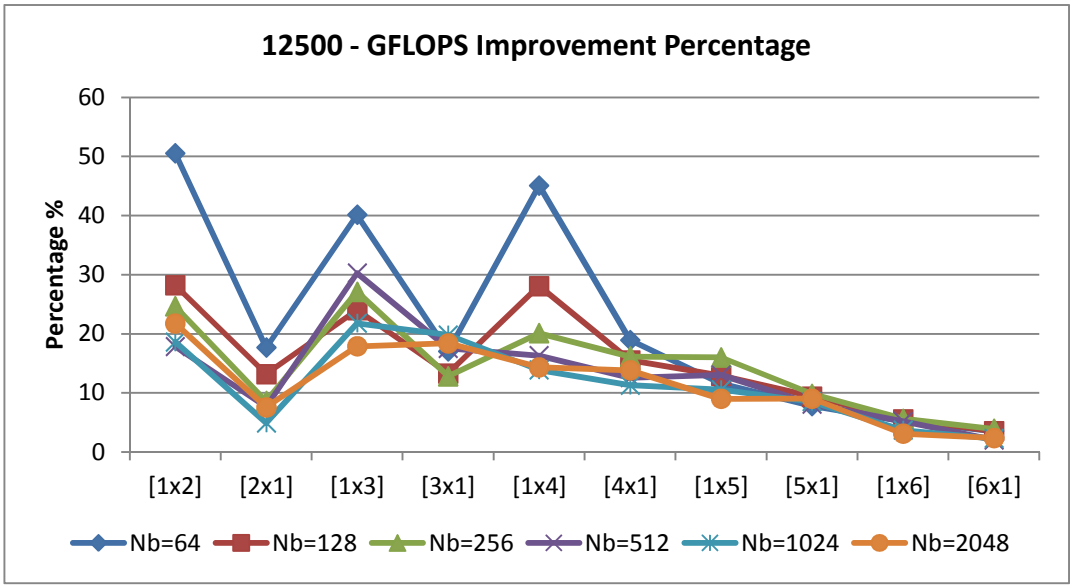


Figure A.54: GFLOPS Improvement of HPL with Problem Size 12500 – Seven Concurrency.

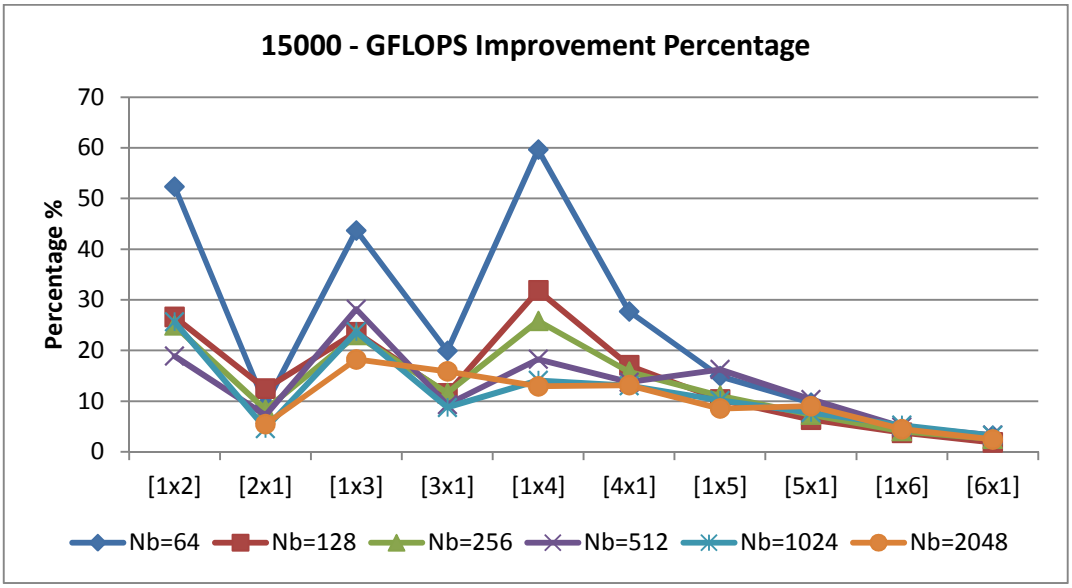


Figure A.55: GFLOPS Improvement of HPL with Problem Size 15000 – Seven Concurrency.

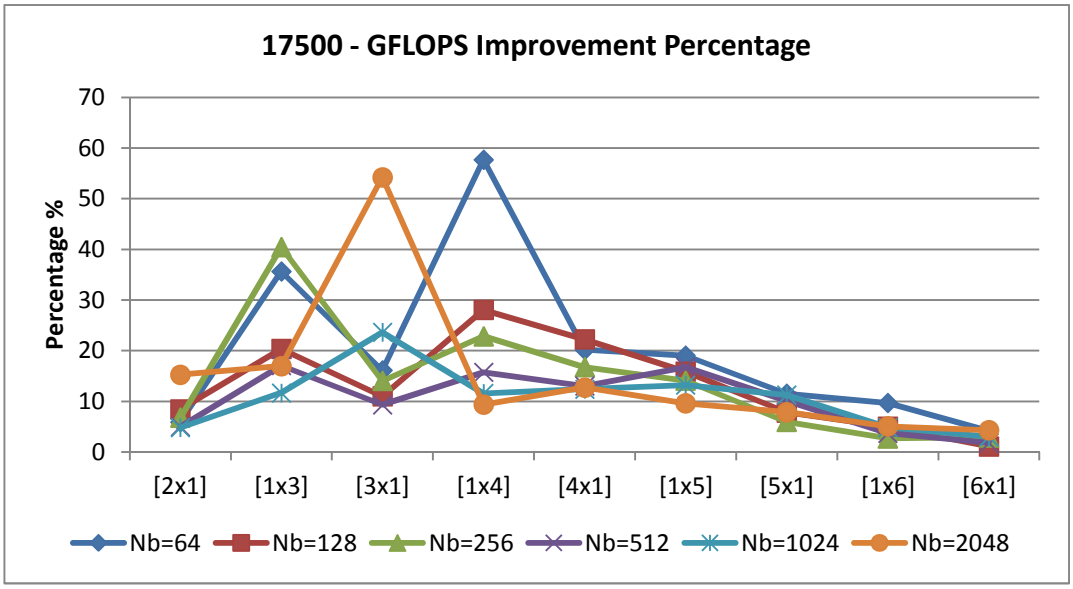


Figure A.56: GFLOPS Improvement of HPL with Problem Size 17500 – Seven Concurrency.

A.8 IMPROVEMENT PERCENTAGE OF GLOPS: SIX PROCESS CONCURRENCY

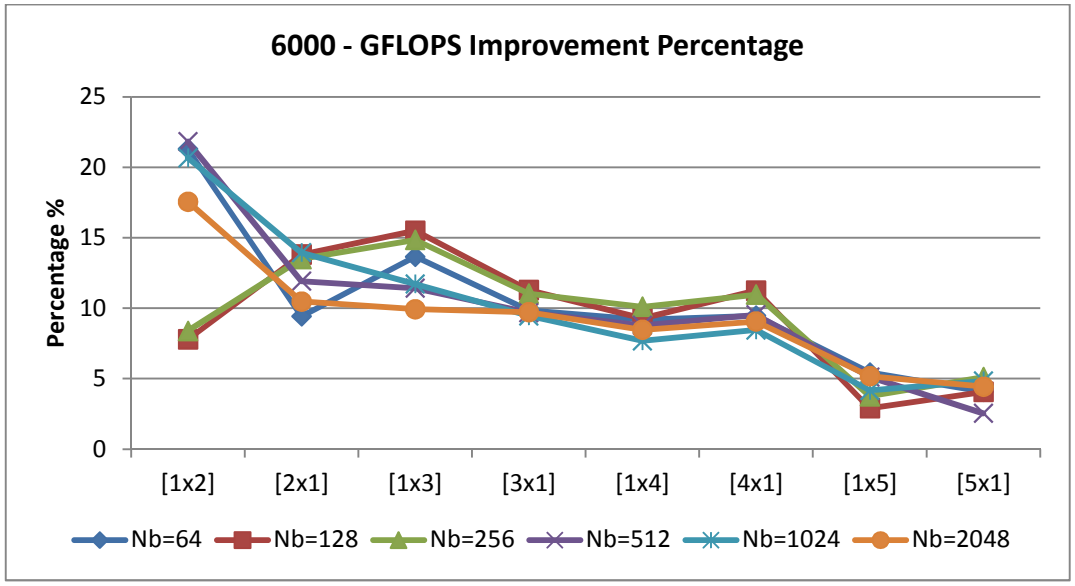


Figure A.57: GFLOPS Improvement of HPL with Problem Size 6000 – Six Concurrency.

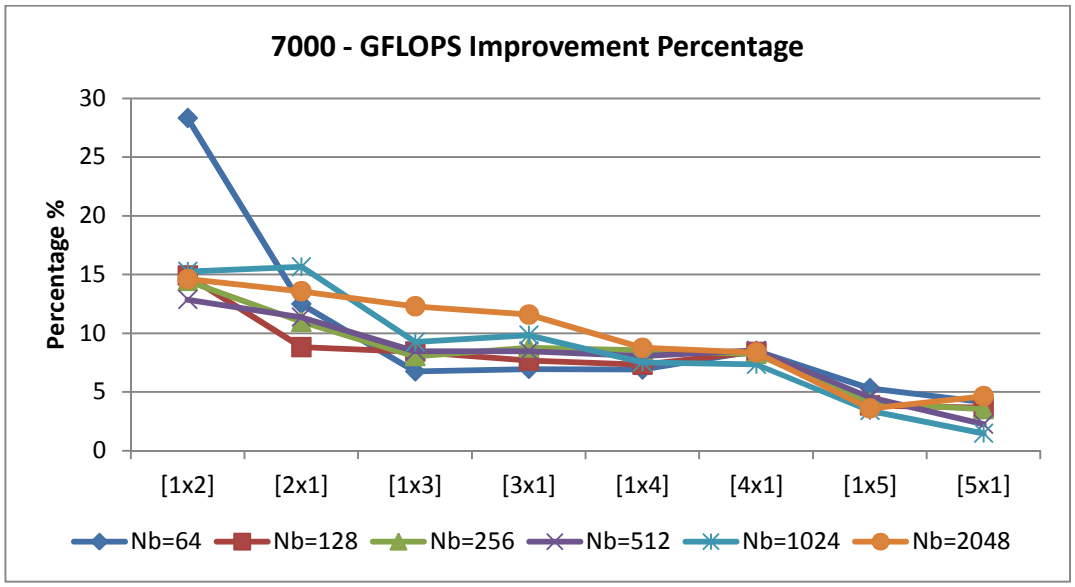


Figure A.58: GFLOPS Improvement of HPL with Problem Size 7000 – Six Concurrency.

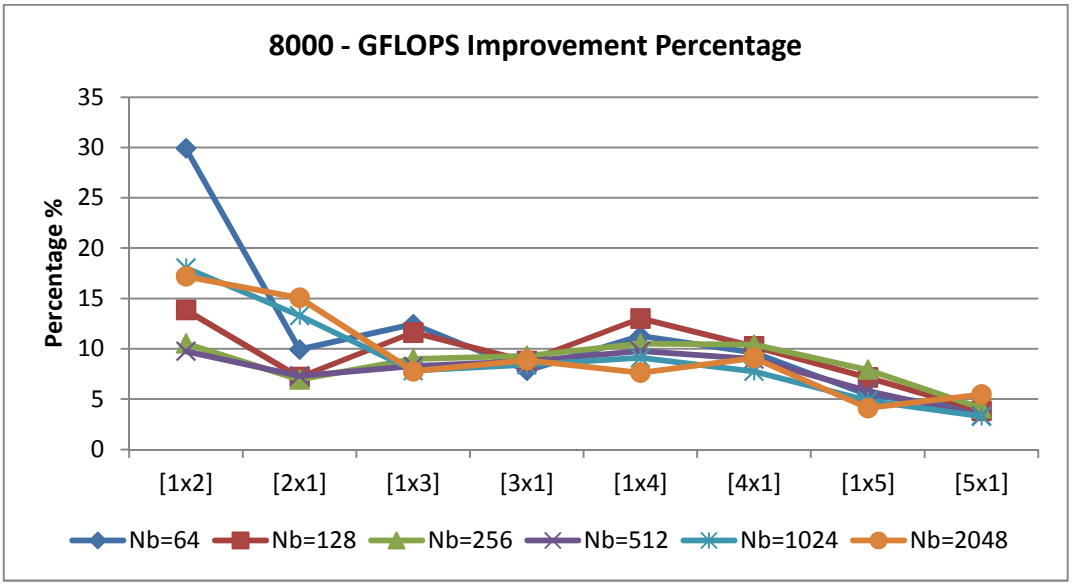


Figure A.59: GFLOPS Improvement of HPL with Problem Size 8000 – Six Concurrency.

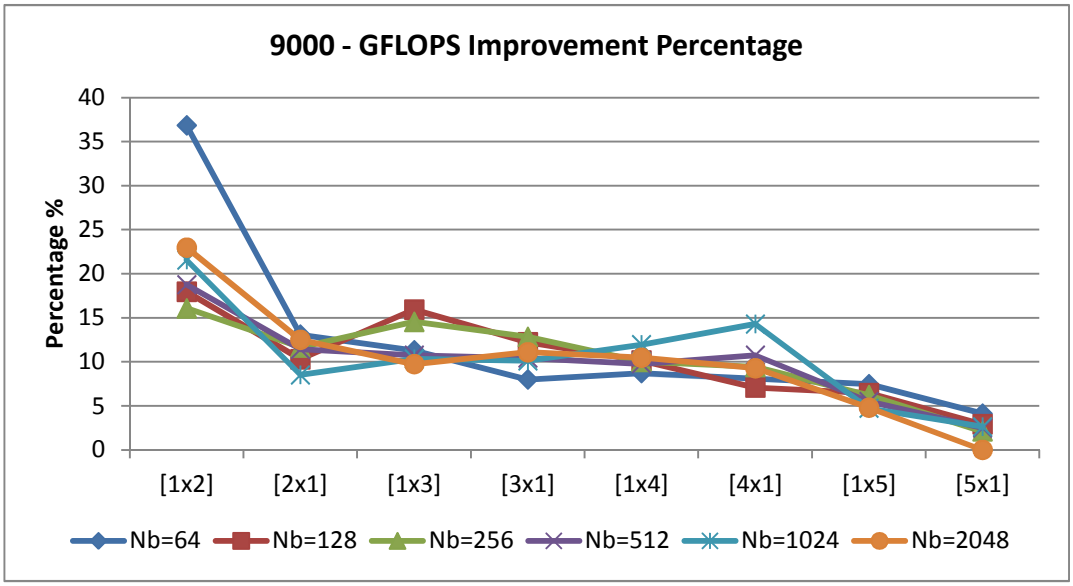


Figure A.60: GFLOPS Improvement of HPL with Problem Size 9000 – Six Concurrency.

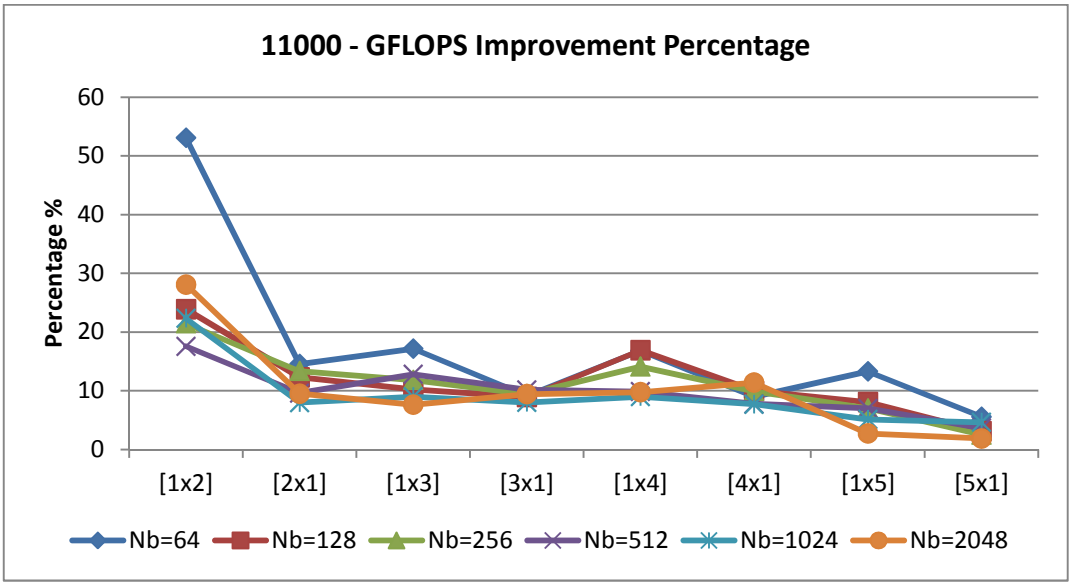


Figure A.61: GFLOPS Improvement of HPL with Problem Size 11000 – Six Concurrency.

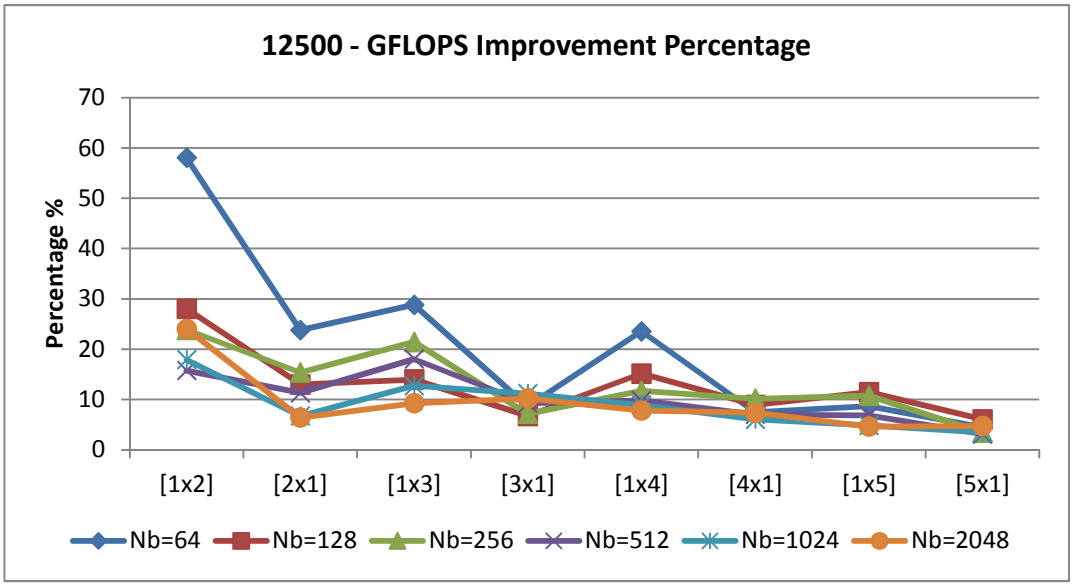


Figure A.62: GFLOPS Improvement of HPL with Problem Size 12500 – Six Concurrency.

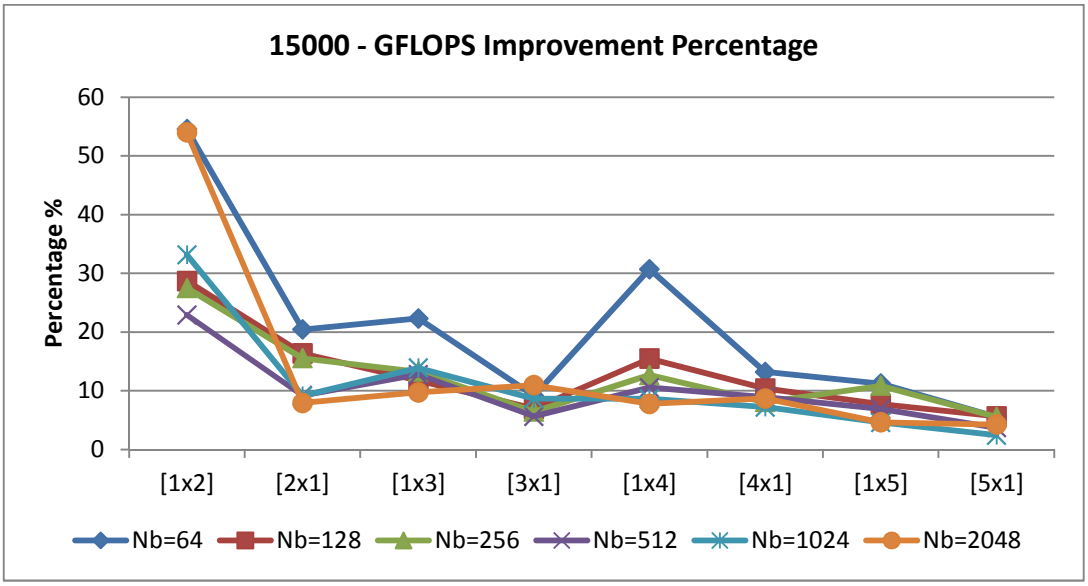


Figure A.63: GFLOPS Improvement of HPL with Problem Size 15000 – Six Concurrency.

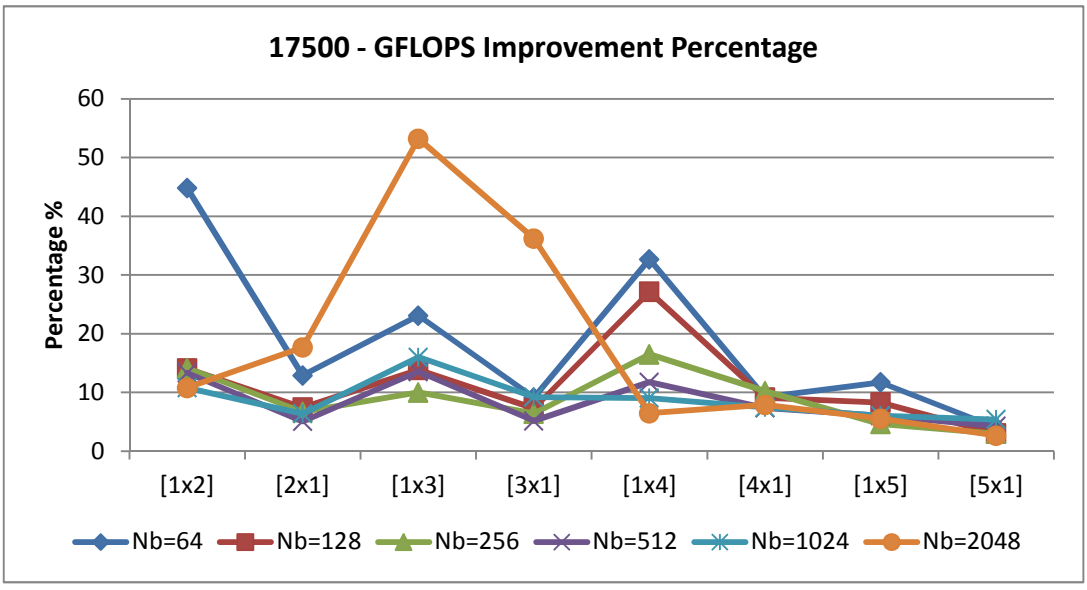


Figure A.64: GFLOPS Improvement of HPL with Problem Size 17500 – Six Concurrency.

A.9 IMPROVEMENT PERCENTAGE OF GLOPS: FIVE PROCESS CONCURRENCY

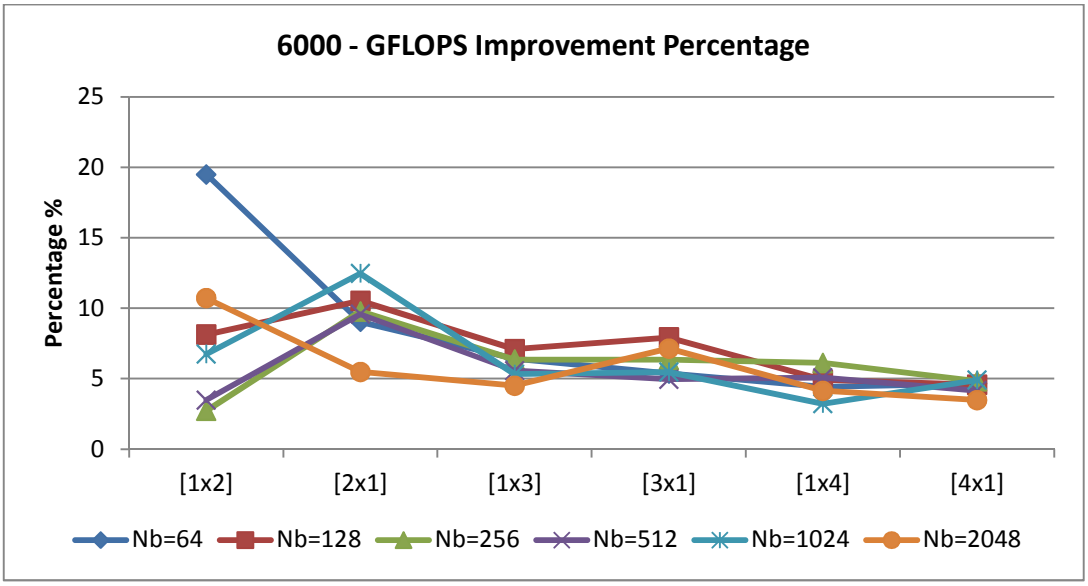


Figure A.65: GFLOPS Improvement of HPL with Problem Size 6000 – Five Concurrency.

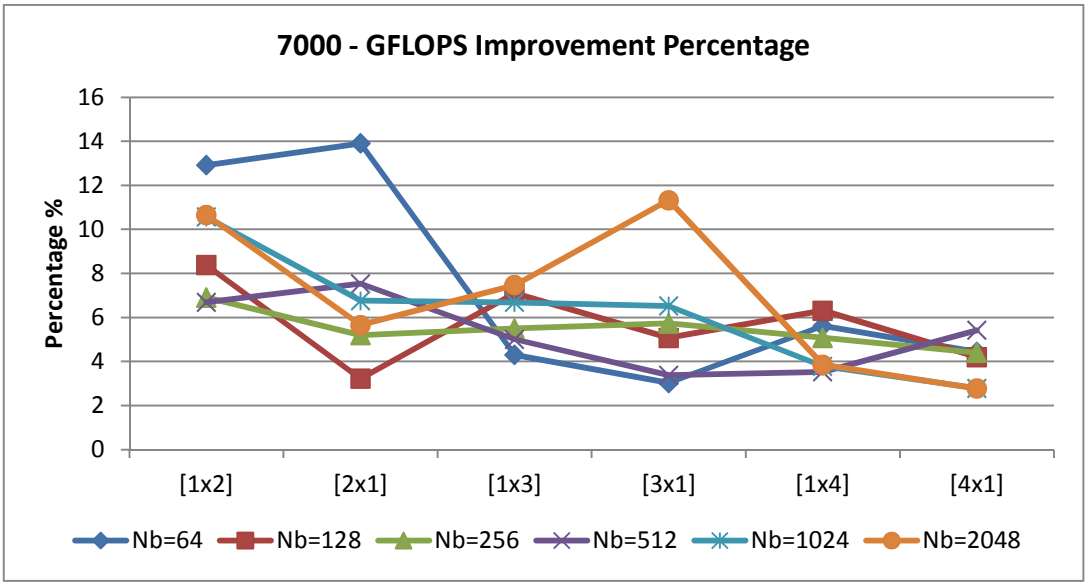


Figure A.66: GFLOPS Improvement of HPL with Problem Size 7000 – Five Concurrency.

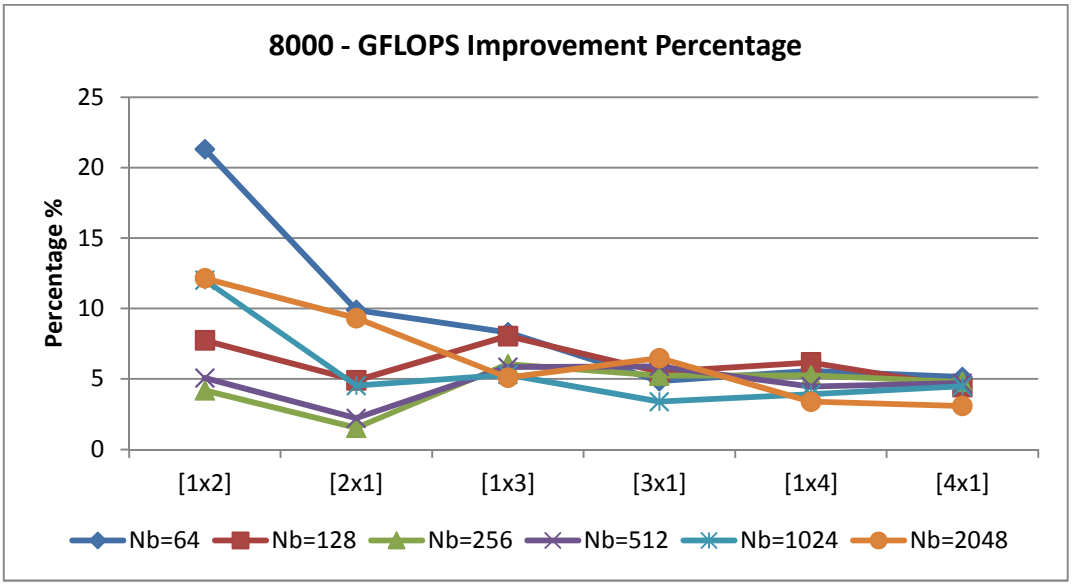


Figure A.67: GFLOPS Improvement of HPL with Problem Size 8000 – Five Concurrency.

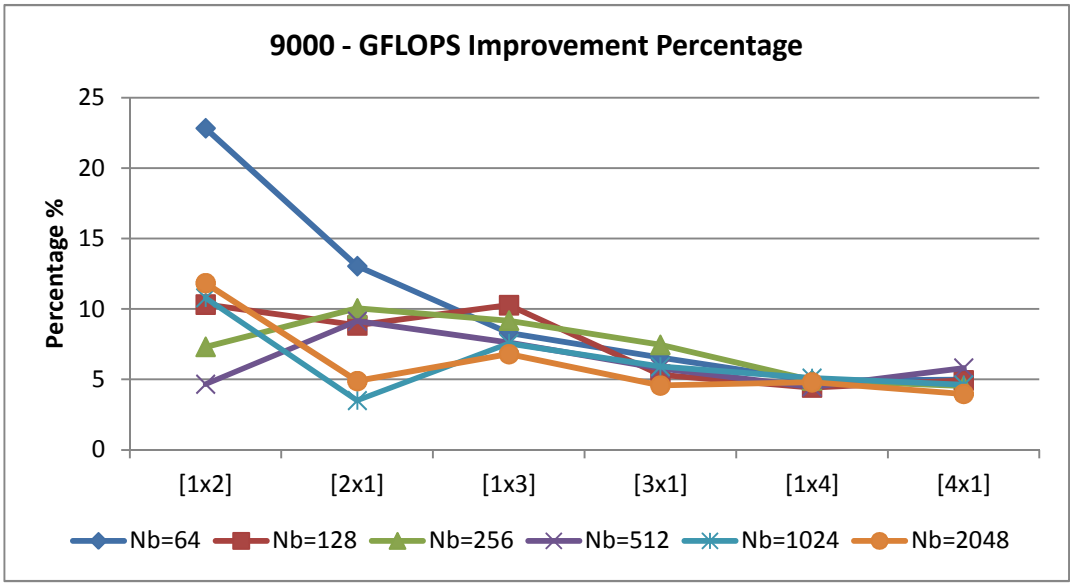


Figure A.68: GFLOPS Improvement of HPL with Problem Size 9000 – Five Concurrency.

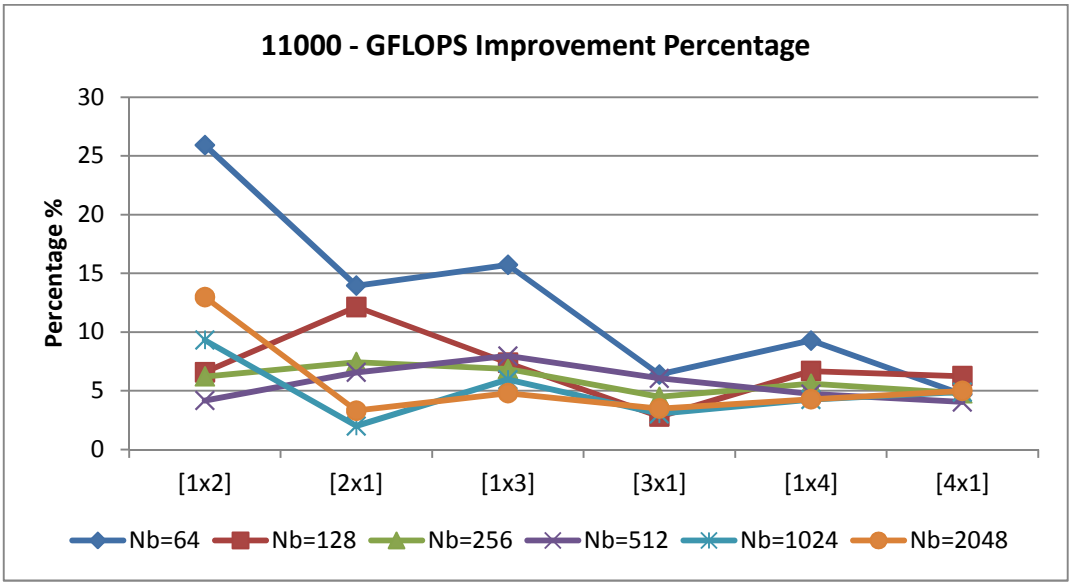


Figure A.69: GFLOPS Improvement of HPL with Problem Size 11000 – Five Concurrency.

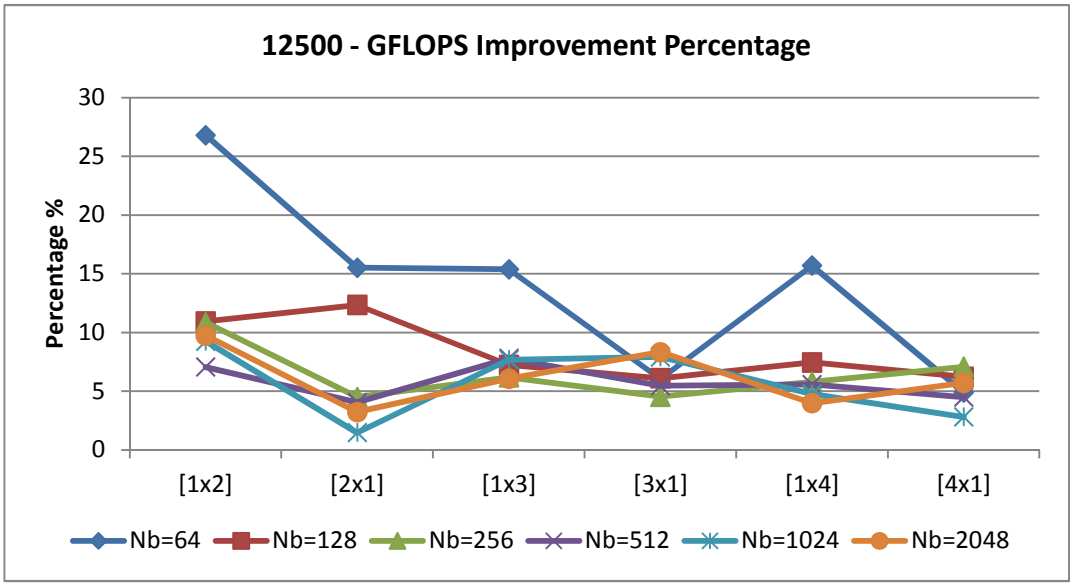


Figure A.70: GFLOPS Improvement of HPL with Problem Size 12500 – Five Concurrency.

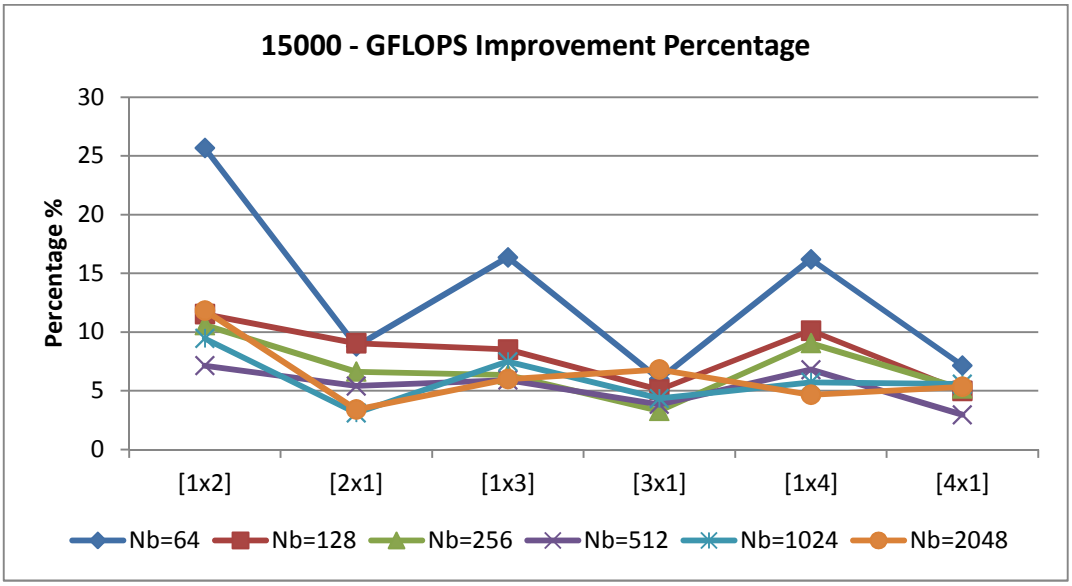


Figure A.71: GFLOPS Improvement of HPL with Problem Size 15000 – Five Concurrency.

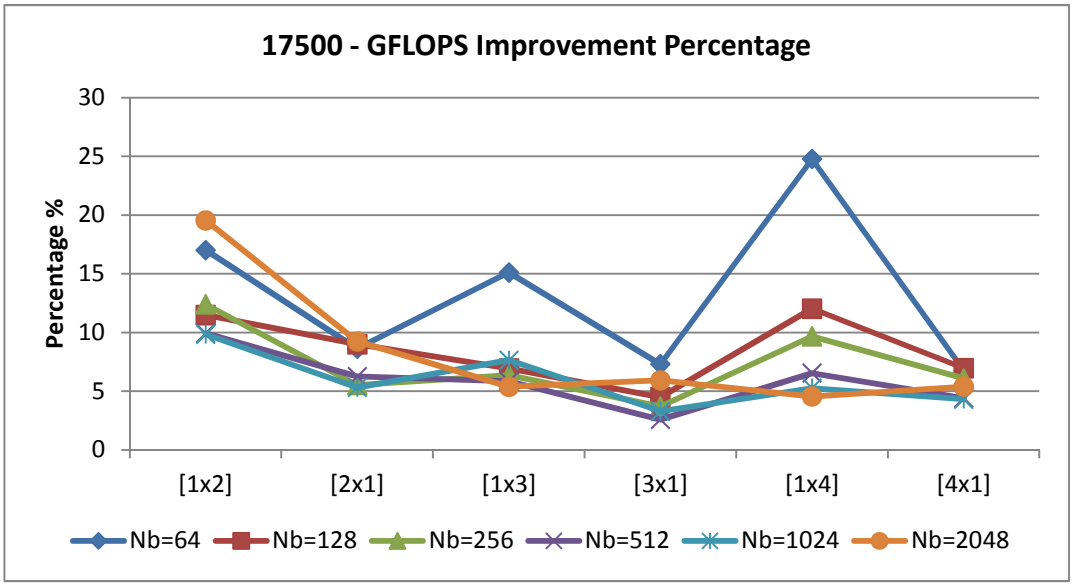


Figure A.72: GFLOPS Improvement of HPL with Problem Size 17500 – Five Concurrency.

A.10 IMPROVEMENT PERCENTAGE OF GLOPS: FOUR PROCESS CONCURRENCY

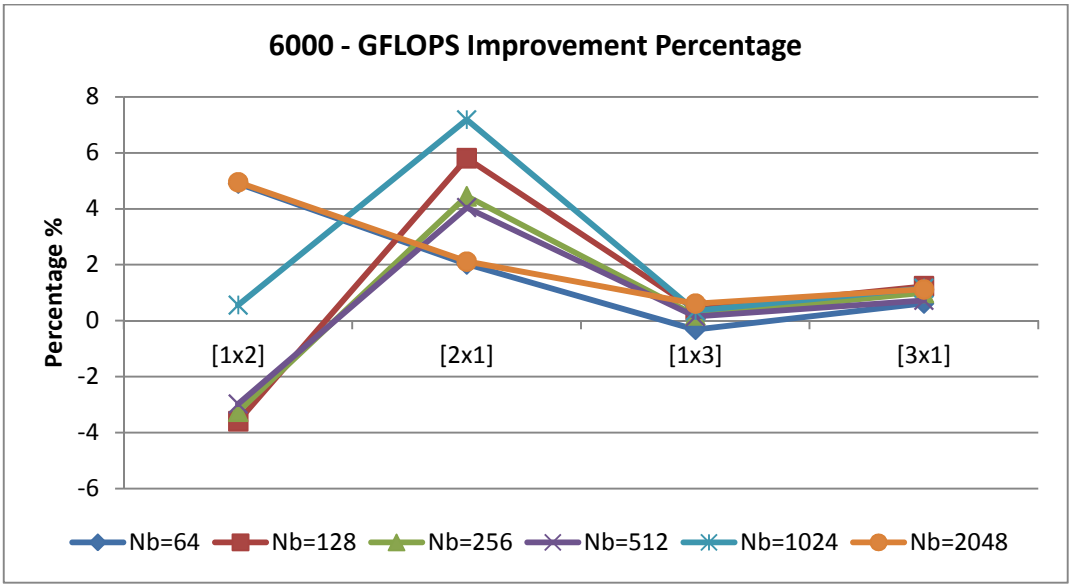


Figure A.73: GFLOPS Improvement of HPL with Problem Size 6000 – Four Concurrency.

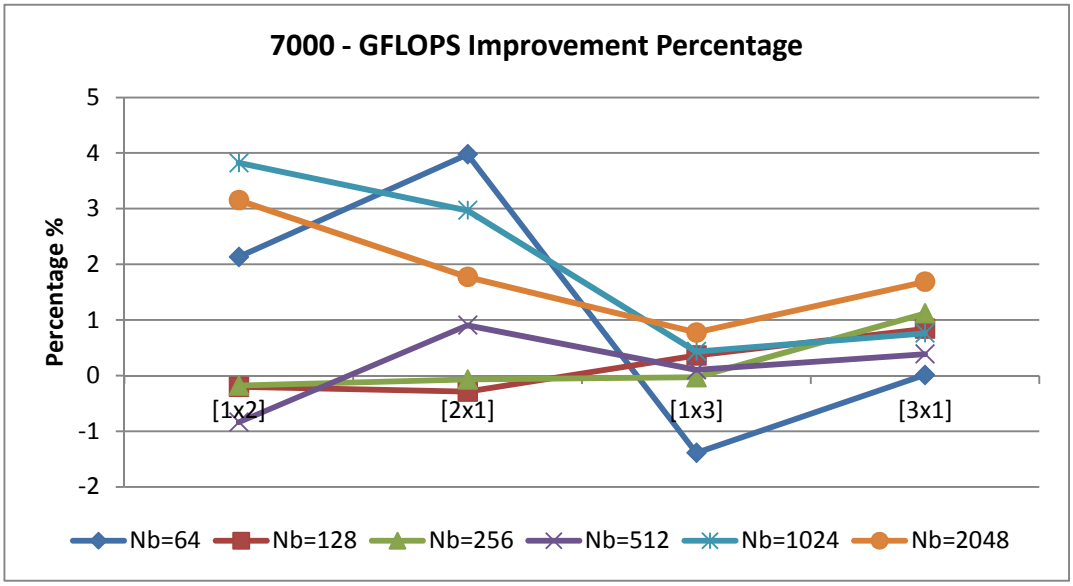


Figure A.74: GFLOPS Improvement of HPL with Problem Size 7000 – Four Concurrency.

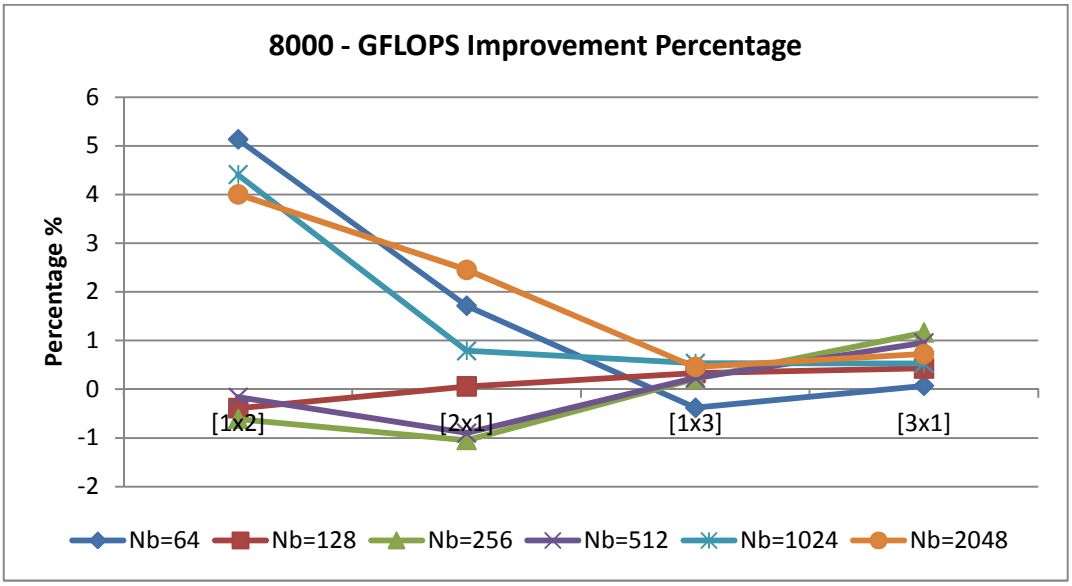


Figure A.75: GFLOPS Improvement of HPL with Problem Size 8000 – Four Concurrency.

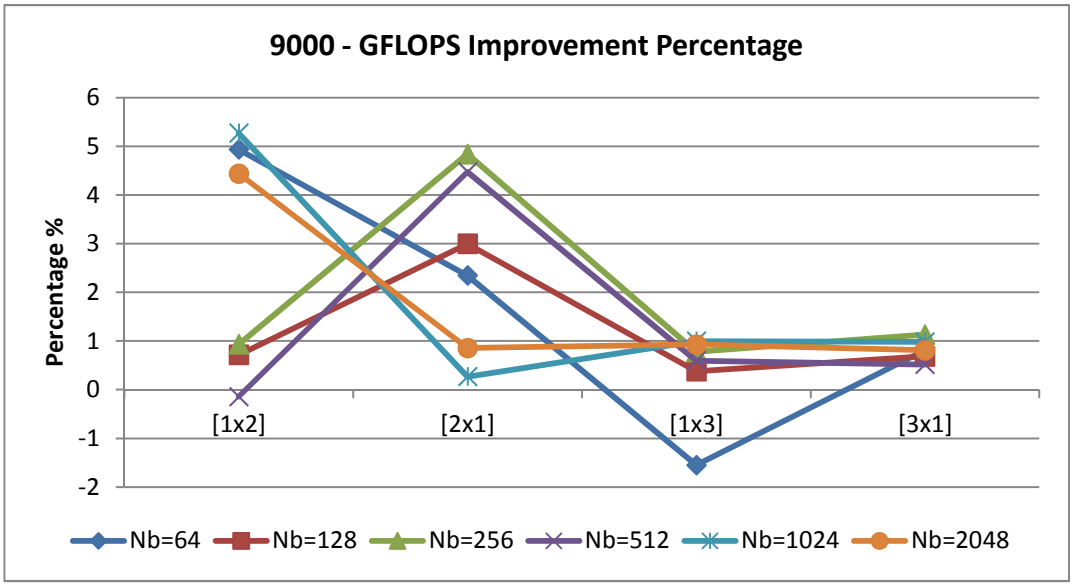


Figure A.76: GFLOPS Improvement of HPL with Problem Size 9000 – Four Concurrency.

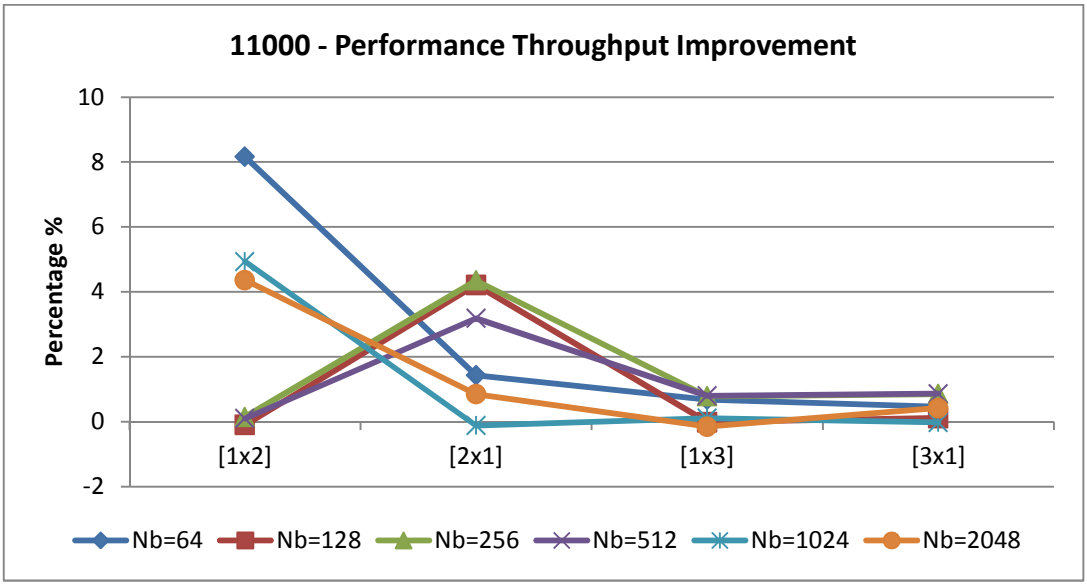


Figure A.77: GFLOPS Improvement of HPL with Problem Size 11000 – Four Concurrency.

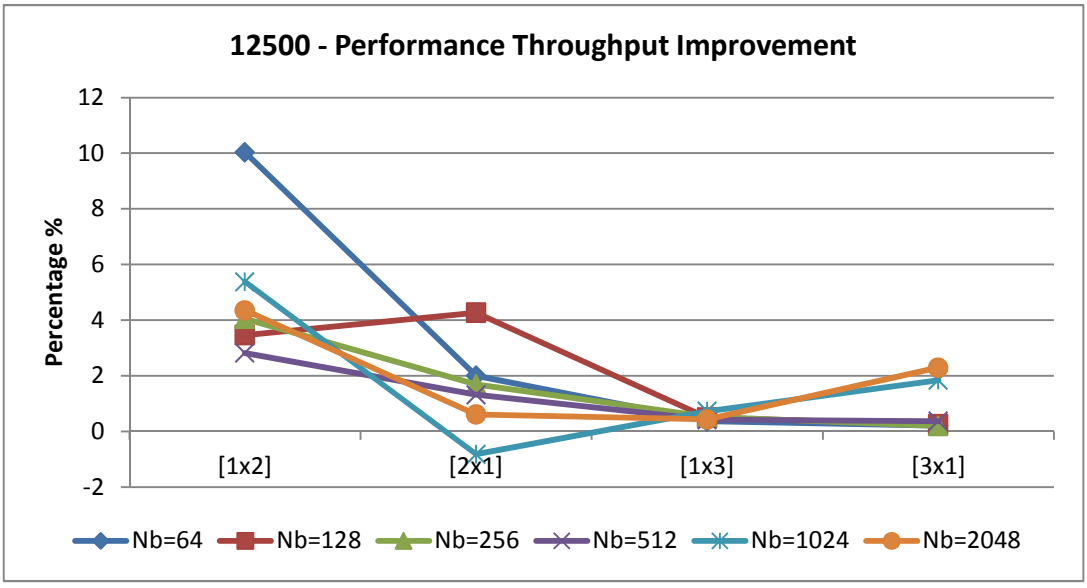


Figure A.78: GFLOPS Improvement of HPL with Problem Size 12500 – Four Concurrency.

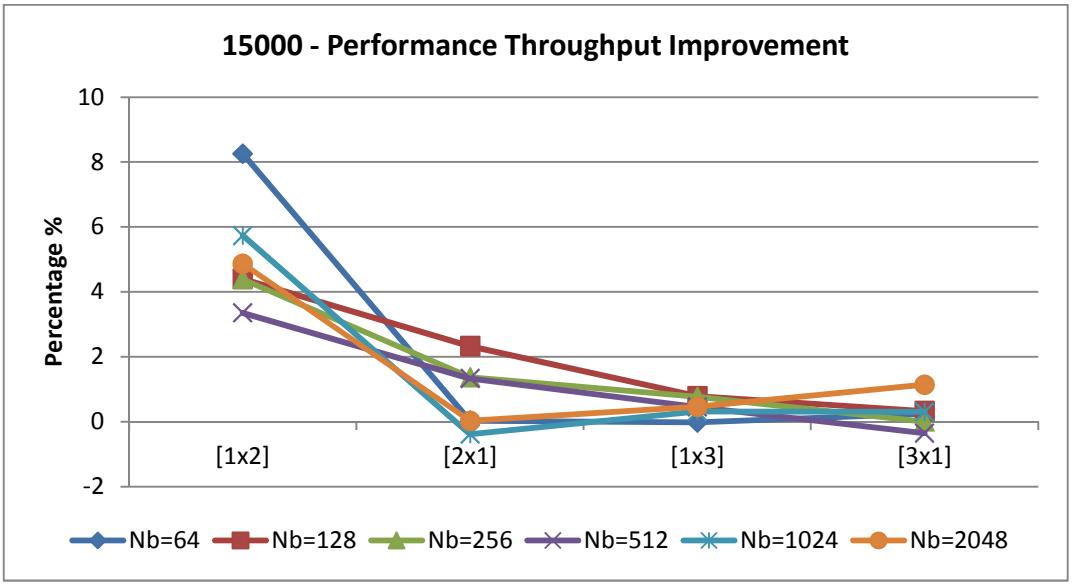


Figure A.79: GFLOPS Improvement of HPL with Problem Size 15000 – Four Concurrency.

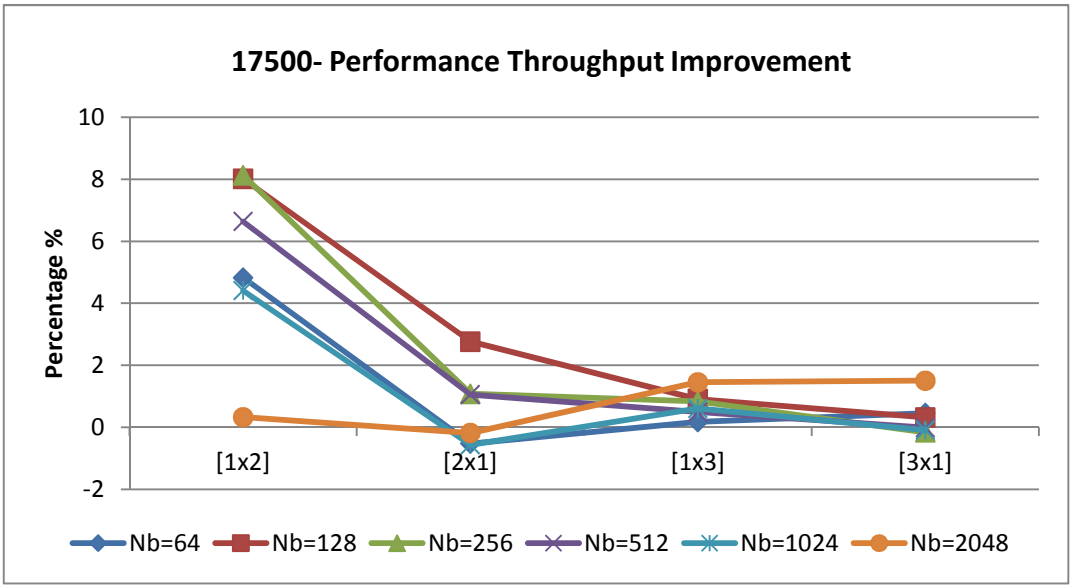


Figure A.80: GFLOPS Improvement of HPL with Problem Size 17500 – Four Concurrency.

Curriculum Vita

Damian Valles was born in Ciudad Juarez, Chihuahua, Mexico in April 16, 1979. His is the eldest of parents Eduardo and Adriana Valles. He earned his Bachelor of Engineering in Electrical Engineering from the University of Texas at El Paso in 2002. He received his Masters in Science degree in Computer Engineering in 2004 from the University of Texas at El Paso. He joined the doctoral program in 2005. Dr. Valles has been the recipient of various awards such as the Texas Instruments Foundation Scholarship Award and Supercomputing Educational Program Scholarship. He was also a recipient of stipend awards from the Distributed Computing Lab. While pursuing his degree, Dr. Valles owned his company Inter tech Reps with his father in the field of Surface Mount technology until 2007. He worked as an Assistant Instructor and System Administrator for the Electrical & Computer Engineering Department during the doctorate years. Dr. Valles has presented his research at international conferences such as 2007 Educational Poster Presentation at Supercomputing Conference and 2009 in Parallel and Distributed Processing Techniques and Applications Proceedings.

Permanent address: 11563 James Grant
El Paso, Texas 79936

This dissertation was typed by Damian Valles.