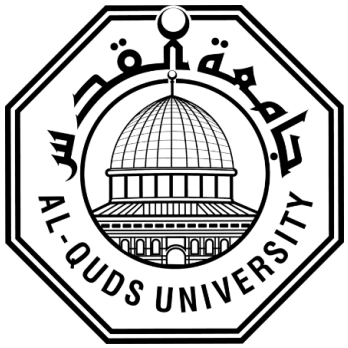
Deanship of Graduate Studies
Al-Quds University



Two Directional Parallel Mergesort Algorithm for Multi-core Computing (CPUs & GPUs)

Manal Ameen Dawood Abu Hijji

M.Sc Thesis

Jerusalem-Palestine

1438-2017

Two Directional Parallel Mergesort Algorithm for Multi-core Computing (CPUs & GPUs)

Prepared By:

Manal Ameen Dawood Abu Hijji

B.Sc.: Computer Engineering, AlQuds University, Palestine

Supervisor: Dr. Nidal Al-Kafri

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Electronic and Computer Engineering/ Department of Electronic and Computer Engineering/Faculty of Engineering/Graduate Studies

Jerusalem-Palestine

1438-2017

Al-Quds University
Deanship of Graduate Studies
Master of Electronics and Computer Engineering

Thesis Approval

Two Directional Parallel Mergesort Algorithm for Multi-core Computing (CPUs & GPUs)

Prepared By: Manal Ameen Dawood Abu Hijji

Registration No: 21112485

Supervisor: Dr. Nidal Al-Kafri

Master thesis submitted and accepted, Date: 10/01 /2017

The name and signatures of examining committee members are as follows:

1. Head of committee	**Dr. Nidal Al-Kafri**	**Signature:** .....
2. Internal Examiner	**Dr. Rushdi Hamamreh**	**Signature:** .....
3. External Examiner	**Dr. Ahmad Hasasneh**	**Signature:** .....

Jerusalem-Palestine

1438-2017

Dedication

I dedicate my thesis to my precious homeland Palestine, to my dearest parents, my lovely husband Tariq and my sweet kids Jannat and Hussein and my coming baby.

Declaration

I certify that this thesis submitted for the degree of Master, is the result of my own research, except where otherwise acknowledged, and that this study (or any part of the same) has not been submitted for a higher degree to any other university or institution.

Signed..... 

Manal Abu Hijji

Date: 15 /01 /2017

Acknowledgments

I thank God for everything.

For his support and guidance, I would like to thank my supervisors, Dr. Nidal Kafri and teachers in the program. Without their meticulous effort and support, this thesis would not have been the same.

I would also very much like to express my gratitude to all people in the Computer Engineering and Computer Science departments at Al-Quds University for their endless help during study.

In addition, I would like to dedicate special and great thanks to my family especially my parents, brother, sister. Also my deepest appreciation to Dr. Rushdi Kittani and his family for the endless care they have provided me, advices, support and patience.

Special thanks to my lovely husband for his endless support and care and to his parents and also I don't forget my lovely daughter Jannat and my son Hussein and my coming baby for their inspiration throughout my life.

Abstract

The sorting algorithms are arguably important in computer science and for information retrieval in vast number of applications. In the continuous increasing of information and the need for high performance sorting algorithms; parallel sorting is a reasonable approach for increasing the performance. Whilst the parallel processing was on super computers, grids, and clusters, the development and spread of modern computing architectures encourage researcher to customize existing algorithms or develop new algorithms to achieve higher performance on such architectures.

In this work we developed a new approach for parallel mergesort algorithm to be implemented on multi-core CPUs and many-core GPUs. We introduced a two directional merging process for this algorithm. Whilst the original parallel merge sorting, in the merging phase, fills the target array starting from the left to the end using one thread, the new proposed two directional mergesort algorithm fills the target array from left to the middle and from right to the middle of the target array. This process can reduce the merging time to the half time needed by the one directional one. Therefore, we recorded the elapsed time for accomplishing sorting data using sequential algorithm, one directional and two directional parallel mergesort algorithms. Furthermore, we computed the speed up of these parallel algorithms. The obtained results of the experiments show that it satisfies our goal in achieving higher performance over the existing one directional approach. In order to carry out these experiment, an application were developed using C programming language using supporting libraries for the communication between the processes (MPICH2). Also, we used supporting library (CUDA) for programming the GPU and OpenMP library that supports multithreading programming.

العنوان :خوارزمية الترتيب/التصنيف الدمجي المتوازية ثنائية الاتجاه (بالاتجاهين) على وحدات المعالجة المركزية متعددة النوى وعلى وحدات معالجة الرسومات

إعداد: **منال أمين داوود أبو حجة**

إشراف: **د. نضال الكفري**

ملخص:

تعتبر خوارزميات ترتيب المصفوفات من الخوارزميات الهامة في علم الحاسوب نظرا لكثرة استخدامها في العديد من التطبيقات ومنها قواعد البيانات وعملياتها ورسومات الحاسوب وأنظمة المحاكاة والعديد من العمليات الرياضية، ونظرا للقدر الهائل من المعلومات التي يتم التعامل معها والحاجة الماسة لترتيب هذه المعلومات وبسرعة عالية ظهرت خوارزميات الترتيب المتوازية والتي تساعد على تحسين الأداء بشكل ملحوظ.

بينما كانت عمليات المعالجة المتوازية على أجهزة الحاسوب الفائقة، والشبكات، والتجمعات، فان ظهور الحواسيب ذات البنية الحديثة شجعت الباحثون نحو تخصيص الخوارزميات الموجودة أو تطوير خوارزميات جديدة تعمل بشكل متوازي لتحقيق مستوى أعلى من الأداء في مثل هذه البنى.

تهدف هذه الدراسة إلى تطوير نهج جديد لخوارزمية "الترتيب عن طريق الدمج" "Merge Sort" المتوازية حيث تعتمد الخوارزمية الجديدة على مبدأ الدمج ثنائي الاتجاه (من اتجاهين) بخلاف الخوارزميات التي تعتمد على الدمج من اتجاه واحد وبالتالي تقليص الوقت المستغرق لتنفيذ عملية الدمج المتوازي بشكل فعال، وقد تم تنفيذها على وحدات المعالجة المركزية CPU ووحدات معالجة الرسومات GPU.

من خلال تطبيق هذه الخوارزمية على مجموعات كبيرة من المعلومات، أظهرت النتائج التي تم الحصول عليها فعالية هذه الخوارزمية في تحسين الأداء وتسريع عملية ترتيب المصفوفات عن طريق الدمج الثنائي الاتجاه والذي تفوق على نظيره أحادي الاتجاه بشكل واضح وملحوظ. لقد تم تطوير تطبيق لهذا الغرض باستخدام لغة البرمجة C واستخدام بعض الحزم البرمجية المساندة لعمليات الاتصال بين البرمجيات (MPICH2) ودعم البرمجة المتوازية لوحدات معالجة الصور (CUDA). حيث تم قياس الزمن اللازم لانجاز عمليات تصنيف وترتيب البيانات بواسطة الطريقة المقترحة ونظيرتها والزمن اللازم لانجاز هذه العمليات باستخدام المعالجة المتسلسلة. كما تم حساب

مدى التسريع الناتج. وهو معيار أساسي لقياس فعالية الخوارزميات المتوازية ويحسب بقسمة الزمن اللازم لانجاز العمليات بواسطة الخوارزميات المتسلسلة على الزمن اللازم للانجاز بواسطة الخوارزمية المتوازية.

ويمكن مستقبلا تطوير هذه الخوارزمية الجديدة باستخدام مجموعة من أجهزة الحاسوب متعددة المعالجات والتي تحتوي في نفس الوقت على وحدات معالجة الصور.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Abbreviation	Full word
M.Sc.	Master Degree
SIMD	Single Instruction Multiple Data
MIMD	Multiple Instruction Multiple Data
SPMD	Single Program Multiple Data
SIMT	Single Instruction Multiple Thread
PRAM	Parallel Random Access Machine
MPI	Message Passing Interface
FLOP	Floating-Point Operation
ILP	Instruction-Level Parallelism
GPGPU	general purpose Graphics Processing Units
GPUs	Graphics Processing Units
CPU	Central Processing Unit
CUDA	Computer Unified Device Architecture
SM	Streaming Multiprocessors
SP	Streaming Processor
DRAM	Dynamic Random Access Memory
TLP	Thread Level Parallelism
2DPMS-CPU	Two Directional Parallel Mergesort on CPUs
1DPMS-CPU	One Directional Parallel Mergesort on CPUs
2DPMS-GPU	Two Directional Parallel Mergesort on GPUs
1DPMS-GPU	One Directional Parallel Mergesort on GPUs

List of Appendices

Chapter 1

Introduction

1.1. Background and Motivation

Sorting is arguably one of the most studied problem in computer science due to its high importance for data analysis in several application domains, sorting is used for sorting data and for database operations such as creation of indices and binary searches, it helps in statistics like finding closest pair, determining an element's uniqueness, finding k^{th} largest element, and identifying outliers. Sorting is used to find the convex hull, and it is an important algorithm used in computational geometry. Other applications that use sorting include computer graphics, computational biology, supply chain management and data compression, N-body simulations, some high performance sparse matrix-vector multiplication implementations and machine learning algorithms (Chhugani, et al., 2008) (Leischner, Osipov, & Sanders, 2009) (Tanasic, et al., 2013).

1.2. Problem Statement

In general, the emerging modern and relatively inexpensive multi-core technologies encourage researchers to move the implementation of the existing sequential algorithms to parallel processing on single and cluster of these systems. Specifically, the emerging growth of information and the need for sorting them based on a given key increased the need for high performance sorting algorithms on available platforms. Thus, these modern computational systems attract researchers to develop parallel sorting algorithms on single and clusters of multi-core and recently on many-cores machines. Therefore; in order to achieve higher performance, some of these algorithms require further improvements to optimize the utilization of the available computing capacity of these systems i.e., the available processors/computers and memory. Since, many existing applications and does not benefit from the multi-core CPUs, the available networked computers, and the installed graphical processing units (GPUs).

1.3. Related Work

In the past decades, many works were carried out to improve and customize the classical sorting algorithms to work on newly developed platforms. For parallel mergesort algorithm on CPUs; there were several approaches have been proposed and developed. For instance, in (Rashid & Qureshi, 2006), (Trimananda & Haryanto, 2-3 Aug. 2010) and (El-Nashar, May 2011), the parallel mergesort is implemented and evaluated on multi-core CPUs. The list is divided into 2 equally sized lists and the generated sub-lists are further divided until each number is obtained individually, the sub-lists are then merged until the whole list is constructed. The algorithm is parallelized by distributing n/p elements (where n is the list size and p is the number of processors) to each slave processor. In (Rashid & Qureshi, 2006) and (El-Nashar, May 2011), the slave can sequentially sort the sub-list using sequential mergesort and then return the sorted sub-list to the master, where the master is responsible of merging all sorted sub-lists into one sorted list, where in (El-Nashar, May 2011) the sorted sub-lists are distributed among dual-cores processors using MPI platform. The authors in (Trimananda & Haryanto, 2-3 Aug. 2010) sort the sub-lists with parallel quick-sort using multi-threading on each processor, and later merged in parallel by using mergesort algorithm with MPICH platform.

On the other hand, In (Radenki A., 2011), the article introduces three parallel versions of recursive mergesort: the first is shared memory (with OpenMP) on mainstream SMP-based systems, such as multi-core computers and multi-core clusters, the second is message-passing that runs on computer clusters (with MPI) and the third is a hybrid (with MPI and OpenMP) that runs on clustered SMP.

In (Chhugani, et al., 2008), the mergesort is parallelized on multi-core CPUs using the multi-way merge, the dataset is divided into chunks (or blocks), where each block can reside in the cache memory, sorting each block separately, and then merge them in the usual fashion, each block of data is divided amongst the available threads (or processors). Each thread sorts the data assigned to it using merging networks like odd-even mergesort.

For mergesort algorithm works on GPUs; (Satish, Harris, & Garland, 23-29 May 2009) describes the design of an efficient sorting algorithms for such many-core GPUs using CUDA, it introduces the radix sort that directly manipulates the binary representation of keys, and the mergesort which requires only a comparison function on keys.

For mergesort, data is partitioned into equally sized blocks that will be sorted with parallel Batcher's odd-even mergesort instead of bitonic sort that proposed in the previous work for the same work that published in 2008 on the shared memory of GPU (Satish, Harris, & Garland, 2008). After that the sorted blocks are merged using the proposed mergesort.
The key to the mergesort algorithm is to compute the ranks r1 and r2 of an element e in the even and the odd sorted blocks. Then the even block is divided into its first r1 elements A and the remainder B. Also, the odd block is divided into its first r2 elements C and remainder D. The sub-block pairs A:C and B:D are independently merged and all sub-lists are merged in parallel.

(Dominik Zurek, 2013) describes the results from the implementation of a few different parallel sorting algorithms on GPU cards and multi-core processors CPUs and a hybrid algorithm is executed on both platforms (a standard CPU and GPU).

On GPU, sorting algorithm is based on the bitonic sort that merge elements until all input data in the global memory is sorted, where on CPU; the data is divided into equal parts for each core and then sorted in parallel with the quick sort with OpenMP, and then the results of each core are merged by an efficient merge-sort algorithm.
In the hybrid, the algorithm consists of two main steps; the first one executed on a GPU and the second on a CPU, data is transferred from CPU to GPU and blocks of data are sorted in parallel using the bitonic sort, then the partially sorted data is transferred back to CPU to be merged in parallel.

In (Peters, Schulz-Hildebrandt, & Luttenberger, 19-23 April 2010), the paper presents a novel merge-based external sorting algorithm for one or more CUDA-enabled GPUs using an approach similar to sample sort and by overlapping memory transfers with GPU computation. In the first phase an input sequence is divided into shorter subsequences, each of them is transferred to the GPU, sorted using a GPU-sorting algorithm (internal sort), and then transferred back.
After the first phase is completed, the CPU pre-computes the data sets for the internal merge runs that form the k-way merging process. In the next phase, single sets are transferred to the GPU, merged on the GPU internally and transferred back. The concatenation of the output of these merge runs is a sorted sequence.

In (Ye, Fan, & Lin, 2010), it presents an efficient comparison-based sorting algorithm for CUDA-enabled GPUs, it is composed of two parts: firstly, the data is divided into equal

sizes subsequences and sorted by using the bitonic network, and then a mergesort follows to merge all small subsequences into the final result, the algorithm takes advantage of the synchronous execution of threads in a warp, and organizes the bitonic sort and mergesort using one warp as a unit. Thus, it is called GPU-WarpSort. When merging two subsequences, the warp fetches t/2 elements from sequence A or B and stores them into the buffer in shared memory. It uses a barrier-free bitonic merge network to sort them and outputs the lower half part. The others are sorted with other t elements from either sequence A or B.

In (Tanasic, et al., 2013), it presents a high performance mergesort algorithm on multi-GPU. The mergesort algorithm first sorts the data on each GPU using an existing single GPU sorting algorithm. Then, a series of merge steps produce a globally sorted array distributed across all the GPUs in the system using a novel pivot selection algorithm that ensures that merge steps always distribute data evenly among all GPUs.

For a single-GPU sorting, the paper uses an implementation based on the parallel mergesort described in (Satish, Harris, & Garland, 23-29 May 2009) and (Hagerup & Rüb, 1989). Merging the data distributed among GPUs depends on a pivot point in one sorted array and its 'mirrored' in the other array that partition the input arrays into two parts lower and upper, then merging lower parts and merging upper parts will result in two sorted arrays when concatenated.

1.4. Contribution

The goal of our work is to satisfy the following question: how can we improve the performance of the mergesort on the modern multi-core and many-core computational devices (CPUs and GPUs)? In other words, the aim of this work is two folds. The first is to improve the performance of parallel mergesort algorithm implementation on multi-core technologies. We achieved this goal by utilizing the hybrid approach on multi-core Central Processing Units (CPUs). This work is carried out using Massage Passing (Distributed Memory) with Message Passing Interface MPI and Shared Memory programming paradigms and with multi-threading approaches on CPUs. The second goal is to improve the performance of this algorithm on GPU using shared memory paradigm with Single Instruction Multiple Threads SIMT. The achievement of better performance, were realized by using our two ways/directional merging technique of sorted sub-lists instead of the regular one directional merge. Our algorithm can be used to sort any type of data after

preprocessing such as using indexing technique and use the indices as input elements for sorting algorithm.

In this work we followed/adapted the formal research methodology, experimental and process methodologies proposed in (Elio, 2011) .

1.5. Thesis Organization

This thesis is organized as follows: in the next chapter we introduce a background regarding to the parallel processing platforms and their supporting APIs that utilized in our work/experiments. Chapter three presents literature review and background on sequential/parallel mergesort. Chapter four introduces our new approach for parallel mergesort. In chapter five we present the experiments, results and discussion/analysis. Finally, the conclusion and future works are presented in Chapter 6.

Chapter 2

Background and Related Work

In this chapter we present the main parallel processing hardware architecture for message passing and shared memory paradigms. We will focus on multi-core CPUs and the relatively modern Graphical Processing Units GPUs and their supporting software libraries and Application Programming Interfaces APIs.

Many computing fields were evolved by using the parallel processing in order to increase performance and decrease computing elapsed time, such applications are multimedia computing, signal processing, scientific computing, engineering, general purpose application, industry, computer systems, statistical applications, and simulations.

In parallel processing, mainframes and supercomputers are used to implement shared memory parallel computing, but these architectures are expensive. On the other hand, inexpensive clusters of on-shelf computers and grid computing attract researchers to be utilized to speed up the computation-using message passing paradigm.

2.1. Introduction

In the ’70s and ’80s, parallel algorithms have focused on fine-grain models such as Random Access Machine (PRAM) or network-based models where the ratio of memory to processors is relatively small and the processor-to-processor communication is the most important bottlenecking parallel computing. Thus, efficient parallel algorithms are more likely to be achieved on coarse-grain parallel systems and in most situations algorithms

originally developed for parallel PRAM-based models are substantially redesigned (Ionescu, 1997).

Computer performance trends had changed by different rates over time. Fig 2.1 illustrates the growth in processor performance over time for several classes of computers of micro, mini, mainframe and supercomputer processors. Performance of microprocessors has been increasing at a rate of nearly 50% per year since the mid 80's. More traditional mainframe and supercomputer performance have been increasing at a rate of roughly 25% per year. As a result, we are seeing the processor that is best suited to parallel architecture become the performance leader as well (Culler, Gupta, & Singh, 1997).

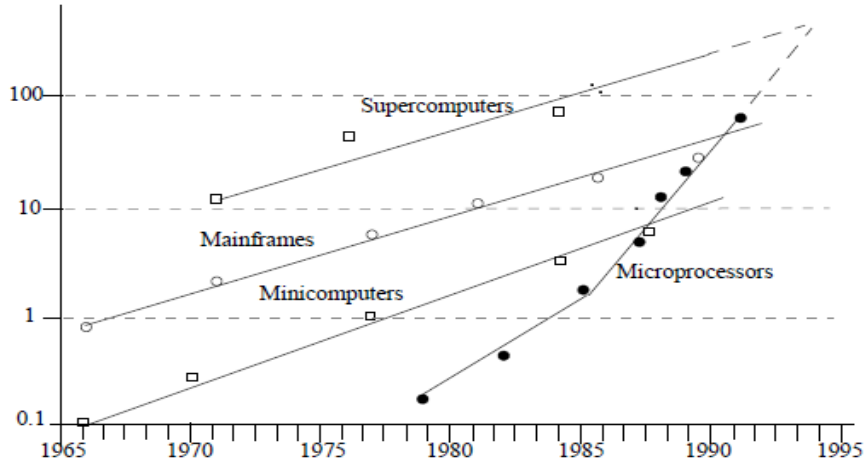


Figure 2.1: Performance trends over time of micro, mini, mainframe and supercomputer processors.

Obviously, parallel processing is implemented on shared memory computer architectures using Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), Single Program Multiple Data (SPMD) Techniques, or multi-threading. Whilst message passing paradigm can be used on distributed memory architectures by means of SPMD and MIMD, a hybrid approach using both paradigms can also be implemented on both architectures (Khalilieh, Kafri, & Mohammad, 2014) (Culler, Singh, & Gupta, 1997).

Modern multi-core CPUs with continuous increasing number of cores per system can be used to implement parallel algorithms/tasks using shared memory. For more efficiency, hybrid of shared/distributed memory and cluster of multi-core systems can be employed using message passing interface MPI and shared memory with multi-threading (OpenMP) paradigms (Khalilieh, Kafri, & Mohammad, 2014) (El-Nashar A. I., 2011) .

On the other hand, the modern and promising many-core computer architecture (GPUs) attracts the researchers to utilize the SIMD architecture as an excellent solution to gain high performance computation for many problems and applications. Therefore, this architecture shifted the interest of many researchers towered parallel computing. Parallel applications usually have the problem that requires an amount of communications as well as computations, where converting the theoretical model to an efficient implementation is not straightforward (Ionescu, 1997).

Parallel Hardware (CPUs and GPUs)

Next in this chapter we present the main architectures and features/characteristics of parallel programming platforms. We will focus on CPUs and GPUs and their supporting parallel languages/APIs. Therefore, in the next section, we introduce a brief introduction about general architectures and paradigms for parallel processing of multi-core CPUs and their parallel APIs for message passing with Message Passing Interface MPI such as MPICH2. Also, we introduce shared memory parallel programming paradigm with multi-threading supporting interfaces namely the (OpenMP). We choose OpenMP because it provides high level abstraction and simple to use. After that we concentrate on shared memory parallel processing paradigm on Graphical Processing Units (GPUs) and their parallel APIs specially (CUDA).

2.2. CPUs Architectures and Parallel Processing APIs

The microprocessor industry continues to have great importance in the course of technological advancements ever since their coming to existence in 1970s, so the trends were focused to increase the performance of these chips by increasing the clock rate and by increasing the number of transistors of the chip, but there were physical limitations of modern processor designs as power consumption and heat dissipation. Additional techniques for increasing the performance are parallel processing, Data Level Parallelism (DLP, e.g., SIMD, vector computation), instruction level parallelism (ILP, e.g., pipelining, out-of-order super-scalar execution, excellent branch prediction), thread-level parallelism (TLP, e.g., simultaneous multi-threading, and multi-core), and memory-level parallelism (MLP, e.g., hardware pre-fetcher) which have all proven to be very effective. One such technique which improves significant performance boost is multi-core processors, these processors are typically a single processor which contains several cores on a chip that made up of computation units and caches. These multiple cores on a single chip improve

overall performance by handling more tasks in parallel (Venu, 2011), (Chhugani, et al., 2008).

There are many libraries for parallel programming For example, PVM, JPVM and MPI for message passing on distributed memory. POSIX and OpenMP are also used for multi-threading on shared memory (Culler, Singh, & Gupta, 1997) (Lee & Downar, 2001). These libraries can be implemented with C, C++ and FORTRAN programming languages the C language is preferred with it is fast and available on virtually any platform and can be used to implement OpenMP and MPI or hybrid of them (El-Nashar, May 2011).

The multi-core began as two-core processors, with the number of cores approximately doubling with each semiconductor process generation. We have dual, tri, quad, hex and soon even 16 and 32 cores and so on. But the number of cores in the CPU was limited due to many factors; one of them is the cache coherence. In a cache-coherent system 'that CPU follows' where GPUs are not, a write to memory location needs to be communicated to all levels of cache in all cores, thus all processor cores see the same view of memory at any time. Another factor is that when the number of cores increased, the communications between cores will evitable increase. Typically, CPU programs will have no more than twice the number of threads active than the number of physical processor cores, even with the existence of the hyper-Threading technique. In the single core CPUs, to run more tasks than the number of the physical cores, they were time sliced by the operating system, but as the number of threads grows, the OS must do context switching. Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off of CPU execution channels to provide multi-threading capability. Context switches (when two threads are swapped) are therefore slow and expensive (Cook, 2013).

2.2.1. Message Passing Interface MPICH2

MPI (Message Passing Interface) has been the de facto standard for writing parallel applications, the first standard is MPI-1, then the MPI-2 standard with additionals such as dynamic process management, one-sided communication and I/O, it is the most commonly used paradigm in writing parallel programs since it can be employed not only within a single processing node but also across several connected ones.

MPI offers several functions such as point-to-point rendezvous-type send/receive operations, logical process topology, data exchange, gathering and reduction operations to

combine partial results from parallel processes, and synchronization capabilities manifested in barrier and event operations (El-Nashar, May 2011).

MPICH is the implementation of MPI developed at the Argonne National Laboratory which supports the MPI-1 standard.

MPICH2 is a successor of MPICH that facilitates the MPI-2 standard; it is completely redesigned and developed to achieve high performance, maximum flexibility, and good portability.

Fig 2.2 shows the hierarchical structure of the MPICH2 implementation, that has four layers: message passing interface 2 (MPI-2), the abstract device interface (ADI3), the channel interface (CH3), and the low-level interface (Xiaojun, et al., 2010).

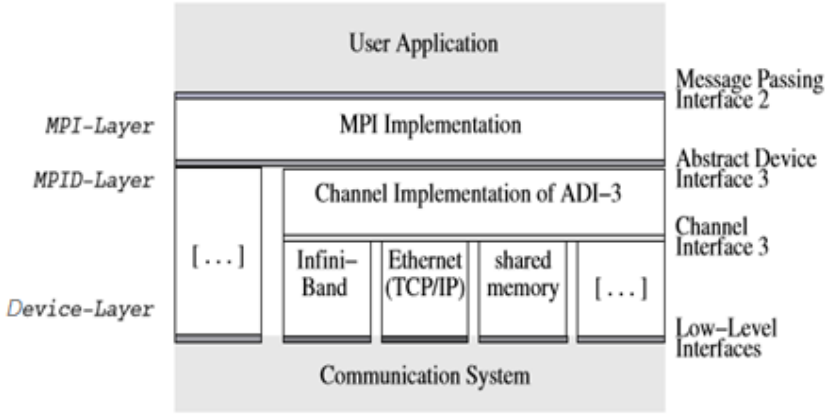


Figure 2.2: MPICH2 Model.

MPICH2 supports different platforms using of C/C++ and FORTRAN programming languages. In contrast to MPICH2 for Windows, the implementation for UNIX and LINUX offers built-in network topology support.

In windows, MPICH2 can be used on a one or more interconnected machines that have one or more cores, but in both cases, MPI programs performance is affected by the number of cores (machines), number of running processes and the programming paradigm which is used.

MPI is more preferred API since it is implemented for a broad variety of architectures, including implementations that are freely available and it is well documented; it has grown much more popular than alternative platforms, such as PVM. I (see Section 2.3). (El-Nashar, May 2011).

2.2.2. Shared Memory Multithreading with OpenMP

OpenMP has established itself as an important method and language extension for programming shared memory parallel computers. On these platforms, OpenMP offers an easier programming model than the currently widely-used message passing paradigm (Basumallik, Min, & Eigenmann, 2007). It is an application program interface API, which is an open source programming model defined by major computer vendor companies. It includes and defines several programming directives, run time library functions/procedures and environment variables that are used to explicitly express direct multi-threaded, shared memory parallelism. It can be used with C/C++/FORTRAN. OpenMP implements the details like work load partitioning, worker management and communication and synchronization transparently. Thus, the developers only need to specify the directives in order to parallelize the application (Ajkunic, Fatkic, Omerovic, Talic, & Nosovic, 21-25 May 2012).

OpenMP is preferred for shared-memory on SMPs because OpenMP is standardized and comes ready-to-use with contemporary C/C++ compilers, including compilers that are freely available and it is easier to use than various thread libraries because it supports a higher level parallel programming model, OpenMP is can work on a wider number of shared-memory computers as opposed to other interesting yet less available higher-level frameworks (El-Nashar, May 2011).

2.3. GPUs Architectures and parallel Processing APIs

A new architecture, which based on large number of cores called Graphics Processing Units (GPUs) that dramatically improved programmability of the chips using high-level APIs and turned this type of processors into an interesting choice for supercomputers targeting massively data parallel applications. NVIDIA's CUDA (Computer Unified Device Architecture) programming model was the first that enabled high level programming of GPUs; more recently the OpenCL API is set to become the de facto standard for data-parallel computing in general and for programming GPUs in particular. (Vajda, 2011).

Many companies have produced GPUs under a number of brand names such as Intel, Nvidia and AMD/ ATI.

2.3.1. What is GPU

GPU is a graphical processing unit ‘a single-chip processor’ which enables users to run high definitions graphics on PC. The primary job of the GPU is to compute 3D functions since these types of calculations are very heavy on the CPU.

At the early stages of GPUs, they were dedicated for graphical purpose, but now, they evolved into computing, precision and performance to be called general purpose GPU (GPGPU) (Ghorpad, Parande, Kulkarn, & Bawaska, January 2012).

GPUs are processors that can operate in parallel by running a single kernel (function) on many records in a stream at once ‘have thousands of small and efficient cores'. This enables the GPU to have Data-parallel processing and results in tremendous computational power for the GPU.
GPGPU is used for performing complex mathematical operations in parallel and its forte for floating point operations for achieving high performance with less time. The arithmetic power of the GPGPU is a result of its highly specialized architecture.

CPUs have few numbers of cores and it is specialized for handling different tasks of the operating systems such as job scheduling and memory management and were optimized for high performance on sequential codes (caches and branch prediction), and since all developers and users seek for high performance for programs and applications, GPGPU was used as a coprocessor with the CPU. However, threads on GPUs are extremely lightweight, so thousands of threads are queued up for work.

Since CPUs and GPUs are built based on very different philosophies each of them is best to do something rather another, so GPUs can’t replace CPUs (Ghorpad, Parande, Kulkarn, & Bawaska, January 2012). Fig 2.3 shows the difference between cores in both CPU and GPU.

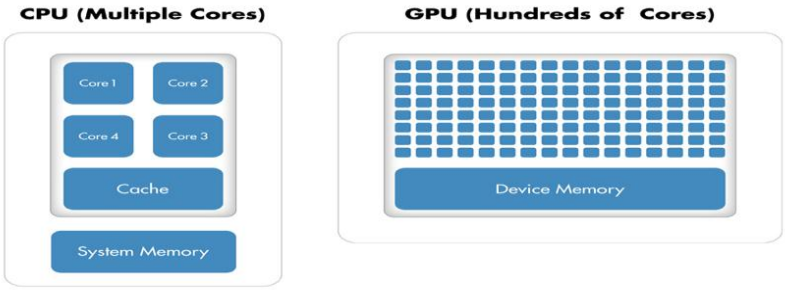


Figure 2.3: Cores comparison between CPU and GPU (The MathWorks, 1994-2016).

Many-core processors, especially the GPUs, have led the race of floating-point performance since 2003 as illustrated in Fig 2.4 while the performance improvement of general-purpose microprocessors has slowed significantly.

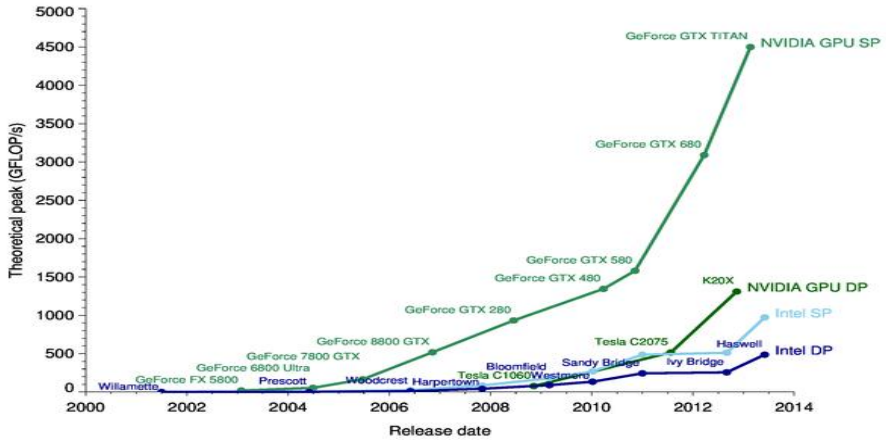


Figure 2.4: CPU and GPU peak performance in gigaflops (Galloy, 2013)

2.3.2. GPU Structure

The many-core began as a large number of much smaller cores, and, once again, the number of cores doubles with each generation. In the GPU, we have many-streaming multi-processors (SMs); each has many streaming processors (SPs) or CUDA cores, they are connected to a shared memory/L1 cache. This is connected to an L2 cache that acts as an inter-SM switch. Data is stored in the global memory (DRAM) in which data can be transferred between host and device or to other GPUs through the PCI-E bus where it is many times faster than any network's interconnect (Cook, 2013).

SM also contains Warp schedulers that can quickly switch contexts between threads and issue instructions to warps that are ready to execute, execution cores for integer and floating-point operations, and Special Function Units (SFUs) for single-precision floating-point transcendental functions (Wilt, 2013). The SPs execute work as parallel sets of up to 32 units. NVIDIA hardware will increase in performance by growing a combination of the number of SMs and number of cores per SM (Cook, 2013), a single SM for example is shown in Fig 2.5 contains many scalar processors, and each has many registers. Each SM is also equipped with on-chip memory that has very low access latency and high bandwidth, similar to an L1 cache (Satish, Harris, & Garland, 23-29 May 2009).

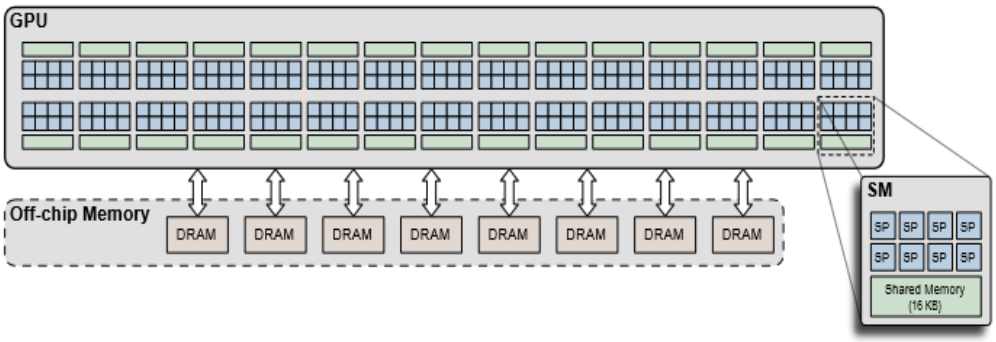


Figure 2.5: GeForce GTX 280 GPU with 240 scalar processor cores (SPs), organized in 30 multi-processors (SMs).

The number of SMs in a building block can vary from one generation of CUDA GPUs to another generation where the number of SMs depends on the card and the number of SP per SM depends on the architecture.

The different types of memory are register, shared, local, global, and constant memory. Each GPU currently has global memory (RAM); these RAMs of the GPU are different from RAMs of the CPU, for graphics applications, they hold video images, and texture information for three-dimensional (3D) rendering, but for computing they function as very-high-bandwidth, off-chip memory, though with somewhat more latency than typical system memory. The constant memory supports short-latency, high-bandwidth, and read-only access by the device when all threads simultaneously access the same location. Registers and shared memory are on-chip memories. Variables that reside in these types of memory can be accessed at very high speed in a highly parallel manner. Fig 2.6 and Fig 2.7 show CUDA device memory model.

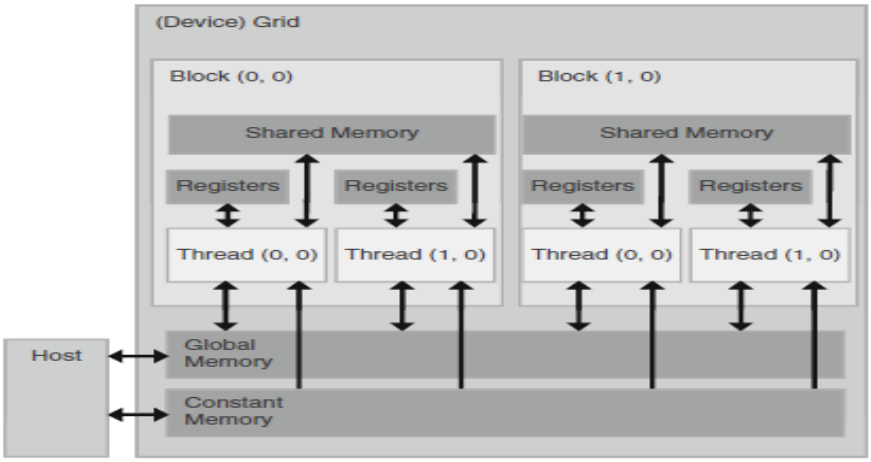


Figure 2.6: CUDA Programming Model and Memory Hierarchy (Kirk & Hwu, 2010).

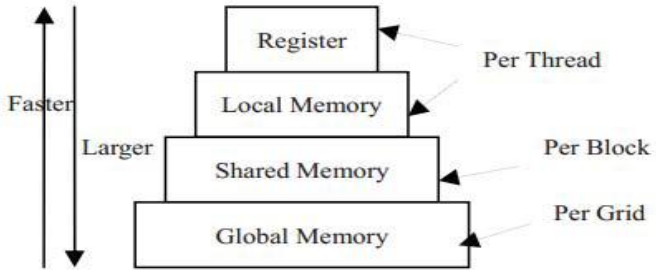


Figure 2.7: Simplified CUDA memory model (Yildiz, Aydin, & Yilmaz, 7-9 Nov. 2013).

There are many factors that affect the GPU throughput such as the bandwidth of global memory, the number of SPs in GPU, and how the GPU parallelism had been correctly exploited by the programmer.

When we want to use the GPU for programming, some points of consideration must be in mind to exploit the GPU efficiently. One of these points is that the complexity of operations must justify the cost of moving data to and from the device, because if the GPU used to perform little parallel operations, it will show little or no performance benefit because transferring data to and from GPU is expensive.

2.3.3. Shared Memory Parallel Programming on GPUs

Many parallel programming APIs for shared memory parallel programming paradigm had been developed for shared-memory on multi-core systems such OpenMP and POSIX. The amount of effort required to port an application into MPI, however, can be extremely high due to the lack of shared memory across computing nodes. For example, OpenMP offers shared memory for multi-core, but the communication is also expensive due to high costs of context switching. CUDA passes this problem by providing shared memory for parallel execution in the GPU. It achieves much higher scalability with simple, low-overhead thread management and no cache coherence hardware requirements.

There is another parallel API for GPUs such as OpenCL .OpenCL (The open standard for parallel programming of heterogeneous systems, 2016) is heterogeneous on both CPUs and GPUs supported by NVIDIA, AMD, and others. CUDA is more preferred since it is somewhat of a higher-level language extension than OpenCL.

There are also several programming libraries alternative to CUDA such as 'DirectCompute' from Microsoft. 'Pthreads' is a library that is used significantly for multi-thread applications on Linux. Also, 'ZeroMQ' (0MQ) which is a simple library that can be linked

to for developing a multi-node, multi-GPU. It's designed for distributed computing, so the connections are dynamic and nodes fail gracefully (Cook, 2013).

Cuda was introduced in 2006 (About CUDA, 2015), it comes with a software environment that allows the programming of GPUs for parallel computation without any graphics knowledge and enables the programmers to write scalable parallel programs by simply scaling the number of multi-processors and memory partitions. CUDA provides great fine-grained parallelism that sufficient for utilizing massively multi-threaded GPUs; it also provides a fairly simple, minimalist abstraction of parallelism with familiar languages to make the programming relative easy (Garland, et al., July-Aug. 2008).

CUDA can be developed with familiar programming languages such as C, C++, it supports various languages and application programming interfaces, other languages, or directives-based approaches is supported, such as FORTRAN, DirectCompute (CUDA C Best Practices Guide, 2015) .

CUDA C/C++ is an extension of the C/C++ programming languages for general purpose computation. It was first introduced by NVIDIA in 2007, Cuda program contains one or more parts that can be executed on GPU (device) or CPU (host), the part of code that with no or little parallelism will be executed on the CPU straight forward C code, where the part of code that have high parallelism will be executed on GPU using CUDA keywords (Ghorpad, Parande, Kulkarn, & Bawaska, January 2012).

The CUDA programming model guides the programmer to expose substantial fine-grained parallelism sufficient for utilizing massively multithreaded GPUs, while at the same time providing scalability across the broad spectrum of physical parallelism available in the range of GPU devices. Because it provides a fairly simple, minimalist abstraction of parallelism and inherits all the well-known semantics of C, it lets programmers develop massively parallel programs with relative ease. (Garland, et al., July-Aug. 2008)

CUDA is preferred rather than OpenCL since it performs better when transferring data to and from the GPU, also CUDA's kernel execution was also consistently faster than OpenCL's. CUDA seems to be a better choice for applications where achieving as high a performance as possible is important. Otherwise the choice between CUDA and OpenCL can be made by considering factors such as prior familiarity with either system, or

available development tools for the target GPU hardware. (Karimi, Dickson, & Hamze, 2010)

CUDA compilation trajectory separates the compilation of host code from device code (kernel), where the device code is compiled with NVIDIA compiler "*nvcc*", and the host code with a general purpose C/C++ compiler that is available on the host platform**,** *'nvcc'* uses the following compilers for host code compilation:

For Linux platforms**:** The GNU compiler, '*gcc'*, and '*arm-linux-gnueabihf-g++'* for cross compilation to the ARMv7 architecture. On Windows platforms**:** The Microsoft Visual Studio compiler, '*cl'* (CUDA Toolkit Documentation v7.5, 2015).

CUDA offers several APIs to use when programming. They are from highest to lowest level (Farber, 2012):

1. The data-parallel C++ Thrust API
2. The runtime API, which can be used in either C or C++
3. The driver API, which can be used with either C or C++

Regardless of the API or mix of APIs used in an application, CUDA can be called from other high-level languages such as Python, Java, FORTRAN, and many others.

CUDA program creates thousands of threads to hide memory access latency or math pipeline latency. A thread is the fundamental building block of a parallel program, for this purpose, CUDA splits problems into grids of blocks; each block contains many threads in which it is assigned to one SM at execution time. The block must execute from start to completion and may in any order. Initially this is done on a round-robin basis so each SM gets an equal distribution of blocks.

Threads in the same block can share data through the on-chip shared memory and can perform barrier synchronization. In the Fig 2.8 (Cook, 2013), it shows the organization of threads, blocks and grids. Each grid consists of N blocks, where each block has N warps.

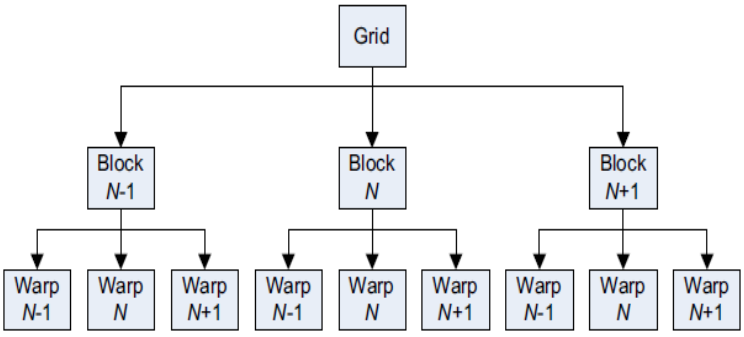


Figure 2.8 GPU based view of threads.

Fig 2.9 shows a CUDA Architecture which splits the GPU into three main parts: Grids, Blocks and Threads. Here we have two grids which are different, the first grid for kernel 1 is 2D array with total four blocks, each block is a 3D array of total 16 thread where typical CUDA grids contain thousands to millions of threads (Ghorpad, Parande, Kulkarn, & Bawaska, January 2012) (Kirk & Hwu, 2010).

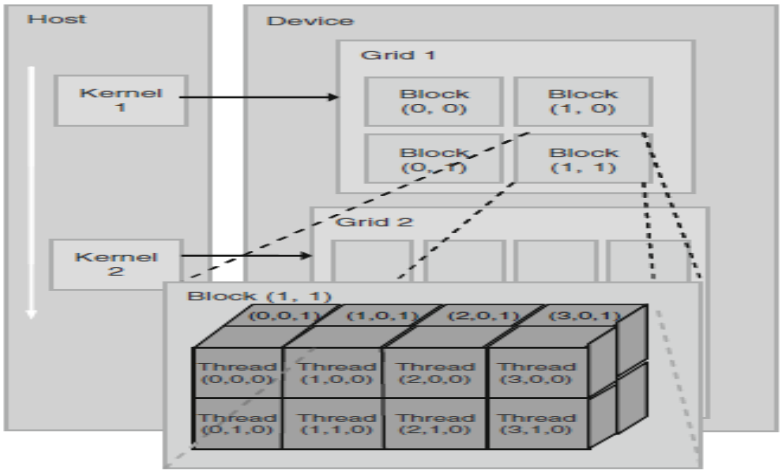


Figure 2.9: CUDA Architecture.

A grid is a group of threads that all execute the same kernel which are not synchronized. They can be organized in one or two-dimensional array of blocks. All blocks in a grid use the same program and have the same dimensions. Blocks also can be organized in one, two or three dimensional array of threads depending on the execution configuration provided at kernel launch which takes two parameters, the first one is the dimension of the grid which is the number of blocks, and the second is the dimension of the block which is the number of thread in each block.

Each thread has an identification number 'ID' (threadIdx) and is relative to the block it is in. Thread IDs can be 1D, 2D or 3D (based on block dimension), so it have three components: the x coordinate threadIdx.x, the y coordinate threadIdx.y, and the z coordinate threadIdx.z , where all threads in a block share the same blockIdx value.

On Fermi GPUs for example, you can define $65{,}535 \times 65{,}535 \times 1536$ threads in total, 24 K of which are active at anytime. This is usually enough to cover most problems within a single node (Cook, 2013).

Given the heterogeneous nature of the CUDA programming model, a typical sequence of operations for a CUDA C program is:

1. Declare and allocate host and device memory.
2. Initialize host data.
3. Transfer data from the host to the device.
4. Execute one or more kernels.
5. Transfer results from the device to the host.

Thread blocks consist of continuous warps, each warp consists of 32 threads, and for example the first warp starts with thread0 and continuously ends with threads31, the second warp starts with thread32 and end with thread63 and so on. For a block of which the size is not a multiple of 32, the last warp will be padded with extra threads to fill up the 32 threads. For example, if a block has 48 threads, it will be partitioned into two warps, and its warp1will be padded with 16 extra threads (Kirk & Hwu, 2011). The execution of warps is not truly SIMD hardware, but they are SIMT (Single Instruction Multiple Thread) devices, because GPUs contains many SMs that may be running one or more different instruction (Farber, 2012).

Recall that the control statements like the if-statement and *for* loops can decrease the performance inside an SM, since each thread can follow different path during the evaluation of the control statement, we say that these threads diverge in their execution. In the *if-else* example, 'divergence' arises if some threads in a warp take the 'then' path and some the 'else' path, where in the *for* loop some threads inside the warp may iterate different time from others, where maximum 32-time slowdown can occur when each thread in a warp executes a separate condition when one thread needs to perform an expensive computational task or each thread performs a separate task.
The cost of divergence is the extra pass the hardware needs to take to allow the threads in a warp to make their own decisions. Fig 2.10 shows the GPU warp execution.

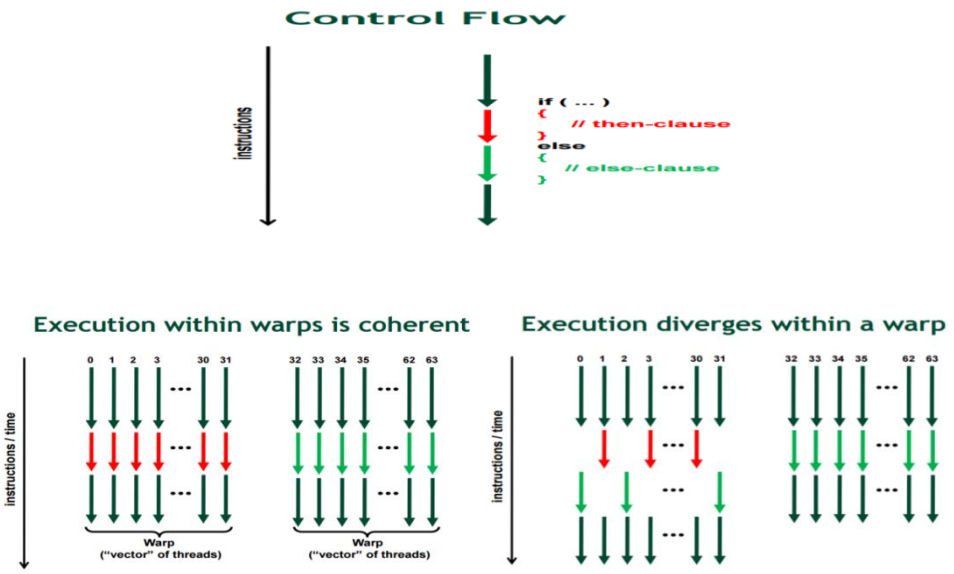


Figure 2.10: GPU Warp Execution.

2.3.4. Warp Scheduling and TLP

To hide the latencies of the ALU and memory of the GPU and to keep the execution units busy, many warps of threads should be running per SM. The hardware SM is fast enough that it basically has no overhead, it can detect the warps whose instructions are ready for execution and so the dependencies are resolved. For these eligible warps, the SM scheduler selects warps regarding to an internal prioritized schedule, then it issues the instruction from that warps to the SIMD cores. When all warps are stall, the hardware become idle and the performance decreased.

TLP (thread level parallelism) is the idea for increasing the performance by giving the scheduler as many threads as possible to choose, the occupancy is a measure of TLP which is the number of active warps running concurrently on a multi-processor divided by the maximum number of warps resident on an SM. When the occupancy is high, this means that the scheduler has many warps to choose and to hide the latencies. Though simple in concept, occupancy is complicated in practice, as it can be limited by on-chip SM memory (Farber, 2012).

Chapter 3

Parallel Sorting Algorithms and Related Work

This chapter introduces a background and literature review regarding parallel algorithms and programming. Also, it presents formal definition and analysis for some relevant sequential and parallel sorting algorithms mainly the mergesort.

3.1. Introduction

Parallel programs that parallelize tasks were appeared in 1958 by Gill, and in 1959 many sub programs ran on a computer pointed about by Holland, where in 1963 Conway presented a parallel computer and its programming (Ravela, 2010).

Parallel sorting is the process of using multiple processing units to collectively sort an unordered sequence. Each part of the unsorted sequence is treated by a unique processing unit, where collectively produces a fully sorted sequence (Kale & Solomonik, 2010) .

Many sorting techniques had evolved like external/internal sorting, data-specific and hardware-specific sorting, etc, so when designing an efficient implementation of any algorithm, the underlying hardware must be considered. Therefore, some basic knowledge about cache/memory interaction should be embedded in the algorithm structure (Pasetto & Akhriev, 2011). Also, these algorithms must be carefully tuned on CPU cores in which the SIMD design must not waked by conditional branches, each thread that belong to its corresponding core must be very careful about data partitioning so that multiple threads can work effectively together to avoid the problem of race conditions, Moreover, due to partitioning, the starting elements for different threads must align with the cache-line boundary (Chhugani, et al., 2008). GPUs also have provided a huge edge over the CPU with respect to computation speed.

3.2. Sorting Algorithms

Sorting is arguably one of the most studied problem in computer science due to its high importance for data analysis in several application domains, sorting is used for sorting data and for database operations such as creation of indices and binary searches, it helps in statistics like finding closest pair, determining an element's uniqueness, finding k^{th} largest element, and identifying outliers. Sorting is used to find the convex hull, and it is an important algorithm used in computational geometry. Other applications that use sorting include computer graphics, computational biology, supply chain management and data compression, N-body simulations, some high performance sparse matrix-vector multiplication implementations and machine learning algorithms (Chhugani, et al., 2008) (Leischner, Osipov, & Sanders, 2009) (Tanasic, et al., 2013).

Sorting is the process of reordering an unordered list of objects of size N to produce an ordered one; either in ascending or descending order. In general the sorting algorithms can be formally described as follows:

Pre-condition: unordered list of size N.

Post-condition: ordered list of size N.

Algorithm: (any correct sorting algorithm such as e.g., mergesort, quicksort).

Definition 3.1: The elements of set *S* are said to satisfy a *linear order* *'<'* if and only if:

1. For any two elements *a* and *b* of *S*, either $a < b$, $a = b$, or $b < a$; and
2. For any three elements *a*, *b*, and *c* of *S*, if $a < b$ and $b < c$, then $a < c$.

The linear order '<' is usually read "precedes".

Definition 3.2: Given a sequence $S = \{a_1, a_2, \ldots, a_n\}$ of N items on which a linear order is defined, the purpose of *sorting* is to arrange the elements of A into a new sequence $S' = \{ a_1', a_2', \ldots, a_n'\}$ such that $a_i' < a_{i+1}'$ for $i = 1, 2, \ldots, N - 1$.

Numerous sorting algorithms were developed along the past decades. Most of these algorithms are comparison based algorithms. However, they are differing on the usage of memory some are:

- In-place/internal (does not need external memory) such as insertion, selection, bubble, shell, heap, and quick sort.

- Out-place/external (which need auxiliary memory for implementing the sorting process) such as mergesort.

Also, they differ in their performance by means of their complexity (running time). Asymptotical running time (complexity) is the main tool for evaluating their performance. Some has complexity $O(N^2)$ such as insertion, selection, bubble, and shell sorting algorithms where N is the problem size (number of elements). Asymptotically optimal running time O(N log N) can be achieved by optimal algorithms such as (quicksort, heapsort, and mergesort).

Asymptotically optimal algorithms have been adapted and developed for parallel computing on varying parallel processing systems models as well. Parallel sorting algorithms have already been proposed for a variety of parallel architectures.

3.3. Sequential Mergesort

Mergesort is one of the optimal comparison based sorting algorithms. It is also based on the divide-and-conquer paradigm. Mergesort on an input sequence ***S*** with ***N*** element consists of three steps:

- Divide: partition ***S*** into two sequences S_1 ***and*** S_2 of about ***N/2*** elements each.
- Recur: recursively apply mergesort on S_1 and S_2
- Conquer: merge the sorted S'_1 and S'_2 into a unique sorted sequence ***S'***

In the merging phase, the algorithm starts merging every two adjacent numbers sequentially until reaching the end of the array, then it recursively call itself to merge every four adjacent numbers and so until sorting all the array, and so for every power of two elements. Fig 3.1 shows the sequential mergesort for eight elements.

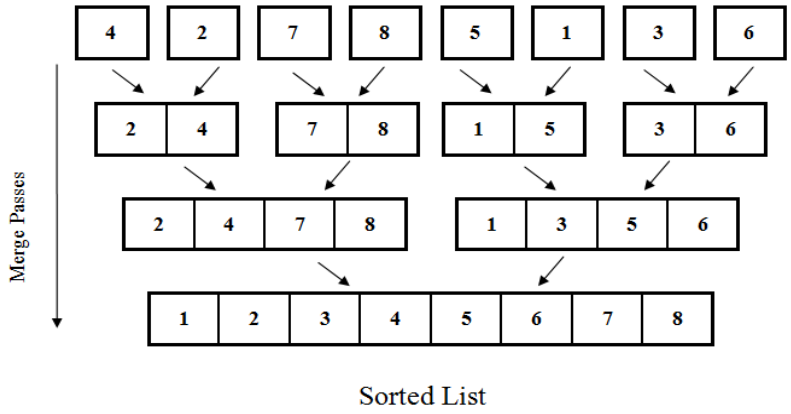


Figure 3.1: Sequential Mergesort Design.

Fig 3.2 shows the recursive sequential mergesort algorithm.

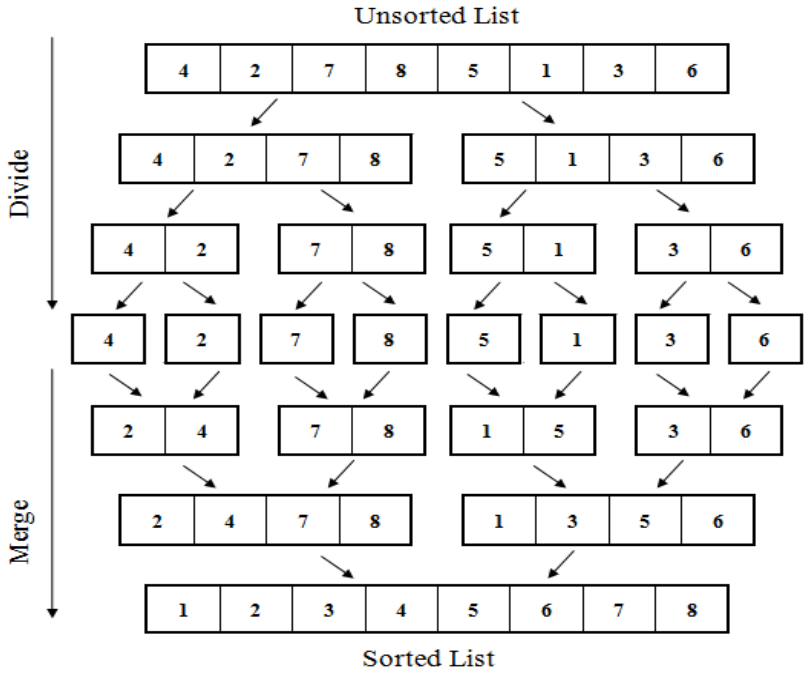


Figure 3.2: Sequential Recursive Mergesort algorithm.

Now, Fig 3.3 shows the pseudocode for recursive sequential merge sort algorithm.

```
1. Algorithm: MergeSort (S[0..N-1])
   // Sorts Array S[0..N-1]) by Recursive mergesort
   // Input :  An array S[0..N-1]of non-orderable elements
   //Output: An array S[0..N-1] sorted in non-decreasing
   //order
2. If  N > 1
3.     Copy S[0..N/2-1] to S1[0..N/2-1]
4.     Copy S[N/2..N-1] to S2[0..N/2-1]
5.     Mergesort(S1[0..N/2-1])
6.     Mergesort(S2[0..N/2-1])
7.     Merge(S1,S2,S)
8.  Algorithm: Merge  (S1[0..k-1], S2[0..m-1], S[0..k+m-1])
   // Merges two sorted lists into one sorted list/array
   // Input: Sorted  lists S1[0..k-1], S2[0..m-1]
   // Output: Sorted  lists S[0..k+m-1] of the elements of S1
   //and S2
9. i←0, j←0, q←0
```

```
10. While  i < k  and   j < m  Do
11.     If  S1[i] ≤ S2[j]
12.         S[q] = S1[i]          i ← i+1
13.     Else
14.         S[q] = S2[j]          j ← j+1
15.     q ← q+1
16. If  i = k
17.     Copy S2[j..m-1] to S[k..k+m-1]
18. Else
19.     Copy S1[i..k-1] to S[k..k+m-1]
```

Figure 3.3: The Pseudocode for Recursive Sequential Mergesort Algorithm.

Analysis

Let $T_s(N)$ be the runtime of sequential mergesort on N items, then: $T_s(N) = 2*T_s(N/2) + N$, where the first part of equation is the cost of merging the two parts of the array recursively, and the second part is the cost of the final merge step of the sorted two parts.

- $T_s(1) = 2*T_s(0) + 1$
- $T_s(2) = 2*T_s(1) + 2$
- $T_s(4) = 2*T_s(2) + 4$
- $T_s(8) = 2*T_s(4) + 8$
- ...
- $T_s(N/2) = 2*T_s(N/4) + N/2$
- $T_s(N) = 2*T_s(N/2) + N$

- $T_s(N) = 2*T_s(N/2) + N$
- $= 2*(2*T_s(N/4) + N/2) + N$
- $= 4*T_s(N/4) + 2N$
- $= 8*T_s(N/8) + 3N$
- ...

Since $N=2^K$ where K is the number of merging levels, then:

- $T_s(N) = 2^k T_s(N/2^k) + kN$ ..... 3.1)

- To get to a more simplified case, let's set k = $\log_2 N$.
  - $T_s(N) = 2^{\log N} T_s(N/2^{\log N}) + (\log N) N$
  - $T_s(N) = N * T_s(N/N) + N \log N$
  - $T_s(N) = N * T_s(1) + N \log N$
  - $T_s(N) = N * 1 + N \log N$
  - $T_s(N) = N + N \log N$
  - $T_s(N) = O(N \log N)$ ..... 3.2)

So, sequential merge sort algorithm behaves as O(*N log N*) computational complexity in all of its cases, worst, average and best cases (El-Nashar A. I., 2011).

3.4. Parallel Mergesort

This section presents and explores the parallel mergesort algorithm on CPUs and GPUs. Also, it presents some related work.

In CPUs, the parallel mergesort has three phases. In the first phase, the master process (holding the unsorted list/array) of size N elements distributes the array amongst the participating processes P with Message Passing Interface MPI using a suitable one-to-all Personalized/Scatter communication module depending on the parallel architecture, at the end each process will hold N/P elements. In the second phase, each processor sorts its data/sub-array locally using a quick-sorting algorithm. Finally, in this phase the sorted data is collected/ gathered in a reverse manner and merged until all data is collected forming N sorted elements (Jeon & Kim1, 2003).

The parallel mergesort algorithm on loosely coupled architecture can be divided into three phases:

- Distribution/division phase: in this phase the data is distributed amongst the available processors using a suitable communication technique.
- In the second phase, each process sorts its local data using a suitable sequential sorting algorithm.
- Merge phase (conquer phase): where the reverse communication take place and each process receives sorted data and merges its local data with the received data.

Fig 3.4 pseudocode describes the mergesorting algorithm.

```
1.  Algorithm: Parallel mergesort(DataArray,SizeofData)
2.  Begin
3.        MyData=LeftHalfof[DataArray]
4.        TempData=RightHalfof[DataArray]
5.        Send(TempData)
6.        MyData = Mergesort(MyData,i,j)
7.        Receive(TempData)
8.        Merge(MyData,Temp Data, DataArray) // see Fig 3.3
9.  End

10. procedure Mergesort(MyData,i,j)
11. Begin
12.       If ( j- i>N/P )
13.         {
14.           MergeSort(MyData,i,(i+j)/2)
15.           MergeSort(MyData,(i+j)/2,j)
16.           }
17.       Else
18.            Sequential_Sort(MyData,i,j)
19. End

20. procedure Sequential_Sort (MyData,i,j)
21. Begin
     //Sequential_ Sort
22. End
```

Figure 3.4: Pseudocode for Parallel MergeSort algorithm.

3.4.1 Related Work on Parallel Mergesort on CPUs

In (Dawra & Priti, July 2012), it introduces the parallel implementation of sorting algorithms on CPU, it presents an algorithm that will split large array into sub parts and apply efficient sorting functions on them, all parts are processed in parallel multi-threading using existing sorting algorithms and finally outcome would be merged.

The algorithm depends on master read/write indices, if the algorithm merges two arrays *S1[0..k-1]* and *S2[0..m-1]* into *S[0..k+m-1]*, it uses read indices i and j , and write index q to store the result, if both i and j are in range then choose *S1[i] and S2[j]* and write to *S[q]*, otherwise copy the rest values from the array to the resulting array. Then increment q and index of array algorithm located minimal value at.

In (Rashid & Qureshi, 2006), (Trimananda & Haryanto, 2-3 Aug. 2010) and (El-Nashar, May 2011), the parallel mergesort is implemented and evaluated on multi-core CPUs. The list is divided into 2 equally sized lists and the generated sub-lists are further divided until each number is obtained individually. The numbers are then merged together as pairs to form sorted lists of length 2. The lists are then merged until the whole list is constructed. The algorithm is parallelized by distributing n/p elements (where n is the list size and p is the number of processors) to each slave processor. In (Rashid & Qureshi, 2006) and (El-Nashar, May 2011), the slave can sequentially sort the sub-list using sequential mergesort and then return the sorted sub-list to the master, where the master is responsible of merging all sorted sub-lists into one sorted list, where in (El-Nashar, May 2011) the sorted sub-lists are distributed among dual-cores processors using MPI platform. In (Trimananda & Haryanto, 2-3 Aug. 2010) it sort the sub-lists with parallel quicksort using multi-threading on each processor, and later merged in parallel by using mergesort algorithm with MPICH platform.

In (Radenki A., 2011), it introduces three parallel versions of recursive mergesort: the first is shared memory (with OpenMP) on mainstream SMP-based systems, such as multi-core computers and multi-core clusters, the second is message-passing that runs on computer clusters (with MPI) and the third is a hybrid (with MPI and OpenMP) that runs on clustered SMP.

In the shared memory implementation, it uses multi-threaded recursive mergesort where independent sections of code have to be divided between automatically generated threads. Where in the MPI version; all MPI processes start at once at the very beginning of program execution, and all processes concurrently execute the same code. The root process splits the data and sends half of it to a helper process which sorts the data and returns it to the root process. The other half of data is retained by the root process for further sorting by using this same procedure .Then, the two halves of data are merged by the root process. In the hybrid parallel architecture, it combines distributed and shared memory in the same computing system.

In (Chhugani, et al., 2008), the mergesort is parallelized on multi-core CPUs using the multi-way merge, the dataset is divided into chunks (or blocks), where each block can reside in the cache memory. Each block is sorted separately, and then they merged in the usual fashion, each block of data is divided amongst the available threads (or processors). Each thread sorts the data assigned to it using merging networks like odd-even mergesort.
In the merging phase, it requires multiple threads to work simultaneously to merge two lists using multi-way merge.

In (Dominik Zurek, 2013), it describes the results from the implementation of a few different parallel sorting algorithms on GPU cards and multi-core processors. Then, a hybrid algorithm will be presented, consisting of parts executed on both platforms (a standard CPU and GPU). On GPU, sorting algorithm is based on the bitonic sort, in the first stage; pairs of subsequences of length 1 are merged, which results in sorted subsequences of length 2. In the second stage, pairs of these sorted subsequences are merged, resulting in sorted subsequences of length 4 and so. In each stage, the merging process of two bitonic subsequences is performed. These steps of the algorithm are run until all input data in the global memory is sorted. On CPU, the data is s divided into equal parts for each core and then sorted in parallel with the quicksort with OpenMP, and then the results of each core are merged by an efficient mergesort algorithm.
In the hybrid, the algorithm consists of two main steps; The first one executed on a GPU and the second on a CPU, data is transferred from CPU to GPU, and blocks of data with maximum size of 4096 is sorted in parallel using the bitonic sort, then the partially sorted data is transferred back to CPU to be merged in parallel.

In (D.Abhyankar, 2011), the study shows that the proposed algorithm is excellent for large input sizes and multiple free cores, when the size of array is small, no parallelization is needed and the heapsort is invoked, else it will divide the array into n equal parts and sort them in parallel on different cores using heapsort. Repeatedly use free cores to merge consecutive sub-arrays using an adaptive in place merges until there are sub arrays to merge. in the merging step; when there are 2 sorted consecutive sub arrays A and B. A can be divided into two types of sub arrays A1 and A2. A1 will have elements which do not involve inversions with elements of B. A2 will have inversions with elements of B. In the same way B can be divided into B1 and B2. B1 will have elements which do not involve inversions with elements of A. B2 will have inversions with elements of A and transfer A2 into B and transfer B2 into A. Restore the sorting order of transferred elements. Now we

have at most 2 A components and 2 B components to merge which can be merged by the algorithm on free cores.

3.4.2. Related Work on Parallel Mergesort on GPUs

In this section we present some related work on parallel sorting algorithms and approaches on GPUs.

In (Satish, Harris, & Garland, 23-29 May 2009) the authors claimed that mergesort is the best external sorting technique to be implemented on GPUs. In GPUs the data is stored in large external memory and each processor access a smaller memory. The data is loaded into the global memory which has large size, and every block of thread have access to its own small and fast shared memory that fairly small parts of the global memory data transferred to. This fits the situation of mergesort algorithm, where the data is loaded into global memory and small parts of the data are transferred to the shared memory by block of threads to be sorted on these on-chip memories. Finally, these sorted parts are merged to form one sorted list. Data is partitioned into equally sized blocks that will be sorted with parallel Batcher's odd-even mergesort instead of bitonic sort that proposed in the previous work for the same paper published in 2008 on the shared memory of GPU, then the sorted blocks are merged using the proposed mergesort.
The key to the mergesort algorithm is to compute the ranks r1 and r2 of element e in the even and odd sorted blocks, then the even block is divided into its first r1 elements A and the remainder B, then divide the odd block into its first r2 elements C and remainder D. The sub-block pairs A;C and B;D then they are merged independently in parallel.

The parallel mergesort in (Davidson, Tarjan, Garland, & Owens, 13-14 May 2012) is designed on GPU to use more register communication (not shared memory) for sorting variable-length key/value pairs, specially the string keys. It implements the mergesort as follows: First it sorts t blocks of size m using block sort, then each CUDA block merges independently each two of these sorted sequences together, so at each step the number of blocks is halved and the size of each sequence is doubled. So the goal in this paper is to create a merge which utilizes shared memory at every stage for binary and linear searches even though the hardware shared memory size remains fixed. The algorithm uses two moving memory windows: one in registers, and one in shared memory.

The authors in (Amirul, et al., 23-25 Nov. 2012) proposed a technique to sort huge text data with different hybrid algorithms applied on multiple Graphic Processor Unit (GPU) cards. One of the proposed hybrid algorithms will combine the Radix sort algorithm with simple mergesort Algorithm. The basic idea is to divide the text data into multiple buckets and then sort individual buckets in different GPUs in parallel eventually merging the sorted bucket into a final sorted result. In this approach, the input text data is partitioned into chunks and assigned to different GPUs to be sorted in parallel using radix sort, and then the sorted chunks are returned back to the memory of CPU to be merged into one array.

The researchers in (Peters, Schulz-Hildebrandt, & Luttenberger, 19-23 April 2010), introduced a novel merge-based external sorting algorithm for one or more CUDA-enabled GPUs using approach similar to sample sort and by overlapping memory transfers with GPU computation. In the first phase an input sequence is divided into shorter subsequences, each of them is transferred to the GPU, sorted using a GPU-sorting algorithm (internal sort), and then transferred back.

After the first phase is completed, the CPU pre-computes the data sets for the internal merge runs that form the k-way merging process. In the next phase, single sets are transferred to the GPU, merged on the GPU internally and transferred back. The concatenation of the output of these merge runs is a sorted sequence.

In this paper, the k-way merge algorithm is modified to achieved best performance when using (s_{seq}/s_{gpu}) -way merge where the s_{seq} is the size of all the input array and s_{gpu} is the size of the shorter lengths resulting in transferring each element only once to the GPU and back in the second phase.

Like the sample sort which uses only a subset of the splitters for partitioning the sequences, the proposed algorithm uses all of them.

In the next chapter we present and illustrate our two directional parallel mergesort approach.

Chapter 4

The Proposed Algorithm (Two Directional Parallel Mergesort)

In this chapter we introduce our new two directional parallel mergesort algorithm on two architectures, namely on multi-core (CPUs) and on Graphical Processing Units (GPU). On the CPUs, we use hybrid paradigm i.e., utilize message passing interface (MPI) with MPICH2 and shared memory paradigm using multi-threading with (OpenMP). This approach i.e., two directional parallel sorting algorithm, we will call (2DPMS-CPU). On the other hand, we utilize shared memory paradigm with multithreading on GPU. The two-directional parallel sorting algorithm on GPU will be called (2DPMS-GPU). To distinguish between our approach and the originally developed algorithms, we start with exploring the existing related parallel mergesorting algorithms on multi-core CPUs i.e., one directional parallel mergesort algorithm (1DPMS-CPU) and then the one directional mergesort on GPUs (1DPMS-GPU). It must be notable that there is no relation between the dimensions of Cuda grids/blocks/threads and our model name.

It should be noted that similar to the other sorting algorithms our algorithm can sort different types of discrete data/elements. Data such strings and database keys, images need to be preprocessed such as indexing. After that the indices can be use as input data/element to our algorithm.

4.1. Parallel Mergesort on CPUs

In this section we present the trivial one directional and our two directional parallel mergesort on multi-core CPUs using message passing with MPI and OpenMP for (1DPMS-CPU) and (2DPMS-CPU).

4.1.1. One Directional Parallel Mergesort on CPUs (1DPMS-CPU)

Recall that the parallel mergesort algorithm on multi-core and loosely coupled architecture using message passing paradigm can be divided into three phase:

1- Distribution/division (Scatter) phase: in this phase the data is distributed amongst the available processors using a suitable communication technique such as One-to-All personalizes/Scatter communication model.

2- In the second phase, each process sorts its local data (block) using a suitable sequential sorting algorithm such as quicksorting algorithm.

3- Merge phase (Gather/Conquer) phase: where the reverse communication take place and each process receives sorted data using Gather communication model and merges its local data with the received data.

Fig 4.1depicts the process of divide and conquer of parallel mergesort of 8 elements and process allocation in four-level mergesorting tree.

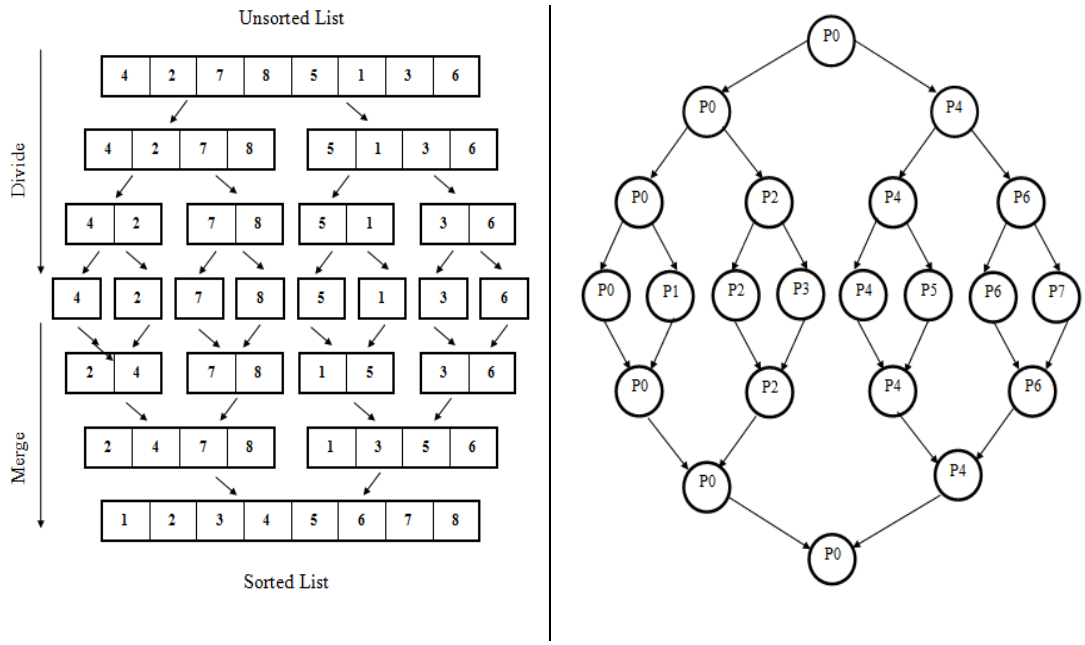


Figure 4.1: Parallel Mergesort and Process Allocation of 8 elements and 8 processes.

For example, the following is the distribution to 8 processes P_i for i= 0 to 7 as follows:

Phase 1:

Step 1: $P_0 \rightarrow P_4$ (P_0 sends N/2 of the data to P_4)

Step 2: $P_0 \rightarrow P_2$ $P_4 \rightarrow P_6$ (P_0 and P_4, send N/4 of the data to P_2 and P_6, respectively)

Step 3: $P_0 \rightarrow P_1$ $P_2 \rightarrow P_3$ $P_4 \rightarrow P_5$ $P_6 \rightarrow P_7$

Phase 2:

When each process receives its block (sub-list) of data, it sorts it sequentially using quicksorting function.

Phase 3:

This phase is the conquer phase where the sorted blocks are collected and merged in (log*P*) steps in a reverse order of communication that done in phase 1.

Step 1: $P_1 \rightarrow P_0$ $P_3 \rightarrow P_2$ $P_5 \rightarrow P_4$ $P_7 \rightarrow P_6$

Step 2: $P_2 \rightarrow P_0$ $P_6 \rightarrow P_4$

Step 3: $P_4 \rightarrow P_0$

In phase 3, when P_0, P_2, P_4, and P_6, receive the sorted blocks from P_1, P_3, P_5, and P_7, they merge their sorted blocks with the received blocks. The merging process is done using regular merging process (from head to tail). The merging process in each step is depicted in Fig 4.2.

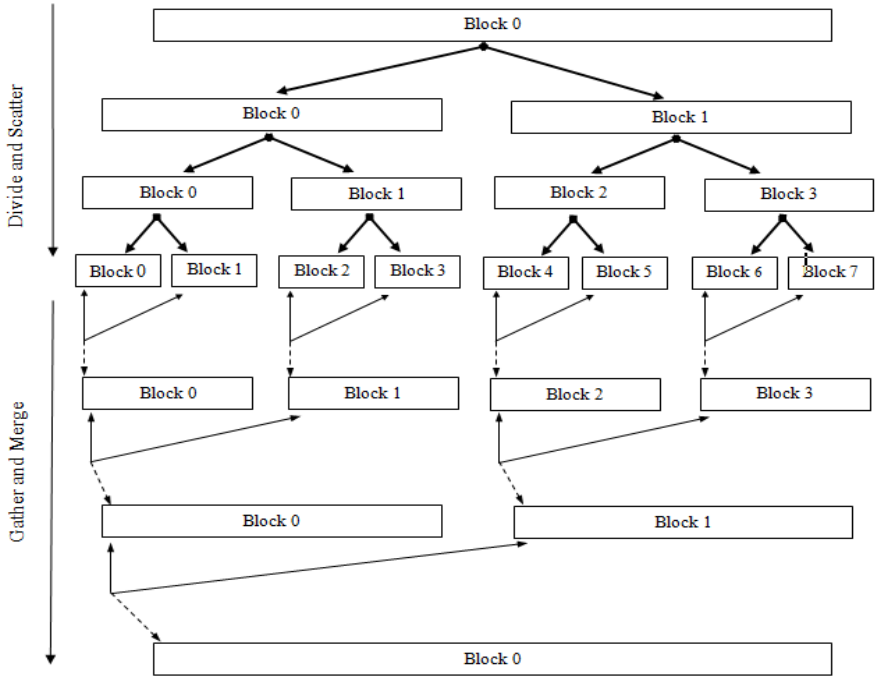


Figure 4.2: 1DPMS-CPU Mergesort Design.

The pseudo-code in Fig 4.3 describes the parallel mergesorting algorithm 1DPMS-CPU.

```
1.  Algorithm: parallel_mergesort(DataArray,N)
2.  Begin
3.        MyData=LeftHalfof [DataArray]
4.        TempData=RightHalfof [DataArray]
5.        Send(TempData)
6.        MyData = Mergesort(MyData,i,j)
7.        Receive(TempData)
8.        Merge (MyData, TempData, DataArray) //see Fig 3.3
9.  End

10. procedure Mergesort(MyData,i,j)
11. Begin
12.       If ( j- i>N/p )
13.         {
14.          MergeSort(MyData,i,(i+j)/2)
15.          MergeSort(MyData,(i+j)/2,j)
16.          }
17.       Else
18.           Sequential_Sort(MyData,i,j, N/p)
19. End

20. procedure Sequential_Sort (MyData,i,j)
21. Begin
    //Sequential_Sort
22. End
```

Figure 4.3: Pseudocode for 1DPMS-CPU Algorithm.

Analysis of One Directional Mergesort on CPUs (1DPMS-CPU)

It is well known that the optimal sequential time complexity $T_s(N)$ is O(N log N). In case of parallel mergesorting algorithm the complexity composed of both communication cost and computational cost.

1. Communication cost in the distribution/division phase.

$$T_{comm1} = t_s \log P + t_w \cdot \sum_{i=1}^{\log P} \frac{P}{2^i} = t_s \log P + N t_w \quad (4.1)$$

Where t_s is the startup up time and t_w is data transfer time.

2. Sorting local data using quick sort:

$$T_{sort}(N/P) = O(N/P \log N/P) \quad (4.2)$$

3. Gathering and merging phase
   a. Communication cost in gathering lists (conquer phase)

$$T_{comm2} = t_s \log P + t_w \sum_{i=1}^{\log P} \frac{N}{2^i} = t_s \log P + N t_w \quad (4.3)$$

Thus the total communication cost for scatter and gather will be:

$$T_{comm} = T_{comm1} + T_{comm2} = 2(t_s \log P + N t_w) \quad (4.4)$$

   b. Merging data T_{comp}

$$T_{comp} = \sum_{i=1}^{\log P} 2^i \left(\frac{N}{P}\right) - 1 = \frac{2^{P+1}-1}{P} N - P$$

$$T_{comp} \approx 2N, \quad where\ N >> P \quad (4.5)$$

Thus, the total parallel time

$$T_P(n) = T_{sort}(N/P) + 2(t_s \log P + N t_w) + 2N \quad (4.6)$$

4.1.2. Two Directional Mergesort on CPUs (2DPMS-CPU)

The main difference between the trivial one directional parallel mergesorting algorithm on multi-core systems 1DPMS-CPU and our hybrid two directional parallel mergesorting algorithm 2DPMS-CPU is in the merging phase. While the 1DPMS-CPU algorithm start filling the destination sorted array from one direction i.e., head to tail (see Fig 4.2). On the other hand, our 2DPMS-CPU algorithm start from both ends of the array simultaneously. This way is accomplished using multi-threading on shared memory paradigm with OpenMP. This technique is realized using two threads; one thread is responsible of merging the array from head to middle of the destination array and the second from the tail

to the middle of the destination array simultaneously (the difference can be seen in lowest part of Gather and merge phase 'thick arrows' in Fig 4.2 and Fig 4.4) . Fig 4.4 depicts the process of 2DPMS-CPU. In Fig 4.2 and Fig 4.4, the upper part (solid) arrows represent partioning the array into sub-arrays and scattering them to processes, while the lower parts of arrows represents threads for merging the sorted sub-arrays.

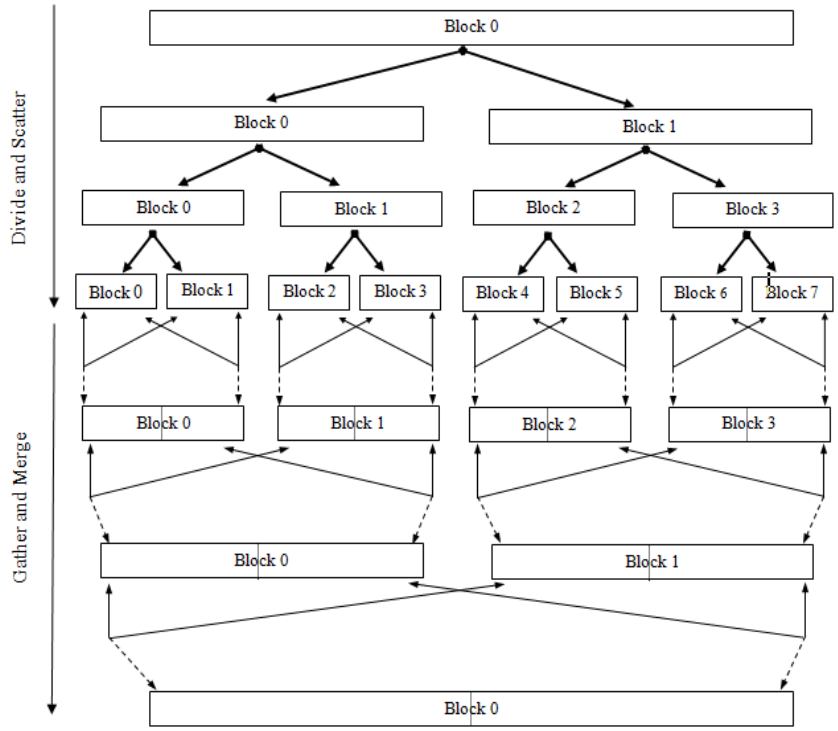


Figure 4.4: 2DPMS-CPU Mergesort Design.

Fig 4.5 shows the pseudocode of our two directional merging process (2DPMS-CPU) i.e., the modified one directional shown in Fig 4.3.

1. **Algorithm:** Merge (S1[0..k-1], S2[0..m-1], S[0..k+m-1])
   // Merges two sorted lists into one sorted list/array using
   //two threads
   // Input: Sorted lists S1[0..k-1], S2[0..m-1]
   // Output: Sorted lists S[0..k+m-1] of the elements of S1
   // and S2

Begin

   //Thread 1 operations

2. **i←0, j←0, q←0**
3. **While** q < (k+m-1)/2 **Do**

```
4.    If  S1[i] ≤ S2[j]
5.        S[q] = S1[i]        i ← i+1
6.    Else
7.        S[q] = S2[j]        j ← j+1
8.    q ← q+1
// Thread 2 operations
9.  ii←k-1, jj←m-1, qq←k+m-1
10. While  q > (k+m+1)/2   Do
11.    If  S1[ii] ≤ S2[jj]
12.        S[qq] = S2[jj]        jj ← jj -1
13.    Else
14.        S[qq] = S1[ii]        ii ← ii -1
15.    qq ← qq – 1
16. End
```

Figure 4.5: The Pseudocode for 2DPMS-CPU Algorithm.

Analysis of (2DPMS-CPU)

The complexity of our approach 2DPMS-CPU is similar to the 1DPMS except for the merging process, while the parallel running time cost (complexity) of the 1DPMS-CPU (see section 4.1.1 Analysis of (1DPMS-CPU)) is:

$$T_P(N) = T_{sort}(N/P) + 2(t_s \log P + N t_w) + 2N \quad \text{4.7)}$$

The parallel running time cost of our approach 2DPMS-CPU is:

1. Communication cost in the distribution/division phase.

$$T_{comm1} = t_s \log P + t_w \cdot \sum_{i=1}^{\log P} \frac{P}{2^i} = t_s \log P + N t_w \quad \text{(4.8)}$$

Where t_s is the startup up time and t_w is data transfer time.

2. Sorting local data using quick sort:

$$T_{sort}(N/P) = O(N/P \log N/P) \quad \text{(4.9)}$$

3. Gathering and merging phase
    a. Communication cost in gathering lists (conquer phase)

$$T_{comm2} = t_s \log P + t_w \sum_{i=1}^{\log P} \frac{N}{2^i} = t_s \log P + N t_w \quad (4.10)$$

Thus the total communication cost for scatter and gather will be:

$$T_{comm} = T_{comm1} + T_{comm2} = 2(t_s \log P + N t_w) \quad (4.11)$$

b. Merging data T_{comp}

$$T_{comp} = (\sum_{i=1}^{\log P} 2^i (\frac{N}{P}) - 1)/2 = (\frac{2^{P+1}-1}{P} N - P)/2$$

$$T_{comp} \approx N, \quad where\ N >> P \quad (4.12)$$

Thus, the total parallel time

$$T_P(N) = T_{sort}(N/P) + 2(t_s \log P + N t_w) + N \quad (4.13)$$

Thus the difference is in the last term which is N (eq.4.13), whilst in the 1DPMS-CPU is 2N (eq.4.7). It should be noted that the dominant cost is in scattering, local cost and gathering.

4.2. Parallel Mergesort on GPU

In this section we present the trivial one directional and our two directional parallel mergesort on Graphical Processing Units GPUs.

4.2.1. One Directional Parallel MergeSort on GPU (1DPMS-GPU)

The original one directional parallel mergesort on GPU (1DPMS-GPU) accomplished by partitioning the input array into equal sized chunks or blocks to be sorted on shared memory of GPU. This is done using parallel sorting networks algorithms such as the Batcher's Bitonic (Batcher, 1968) or the Odd-Even mergesort (Rüb, 1998) which are very efficient for small sizes. In our implementation we used the batcher's odd-even mergesort since it is faster than the bitonic sort.

After each block has been sorted in parallel, the merging phase of the sorted blocks is started. In this phase each thread is responsible on merging every two adjacent odd-even blocks into another array.

The merging process is dependent on the idea of comparing each two elements from the odd-even blocks starting from the start of the two blocks: if the first element from the even

block is smaller than the first one from the odd block then this element is copied to the corresponding output array and the even index is increased and the second element from the even block is compared to the same element from the odd block. The same procedure continues until the odd element become larger than the even block element, then the element is copied and the odd index increased.

This procedure continues until all elements of the odd and even blocks are copied to the output block. In some cases all block elements are copied to the output block and the other block elements still not copied to the output array. This is because they are larger than the copied ones. Thus, the rest of un-copied elements are moved to the output array directly. Fig 4.6 shows the 1DPMS-GPU design. Note that in our implementation we didn't scatter data to processes as in the work on CPU. Instead, data is virtually divided into equal sized blocks. Every group of threads is responsible of a single block of data. The number of threads is equal to the half size of the block. Then, the block is transferred to the shared memory for sorting. After that, the sorted data is returns to the global memory and every odd and even block are merged by one new/distinct thread.

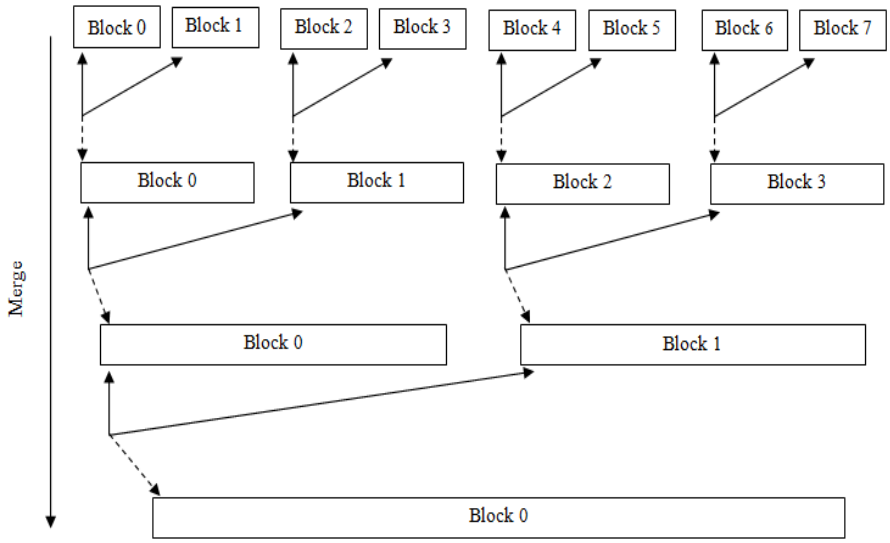


Figure 4.6: 1DPMS-GPU Mergesort Design.

The pseudocode for the 1DPMS-GPU algorithm is the same as one of 1DPMS-CPU in Fig 4.3.

Analysis of (1DPMS-GPU)

Now we will analyze our algorithm running time. In 1DPMS, the work goes in three phases; in the first phase the blocks of data in shared memory are sorted using the odd-

even sorting. In the second phase the sorted blocks are merged using 1DPMS-GPU. It should be noted that the merging process on the GPU is not log N steps since not all merging steps are done on GPU, the 1DPMS-GPU merges 'm' steps until the number of threads per block is less than 32. Finally, in the third phase, the merging process is completed on the CPU. This is because the GPU will work efficiently if there is no warp divergence, this divergence happens if a kernel launch blocks of threads less than 32 thread in each block. Thus, in the mergesort algorithm, the divergence happens in the last phases of merging.

For phase 1: the odd-even merge sort is based on a bitonic sorting network, so for a sequence of size Z which is the size of shared memory, it proceeds with O(*log*Z) phases. Thus, the total computation and memory transfer time is:

$$T_{comp} \approx \sum_{i=1}^{\log Z} 2^i. \quad (4.14)$$

$$T_{mt} \approx \frac{2Z}{C} \quad (4.15)$$

where T_{mt} is the time for transferring data from global memory to shared memory and vice versa in parallel and C is the memory access width. This is because the data is transferred from global to shared memory in chunks of maximum size C.

For phase 2 and phase 3, the N/Z subsequences are merged in a parallel pair-wise merge tree of specific number of levels 'm' on the GPU and log (N/Z) –m on CPU. Thus the total computation time for the hybrid merging on GPU (1DPMS-GPU) and CPU (1DPMS-CPU) is:

$$T_{comp} \approx \sum_{i=1}^{\log \frac{N}{Z}} 2^i Z. \quad (4.16)$$

By combining times of all phases, the total time is:

$$T_{comp} \approx \sum_{i=1}^{\log Z} 2^i + \sum_{i=1}^{\log \frac{N}{Z}} 2^i Z.$$

$$T_{mt} = \frac{2Z}{C}$$

In our implementation, we included the time of transferring the data to and from the GPU (T_{CG}), so the total work of 1DPMS-GPU including the start up time at each level is:

$$T_P \approx \sum_{i=1}^{\log Z} 2^i + \sum_{i=1}^{\log \frac{N}{Z}} 2^i Z + \frac{2Z}{C} + 2T_{CG} + t_s \log N \quad (4.17)$$

4.2.2. Two Directional Parallel Mergesort on GPU (2DPMS-GPU)

In our mergesort algorithm which is called (Two Directional mergesort), the first phase of sorting is the same as 1DPMS-GPU in which each equal sized block is sorted with batcher's odd-even mergesort in parallel. After that, every two odd-even blocks are merged in parallel with two threads. The first thread is responsible on merging starting from the left of the two blocks filling the output array from left to the middle of that array.

The second thread starts filling the output array from the right/end to the middle by selecting the larger element from the sorted blocks by comparing their element from the end; Fig 4.7 depicts the merging process of the 2DPMS-GPU.

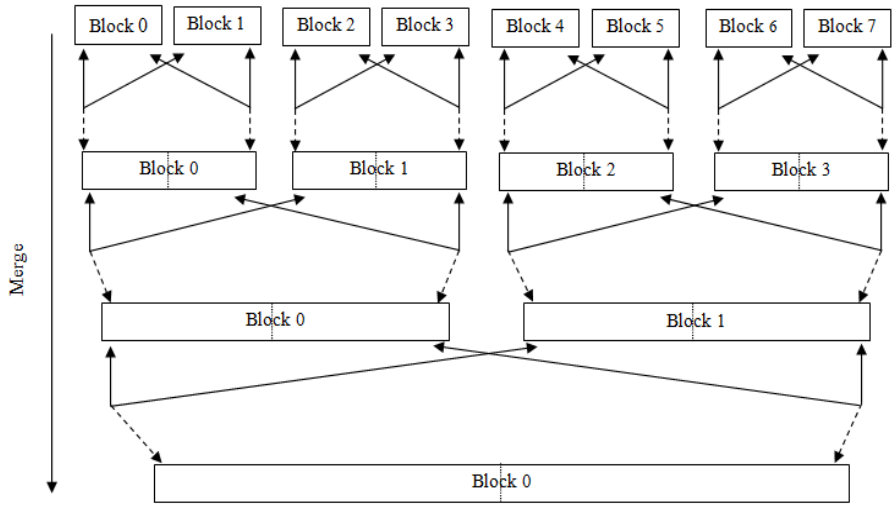


Figure 4.7: 2DPMS-GPU Mergesort Design.

The pseudocode for the 2DPMS-GPU algorithm is the same as for 2DPMS-CPU in Fig 4.5. More, Fig 4.8 shows the pseudocode of our 2DPMS-GPU implementation.

1. Pre-define the size of array (N), size of shared memory (Z) and number of chunks to be merged.
2. Initialize host (CPU) and device (GPU) arrays.
3. Fill the target 'host' array with random integer numbers.
4. Transfer data from host to device.

/*……Sorting Phase 1 on GPU……..……..*/

5. Sort on shared memory each chunk of numbers in the array

in parallel using:

6. Number of thread blocks = number of chunks.
7. Number of threads in each block = half the size of each shared memory.

/*…… End of Phase 1……..……..*/

/*…….. Sorting Phase 2 on GPU…….……*/

8. **If** (N/Z) < 32 then
9. Transfer data to host.
10. Go to phase 3.
11. **End If**

12. **If** 32 < (N/Z) < 1024 then
13. recursion_times = log (N/Z).
14. **while** recursion_times > 5 do
15. Call 2D merge_sort(array) to merge every odd-even chunks using:
16. Number of thread blocks = 1.
17. Number of threads in each block = number of chunks.
18. Halving the number of threads.
19. Decrease recursion_times.
20. Double merged size.
21. **EndWhile.**
22. Transfer data to host.
23. Go to phase 3.
24. **End If.**

25. **If** (N/Z) > =1024 then
26. recursion_times = 5.
27. **While** recursion_times > 1 do
28. Call 2D merge_sort(array) to merge every odd-even chunks using:

```
29.          Number of thread blocks =number of chunks/ 1024.
30.          Number of threads in each block = 1024.
31.          Halving the number of threads.
32.          Decrease recursion_times.
33.    EndWhile.
34.     Transfer data to host.
35.     Go to phase 3.
36.  End If.
     /*……..… End of Phase 2…….……..*/
     /*……. Sorting Phase 3 on CPU……*/
37.  If  (N/Z) < 32 then
38.    looping _times = log (N/Z).
39.    number of threads =(N/Z)/ 2;
40.    For counter = 0 to looping _times
41.          Merge chunks in parallel on host:
42.          Merge left half part
43.          Merge right half part.
44.          Halve the number of threads.
45.          Double merged size.
46.     End For
47.  End If.

48.  If  32 < (N/Z)< 1024 then
49.     looping_times = 5.
50.     number of threads =16.
51.    For counter = 0 to looping _times
52.        Merge chunks in parallel on host:
53.        Merge left half part
54.        Merge right half part.
55.        Halve the number of threads.
56.        Double merged size.
57.    End for
58.  End If.
```

```
59. If size > =1024 then
60.     number of threads_A =number of chunks /1024;
61.     Do in parallel with number of threads_A :
62.         Looping_times = 5.
63.         number of threads_B =16.
64.         For counter = 0 to looping _times
65.             Merge chunks in parallel on host:
66.             Merge left half part
67.             Merge right half part.
68.             Halve the number of threads_B.
69.             Double merged size.
70.         End For
71.    End Do
72. looping_times = log (number of chunks / 1024).
73. Number of threads = (number of chunks / 1024) / 2.
74. For counter = 0 to looping _times
75.       Merge chunks in parallel on host:
76.       Merge left half part
77.       Merge right half part.
78.       Halve the number of threads.
79.       Double merged size.
80.  End for
    /…….…… End of Phase 3……..……..*/
81.Flush host (CPU) and device (GPU) arrays.
82.Reset device.
```

Figure 4.8: Pseudo-code for 2DPMS-GPU.

Analysis of (2DPMS-GPU)

The complexity of our approach 2DPMS-GPU is similar to the 1DPMS except for the merging process, while the parallel merging running time cost (complexity) of the 1DPMS-GPU is:

$$T_{comp} \approx \sum_{i=1}^{\log \frac{N}{Z}} 2^i Z.$$

Since the merging process of our 2DPMS-GPU is done in two directions the computational time becomes approximately half of that in the 1DPMS-GPU.)

$$T_{comp} \approx (\sum_{i=1}^{\log_Z^N} 2^i Z)/2 \quad (4.18)$$

Thus, the parallel running time cost (complexity) of the 2DPMS-GPU is:

$$T_P \approx \sum_{i=1}^{\log Z} 2^i + (\sum_{i=1}^{\log_Z^N} 2^i Z)/2 + \frac{2Z}{C} + 2T_{CG} \quad (4.19)$$

In the next chapter is the experimental results and the discussion of these results.

Chapter 5

Experimental Results and Discussion/Analysis

Now we present and discuss the results of our experiments. The experiments are divided into two main parts. The first part presents the results efficiency of our 'two directional' parallel mergesort on message passing and multi-threading compared with the original 'one directional' approach and with the serial sort. In the second part we discuss the results of our 'two directional' approach on GPU compared with the original 'one directional' method, with the serial sort and with the mergesort mentioned in (Satish, Harris, & Garland, 23-29 May 2009), all done on the GPU.

To ensure that our results are reproducible we describe the platform/environment (i.e., hardware and software) setup as well as the utilized methods and algorithms to generate input data samples for the carried out experiments in each part.

5.1. Experiments Platform

Since the results of the experiments are measured in time, hence, they are very dependent on the used platform i.e., hardware and the software. Therefore in the next section we describe the platform specifications.

5.1.1. Hardware Setup

Our test hardware consist of an Intel® Core™ i7-4702 MQ CPU at rate of 2.2 GHz with 4.00 GB RAM memory.

The Display card (GPU) used to run our program in the second part is Nvidia GeForce GT 740M of family GK208, based on the Kepler architecture which is Cuda capable GPU. It have two SMs all of 384 Cuda cores, driver type is WDDM , PCI Express 3.0 bus support

with bandwidth of 7.336 GB/s, DDR3 RAM of 2048 MB using a 64-bit memory interface. The shared memory per thread block is maximum of 48 KB, the compute capability is 3.5, L2 cache memory of 512, graphics and processor clock of 1032.5 MHz and memory clock of 900 MHz.

The used GPU supports 1024 threads per block in the X and Y axis and 64 in the Z axis, 2147483647 blocks in the X axis and 65535 in the Y and Z axis, and with warp size of 32 threads.

5.1.2. Software Setup

The system runs Windows 10 Pro of 64 bit operating system. The codes are compiled using Visual C++ 12.0 compiler in Visual Studio 2013 Ultimate with runtime environment. For implementation of Part 1 experiments we used the MPICH2 software/library version 1.4.1p1 to enable the message passing and the OpenMP library for multi-threading on CPU with visual studio and both APIs are implemented with C language. For implementation of Part 2 experiments we used Cuda runtime API implemented with C language, the installed Nvidia Cuda Toolkit version is 7.5 and the Cuda driver for the GPU. Also, we used the Nsight software that is embedded in the Visual studio to estimate the performance issues for Cuda program.

5.1.3. Input Data

We used a uniform random number generator to produce random integers whose lengths range from 2^5 elements to 2^{25} elements, only of power-of-2 sizes. Since sorting is frequently the most important as one building block of larger-scale computation, the data is generated by a function on the CPU and the resulting sorted array will be consumed by a function on the CPU.

5.2. Part 1: Two Directional Parallel Mergesort on CPU using MPI and Multi-threading (2DPMS-CPU)

In this section we introduce and discuss the obtained results by means of the performance of our two directional parallel mergesorting algorithm on multi-core CPUs using message passing with MPICH2 and shared memory with OpenMP. We compare these results with the performance of the original/trivial one directional parallel mergesorting algorithm using only message passing paradigm with MPICH2 and the sequential quicksorting algorithm. Therefore, the figures in this part show the running time in milli-seconds and

the speed up of these algorithms. Where the speed up is well known term in parallel processing which is the ration between the performance of the best know sequential algorithm to the performance of parallel algorithm (Sharma & Gupta, 2012) i.e.,

$$\text{Speed up} = \frac{T_{sequential}}{T_{parallel}} \quad 5.1)$$

5.2.1. Work Methodology

In this experimental work we tested the performance of the compared algorithms using different data sizes. Also, we ran the experiments using different number of processes. For each experiment with different combination of data sizes and number of processes, we ran the program more than 15 times for each case. Then we recorded the average running time of these experiments.

5.2.2. Results and Discussion

We present and discuss the experimental results and evaluating the performance of our two directional parallel mergesorting algorithm (2DPMS-CPU) and the one directional (1DPMS-CPU) running on multi-core CPU using message passing interface paradigm (MPICH2). However, in our 2DPMS-CPU algorithm we utilize shared memory paradigm for merging process (OpenMP). Also we compare these results with the performance of the sequential (serial) quick sorting algorithm.

Now, we present the average running time for different reasonable data sizes by number of processes for some obtained results in the figures (Fig 5.1 to Fig 5.18) for 1DPMS-CPU, 2DPMS-CPU and sequential sort. The elapsed running time is measured in milliseconds. We just show the average time for the different recorded running times and the speed up at each data size by number of processes, the complete tables of results and figures are shown in appendix I.

It should be noted that figures show the results of the experiment (running time) for reasonable data sizes (from 256 to 33554432) elements and number of processes (from 2 to 32 processes). Also, these results are very dependent on the utilized platform by means of frequency, number of cores and threads, main memory, cache memory and bandwidth. Note that figures for 2DPMS-CPU and 1DPMS-CPU for each data size (number of processes versus Elapsed time) are shown in Appendix I (from Fig 8.1 to Fig 8.20) and the results are obtained for the running time in millisecond. When sorting very small data

sizes, it is clear that the sequential algorithm performs better than both parallel algorithms. This is because of the overheads of managing the processes and the threads. As a result, we can't obtain a speed up for small sizes of data and it is better to do the sorting using sequential algorithms where the parallel sorting is better to do with large data sizes.

In Fig 5.1, when the array size is 256, we can see that the 1DPMS-CPU is almost better than the 2DPMS-CPU between 4 and 16 processes, but the sequential performs better than both parallel algorithms from 8 to 32 processes.

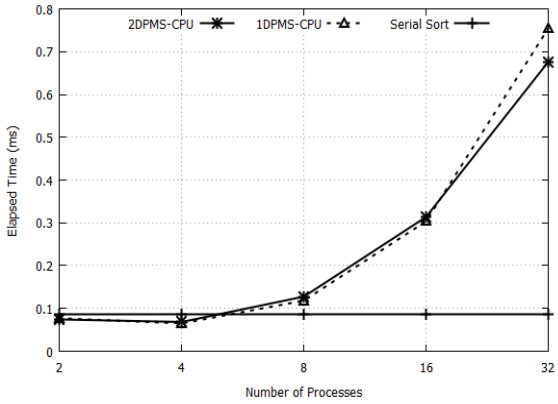


Figure 5.1: Running Time for Data Size 256.

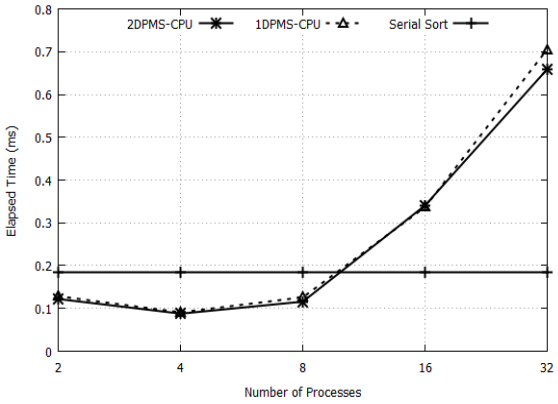


Figure 5.2: Running Time for Data Size 512.

In Fig 5.2, where the array size is 512, we can see that the 2DPMS-CPU is almost better than the 1DPMS-CPU. However, both parallel algorithms perform better than the sequential before 8 processes.

In Fig 5.3, where the array size is 1024, the 2DPMS-CPU is better than 1DPMS-CPU at all number of processes where both 1DPMS-CPU and 2DPMS-CPU perform better than the sequential sort except at 32 processes.

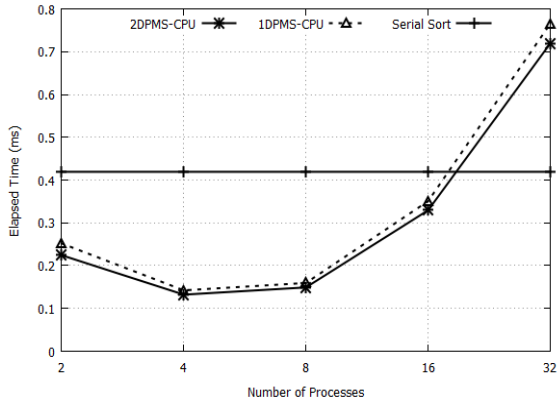


Figure 5.3: Running Time for Data Size 1024.

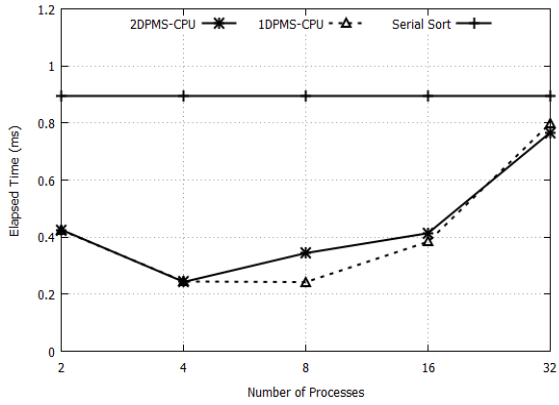


Figure 5.4: Running Time for Data Size 2048.

In Fig 5.4, where the array size is 2048, the 2DPMS-CPU is little better than 1DPMS-CPU except at 8 and 16 processes where both 1DPMS-CPU and 2DPMS-CPU perform better than the sequential sort at all processes.

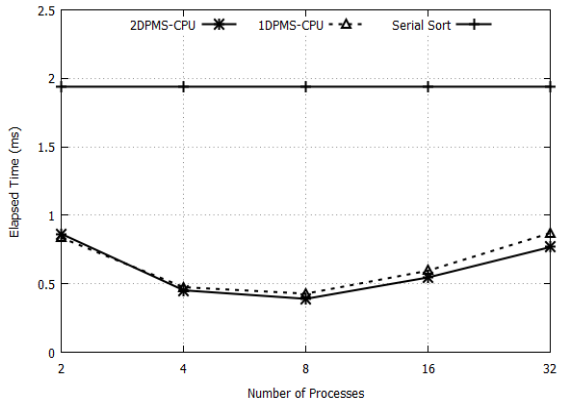


Figure 5.5: Running Time for Data Size 4096.

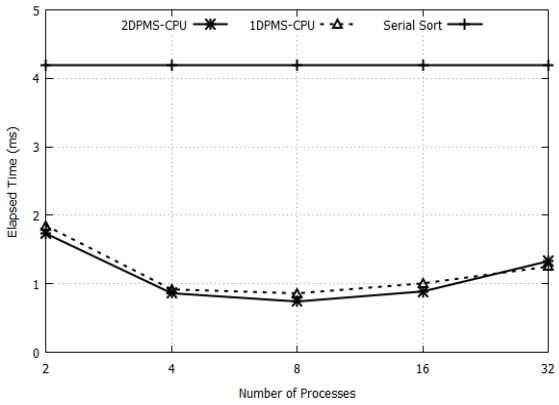


Figure 5.6: Running Time for Data Size 8192.

Now from Fig 5.5 up to Fig 5.18, the 2DPMS-CPU mostly performs better than the 1DPMS-CPU for sizes from 4096 to 33554432.

However, in both algorithms the best running time is at 8 processes (the number of threads of our CPU with hyperthreading) which is the best number of processes to test the algorithm, .So at small data sizes, it is best to do the parallelism with small number of processes. Since increasing the number of processes causes overheads of managing processes and threads. This is due to context switching where the 2DPMS-CPU mostly performs better than 1DPMS-CPU. At large data sizes, also the best is to do the parallelism with 8 processes and then with 16 and 32 processes. Since increasing the number of processes provides enough parallelism. But increasing the processes causes overheads of managing processes. On the other hand, if we decrease the number of processes then there will be no enough parallelism. Therefore, 2DPMS-CPU always performs better than the 1DPMS-CPU at these sizes. It should be mentioned that the provided analysis is applied for the used platform of the experiments.

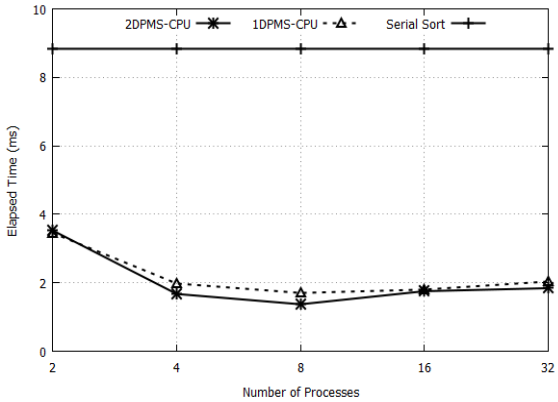


Figure 5.7: Running Time for Data Size 16384.

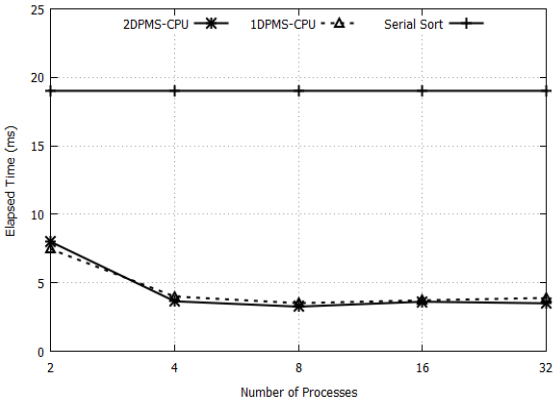


Figure 5.8: Running Time for Data Size 32768.

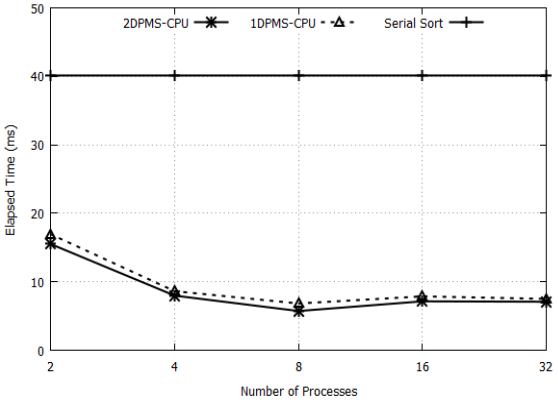


Figure 5.9: Running Time for Data Size 65536.

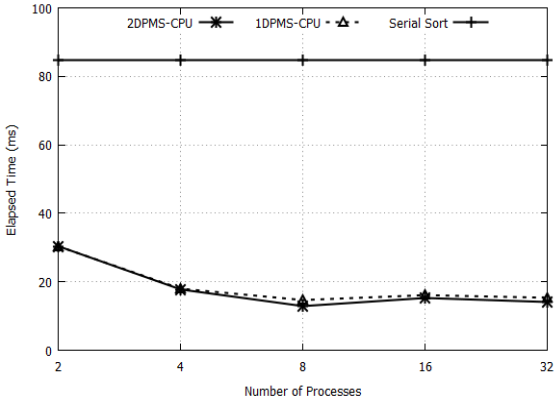


Figure 5.10: Running Time for Data Size 131072.

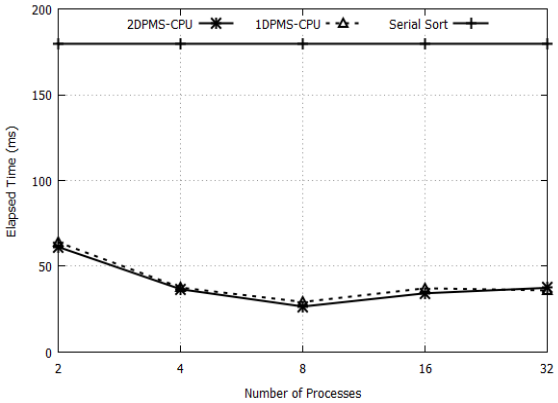


Figure 5.11: Running Time for Data Size 262144.

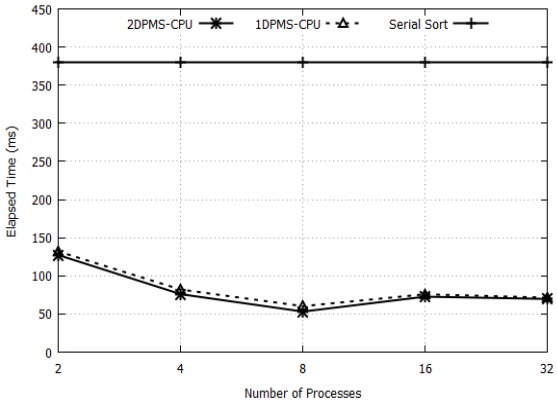


Figure 5.12: Running Time for Data Size 524288.

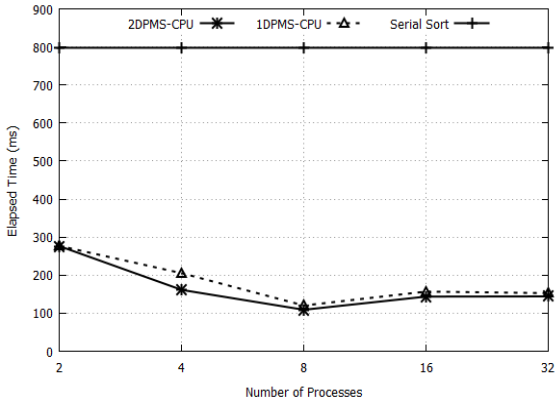


Figure 5.13: Running Time for Data Size 1048576 .

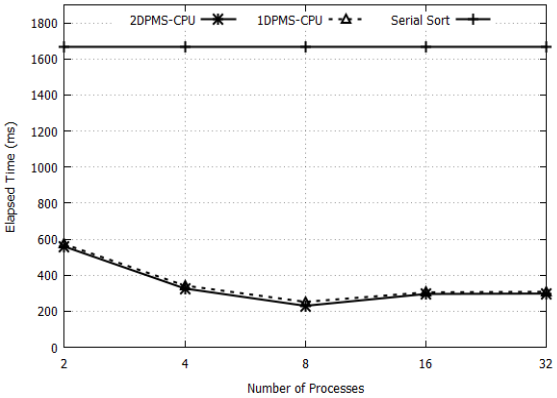


Figure 5.14: Running Time for Data Size 2097152.

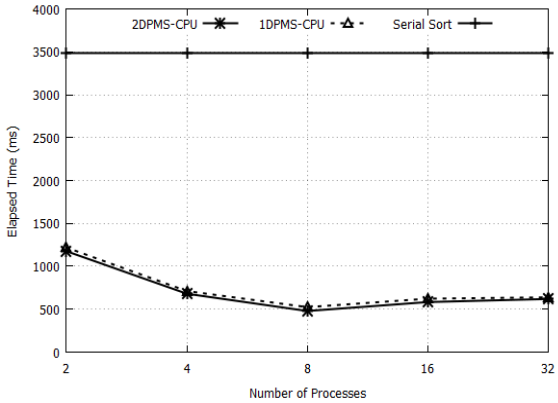


Figure 5.15: Running Time for Data Size 4194304.

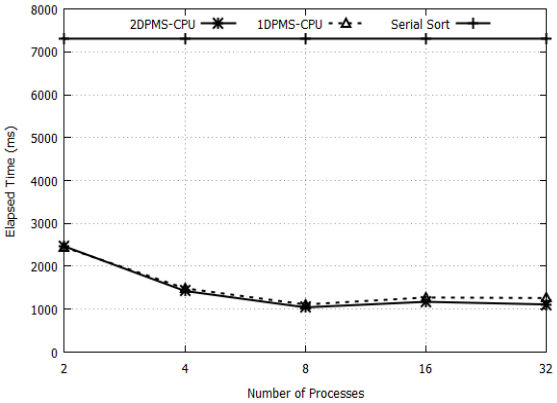


Figure 5.16: Running Time for Data Size 8388608.

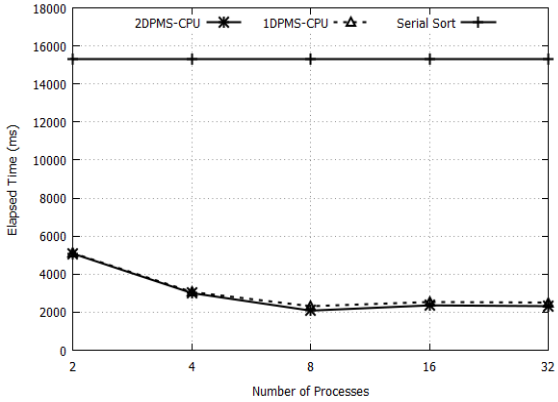


Figure 5.17: Running Time for Data Size16777216.

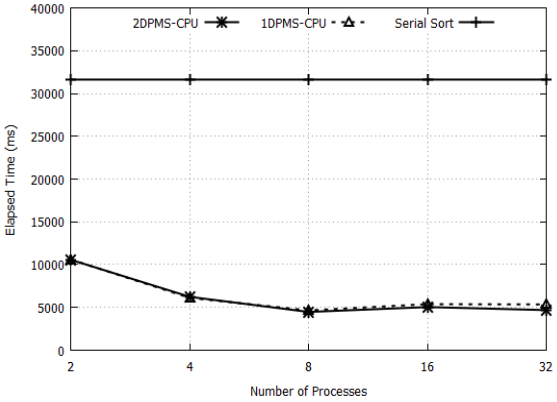


Figure 5.18: Running Time for Data Size 33554432.

Now we will examine the speed up of 2DPMS-CPU and 1DPMS-CPU each over its corrosponding sequential times for different sizes, the speed up of 2DPMS-CPU over sequential is the value of sequential time divided by the 2DPMS-CPU time and the same for 1DPMS-CPU,it is a measure of the percent of enhancment done on the time between the two compared values.

When the value of speed up is 1, this means that the parallel and sequential times are the same, if greater than 1; it means that parallel time is less than sequential, or in other words, the parallel is more fast than sequential. If the speed up less than 1, the sequential is more fast than parallel.

For small data sizes that are less than 2048, the parallel mergesort is not efficient which have speed ups less than 1, but after that and at large data sizes, the 2DPMS-CPU at most cases have speed ups more than 1DPMS-CPU and all are more than 1 specialy at 8 processes.

Fig 5.19, Fig 5.20, Fig 5.21 and Fig 5.22 show the 2DPMS-CPU speed ups at different data sizes. We can see that in general the best speed up happens at 8 processes followed by 16 and 32 and the worst is at 4 and 2 processes. The best speed ups occure with 8 processes are at sizes 16777216, 1048576, 4194304, 2097152 , 524288, 33554432, 65536 and 8388608 respectively that have speed up more than 7.

The speed ups of more than 6.5 and less than 7 occure at sizes 33554432, 16777216 and 8388608 with 32 processes , then the sizes of 16777216 with 16 processes. So the best speed up with largest size is at 16777216 with 8 processes, and after this size and for large data sets the performance decreases since we have more overheads in swapping data between main memory and cahe memory which is bandwidth limitation and bottleneck.

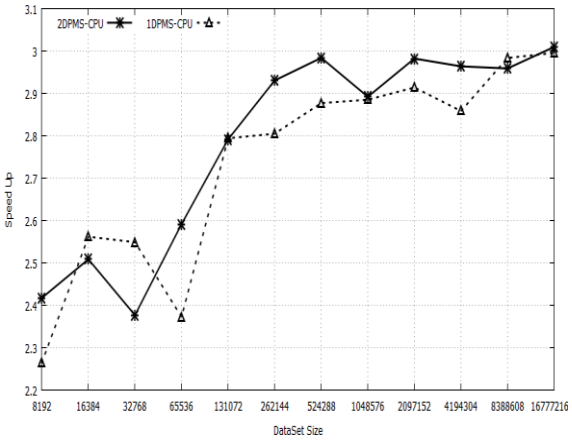


Figure 5.19: Speedup on 2 Processes by Data Size.

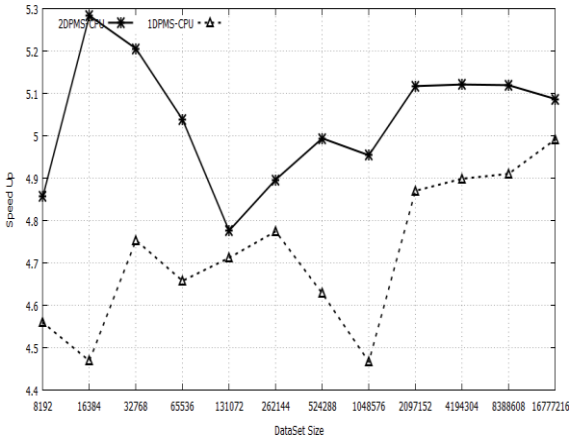


Figure 5.20: Speedup on 4 Processes by Data Size.

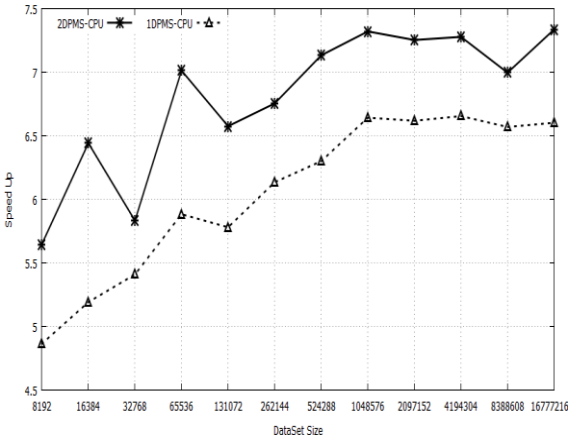


Figure 5.21: Speedup on 8 Processes by Data Size.

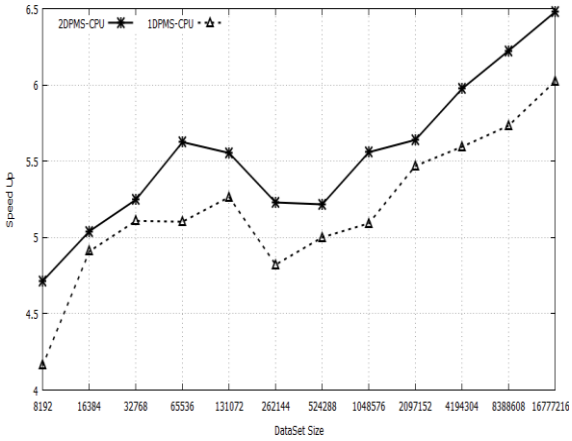


Figure 5.22: Speedup on 16 Processes by Data Size.

5.2.3 Analysis of 2DPMS-CPU

It was expected that the 2DPMS-CPU will have half the time of the 1DPMS-CPU since for every odd-even chunks there are two concurent threads each do half of the work of the original one thread in the 1DPMS-CPU in the parallel, but in practice that is not the case on CPU, since we have communication time added to the total running time such as the startup time of processes and the additional threads in our algorithm**.**

Also we can see that the 2DPMS-CPU is not efficient at small sizes less than or equals 2048 elements, the performance enhances after this value, the best speed up at these sizes occurs at 4 processes and it decreases as we increase them where the speed up at these sizes mostly less than 1 specially before size of 512 because of the overheads of managing large number of processes and threads and due to context switching at these sizes.

From sizes 16384 until 33554432, the best performance of 2DPMS-CPU is at 8 processes since we have maximum of 8 logical cores for our running i7 CPU in which the CPU works the best at one process for each core in parallel. The performance has tradeoff between 16 and 32 processes since they provide large parallelism where at most cases they have better performance than 2 and 4 processes where there is no enough parallelism.

In our work of 2DPMS-CPU, the experimental results are as expected with the theoretical analysis explained in section 4.1.2; since when N is small, the dominant factor that affects the running time is the communication time, but for large data sizes the dominant factor is the size of array. Also at 8 processes, the 2DPMS-CPU gives the highest speed ups over 1DPMS-CPU.

5.3. Part 2: Two Directional Parallel Merge Sort on GPU (2DPMS-GPU)

In this part we present and analyze the obtained results of our approach using two directional parallel mergesort on GPU platform (2DPMS-GPU). We compare our results with the obtained results for one directional approach (1DPMS-GPU), the result of (Satish et al.) on the GPU and the result of sequential quicksort.

5.3.1. Work Methodology

In this experimental work we tested the performance of the compared algorithms using different data sizes. We ran the program fifteen times for each experiment (to achieve fair average) and then we recorded the average running time of these experiments. In all measured performance values (running time), the time for transferring input to the GPU memory and back to the system main memory across the PCI-E bus is included. The figures in this part show the elapsed time in milli-seconds for different data sizes and different shared memory sizes**,** also we take the times at maximum possible number of chunks and maximum shared memory sizes that we couldn't increase them more due to the limitation of resources.

Recall that N is the total input data size (unordered list) in the experiments on GPU. Note that N equals to the product of the shared memory size S_{sh} times the number of chunks N_{ch}. i.e., $N = S_{sh}*N_{ch}$. Further, the number of chunks N_{ch} equals to the product of chunk base B and an integer number K i.e., $N_{ch} = K*B$. Thus, $N= S_{sh}*K*B$.

In the conducted experiments and for simplicity, all values are of power of 2. The shared memory size S_{sh} ranges from 32 to 2048 and the chunk base B ranges from 2 to 1024. If chunk base B is less than or equal 1024, then the number of chunks must be the chunk base B i.e., $N_{ch} =1*B$. For bigger sizes, the chunk base must be fixed to 1024 and the number of chunks N_{ch} equals the chunk base B multiplied by an integer K, i.e., $N_{ch} =K*B$.

After the data is transferred to the GPU, the first phase of sorting is started. Since mergesort is not so efficient on small sizes, we used batcher's odd even mergesort where it is very efficient parallel sorting algorithm on these sizes. In the first phase, the data is logically divided into chunks of the shared memory size S_{sh} to be sorted in parallel on device, the number of Cuda blocks in this kernel equals to the number of shared memory chunks N_{ch} and the number of Cuda threads N_{th} equals to the size of shared memory divided by two $N_{th} =S_{sh}/2$.

After finishing this phase, the array consists of sorted equal sized chunks that are ready for the second phase to merge them, the second kernel is launched with K' Cuda blocks and with 'B' Cuda threads. Our 2DPMS-GPU is now working and it recursively calls itself for each stage of merging with the same number of thread blocks and with halving the number of threads.

There is an important point to get in mind in which that the GPU will work efficiently if there is no warp divergence, this divergence happens if a kernel launch blocks of threads less than 32 thread in each block. In the mergesort algorithm, the divergence happens in the last phases of merging.

In 2DPMS-GPU, if the chunk base B is less than or equals 1024, we need number of threads N_{th} equals to the chunk base value B and the mergesort will works efficiently until the case that needs 16 threads, for example; if B=1024, then at first phase we will need 1024 threads for our 2DPMS-GPU or 512 for 1DPMS-GPU. In the next phases we need '512', '256', '128', '64', '32', '16', 8', '4', '2' threads. So, the first five phases works efficiently, but the performance degrades at the last five phases. In general, if B is more than 32, the 2DPMS-GPU will merge the sub-lists until reaching the last five phases, it stops and the merge steps are continued in parallel on CPU using OpenMP directives since CPU don't have the warp divergence problem when we need small number of threads, but if the B is less than 32, there is no benefit from merging on GPU and the chunks are sorted on GPU only with odd-even mergesort (phase 1) and then transferred to the CPU to be completely merged in parallel using OpenMP directives. In the parallel OpenMP merging, we used 16 concurrent threads at maximum, also we can't use more than 64 multiples of shared memory chunk bases 'K' because the visual studio with C- programming language can't launch more than 64 OpenMP concurrent threads where the mergesort on CPU for this phase needs K number of threads.

When B=1024 and K>1, then each odd-even chunks (which are 1024 chunks) are merged for five levels and the rest five levels are merged on CPU. After that, the K (multiples of 1024 chunks) are merged on CPU since they needs at maximum 16 threads.

5.3.2. Results and Discussion

Now we present and discuss the experimental results by mean of the performance using samples of the obtained results. More tables and figures of the results can be found in

appendix I. We will focus on the running time and speed up of our technique 2DPMS-GPU compared with the original approaches 1DPMS-GPU and of (Satish et al.) also with the sequential (serial) sort.

At first, we explore the average running time for each possible vast number of different data sizes in the figures (Fig 5.23 to Fig 5.29) for 2DPMS-GPU, 1DPMS-GPU and (Satish et al.), compared with the sequential quicksort. We chose the serial quicksort since it is the fastest sequential sorting algorithm.

In these figures, we chose different shared memory sizes S_{sh} with different number of chunks (blocks) N_{ch}, where the total size of the array to be sorted N is the shared memory sizes S_{sh} multiplied by the number of chunks (the number of shared memory blocks used) i.e., $N = S_{sh} * N_{ch}$. We used different shared memory sizes ranging from 32 elements up to 2048 elements, and the number of chunks ranging from 64 until 65536 chunks where these chunks are merged on the GPU and the chunks from 2 to 32 are merged on CPU. More details specially for figures are shown in appendix I (Fig 8.25 to Fig 8.39).

Now we present the recorded results for 2DPMS-GPU, 1DPMS-GPU, (Satish et al.) and sequential sorting at different shared memory sizes.

Fig 5.23 shows the running time when using shared memory $S_{sh} = 32$. It is clear that both 2DPMS-GPU and 1DPMS-GPU perform better than the sequential sort from $N_{ch} = 256$ to 65536. Also, the 2DPMS-GPU is little better than 1DPMS-GPU from $N_{ch} = 1024$ to 65536. It should be noted that the parallel mergesort of (Satish et al.) doesn't work at this shared memory size.

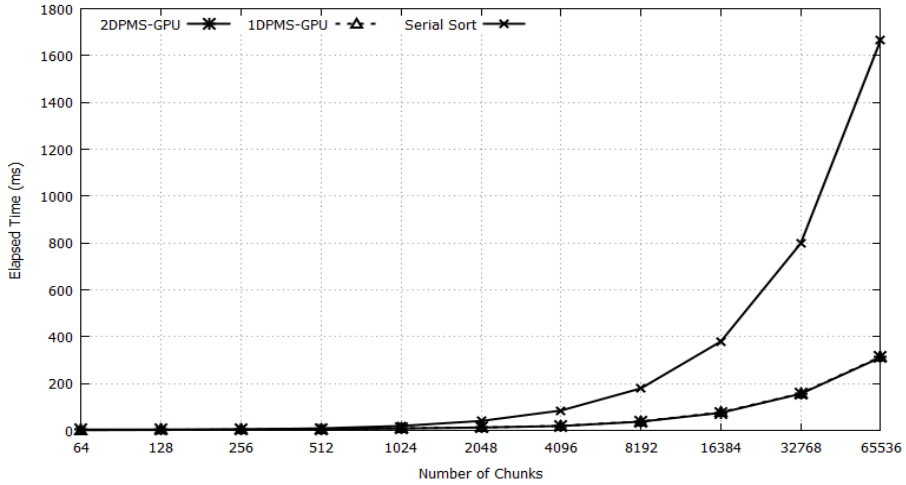


Figure 5.23: Running Time for Shared Memory 32.

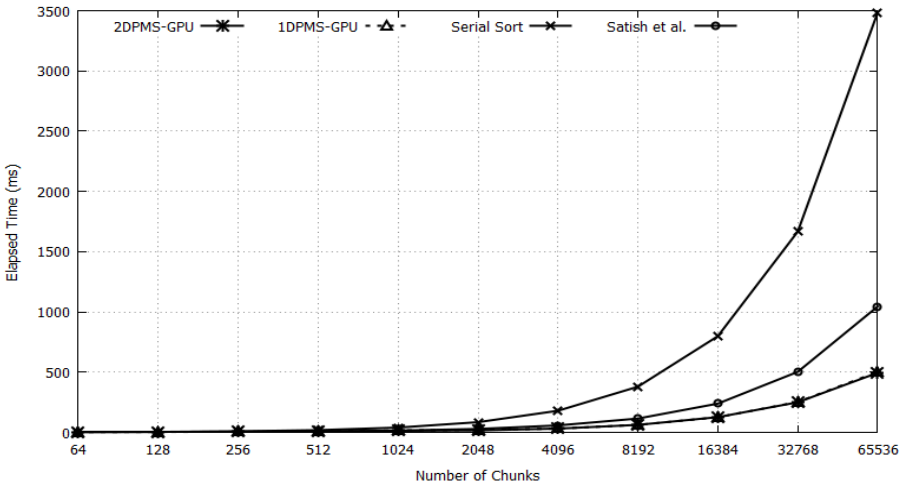


Figure 5.24 : Running Time for Shared Memory 64.

By running the "*Nsight*" software for performance analysis at this shared memory size, we found that the occupancy of the odd-even mergesort for 2DPMS-GPU and 1DPMS-GPU have occupancy of 25%. Also it has bad utilization of GPU to be no more than 43% that happens at 65536 chunks.

In Fig 5.24, when using shared memory size of 64, we can see that 2DPMS-GPU performs better than 1DPMS-GPU for N_{ch} = 512 to 65536. Also, both 2DPMS-GPU and 1DPMS-GPU are better than the sequential sort at N_{ch} = 128 and better than (Satish et al.) merge at all chunks. The utilization of GPU becomes little better and reached at most 61% at last chunks set.

Similarly, in Fig 5.25 when using shared memory of size 128, the 2DPMS-GPU and 1DPMS-GPU perform better than sequential sort from N_{ch} =64 and better than (Satish et al.) merge at all chunks. Note that the 2DPMS-GPU is better than 1DPMS-GPU from N_{ch} = 256 to 2048 and at 65536.

The occupancy of the odd-even merge is improved and reached 50%, where 2DPMS-GPU has occupancy 50% and the GPU utilization becomes near 63% at last chunks set.

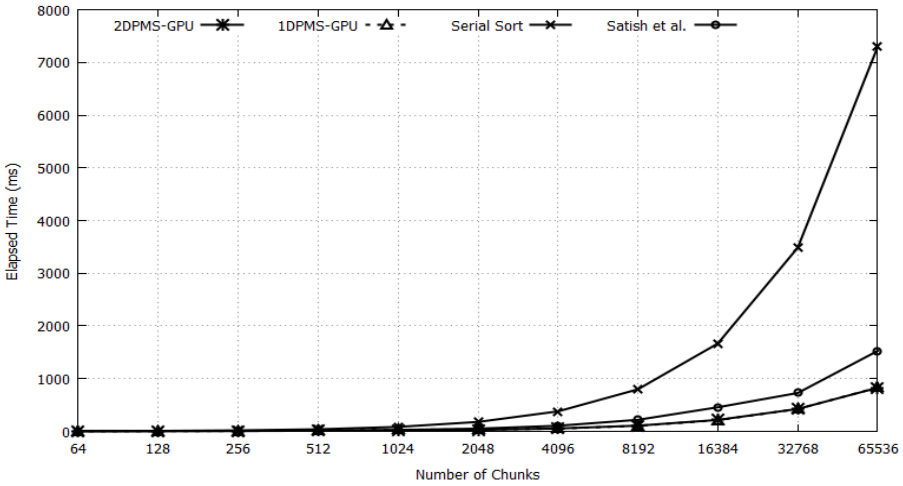


Figure 5.25: Running Time for Shared Memory 128.

Moreover, Fig 5.26 shows the running time when the shared memory size is 256. It could be seen that 2DPMS-GPU and 1DPMS-GPU perform better than sequential sort starting from N_{ch} = 64. Where (Satish et al.) mergesort is better than 2DPMS-GPU from N_{ch}=128 to 1024 only. Also, the 2DPMS-GPU performs better than 1DPMS-GPU from N_{ch}=128 to 2048 only. The occupancy for the odd-even merge enhances to reach 100% where 2DPMS-GPU have 50% and the GPU utilization becomes near 68% at last chunks set.

In Fig 5.27, where the shared memory size is 512, the 2DPMS-GPU and 1DPMS-GPU perform better than the sequential sort from N_{ch} = 64 to 65536, and (Satish et al.) merge performs little better than 2DPMS-GPU from N_{ch} =128 to 1024 only. Note that 2DPMS-GPU performs better than 1DPMS-GPU from N_{ch} =64 to 2048 and at N_{ch} =32768 and 65536. The occupancy of the odd-even merge is 100% where our merge stills 50% and the GPU utilization becomes near 72% at last chunks set.

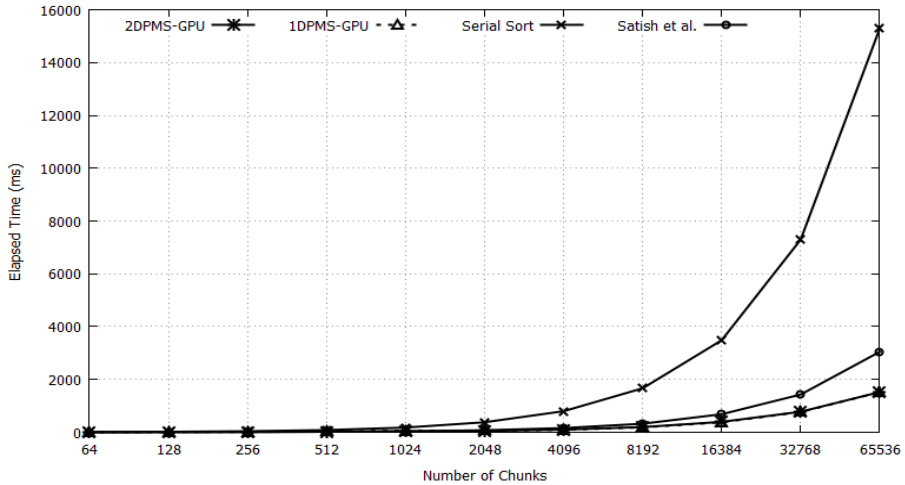


Figure 5.26: Running Time for Shared Memory 256.

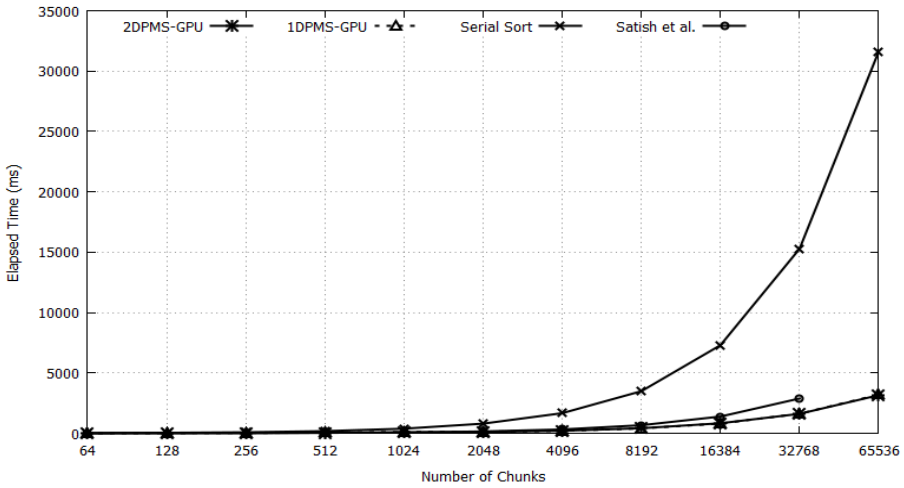


Figure 5.27: Running Time for Shared Memory 512

Fig 5.28, where the shared memory size is 1024, the 2DPMS-GPU and 1DPMS-GPU perform better than sequential sort at all chunks, (Satish et al.) merge is better than 2DPMS-GPU from 128 to 1024 only, the 2DPMS-GPU performs better than 1DPMS-GPU from 64 to 32768 chunks 'all chunks'. The occupancy for the odd-even merge is 100% where our merge still 50% and the GPU utilization becomes near 72% at last chunks set.

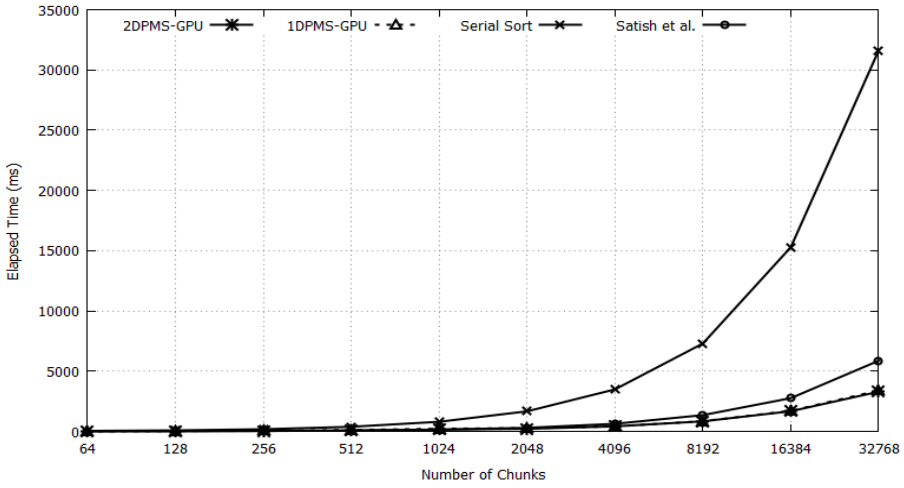


Figure 5.28: Running Time for Shared Memory 1024.

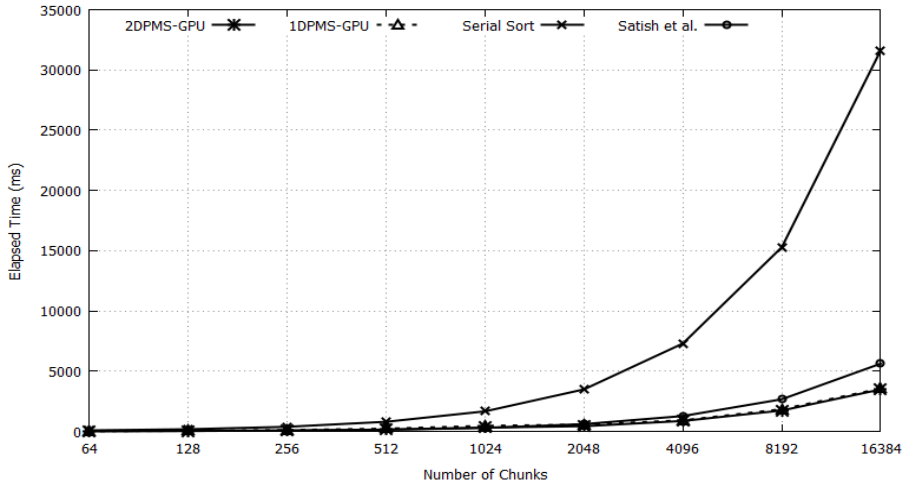


Figure 5.29: Running Time for Shared Memory 2048.

Finally, in Fig 5.29 where the shared memory size is 2048, both 2DPMS-GPU and 1DPMS-GPU perform better than sequential sort at all chunks. 2DPMS-GPU is better than 1DPMS-GPU from N_{ch} = 64 to 16384 chunks 'all chunks' with significant difference. (Satish et al.) mergesort performs better than 2DPMS-GPU from N_{ch} = 128 to 1024 only. The occupancy for the odd-even merge reaches 100%, while our 2DPMS-GPU stills 50% and the GPU utilization almost near 74%.

Now, Each of Fig 5.30 to Fig 5.45 shows the elapsed running time (ms) for a given data size N with 2DPMS-GPU. For any size 'N', there are many possible combinations of shared memory sizes and number of chunks, for example; the size N= 1024, then N_{ch}*S_{sh} : 64*16, 32*32, 16*64, 8*128, 4*256 and 2*512.

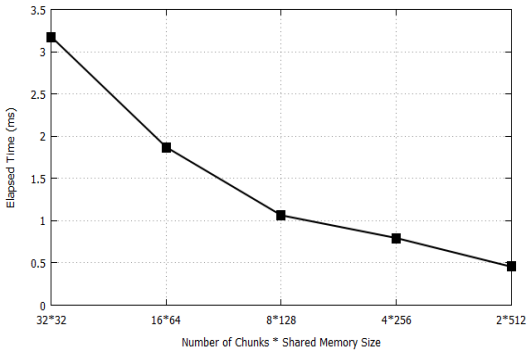


Figure 5.30: Running Time for Size 1024.

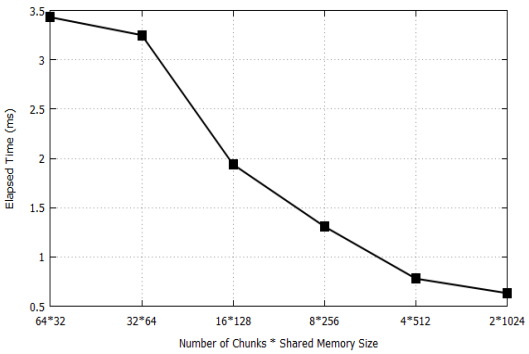


Figure 5.31: Running Time for Size 2048.

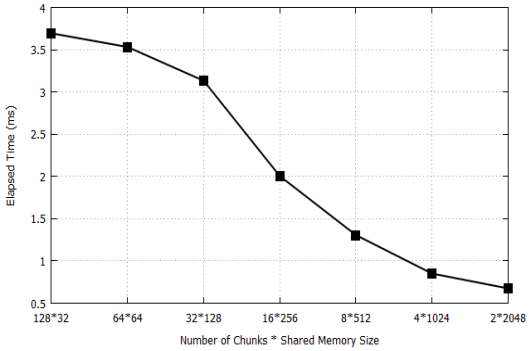


Figure 5.32: Running Time for Size 4096.

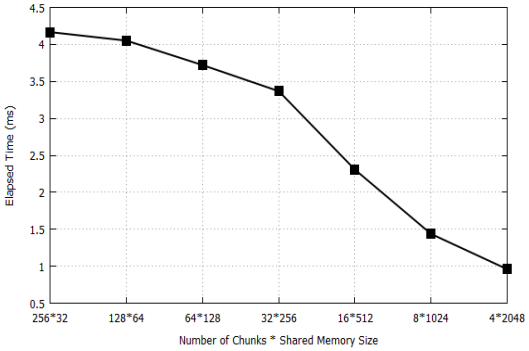


Figure 5.33: Running Time for Size 8192.

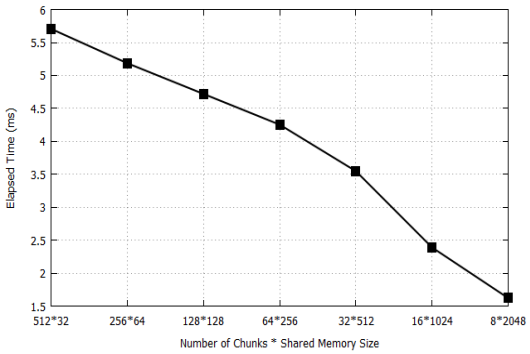


Figure 5.34: Running Time for Size 16384.

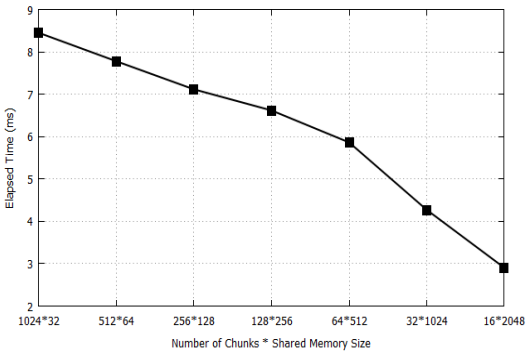


Figure 5.35: Running Time for Size 32768.

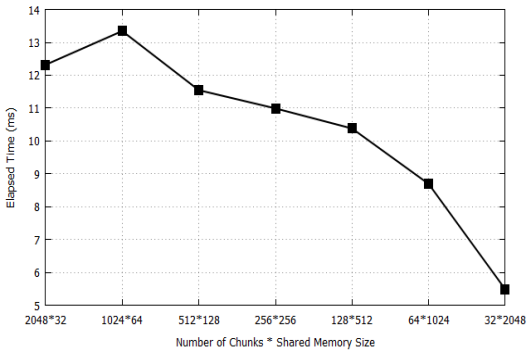


Figure 5.36: Running Time for Size 65536.

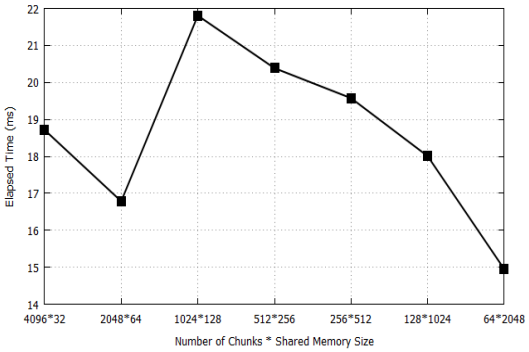


Figure 5.37: Running Time for Size 131072.

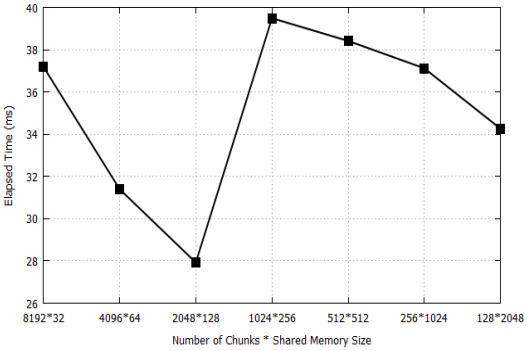


Figure 5.38: Running Time for Size 262144.

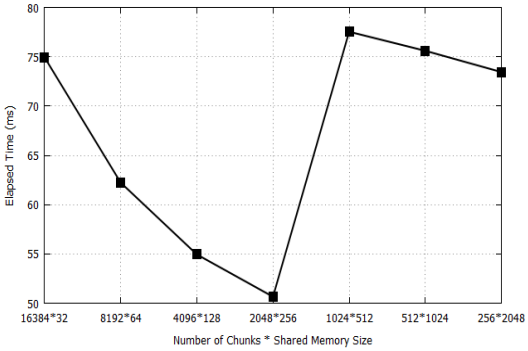


Figure 5.39: Running Time for Size 524288.

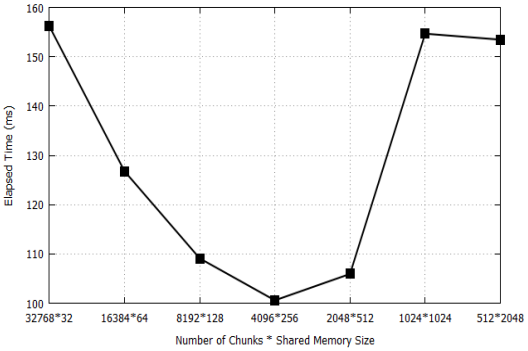


Figure 5.40: Running Time for Size 1048576.

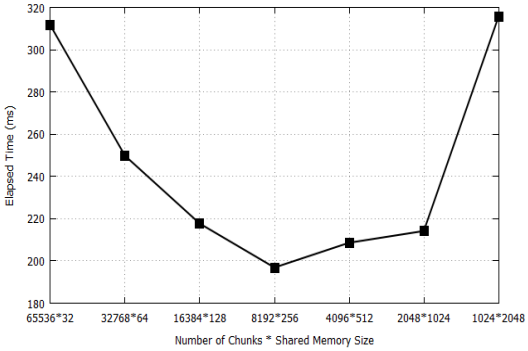


Figure 5.41: Running Time for Size 2097152.

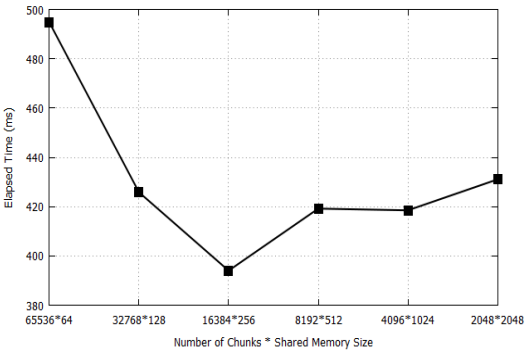


Figure 5.42.3: Running Time for Size 4194304.

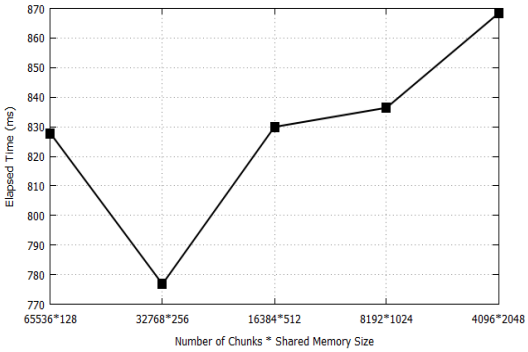


Figure 5.43: Running Time for Size 8388608.

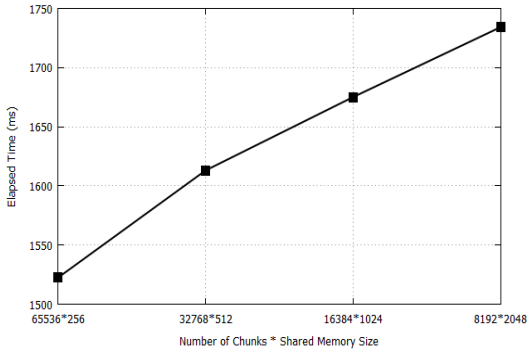


Figure 5.44: Running Time for Size 16777216.

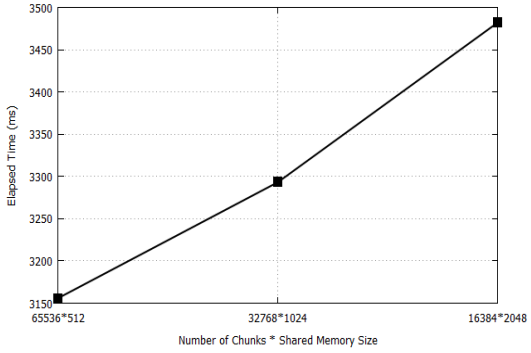


Figure 5.45: Running Time for Size 33554432.

From these results/figures, we can conclude where the best performance can occur at which combination of $N=N_{ch}*S_{sh}$ for a given data size N which shown in Table 5.1.

Table 5.1: Best Performance and Speed Up at Specific Sizes.

N	The best performance at $N_{ch}*S_{sh}$	Speed Up
1024, 2048, 4096	2*(N/2)	0.92, 1.41, 2.88
8192, 16384, 32768, 65536, 131072	(N/ 2048)* 2048	4.36, 5.45, 1.27, 7.33, 5.67
262144	2048*128	6.43
524288, 1048576, 2097152, 4194304, 8388608, 16777216	(N/256) * 256	7.49, 7.94, 8.47, 8.85, 9.39, 10.04
33554432	65536*512	10.02

Next we will examine the speedUp of 2DPMS-GPU and 1DPMS-GPU each over its corrosponding sequential times for different shared memory sizes, the speed up of 2DPMS-GPU over sequential is the value of sequential time divided by the 2DPMS-GPU time and the same for 1DPMS-GPU, it is a measure of the percent of enhancment done on the time between the two compared values. When the value of speed up is 1, this means that the parallel and sequential times are the same, if greater than 1; it means that parallel time is less than sequential, or in other words, the parallel is more fast than sequential. If the speed up is less than 1, the sequential version is faster than parallel.

From Fig 5.46 to Fig 5.52, we show the speed ups at different shared memory sizes. For S_{sh}=32, we can see that 2DPMS-GPU have more speed up than 1DPMS-GPU after N_{ch}=512 until the end except at 2048, to reach the highest speed up of 5.35 at N_{ch} =65536 and the biggest differnce between 2DPMS-GPU and 1DPMS-GPU at N_{ch}=1024 of 2.25

speed up. Where at S_{sh}=64 , the 2DPMS-GPU have speed ups larger than 1DPMS-GPU after N_{ch}= 256 until the end except at 4096 and 16384 with biggest difference at N_{ch}=1024 of 3.01 and to reach the highest speed up at N_{ch}=65536 of 7.05.

At S_{sh}=128, the 2DPMS-GPU have speed ups larger than 1DPMS-GPU from N_{ch}=256 until 2048 and also at 65536 with biggest difference at N_{ch}=2048, and to reach the highest speed up at N_{ch}=65536 of 8.82.

At S_{sh}=256, the 2DPMS-GPU have speed ups more than 1DPMS-GPU only from N_{ch}=128 until 2048 with highest difference at N_{ch}=2048 of speed up 7.49 and the highest speed up is at N_{ch}=65536 with speed up of 10.04.

When using S_{sh}=512, the 2DPMS-GPU have more speed ups than 1DPMS-GPU from N_{ch}=64 until 2048 and at N_{ch}=32768 and 65536, the highest different between 2DPMS-GPU and 1DPMS-GPU is at 512 chunks with speed up of 4.67 and the highest speed up is 10.02 at N_{ch}=65536.

Now at S_{sh}=1024, we have enhancement in speed ups, the 2DPMS-GPU have more speed ups than 1DPMS-GPU from N_{ch}=64 until the end with highest difference at N_{ch}=64 of 4.62 and the highest speed up is at N_{ch}=32768 of 9.602.

At S_{sh}=2048, the best difference of speed ups between 2DPMS-GPU and 1DPMS-GPU happens, 2DPMS-GPU has speed up more than 1DPMS-GPU from N_{ch}=64 to 16384 'the end' with biggest difference at N_{ch}=2048 and the highest speed up is at N_{ch}=16384 of 9.08 speed up.

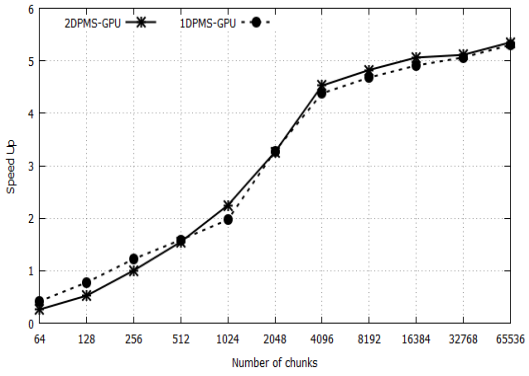


Figure 5.46: Speed Up, shared memory 32.

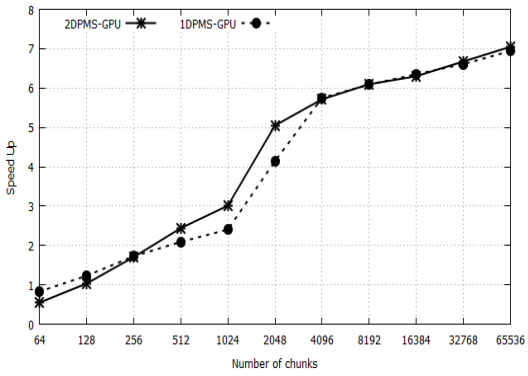


Figure 5.47: Speed Up, shared memory 64.

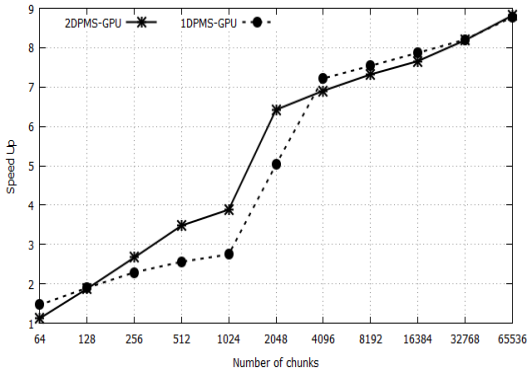


Figure 5.48: Speed Up, shared memory 128.

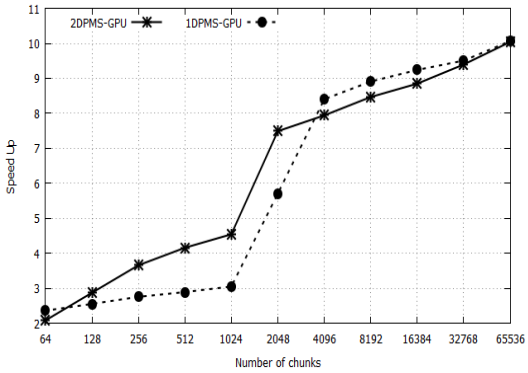


Figure 5.49: Speed Up, shared memory 256.

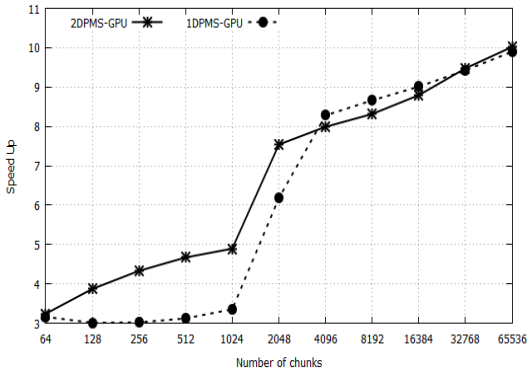


Figure 5.50: Speed Up, shared memory 512.

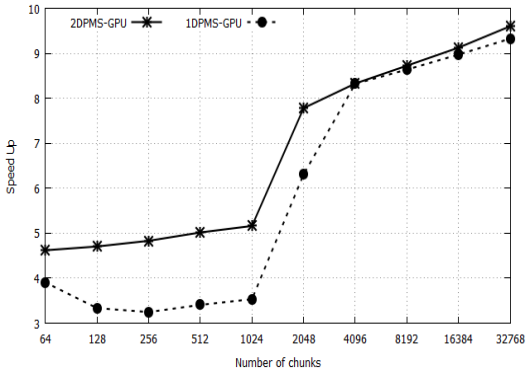


Figure 5.51: Speed Up, shared memory 1024.

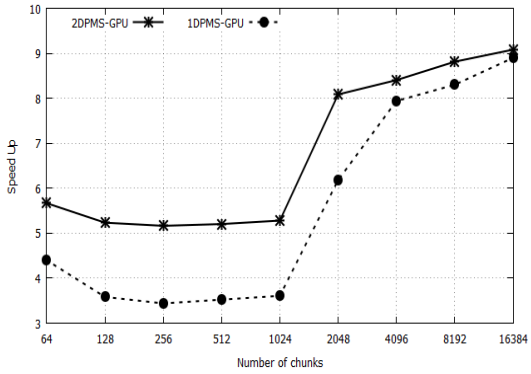


Figure 5.52: Speed Up, shared memory 2048.

Fig 5.53 shows the 2DPMS-GPU speed ups at different shared memory sizes, it demonstares the highest speed ups at shared 128, 256, 512, 1024 and 2048 in which they deliver speed ups more than 1 until 10. We can say that for all number of chunks N_{ch}, the highest speed up is at 2048 shared memory followed by 1024. Other shared memories than 2048; From N_{ch}=64 until 4096, the speed up is best starting from 1024, 512, 256, 128, 64 and 32 shared memory, but after N_{ch}=4096, the best speed up exchange between 512 and 256 shared memory.

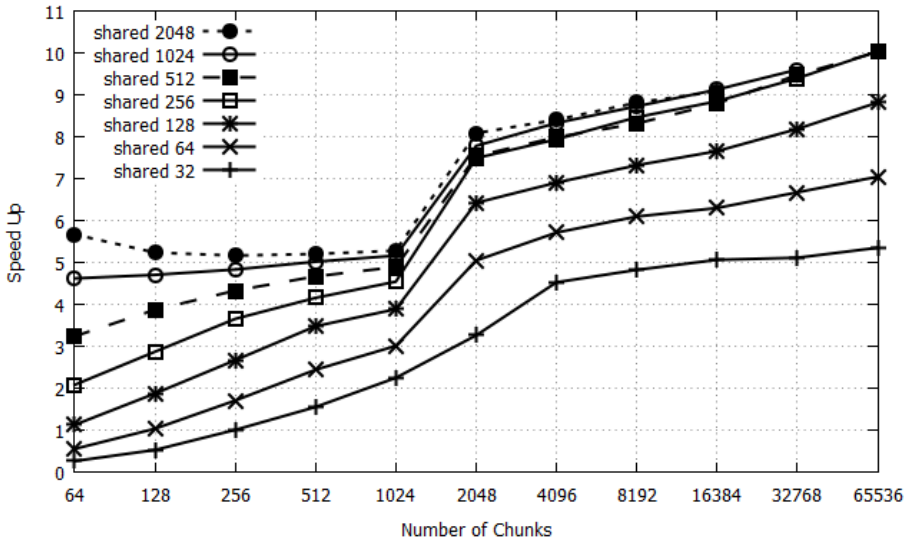


Figure 5.53: Speed Up of 2DPMS-GPU, all possible shared memories.

5.3.3. Analysis of 2DPMS-GPU

It was expected that the 2DPMS-GPU will have half the time of the 1DPMS-GPU since for every odd-even chunk there are two concurent threads each do half the work of 1DPMS-GPU simulanously, but in practice that is not the case on GPU, since the 2DPMS-GPU kernel have conditional branching by using if statements on the threads IDs that could't be avoided when designing this algorithm where the conditional branching degrade the performance, so the speed Up that was expected by the 2DPMS-GPU was consumed in the branching although the efforts that was done to increase the performance when implementing our algorithm as for example using *_device_* identifier for our kernel. To keep the performance high, we leave merging on the GPU the last five stages in which warp divergence happens at all warps, so we complete them on the CPU by openMP parallel directives using the concept of 'GPU-accelerated computing' (Corporation N. , WHAT IS GPU ACCELERATED COMPUTING?, 2016).

From the previous results, we can see that the utilization of the GPU at maximum is 74%. This happens when using many blocks of threads and shared memory size of 2048 since the maximum utilization of the GPU should be carefully balance the number of threads per thread block, the amount of shared memory per block, and the number of registers used by the kernel (Corporation N. , CUDA FAQ, 2016). The Utilization of mergesort can't reach 100% where our mergesort have occupancy of 50% since every phase of merging is dependent on its prevoius phase to finish before it starts, we have a lot of dependencies and synchronization that incure the parallism of GPU.

At small number of chunks, most of shared memories have speed up less than 1, which means that the sequential sort is faster than parallel sort at small sizes, but when the array size becomes big, the parallel sort becomes better than the sequential where the GPU have more work since launching many threads will increase the performance. At these small chunks, the work is done only by the odd-even mergesort on the GPU not with the 2DPMS-GPU mergesort then it is fully merged on CPU using openMP.

At N_{ch}=655536, we have the maximum speed up at all shared memories with 10.4 for shared 256, then the shared memories 512.

The batcher's odd-even mergesort have utilization of 100% at 512, 1024 and 2048 shared memories since it launchs a large amount of threads per CUDA block, and our 2DPMS-GPU mergesort have occupancy of 50% which means that half of the warps that exists SMs are active. Where for 1DPMS-GPU merge is the occupancy is 25%. The occupancy is affected by the shared memory size and number of registers since they are limited in our GPU because we have only 2 SMs where NVIDIA's current products range from 1 SM at the low end to 30 SMs at the high end and currently reaches more than 60 SMs, the mergesort of (Satish et al.) is not efficient with our experiment because of the small number of SMs of our GPU where our 2DPMS-GPU has higher performance than it at small number of SMs.

We are limited to maximum k=64 for all shared memory sizes except shared of 1024 with maximum K=32 and K=16 for shared 2048 because we are limited with number of registers for the 2DPMS-GPU merge and the number of warps for the odd-even mergesort that are available by only the two SMs.

From the prevouis results, we can conclude that our 2DPMS-GPU mergesort have good speed ups for large sizes specially when using more than one block of threads for sorting since the GPU gives good performance when using huge number of threads at the same time in contrast of the CPU and it is more efficient over other compared mergesort for large sizes with small number of SMs.

In our work of 2DPMS-GPU, the experimental results are as expected with the theoretical analysis explained in section 4.2.2; since when N is small, the dominant factor that affects the running time is the communication time, but for large data sizes the dominant factor is the size of array.

Chapter 6

Conclusion and Future Work

The goal of this work was to improve the performance of the parallel mergesort on modern architectures. The first one is on the multi-core CPUs and the second is on the Graphical Processing Units GPUs. Whilst the related works on both architectures use one directional merging process in creating the merged array i.e., from head to tail, in our work we introduced a new approach i.e., the two directional merging process, the creation of the merged array (target array) is done in two directional fashion: from head to middle of the array and from the tail to the middle of the same array using two threads concurrently. Thus, we can speed up the merging process by almost a factor of 2. This approach was tested and evaluated by conducting experiments and compared with existing one directional approaches. For this purpose, a C program/application was developed with some supporting libraries for communication between the processes in the CPU, programming the GPU, and multithreading programming.

The results of the experiments on randomly generated data of different sizes show that an improvement of the performance by mean of running time and better speedup than the speed up of the one directional algorithm is achieved. However, the dominant time is the time needed for scattering data, local sorting and gathering data from the processes in the implementation on CPUs. Therefore, the difference in the performance is relatively small. On the other hand, the limitation that reduces the achieved improvement using this algorithm using GPU is the conditional statement and its disadvantage when using multithreading. Also, because of the overheads in parallel processing we can see that the sequential algorithm performs better than the parallel algorithms when data size is relatively small. Also, the better performance can be achieved when the number of processes matches the actual number of processors/cores. In our case the better

performance was achieved when the number of processes was 8. This is because the used computer on which the experiment carried out has 8 processing threads.

It should be noted that the sorting algorithms are usually tested on data element of integer numbers. However, sorting is needed for different data types such as strings. Therefore, a preprocessing on those data types such as indexing is required before implementing the sorting process.

In the future work we intend to apply this algorithm on different architectures. The two parts of our work can be combined into a hybrid one. Thus the original array that would hold large data elements will be scattered among processors that are connected through a network using MPI and each process do 2DPMS-GPU on distinct GPUs.

References

1. *About CUDA*. (2015). (NVIDIA Corporation) Retrieved January 3, 2016, Available at: https://developer.nvidia.com/about-cuda

2. Adinetz, A. (2014, May 6). *Adaptive Parallel Computation with CUDA Dynamic Parallelism*. (NVIDIA Corporation) Retrieved January 4, 2016, Available at: http://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/

3. Ajkunic, ,. E., Fatkic, H., Omerovic, E., Talic, K., & Nosovic, N. (21-25 May 2012). A comparison of five parallel programming models for C++. *MIPRO, 2012 Proceedings of the 35th International Convention.* IEEE.

4. Amirul, M., Omar, M., Aini, N., Karuppiah, E., Mohanavelu, Meng, S. S., et al. (23-25 Nov. 2012). Sorting Very Large Text Data in Multi GPUs. *Control System, Computing and Engineering (ICCSCE), 2012 IEEE International Conference on.* Penang.

5. Basumallik, A., Min, S.-J., & Eigenmann, R. (2007). Programming Distributed Memory Sytems Using OpenMP . *2007 IEEE International Parallel and Distributed Processing Symposium*, (pp. 1-8). Long Beach, CA.

6. Batcher, K. E. (1968). Sorting Networks and their Applications. *AFIPS '68 (Spring) Proceedings* (pp. 307-314). New York: ACM.

7. Chhugani, J., Nguyen, A. D., Lee, V. W., Macy, W., Hagog, M., Chen, Y.-K., et al. (2008). Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *VLDB Endowment , 1* (2), 1313-1324 .

8. *Coalesced Access to Global Memory*. (2015, September 1). (NVIDIA Corporation) Retrieved January 4, 2016, Available at: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#coalesced-access-to-global-memory

9. *Compute Capability*. (2015, September 1). (NVIDIA Corporation) Retrieved January 4, 2016, Available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability

10. Cook, S. (2013). *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

11. Corporation, I. (n.d.). *Intel® Core™ i7-4702MQ Processor* . Retrieved June 4, 2016, Available at: intel: http://ark.intel.com/products/75119/Intel-Core-i7-4702MQ-Processor-6M-Cache-up-to-3_20-GHz

12. Corporation, N. (2016). *CUDA FAQ*. Retrieved May 5, 2016, Available at: NVIDIA Accelerated Computing: https://developer.nvidia.com/cuda-faq

13. Corporation, N. (2016). *WHAT IS GPU ACCELERATED COMPUTING?* Retrieved June 3, 2016, Available at: What is GPU Computing? : http://www.nvidia.com/object/what-is-gpu-computing.html

14. *CUDA C Best Practices Guide*. (2015, September 1). (NVIDIA Corporation) Retrieved January 3, 2016, Available at: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz3wDjTUQXo

15. *CUDA FORTRAN*. (2015). (NVIDIA Corporation) Retrieved January 4, 2016, Available at: https://developer.nvidia.com/cuda-fortran

16. *CUDA Toolkit Documentation v7.5*. (2015, September 1). (NVIDIA Corporation) Retrieved January 4, 2016, Available at: http://docs.nvidia.com/cuda/index.html#axzz3H5VA0suv

17. Culler, D. E., Gupta, A., & Singh, J. P. (1997). *Parallel Computer Architecture: A Hardware/software Approach 1st* . San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

18. Culler, D., Singh, J. P., & Gupta, A. (1997). *Parallel Computer Architecture A Hardware / Software.* San Francisco: Morgan Kaufmann Publishers Inc.

19. D.Abhyankar, M. (2011). An Efficient Parallel Sorting Algorithm for Multicore Machines. *(IJCSIT) International Journal of Computer Science and Information Technologies* , 1995-1998.

20. Davidson, A., Tarjan, D., Garland, M., & Owens, J. (13-14 May 2012). Efficient Parallel Merge Sort for Fixed and Variable Length Keys. *Innovative Parallel Computing (InPar), 2012.* San Jose, CA.

21. Dawra, M., & Priti. (July 2012). Parallel Implementation of Sorting Algorithms. *IJCSI International Journal of Computer Science Issues , 9* (4), 164-196.

22. Dominik Zurek, M. P. (2013). Comparison of Hybrid Sorting Algorithms Implemented on Different Parallel Hardware Platforms. *Computer Science (AGH)* , 679-691.

23. Ebersole, M. (2012, September 10). *What Is CUDA?* (NVIDIA Corporation) Retrieved January 4, 2016, Available at: http://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/

24. Elio, R. H. (2011). About Computing Science Research Methodology.

25. El-Nashar A. I. (2011). " PARALLEL PERFORMANCE OF MPI SORTING ALGORITHMS ON DUAL–CORE PROCESSOR WINDOWS-BASED SYSTEMS". *Alaa Ismail El-Nashar," PARALLEL PERFORMANCE OF MPI SORTING ALGORITHMS ON DUAL–COREInternational Journal of Distributed and Parallel Systems (IJDPS) , 2 No. 3*.

26. El-Nashar, A. I. (May 2011). PARALLEL PERFORMANCE OF MPI SORTING ALGORITHMS ON DUAL–CORE PROCESSOR WINDOWS-BASED SYSTEMS. *International Journal of Distributed and Parallel Systems (IJDPS) , 2*.

27. Farber, R. (2012). *CUDA Application Design and Development.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc.

28. *Fermi (microarchitecture)*. (2015, December 7). (Wikipedia, the free encyclopedia) Retrieved January 3, 2016, Available at: https://en.wikipedia.org/wiki/Fermi_(microarchitecture)

29. Galloy, M. (2013, June 11). *CPU vs GPU performance*. Retrieved June 5, 2016, Available at: michaelgalloy: http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html

30. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., et al. (July-Aug. 2008). Parallel Computing Experiences with CUDA. *Micro, IEEE , 28* (4), 13 - 27.

31. Ghorpad, J., Parande, J., Kulkarn, M., & Bawaska, A. (January 2012). GPGPU PROCESSING IN CUDA ARCHITECTURE. *Advanced Computing: An International Journal (ACIJ) , 3*.

32. *GPU Accelerated Computing with Python*. (2015). (NVIDIA Corporation) Retrieved January 4, 2016, Available at: https://developer.nvidia.com/how-to-cuda-python

33. *Graphics Cards with NVIDIA Kepler Architecture*. (2015). (NVIDIA Corporation) Retrieved January 3, 2016, Available at: http://www.geforce.com/landing-page/graphics-cards-with-kepler-architecture

34. Gupta, S. (2014, March 25). *NVIDIA Updates GPU Roadmap; Announces Pascal*. (NVIDIA Corporation) Retrieved January 3, 2016, Available at: http://blogs.nvidia.com/blog/2014/03/25/gpu-roadmap-pascal/

35. Hagerup, T., & Rüb, C. (1989). Optimal merging and sorting on the EREW PRAM. *Elsevier* , 181-185.

36. Harris, M. (2012, November 7). *How to Implement Performance Metrics in CUDA C/C++*. (NVIDIA Corporation) Retrieved January 4, 2016, Available at: http://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/

37. Harris, M. (2013, January 28). *Using Shared Memory in CUDA C/C++*. (NVIDIA Corporation) Retrieved January 4, 2016, Available at: http://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/

38. Ionescu, M. F. (1997). *Optimizing Parallel Bitonic Sort.* Technical Report, University of California at Santa Barbara Santa Barbara, CA, USA .

39. Jeon, M., & Kim1, D. (2003). Parallel Merge Sort with Load. *International Journal of Parallel Programming* , 21-33.

40. K.B., M. (2010). Analysis of Parallel Merge Sort Algorithm. *International Journal of Computer Applications* , 66-69.

41. Kale, V., & Solomonik, E. (2010). Parallel sorting pattern. *ParaPLoP '10* . New York, NY, USA.

42. *Kepler (microarchitecture)*. (2015, December 8). (Wikipedia, the free encyclopedia) Retrieved January 3, 2016, Available at: https://en.wikipedia.org/wiki/Kepler_(microarchitecture)

43. Khalilieh, H., Kafri, N., & Mohammad, R. (2014). Performance Evaluation of Message Passing vs. Multithreading Parallel. *International Journal of New Computer Architectures and their Applications (IJNCAA) , 4*, 108-116.

44. Kirk, D. B., & Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

45. Kirk, D., & Hwu, W.-m. W. (2011). *Programming Massively Parallel Processors (Second Edition) A Hands-On Approach.* San Francisco, CA, USA.: Morgan Kaufmann Publishers Inc. .

46. Krewell, K. (2009, December 16). *What's the Difference Between a CPU and a GPU*. (NVIDIA Corporation) Retrieved January 3, 2016, Available at: http://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

47. Lee, D. J., & Downar, T. J. (2001). The Application of POSIX Threads And OpenMP to the U.S. NRC Neutron Kinetics Code PARCS. *Springer-Verlag* (pp. 90-100). London: ACM.

48. Leischner, N., Osipov, V., & Sanders, P. (2009). GPU sample sort. *arXiv* .

49. Ma, L., Chamberlain, R. D., & Agrawal, K. (2014). Analysis of Classic Algorithms on GPUs. *12th IEEE/ACM International Conference on High Performance Computing & Simulation (HPCS)*, (pp. 65-73). Bologna.

50. Manawade, K. (2010). Analysis of Parallel Merge Sort Algorithm. *International Journal of Computer Applications* , 66-69.

51. Manwade, K. (2010, February). Analysis of Parallel Merge Sort Algorithm. *International Journal of Computer Applications 1(1).* , 66-69.

52. *Maxwell Compatibility Guide for CUDA Applications*. (2015, September 1). (NVIDIA Corporation) Retrieved January 3, 2016, Available at: http://docs.nvidia.com/cuda/maxwell-compatibility-guide/#axzz3H5VA0suv

53. Microsoft. (2016, February). Retrieved from Timeout Detection and Recovery (TDR): https://msdn.microsoft.com/en-us/Library/Windows/Hardware/ff570087(v=vs.85).aspx

54. NVIDIA Tesla: A Unified Graphics and Computing Architecture. (March-April 2008). *Micro, IEEE , 28* (2), 39 - 55.

55. Pasetto, D., & Akhriev, A. (2011). A Comparative Study of Parallel Sort Algorithms. *OOPSLA '11*, (pp. 203-204). New York, NY, USA.

56. Peters, H., Schulz-Hildebrandt, O., & Luttenberger, N. (19-23 April 2010). Parallel external sorting for CUDA-enabled GPUs with load balancing and low transfer overhead. *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on.* Atlanta, GA.

57. Radenki A. (2011). Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs. *Proc. PDPTA'11, the 2011, International Conference on Parallel and Distributed Processing Techniques and Applications.* (pp. 267-373). (H. Arabnia, Ed.): CSREA Press.

58. Radenski, A. (2011). Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs. *PDPTA'11, the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications* , (pp. 367-373). H. Arabnia, Ed.

59. Rashid, H., & Qureshi, K. (2006). A practical performance comparison of parallel sorting algorithms on homogeneous network of workstations. *TELE-INFO'06 Proceedings of the 5th WSEAS international conference on Telecommunications and informatics.* Stevens Point, Wisconsin, USA.

60. Ravela, C. S. (2010). *Comparison of Shared memory based.* Retrieved January 6, 2016, Available at: http://www.diva-portal.org/smash/get/diva2:830690/FULLTEXT01.pdf

61. Rüb, C. (1998). On Batcher's Merge Sorts as Parallel Sorting Algorithms. *STACS '98 Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science* (pp. 410-420). London: ACM.

62. Ruetsch, G., & Fatica, M. (n.d.). A CUDA Fortran Implementation of BWAVES.

63. Satish, N., Harris, M., & Garland, M. (23-29 May 2009). Designing efficient sorting algorithms for manycore GPUs. *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.* Rome.

64. Tanasic, I., Vilanova, L., Jordà, M., Cabezas, J., Gelado, I., Navarro, N., et al. (2013). Comparison Based Sorting for Systems with Multiple GPUs. *GPGPU-6 Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (pp. 1-11). New York: ACM.

65. *TESLA GPU ACCELERATORS FOR SERVERS*. (2015). (NVIDIA Corporation) Retrieved January 3, 2016, Available at: http://www.nvidia.com/object/tesla-servers.html

66. The MathWorks, I. (1994-2016). *Technical Articles and Newsletters\ GPU Programming in MATLAB*. Retrieved July 5, 2016, Available at: Math Works |Accelerating the pace of engineering and science: https://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html?requestedDomain=www.mathworks.com

67. *The open standard for parallel programming of heterogeneous systems*. (2016). (Khronos Group) Retrieved January 4, 2016, Available at: https://www.khronos.org/opencl/

68. Trimananda, R., & Haryanto, C. (2-3 Aug. 2010). A parallel implementation of hybridized merge-quicksort algorithm on MPICH. *Distributed Framework and Applications (DFmA), 2010 International Conference on.* Yogyakarta.

69. Vajda, A. (2011). *Programming Many-Core Chips.* Springer Publishing Company.

70. Venu, B. (2011, October 16). *Multi-core processors - An overview.* Retrieved 8 10, 2016, Available at: ARXIV: https://arxiv.org/ftp/arxiv/papers/1110/1110.3535.pdf

71. *What is CUDA?* (2015). (NVIDIA Corporation) Retrieved January 3, 2016, Available at: http://www.nvidia.com/object/cuda_home_new.html

72. Wilt, N. (2013). *CUDA Handbook: A Comprehensive Guide to GPU Programming.* Addison-Wesley.

73. Xiaojun, R., Qing, Y., I. Alghamdi, M., Shu, Y., Zhiyang, D., Jiong, X., et al. (2010). ES-MPICH2: A Message Passing Interface with enhanced security. *International Performance Computing and Communications Conference*, (pp. 161 - 168). Albuquerque, NM.

74. Ye, X., Fan, D., & Lin, W. (2010). High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs. *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on.* IEEE.

75. Yildiz, Z., Aydin, M., & Yilmaz, G. (7-9 Nov. 2013). Parallelization of bitonic sort and radix sort algorithms on many core GPUs. *Electronics, Computer and Computation (ICECCO), 2013 International Conference on.* Ankara.

APPENDIX I

Tables 7.1 to 7.20 show the results of 2DPMS-CPU and 1DPMS-CPU.

Table 7.1: 2DPMS-CPU and 1DPMS-CPU on size 64

Array Size 64		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	0.052655	0.323410
4	0.060712	0.2805046
8	0.123600	0.1377831
16	0.303493	0.0561132
32	0.705582	0.0241361
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	0.048613	0.3503206
4	0.050074	0.3400943
8	0.120925	0.1408315
16	0.454183	0.0374958
32	0.657311	0.0259086

Table 7.2: 2DPMS-CPU and 1DPMS-CPU on size 128

Array Size 128		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	0.057477	0.6483631
4	0.055484	0.6716485
8	0.127955	0.2912403
16	0.329589	0.1130674
32	0.769029	0.0484582
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	0.085313	0.4368124
4	0.097194	0.3834166
8	0.227766	0.1636140
16	0.451446	0.0825474
32	0.696313	0.0535187

Table 7.3: 2DPMS-CPU and 2DPMS-CPU on size 256

Array Size 256		
2D merge sort		
#Processes	Time Average(ms)	Speed Up
2	0.074271	1.1738378
4	0.068424	1.2741508
8	0.127736	0.6825209
16	0.313820	0.2778105
32	0.676189	0.1289320
1D merge sort		
#Processes	Time Average(ms)	Speed Up
2	0.077071	1.1312001
4	0.065066	1.3399183
8	0.118250	0.7372731
16	0.303680	0.2870864
32	0.754256	0.1155874

Table 7.4: 2DPMS-CPU and 1DPMS-CPU on size 512

Array Size 512		
2D merge sort		
#Processes	Time Average(ms)	Speed Up
2	0.121887	1.5123597
4	0.087800	2.0995159
8	0.115979	1.5893995
16	0.340785	0.5409207
32	0.659084	0.2796874
1D merge sort		
#Processes	Time Average(ms)	Speed Up
2	0.128451	1.4350750
4	0.090942	2.0269683
8	0.126119	1.4616171
16	0.336835	0.5472632
32	0.705519	0.2612793

Table 7.5: 2DPMS-CPU and 1DPMS-CPU on size 1024

Table 7.6: 2DPMS-CPU and 2DPMS-CPU on size 2048

Array Size 1024		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	0.224401	1.8703208
4	0.132277	3.1728981
8	0.149133	2.8142769
16	0.329341	1.2743719
32	0.719204	0.5835653
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	0.251927	1.6659709
4	0.142261	2.9502205
8	0.159491	2.6315066
16	0.350582	1.1971592
32	0.765141	0.5485293

Table 7.7: 2DPMS-CPU and 1DPMS-CPU on size 4096

Array Size 2048		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	0.425539	2.1011163
4	0.242937	3.6804088
8	0.343988	2.5992367
16	0.413720	2.1611385
32	0.766416	1.1666069
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	0.423981	2.1088339
4	0.244556	3.6560389
8	0.242284	3.6903231
16	0.383829	2.3294369
32	0.799011	1.1190166

Table 7.8: 2DPMS-CPU and 1DPMS-CPU on size 8192

Array Size 4096		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	0.863579	2.2468115
4	0.452100	4.2917496
8	0.389989	4.9752727
16	0.545997	3.5536803
32	0.767567	2.5278586
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	0.835183	2.3232024
4	0.474866	4.0859947
8	0.427901	4.5344565
16	0.595605	3.2576977
32	0.870235	2.2296284

Table 7.9: 2DPMS-CPU and 2DPMS-CPU on size 16384

Array Size 8192		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	1.731233	2.4173946
4	0.861774	4.8563457
8	0.741099	5.6471141
16	0.887621	4.7149300
32	1.329176	3.1486217
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	1.847927	2.2647394
4	0.917946	4.5591706
8	0.860438	4.8638861
16	1.005032	4.1641186
32	1.249929	3.3482490

Table 7.10: 2DPMS-CPU and 1DPMS-CPU on size 32768

Array Size 16384		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	3.523893	2.5084566
4	1.673010	5.2836097
8	1.372066	6.4424975
16	1.753596	5.0408029
32	1.841987	4.7989102
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	3.450492	2.5618178
4	1.978026	4.4688653
8	1.702463	5.1922030
16	1.800497	4.9094945
32	2.036747	4.3400237

Table 7.11: 2DPMS-CPU and 1DPMS-CPU on size 65536

Array Size 32768		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	7.988528	2.3767006
4	3.647211	5.2057140
8	3.255451	5.8321692
16	3.617635	5.2482734
32	3.501281	5.4226842
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	7.449095	2.5488114
4	3.994217	4.7534567
8	3.509897	5.4093727
16	3.716383	5.1088215
32	3.893790	4.8760563

Table 7.12: 2DPMS-CPU and 1DPMS-CPU on size 131072

Array Size 65536		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	15.495160	2.5916245
4	7.9702110	5.0384670
8	5.7262590	7.0128937
16	7.1386650	5.6253713
32	7.0811250	5.6710821
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	16.926480	2.3724747
4	8.621921	4.6576215
8	6.825995	5.8830458
16	7.869158	5.1031690
32	7.497676	5.3560121

Table 7.13: 2DPMS-CPU and 1DPMS-CPU on size 262144

Array Size 131072		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	30.348130	2.7915443
4	17.738120	4.7760506
8	12.884550	6.5751732
16	15.254120	5.5537871
32	14.095540	6.0102801
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	30.320320	2.7941042
4	17.979280	4.7119873
8	14.655810	5.7805157
16	16.091260	5.2648532
32	15.328520	5.5268324

Table 7.14: 2DPMS-CPU and 1DPMS-CPU on size 524288

Array Size 262144		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	61.202190	2.9306110
4	36.630550	4.8964540
8	26.553150	6.7547477
16	34.296530	5.2296782
32	37.472800	4.7864006
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	63.943650	2.8049668
4	37.567220	4.7743701
8	29.236350	6.1348218
16	37.196460	4.8219601
32	35.972530	4.9860222

Table 7.15: 2DPMS-CPU and 1DPMS-CPU on size 1048576

Array Size 524288		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	127.126300	2.9839179
4	75.957870	4.9940120
8	53.178300	7.1332577
16	72.711471	5.2169831
32	69.733200	5.4397978
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	131.844100	2.8771437
4	81.964740	4.6280211
8	60.205250	6.3006887
16	75.850070	5.0011097
32	71.444760	5.3094802

Table 7.16: 2DPMS-CPU and 1DPMS-CPU on size 2097152

Array Size 1048576		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	276.105000	2.8921462
4	161.190300	4.9539954
8	109.074100	7.3210376
16	143.649400	5.5589246
32	144.154000	5.5394648
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	276.774100	2.8851545
4	205.056600	3.8942225
8	120.220400	6.6422645
16	156.803600	5.0925870
32	152.705700	5.2292485

Table 7.17: 2DPMS-CPU and 1DPMS-CPU on size 4194304

Array Size 2097152		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	558.871000	2.9822231
4	325.702200	5.1171830
8	229.769800	7.2536842
16	295.515700	5.6398956
32	297.868200	5.5953529
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	571.954200	2.9140057
4	342.219700	4.8701993
8	251.830400	6.6182545
16	304.740600	5.4691697
32	307.809200	5.4146457

Table 7.18: 2DPMS-CPU and 1DPMS-CPU on size 8388608

Array Size 4194304		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	1175.989000	2.9639384
4	680.627000	5.1210996
8	478.912300	7.2780728
16	583.383600	5.9747286
32	619.411600	5.6272096
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	1219.336000	2.8585715
4	711.537300	4.8986310
8	523.752500	6.6549733
16	623.040500	5.5944332
32	638.190400	5.4616287

Table 7.19: 2DPMS-CPU and 1DPMS-CPU on size 16777216

Array Size 8388608		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	2466.33900	2.9589343
4	1425.482000	5.1194851
8	1043.034000	6.9966443
16	1172.553000	6.2237998
32	1111.560000	6.5653076
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	2445.902000	2.9836579
4	1486.261000	4.9101302
8	1110.906000	6.5691762
16	1273.011000	5.7326559
32	1259.623000	5.7935854

Table 7.20: 2DPMS-CPU and 1DPMS-CPU on size 33554432

Array Size 16777216		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	5078.734000	3.0096603
4	3004.756000	5.0870244
8	2083.536000	7.3362150
16	2359.360000	6.4785630
32	2318.958000	6.5914373
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	5102.435000	2.9956806
4	3062.516000	4.9910812
8	2314.999000	6.6027104
16	2537.255000	6.0243326
32	2506.792000	6.0975390

Array Size 33554432		
2DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	10560.600000	2.9943407
4	6231.868000	5.0742464
8	4471.067000	7.0725926
16	5044.990000	6.2680066
32	4687.619000	6.7458621
1DPMS-CPU		
#Processes	Time Average(ms)	Speed Up
2	10580.590000	2.9886820
4	6112.007000	5.1737563
8	4652.708000	6.7964799
16	5384.446000	5.8728478
32	5352.630000	5.9077568

Tables from 7.21 to 7.27 show the results for 2DPMS-GPU with different shared memory sizes.

Table 7.21: 2DPMS-GPU with shared memory 32

Shared Memory 32 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	64	0.4575111	0.0372
4	128	0.6579705	0.0566
8	256	1.1438688	0.0762
16	512	1.9104108	0.0964
32	1024	3.1752380	0.1321
64	2048	3.4313147	0.2605
128	4096	3.6942776	0.5252
256	8192	4.1662963	1.0045
512	16384	5.7045260	1.5495
1024	32768	8.4497884	2.2469
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	65536	12.3162689	3.2605
4	131072	18.7139588	4.5270
8	262144	37.1809092	4.8239
16	524288	74.9338953	5.0622
32	1048576	156.1733439	5.1131
64	2097152	311.5151347	5.3502

Table 7.22: 2DPMS-GPU with shared memory 64

Shared Memory 64 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	128	0.3848384	0.0968
4	256	0.7133062	0.1222
8	512	1.1129170	0.1656
16	1024	1.8697433	0.2244
32	2048	3.2497985	0.2751
64	4096	3.5327331	0.5492
128	8192	4.0516953	1.0329
256	16384	5.1839256	1.7051
512	32768	7.7791350	2.4406
1024	65536	13.3442474	3.0093
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	131072	16.7836690	5.0476
4	262144	31.3931612	5.7133
8	524288	62.2266309	6.0960
16	148576	126.7596257	6.2996
32	2097152	249.7758362	6.6726
64	4194304	494.7267049	7.0454

Table 7.23: 2DPMS-GPU with shared memory 128

Shared Memory 128 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	256	0.4757917	0.1832
4	512	0.7218173	0.2553
8	1024	1.0638476	0.3945
16	2048	1.9381314	0.4613
32	4096	3.1332853	0.6192
64	8192	3.7208696	1.1247
128	16384	4.7180823	1.8735
256	32768	7.1161527	2.6680
512	65536	11.5435616	3.4787
1024	131072	21.8062873	3.8850
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average (ms)	Speed Up
2	262144	27.9256460	6.4227
4	524288	54.9422904	6.9042
8	1048576	109.0879847	7.3201
16	2097152	217.6842305	7.6564
32	4194304	425.8017985	8.1858
64	8388608	827.7650960	8.8161

Table 7.24: 2DPMS-GPU with shared memory 256

Shared Memory 256 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	512	0.5510829	0.3345
4	1024	0.7949241	0.5279
8	2048	1.3080766	0.6835
16	4096	2.0044647	0.9679
32	8192	3.3674044	1.2428
64	16384	4.2519996	2.0789
128	32768	6.6171032	2.8692
256	65536	10.988775	3.6544
512	131072	20.385625	4.1557
1024	262144	39.493742	4.5414
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average (ms)	Speed Up
2	524288	50.6549603	7.4885
4	1048576	100.5767141	7.9395
8	2097152	196.8791616	8.4654
16	4194304	394.0132202	8.8462
32	8388608	776.7813233	9.3948
64	16777216	1522.529362	10.039

Table 7.25: 2DPMS-GPU with shared memory 512

Shared Memory 512 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	1024	0.4558272	0.9207
4	2048	0.7791280	1.1475
8	4096	1.3061479	1.4855
16	8192	2.3064797	1.8144
32	16384	3.5467568	2.4922
64	32768	5.8626697	3.2385
128	65536	10.3778191	3.8695
256	131072	19.5751378	4.3278
512	262144	38.4178001	4.6686
1024	524288	77.5400802	4.8920
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	1048576	105.9344193	7.5380
4	2097152	208.6195669	7.9890
8	4194304	419.1905273	8.3149
16	8388608	829.9394287	8.7930
32	16777216	1613.120003	9.4755
64	33554432	3155.956665	10.019

Table 7.26: 2DPMS-GPU with shared memory 1024

Shared Memory 1024 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	2048	0.6323092	1.4140
4	4096	0.8506979	2.2808
8	8192	1.4368577	2.9126
16	16384	2.3890283	3.7000
32	32768	4.2709958	4.4454
64	65536	8.6912611	4.6204
128	131072	18.0061359	4.7049
256	262144	37.1218322	4.8316
512	524288	75.6069595	5.0171
1024	1048576	154.6869507	5.1622
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	2097152	214.1684326	7.7820
4	4194304	418.4889832	8.3289
8	8388608	836.4232056	8.7249
16	16777216	1675.061800	9.1251
32	33554432	3293.215641	9.6021
64	----	-----	-----

Table 7.27: 2DPMS-GPU with shared memory 2048

Shared Memory 2048 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	4096	0.6746102	2.8761
4	8192	0.9599342	4.3597
8	16384	1.6212243	5.4523
16	32768	2.9041878	6.5375
32	65536	5.4820242	7.3253
64	131072	14.9465493	5.6680
128	262144	34.2651303	5.2344
256	524288	73.4505193	5.1644
512	1048576	153.4749217	5.2030
1024	2097152	315.5652974	5.2815
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	4194304	431.0350789	8.0864
4	8388608	868.3356486	8.4042
8	16777216	1734.185474	8.8140
16	33554432	3481.987380	9.0816

Tables from 7.28 to 7.34 show the results of 1DPMS-GPU .

Table 7.28: 1DPMS-GPU with shared memory 32

Shared Memory 32 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	64	0.3493418	0.0487
4	128	0.5137779	0.0725
8	256	0.6657312	0.1309
16	512	1.1308896	0.1630
32	1024	1.9843150	0.2115
64	2048	2.1430064	0.4172
128	4096	2.4974709	0.7769
256	8192	3.4209320	1.2233
512	16384	5.5626525	1.5890
1024	32768	9.6344022	1.9706
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	65536	12.2538751	3.2771
4	131072	19.3435081	4.3796
8	262144	38.3190541	4.6806
16	524288	77.2592819	4.9098
32	148576	157.6855611	5.0641
64	2097152	314.4776713	5.2998

Table 7.29: 1DPMS-GPU with shared memory 64

Shared Memory 64 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	128	0.3754069	0.0992
4	256	0.5567444	0.1565
8	512	0.7384654	0.2496
16	1024	1.2295652	0.3413
32	2048	1.9912142	0.4490
64	4096	2.3368812	0.8302
128	8192	3.4006537	1.2306
256	16384	5.1181112	1.7271
512	32768	9.0735899	2.0924
1024	65536	16.6993330	2.4047
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	131072	20.4646375	4.1397
4	262144	31.2053383	5.7477
8	524288	62.3151471	6.0873
16	1048576	125.7363871	6.7375
32	2097152	252.3108266	6.6056
64	4194304	502.2904521	6.9393

Table 7.30: 1DPMS-GPU with shared memory 128

Shared Memory 128 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	256	0.4107634	0.2122
4	512	0.5269565	0.3498
8	1024	0.7805997	0.5376
16	2048	1.2318238	0.7258
32	4096	1.9608477	0.9895
64	8192	2.8595196	1.4635
128	16384	4.6603992	1.8967
256	32768	8.2900676	2.2902
512	65536	15.6900499	2.5594
1024	131072	30.8385929	2.7471
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	262144	35.6835782	5.0263
4	524288	52.5577781	7.2174
8	1048576	105.9147929	7.5394
16	2097152	211.858960	7.8669
32	4194304	424.8822815	8.2035
64	8388608	831.6508667	8.7749

Table 7.31: 1DPMS-GPU with shared memory 256

Shared Memory 256 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	512	0.4148275	0.4443
4	1024	0.5924918	0.7083
8	2048	0.8051487	1.1104
16	4096	1.2767597	1.5197
32	8192	2.0528764	2.0386
64	16384	3.7506833	2.3567
128	32768	7.4468619	2.5495
256	65536	14.5956768	2.7513
512	131072	29.3289260	2.8885
1024	262144	58.8663861	3.0468
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	524288	66.4690068	5.7069
4	1048576	95.0275660	8.4032
8	2097152	187.1190847	8.9070
16	4194304	377.2776184	9.2387
32	8388608	767.7275472	9.5056
64	16777216	1519.106226	10.062

Table 7.32: 1DPMS-GPU with shared memory 512

Shared Memory 512 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	1024	0.4352874	0.9641
4	2048	0.6289165	1.4216
8	4096	0.9266485	2.0938
16	8192	1.4147367	2.9581
32	16384	2.3866037	3.7038
64	32768	6.0108095	3.1586
128	65536	13.3412627	3.0100
256	131072	28.0670701	3.0184
512	262144	57.4761561	3.1205
1024	524288	113.1884084	3.3513
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	1048576	129.267246	6.1774
4	2097152	201.289931	8.2799
8	4194304	402.451757	8.6608
16	8388608	809.835319	9.0113
32	16777216	1621.900244	9.4242
64	33554432	3195.038314	9.8972

Table 7.33: 1DPMS-GPU with shared memory 1024

Shared Memory 1024 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	2048	0.4689258	1.9067
4	4096	0.6862943	2.8272
8	8192	1.1075780	3.7785
16	16384	1.7255109	5.1228
32	32768	3.0073905	6.3132
64	65536	10.2898372	3.9026
128	131072	25.4136861	3.3335
256	262144	55.2427914	3.2467
512	524288	111.3749354	3.4059
1024	1048576	226.0018646	3.5333
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	2097152	264.1635661	6.3092
4	4194304	418.8508606	8.3217
8	8388608	844.7704875	8.6387
16	16777216	1703.718970	8.9717
32	33554432	3391.018188	9.3252
64	---	----	----

Table 7.34: 1DPMS-GPU with shared memory 2048

Shared Memory 2048 elements			
NumOfChunks= 1*chunk base			
Chunk base	Total Size	Time Average(ms)	Speed Up
2	4096	0.5606314	3.4609
4	8192	0.7861664	5.3233
8	16384	1.4113382	6.2632
16	32768	2.3770253	7.9874
32	65536	4.3310076	9.2721
64	131072	19.2637630	4.3977
128	262144	50.0640864	3.5826
256	524288	110.3112850	3.4387
512	1048576	226.7082021	3.5223
1024	2097152	461.9188090	3.6081
NumOfChunks= y*1024			
value of "y"	Total Size	Time Average(ms)	Speed Up
2	4194304	563.8297038	6.1819
4	8388608	919.0211670	7.9407
8	16777216	1841.414111	8.3008
16	33554432	3553.416569	8.8990

Table 7.35 shows the results of sequential sort.

Table 7.35: Sequential sort

Size of Array	Time Average (ms)
64	0.017030000
128	0.037265750
256	0.087182500
512	0.184337500
1024	0.419702500
2048	0.894106250
4096	1.940300000
8192	4.1850725000
16384	8.8395318750
32768	18.9863393800
65536	40.1576437500
131072	84.7181437500
262144	179.3598269000
524288	379.3345350000
1048576	798.5359213000
2097152	1666.677921000
4194304	3485.558748000
8388608	7297.734656000
16777216	15285.265260000
33554432	31622.033680000

Figures from Fig 7.1 to Fig 7.20 show the elapsed time for the 2DPMS-CPU and 1DPMS-CPU and there details

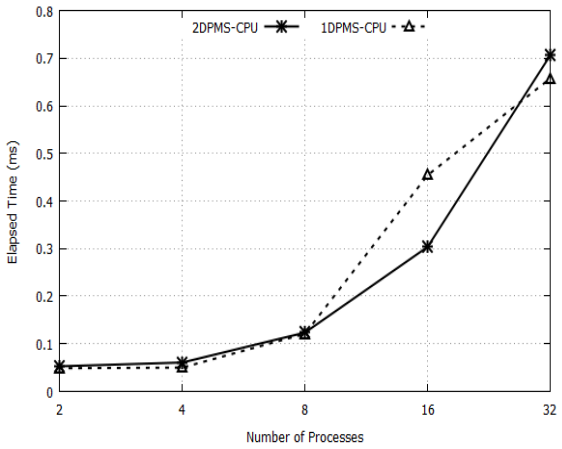


Figure 7.1: Running Time at size 64

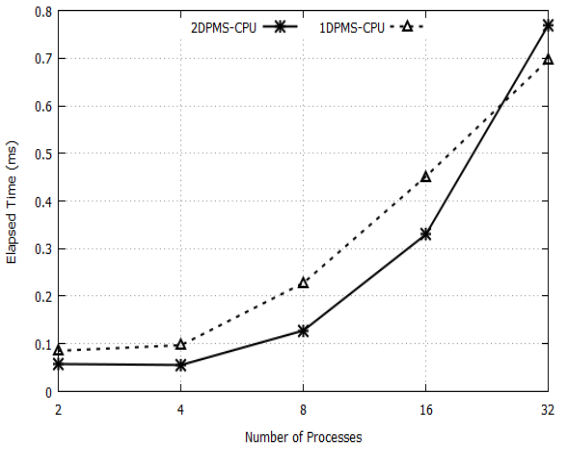


Figure 7.2: Running Time at size 128

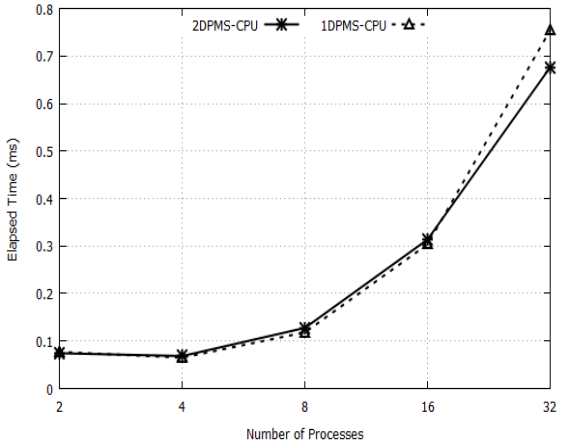


Figure 7.3: Running Time at size 256

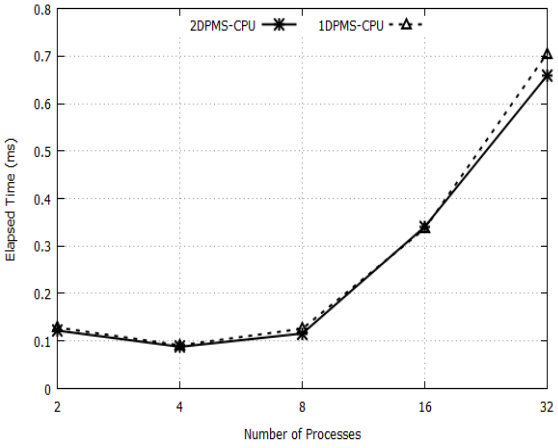


Figure 7.4: Running Time at size 512

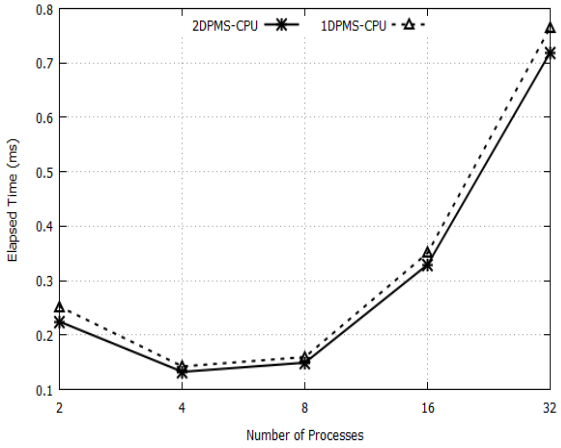


Figure 7.5: Running Time at size 1024

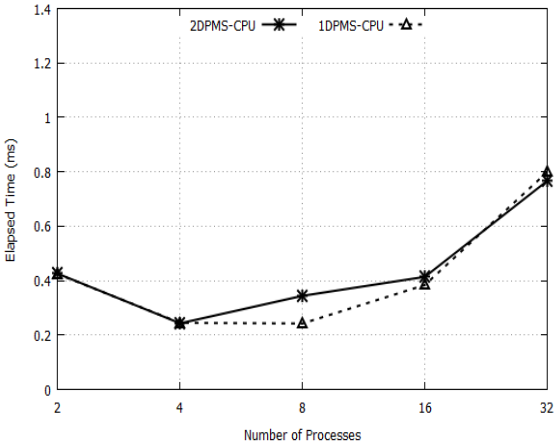


Figure 7.6 Running Time at size 2048

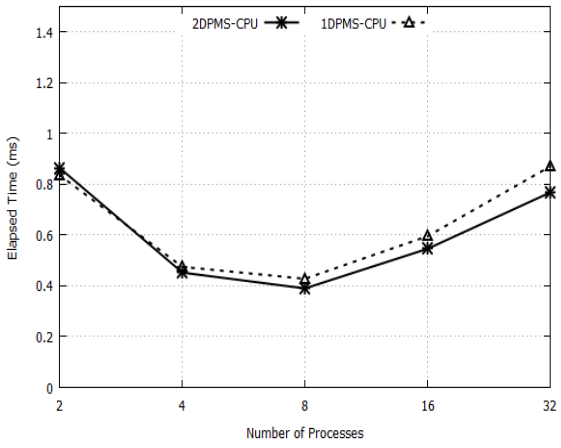


Figure 7.7: Running Time at size 4096

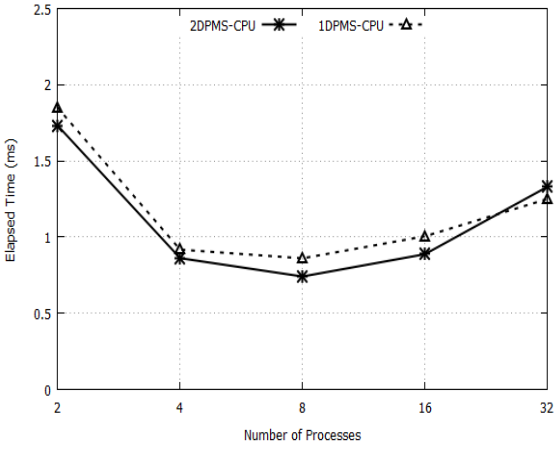


Figure 7.8: Running Time at size 8192

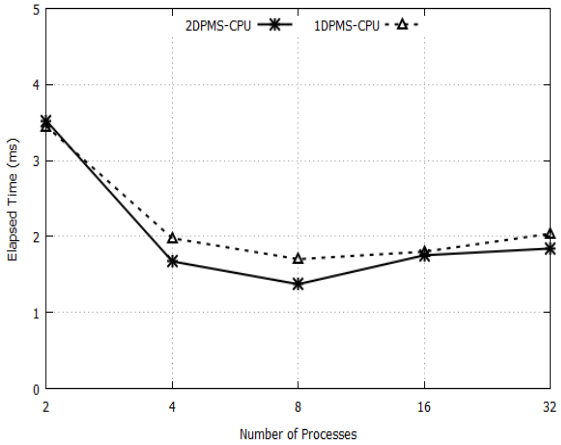


Figure 7.9: Running Time at size 16384

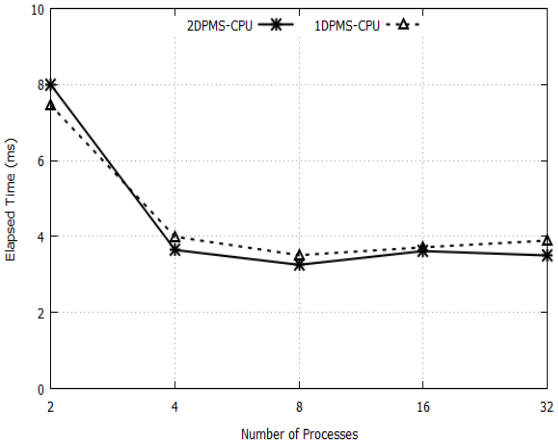


Figure 7.10: Running Time at size 32768

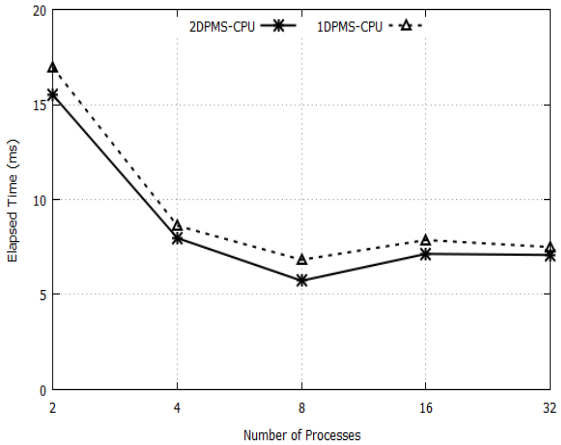


Figure 7.11: Running Time at size 65536

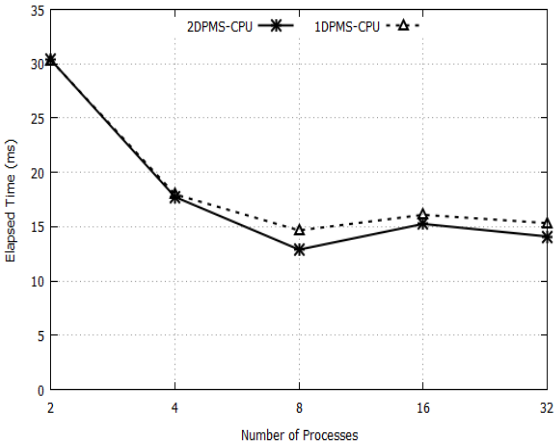


Figure 7.12: Running Time at size 131072

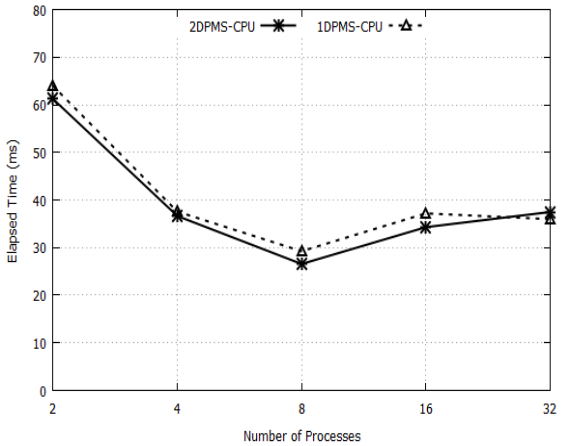


Figure 7.13: Running Time at size 262144

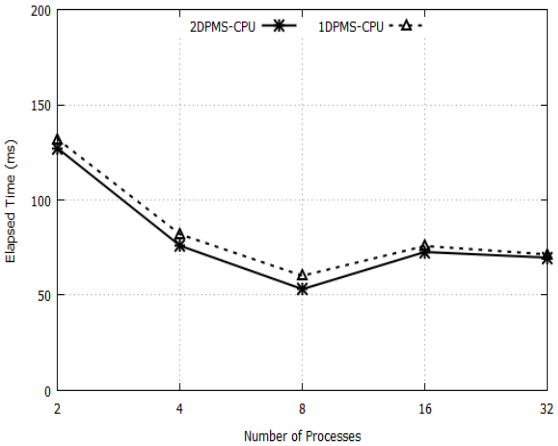


Figure 7.14: Running Time at size 524288

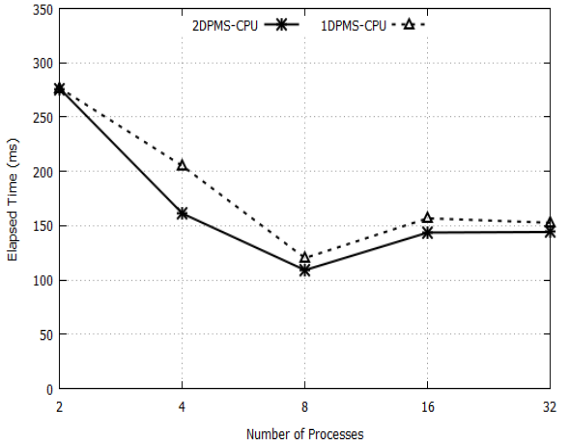


Figure 7.15: Running Time at size 1048576

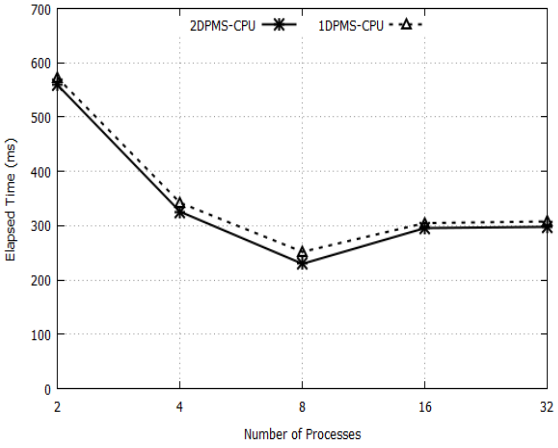


Figure 7.16: Running Time at size 2097152

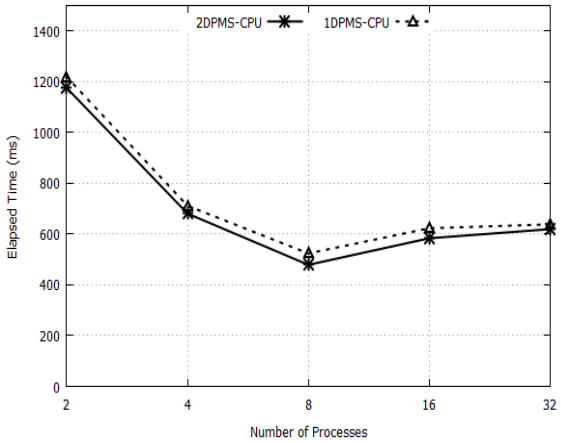


Figure 7.17: Running Time at size 4194304

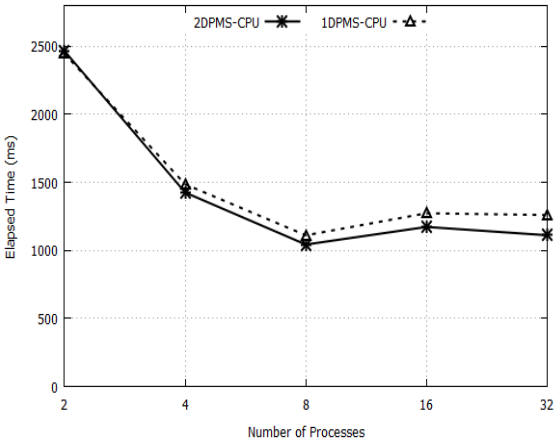


Figure 7.18: Running Time at size 8388608

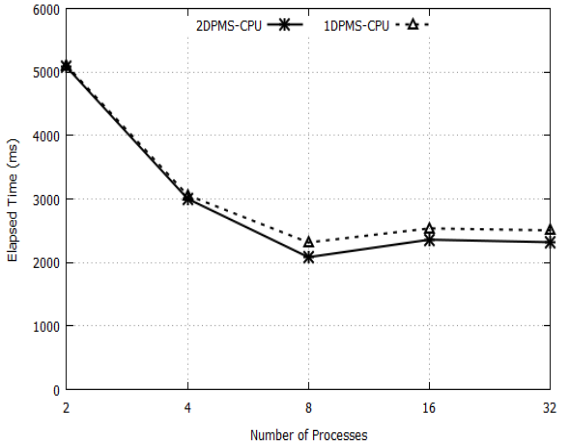


Figure 7.19: Average time at size 16777216

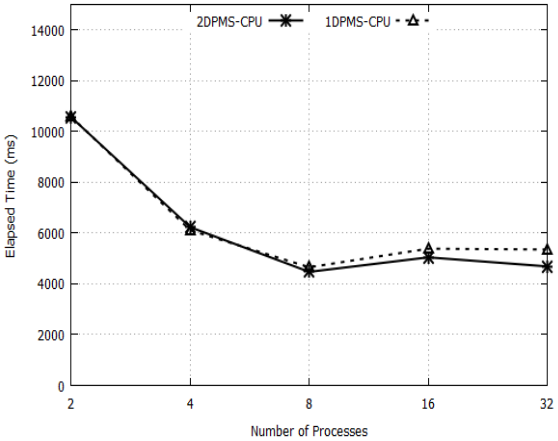


Figure 7.20: Array Size 33554432

Fig 7.21 to Fig 7.24 show the speed up of 2DPMS-CPU at different sizes and different number of processes 'zoomed figures'.

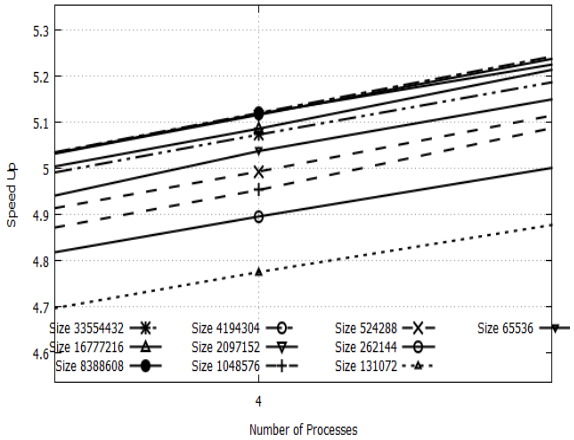


Figure 7.21: Speed Up of 2DPMS-CPU at different sizes and 4 processes

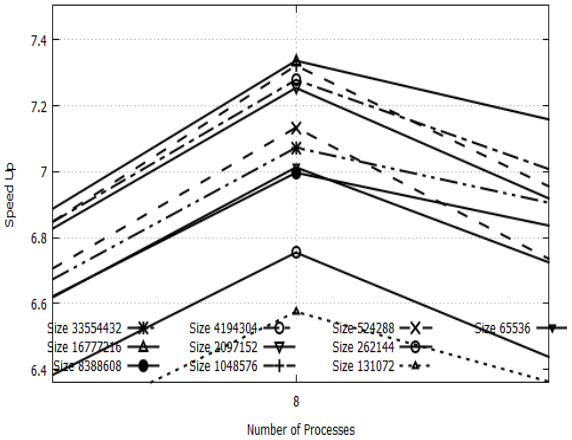


Figure 7.22: Speed Up of 2DPMS-CPU at different sizes and 8 processes

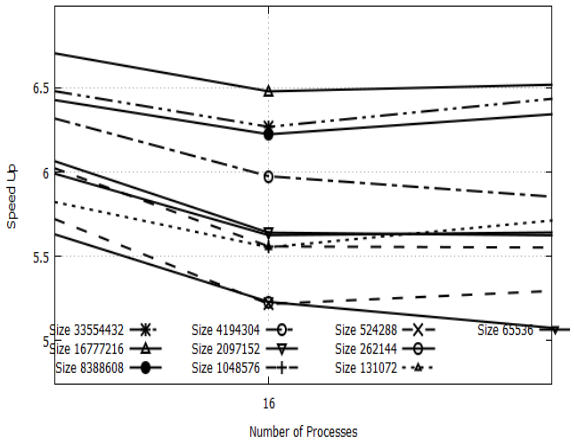


Figure 7.23: Speed Up of 2DPMS-CPU at different sizes and 16 processes

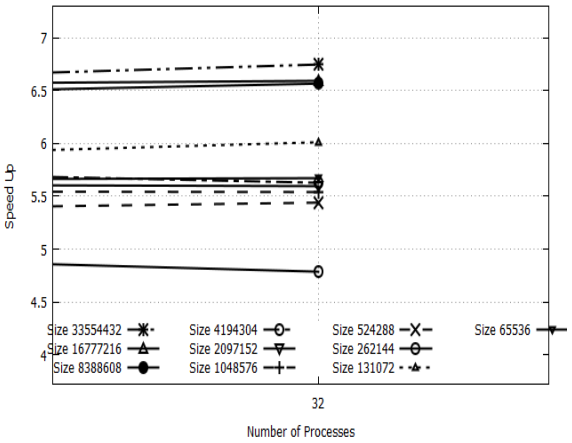


Figure 7.24: Speed Up of 2DPMS-CPU at different sizes and 4 processes

Figures from Fig 7.25 to Fig 7.39 show the elapsed time for the 2DPMS-GPU and 1DPMS-GPU and there details.

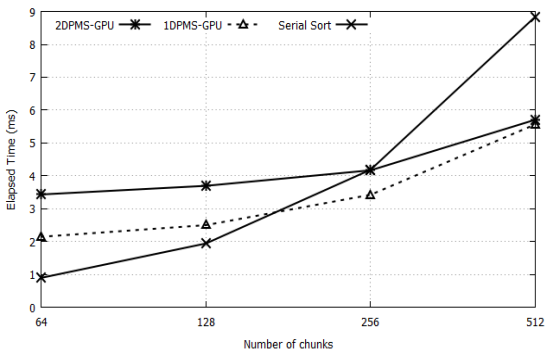


Figure 7.25: Running Time at shared memory 32 from 64 to 512 chunks

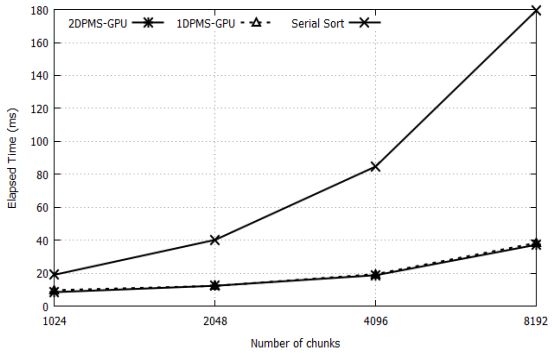


Figure 7.26: Running Time at shared memory 32 from 1024 to 8192 chunks

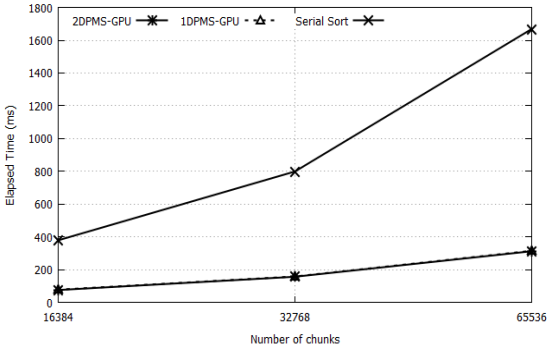


Figure 7.27: Running Time at shared memory 32 from 16384 to 65536 chunks

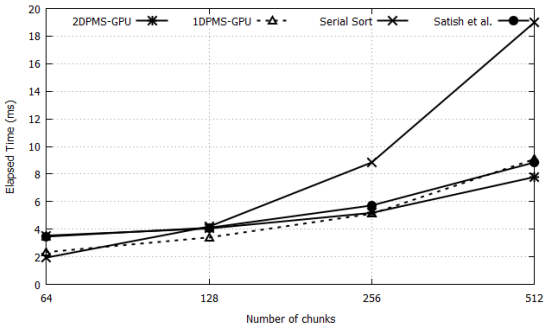


Figure 7.28: Running Time at shared memory 64 from 64 to 512 chunks

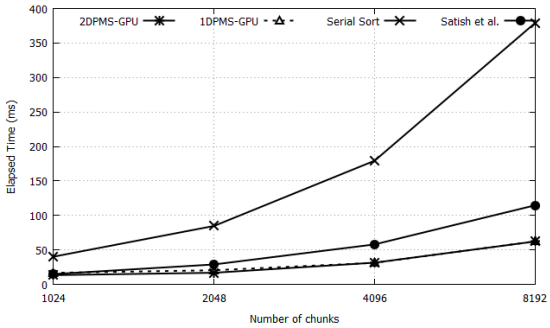


Figure 7.29: Running Time at shared memory 64 from 1024 to 8192 chunks

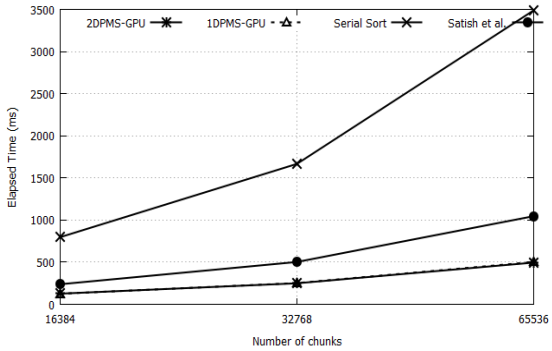


Figure 7.30: Running Time at shared memory 64 from 16384 to 65536 chunks

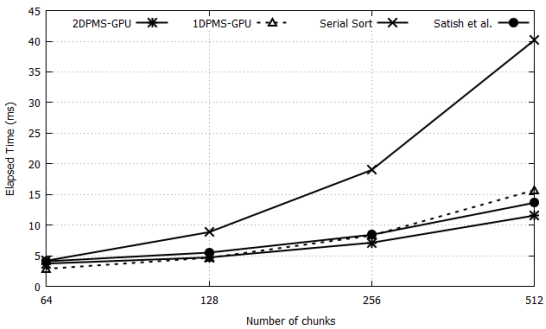


Figure 7.31: Running Time at shared memory 128 from 64 to 512 chunks

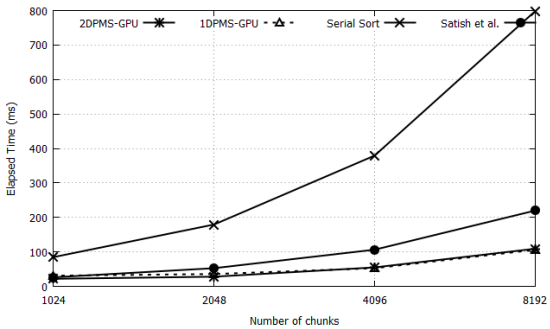


Figure 7.32: Running Time at shared memory 128 from 1024 to 8192 chunks

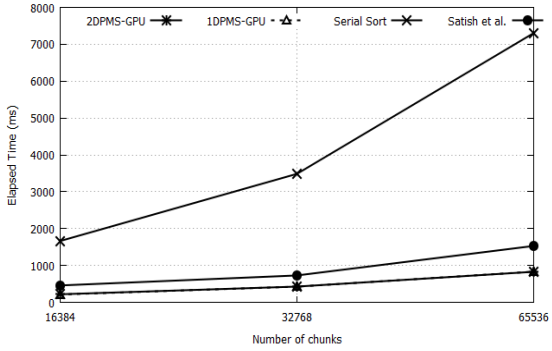


Figure 7.33: Running Time at shared memory 128 from 1024 to 65536 chunks

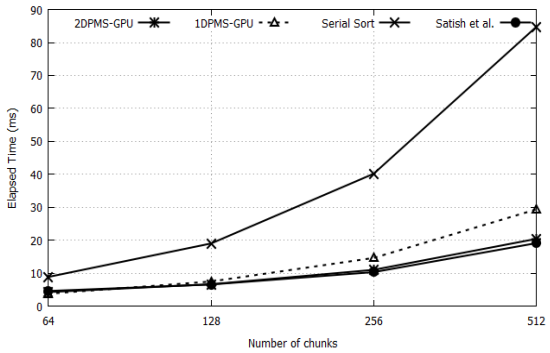


Figure 7.34: Running Time at shared memory 256 from 64 to 512 chunks

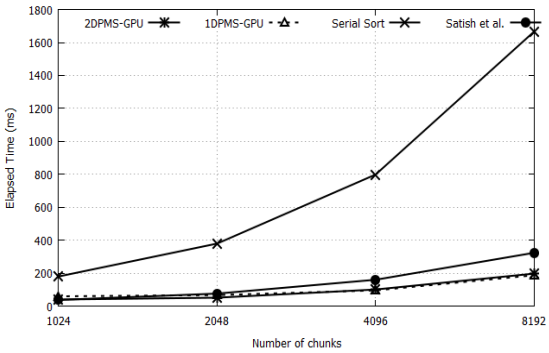


Figure 7.35: Running Time at shared memory 256 from 1024 to 8192 chunks

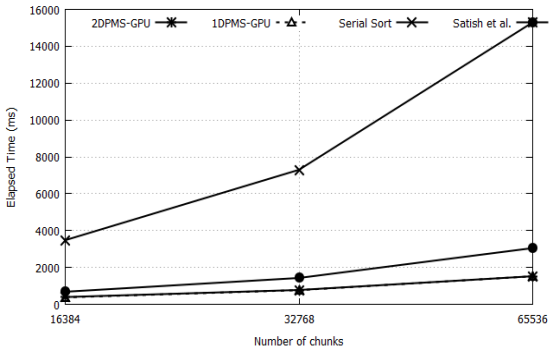


Figure 7.36: Running Time at shared memory 256 from 1024 to 65536 chunks

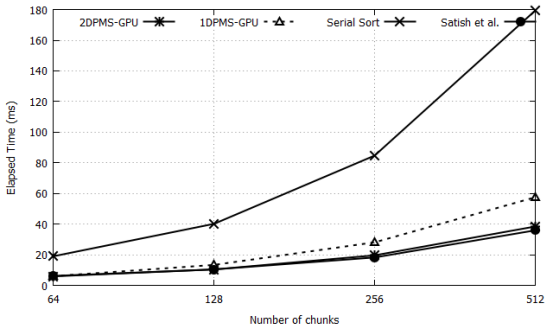


Figure 7.37: Running Time at shared memory 512 from 64 to 512 chunks

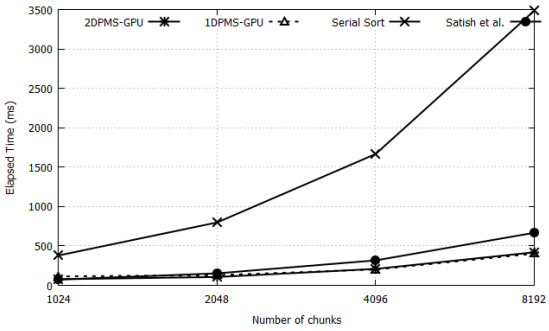


Figure 7.38: Running Time at shared memory 512 from 1024 to 8192 chunks

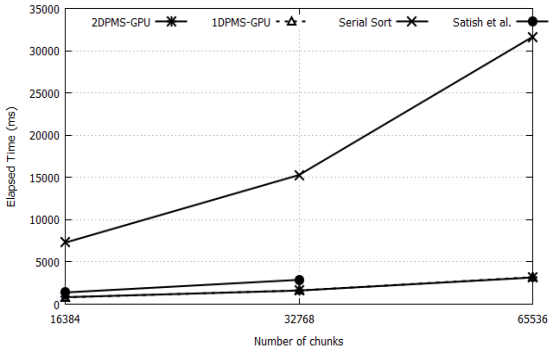


Figure 7.39: Running Time at shared memory 512 from 16384 to 65536 chunks

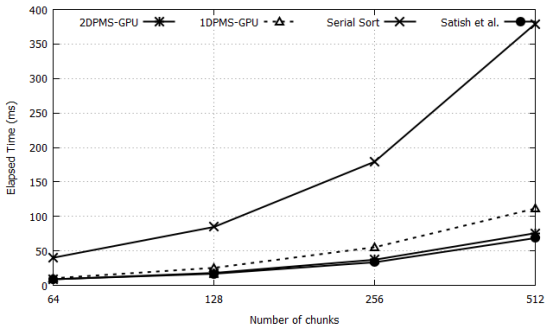


Figure 7.40: Running Time at shared memory 1024 from 64 to 512 chunks

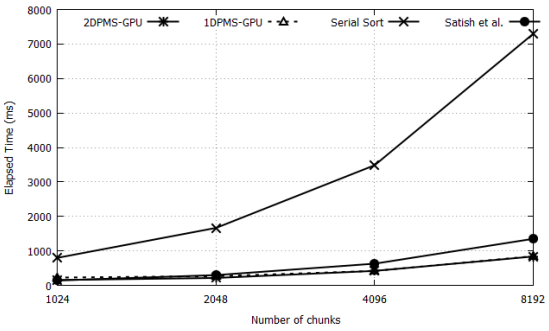


Figure 7.41: Running Time at shared memory 1024 from 1024 to 8192 chunks

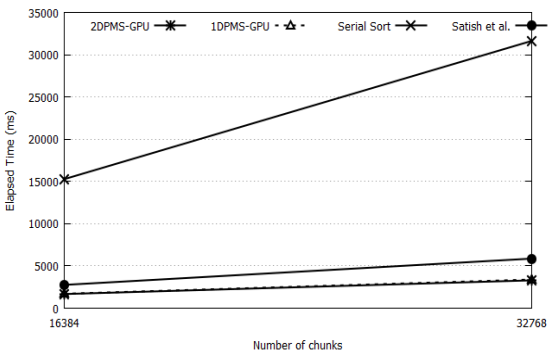


Figure 7.42: Running Time at shared memory 1024 from 16384 to 32768 chunks

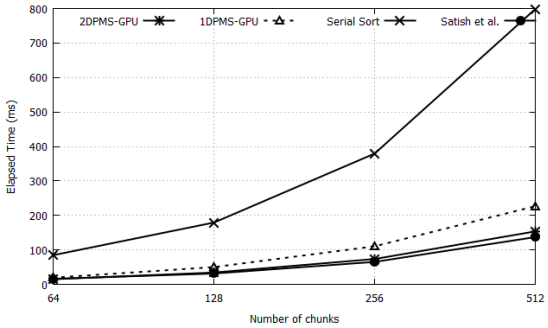


Figure 7.43: Running Time at shared memory 2048 from 64 to 512 chunks

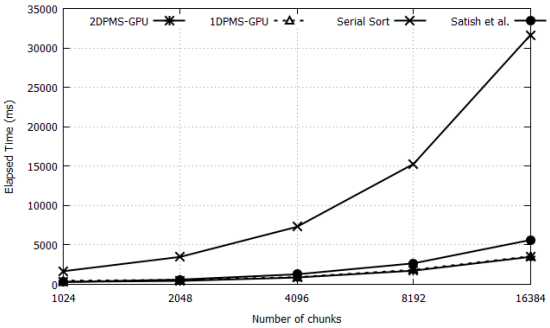


Figure 7.44: Running Time at shared memory 2048 from 1024 to 16384 chunks