DEVELOPMENT OF PARALLEL ALGORITHMS BASED ON OPENMP: SNPHAP AND PPMQSORT CASE STUDIES

RATTHASLIP RANOKPHANUWAT

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF ENGINEERING IN ELECTRICAL ENGINEERING
FACULTY OF ENGINEERING
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG
2016

KMITL-2016-EN-D-018–027

DEVELOPMENT OF PARALLEL ALGORITHMS BASED ON OPENMP: SNPHAP AND PPMQSORT CASE STUDIES

RATTHASLIP RANOKPHANUWAT

A THESIS SUBMITTED IN FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF ENGINEERING IN ELECTRICAL ENGINEERING

FACULTY OF ENGINEERING

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2016

THESIS CERTIFICATION
FACULTY OF ENGINEERING
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

Thesis Title Development of Parallel Algorithms Based on OpenMP : SNPHAP and PPMQSORT Case Studies

Student Mr. Ratthaslip Ranokphanuwat

Student Id. 51060035

Degree Doctor of Engineering

Program Electrical Engineering

Thesis Advisor Asst. Prof. Dr. Surin Kittitornkun

Thesis Reference Number KMITL-2016-EN-D-018-027

EXAMINERS	SIGNATURES
Assoc. Prof. Dr. Boontee Kruatrachue	
Asst. Prof. Dr. Visit Hirankitti	
Prof. Dr. Kosin Chamnongthai	
Assoc. Prof. Dr. Kietikul Jearanaitanakij	
Asst. Prof. Dr. Surin Kittitornkun	

Date 21th March 2016 **Time** 09.00-11.00

Place Building A , Conference Room No.3

(Assoc. Prof. Dr. Komsan Maleesee)
Dean, Faculty of Engineering
21th March 2016

หัวข้อวิทยานิพนธ์	การพัฒนาอัลกอริทึมแบบขนานโดยใช้ OpenMP กรณีศึกษา SNPHAP และ PPMQSORT
นักศึกษา	นายรัฐศิลป์ รานอกภานุวัชร์
รหัสนักศึกษา	51060035
ปริญญา	วิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชา	วิศวกรรมไฟฟ้า
พ.ศ.	2559
อาจารย์ที่ปรึกษาวิทยานิพนธ์	ผศ.ดร. สุรินทร์ กิตติธรกุล

บทคัดย่อ

วิทยานิพนธ์นี้นำเสนอวิธีการสำหรับเปลี่ยนอัลกอริทึมแบบเรียงลำดับให้ประมวลผลแบบขนานโดยใช้ไลบรารี OpenMP 3.0 และทำงานบนระบบคอมพิวเตอร์แบบแชร์หน่วยความจำ หลายคอร์ และหลายซ็อกเก็ตซีพียู วิธีการนี้ทำให้สามารถเพิ่มความเร็วประมวลผล โดยเฉพาะในแอพพลิเคชันด้านไบโออินฟอร์เมติกส์ การวิเคราะห์ข้อมูลขนาดใหญ่ เป็นต้น วิทยานิพนธ์นี้นำเสนอสองกรณีศึกษา กรณีศึกษาแรกเป็นโปรแกรม SNPHAP ซึ่งใช้อัลกอริทึม Expectation Maximization ในการประมวลผล Haplotype inference วิธีการเริ่มจากหาส่วนโปรแกรมที่ใช้เวลาประมวลผลนาน จากนั้นเปลี่ยนเป็นการทำงานแบบขนานโดยใช้คำสั่ง OpenMP ได้แก่ parallel For และ Task ผลการทดสอบบนเครื่อง 8-core Xeon E5405 8-core Hyper-Thread Xeon E5520 และ 32-core AMD Opteron 8356 และใช้ระบบปฏิบัติการลินุกซ์ พบว่าความเร็วเพิ่มขึ้น 316% 410% และ 488% ตามลำดับ

กรณีศึกษาสอง เป็นอัลกอริทึมสำหรับการเรียงลำดับข้อมูลที่เป็นนิยมในด้านวิทยาศาสตร์คอมพิวเตอร์มีชื่อว่า QuickSort ในวิทยานิพนธ์นี้นำเสนออัลกอริทึมใหม่ ที่มีประสิทธิภาพและสามารถขยายขอบเขตการประมวลผลได้ เรียกว่า Parallel Partition and Merge QuickSort หรือ PPMQSort อัลกอริทึม PPMQSort นี้ถูกพัฒนาให้รองรับและเปรียบเทียบประสิทธิภาพกับฟังก์ชันที่เร็วที่สุดในไลบรารีมาตรฐาน Stdlib สำหรับ C และ C++ คือ qsort() โดยอัลกอริทึมจะแบ่งข้อมูลในอาร์เรย์แบบเรียกตัวเองให้เป็นส่วนย่อยๆที่เรียงลำดับแล้ว ด้วยวิธี nested OpenMP Task parallelism จากนั้นข้อมูลภายในแต่ละส่วนย่อยที่อิสระเหล่านี้ จะทำการเรียงลำดับด้วยฟังก์ชัน qsort() นั่นคือไม่ต้องทำ synchronization ผลการทดสอบการทำงานที่ข้อมูลชนิดจำนวนเต็ม 32 บิต 200 ล้าน และจำนวน 16 เทรด บนเครื่อง 8-core HyperThread Xeon E5520 พบว่าความเร็วเพิ่มขึ้น 12.29 เท่า ขณะที่ข้อมูลชุดเดียวกันทดสอบบนเครื่อง 4-core AMD A6-3600 CPU (non-HyperThread) ความเร็วเพิ่มขึ้น 4.67 เท่า ความเร็วแบบ Super linear ดังนั้นสรุปได้ว่าอัลกอริทึม PPMQSort สามารถใช้ประโยชน์จากแคชทุกระดับชั้นและซีพียูคอร์แบบ HyperThread และใช้ประโยชน์จากซีพียูคอร์ถึง 83% และ 96% บนเครื่อง Xeon E5520 และเครื่อง AMD A6-3600 ตามลำดับ

Thesis Title	Development of Parallel Algorithms Based on OpenMP: SNPHAP and PPMQSORT Case Studies
Student	Mr. Ratthaslip Ranokphanuwat
Student ID.	51060035
Degree	Doctor of Engineering
Program	Electrical Engineering
Year	2016
Thesis Advisor	Asst. Prof. Dr. Surin Kittitornkun

ABSTRACT

This thesis has introduced the methodology for parallelizing sequential algorithms using OpenMP 3.0 library to run on any shared memory/multicore/multi-socket systems. It can achieve high Speedups especially on demanding applications such as bioinformatics, Big Data analytics and so on. As the first case study, a computation-intensive problem called SNPHAP, an Expectation Maximization haplotype inference algorithm. The methodology starts with profiling SNPHAP to identify hotspots. Then, those hotspots are parallelized using OpenMP *parallel For* and *Task* constructs. The Speedups are 316%, 410%, and 488% on an 8-core Xeon E5405, an 8-core HyperThread Xeon E5520 and a 32-core AMD Opteron 8356 Linux machines, respectively.

The second case study is the QuickSort, a well-known sequential sorting algorithm in Computer Science. An explosive amount of data has tremendous impacts on run time complexity. This thesis proposes an efficient and scalable algorithm called *Parallel Partition and Merge QuickSort (PPMQSort)*, the PPMQSort is developed to be compatible and benchmarked with the fastest C/C++ Stdlib $qsort()$. The PPMQSort recursively divides an unsorted input data array into partially sorted partitions using nested OpenMP *Task* parallelism. Finally, those independent partitions are sequentially $qsort()$ (conquered). As such, it can be up to 12.29 times faster on an 8-core HyperThread Xeon E5520 while sorting random 200M 32-bit integer data at 16 threads. With the same data type, a 4-core AMD A6-3600 CPU (non-HyperThread) can reach up to 4.67x yielding a superlinear Speedup. It has been proved that the proposed PPMQSort can exploit all available cache levels and HyperThread CPU cores well thus utilizing up to 83% and 96% of CPU on Xeon E5520 and AMD A6-3600, respectively.

ACKNOWLEDGEMENTS

First of all, I would like to deeply thank Assistant Professor Dr. Surin Kittitornkun of King Mongkut's Intitute of Technology Lardkrabang, my Advisor for his helpful suggestions and constant supports during my Doctor of Engineering.

I am also thankful to my thesis committee members in the Faculty of Engineering, King Mongkut's Intitute of Technology Lardkrabang for their insightful comments and helpful discussions which give me a better perspective of this thesis.

I wish to express my acknowledgment to Dhurakij Pundit University for awarding me the scholarship with the financial support four years for my Doctor of Engineering.

My gratefully acknowledge goes also to Dr.Sissades Tongsima and the National Center for Genetic Engineering and Biotechnology (BIOTEC) Thailand for 32-core CPUs system supports.

I wish to thank Mr. Apisit Rattanatranurak and Mr. Surapong Towtiamton for experiments and discussions on some of the algorithms in this thesis.

Finally, I would like to acknowledge the supports of all of my beloved family and friends for all of their helps and encouragements.

Ratthaslip Ranokphanuwat

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivations

Major CPU manufacturers have produced multicore CPUs since 20xx. Several demanding programs can be multithreaded to gain higher performance with less expensive hardware and better efficiency [4]. The recent parallel library for shared memory multiprocessor system and multicore systems, OpenMP library [1] is now the most well-known one. OpenMP can be applied successfully to develop parallel programs such as Generator polynomials search using OpenMP [5], OpenMP model for realizing the image edge detection within the platform of TMS320C6678 DSP [6], Doolittle parallel algorithm for multicore machines [7], Parallel multilevel fast multipole algorithm (MLFMA) for shared memory parallel platform [8], Parallel OpenMP to implement of the image processing application [9], Three OpenMP-based parallel implementations for solving descriptors to be deployed on shared-memory machines [10] and so on.

The difficulty of multithreading may depend on the characteristics of each application. Park *et al.* [11] created a parallel FORTRAN programming environment using OpenMP. However, they focus on general program development and also created a new tool for supporting each step in the methodology. It is not flexible for other free and open-source tools. Moreover, it does not support implementation details to get the highest performance, especially, demanding applications such as bioinformatics, Big Data analyses, etc.

As the first case study, SNPHAP [2] is chosen for this thesis due to its demanding characteristic. SNPHAP is a haplotype inference bioinformatics program using EM (Expectation Maximization) algorithm. An inferred haplotype is not a direct observation but can be obtained from unphased (ambiguous) genotype data (resulting in haplotype ambiguity) through experiment in software techniques. A haplotype provides a snapshot of human evolution to estimate the age and location of disease mutations related to a set of multiple linked markers and to investigate many population processes [12]. There exist many methods for inferring haplotypes from unphased genotype data. One of the most widely used methods is to statistically estimate haplotype frequencies from genotyping markers of a group of

unrelated individuals by employing the Expectation Maximization (EM) algorithm to predict the missing or incomplete data [13, 14]. This problem can take longer computation time caused by a large number of possible haplotype instances. Therefore, if the method can be extended into parallel algorithms, the throughput may be increased yet more accurate.

The second case study, $qsort()$ is still a challenging sorting algorithm because of the explosive growth of data in social/web mining, large-scale scientific, commercial application domains and so on. Among all the sorting algorithms, QuickSort is the most well-known and standard sorting algorithm [15, 16] . To compare with any existing sorting algorithms, QuickSort is the fastest one in practice [17].

1.2 Existing Approaches

Being a contribution, Eronen *et al.* [3] compare runtime per genotype (seconds) of the SNPHAP with other haplotyping software. It can be noticed that run time per genotype of the SNPHAP is the best. Moreover, the SNPHAP reduces memory requirements so that these make parallelization of SNPHAP even more challenging. Recently, there has been little research to implement the EM algorithm for inferring haplotypes in parallel. Huang and Chen [18] have done experiments as a final project by inferring haplotype using EM algorithm with MPI (Message Passing Interface) library for passing information among the processors.

For sorting, numerous parallel architectures can be applied to perform sorting algorithms. Earlier studies have shown [19–21] that sorting can be done at the interconnection level of a particular network of processors named the MultiRing network. Recently, sorting networks have been implemented on FPGAs instead. References [22–24] used FPGAs as sorting kernels for database intensive operations. In addition to FPGAs, hundreds to thousands of processing elements/cores inside the GPUs can be applied as co-processors for sorting [25] based on SIMD parallelism including the Bitonic-Merge Sort on Intel Xeon Phi [26].

A few parallel algorithms have been proposed to enhance the existing QuickSort algorithm. Initially, Heidelberger [27] presented the parallel version on an ideal Parallel Random Access Machine. In practice, the sequential QuickSort can be enhanced with several parallel techniques to run on any shared memory/multicore systems with multithreading operating system. Tsigas and Zhang [28] proposed a fine-grain parallel QuickSort algorithm to fit data into caches. M. Sub and C. Leopold [29] presented several alternative algorithms of parallel QuickSort based on pthreads and OpenMP 2.0. Man *et al.* [30, 31] developed $psort()$

algorithm to be compatible with Stdlib $qsort()$. Meanwhile, Kim *et al.* [32] have shown that a dual core OMAP-4430 can achieve only 1.47x Speedup from their Introspective QuickSort algorithm. Mahafzah [33] splitted the input array with multi-pivot/thread into partitions using extra space and then sorted them in parallel up to 8 threads. Very recently, Bingmann *et al.* [34] proposed multikey QuickSort algorithms for string sorting on NUMA (Non Uniform Memory Access) architectures. Their results show that the Speedup is bounded by memory bandwidth.

1.3 Statement of Problem

Inferring haplotype using EM algorithm of SNPHAP can take longer computation time caused by a large number of possible haplotype instances are widely needed. Moreover, $qsort()$ inside SNPHAP should be further optimized in order to sorting efficiency for a large number of data. Therefore, if the method can be extended both SNPHAP and $qsort()$ into parallel algorithms, the throughput may be increased.

For sorting, moreover, it is still challenging to enhance parallel QuickSort performance and efficiency at the same time. These challenges are due to sequential data partitioning, latency/bandwidth between memory hierarchy, and sequential/recursive nature of QuickSort. Furthermore, the bottlenecks of parallel QuickSort should be further investigated together with some performance characteristics such as CPU utilization and memory bandwidth/latency.

1.4 Contributions

The contributions of this thesis can be divided into two parts:

Firstly, SNPHAP has already been highly optimized for sequential execution and considered as a benchmark for haplotyping in term of speed. This thesis has put tremendous efforts speed SNPHAP up by 4.88x times faster than the original version. The contributions are summarized as follows:

- The development for parallelizing SNPHAP consists of four major steps: Profiling, Multithreading, Verification, and OpenMP Profiling.
- The applied OpenMP For constructs convert functions consuming most execution time to threads according to the Profiler.

- The enhanced SNPHAP is parallelized further with OpenMP $Task$ construct to run $qsort()$.

Secondly, a new $qsort()$ algorithm has been proposed and named Parallel Partition and Merge QuickSort (PPMQSort) for various multicore CPUs. The contributions are summarized as follows:

- The PPMQSort algorithm is compatible and benchmarked with Stdlib $qsort()$ while achieving superlinear Speedup in some CPUs.
- The time complexity of PPMQSort has been analyzed using BigO notation and comparable with others.
- A new efficiency metric of any parallel programs on either HyperThread or non-HyperThread CPUs is proposed. The PPMQSort algorithm can achieve the highest efficiency among previously published algorithms.
- Based on the Linux Perf measurement tool, a system performance model of any shared memory/multiprocessor/multicore systems is proposed to estimate memory bandwidth.
- The Speedups of PPMQSort with Worst-case input data although very rare but can be as high as those of Random cases.

1.5 Thesis Organization

This thesis consists of five main chapters, which covered all of our research. Given briefly summarized as follows.

Chapter 2 describes and background involved in this thesis. Related work summarizes previous work that has been done with similar objectives. Finally, SNPHAP algorithm and parallel QuickSort algorithm are explained.

Chapter 3 presents the parallel SNPHAP and results discussion.

Chapter 4 presents the PPMQSort and results discussion,

Chapter 5 is the last chapter providing conclusion and future work.

Appendix A describes the SNPHAP program

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter presents a review of the research background associated with parallel SNPHAP and parallel QuickSort in order to fulfill the objective of this thesis. This chapter is structured into 4 main sections; OpenMP library, Multicore and Parallel computing technology, SNPHAP algorithm, and Parallel QuickSort algorithms as following.

2.1 OpenMP library

This section begins with the overview of OpenMP library, the parallel programming model of this thesis. The recent parallel language standard for shared memory multiprocessor system (SMP), OpenMP library 3.0 [1] is now the most well-known library that can be applied successfully to develop parallel programs running on multicore CPUs architecture.

OpenMP library is an application program interface (API) for thread based parallelism on shared memory multicore processors. The API consists of a set of compiler directives, library routines, and environmental variables that support FORTRAN and C/C++ on multiple architectures. For programmers, OpenMP provides a portable, scalable model of thread based parallelism application. The model of multithreading execution is the fork-join model. The Master thread initially starts the running program. After the program is executed for a while, the Worker threads are spawned (forked) at a parallel region to form a team together with the Master thread. At the end of each parallel region, all threads are synchronized (joined) before the Master thread can continue further. A parallel region can also nest with other parallel regions. The main advantage of using OpenMP is the ability of all CPU cores to share and access the same memory pool (data) with less communication overhead and network latency compared with other parallel computing paradigms such as cluster computing, grid computing, etc. To improve the performance, this thesis makes use of three OpenMP constructs:

2.1.1 For Construct (Data-level parallelism)

The *parallel For* construct defines that the iterations of the loop must be executed in parallel by the thread team. Moreover, it could be used to split the iterations of a loop across multiple threads. When OpenMP encounters the *parallel For* construct, threads are

created and the iterations of the loop are divided to distribute over available threads. At the end of the parallel region, all threads must wait until all iterations of the loop have completed. The syntax of *parallel For* construct is illustrated in Figure 2.1.

```
#pragma omp parallel for [clause [[,] clause] ... ] new-line
        for_loop
```

Figure 2.1 $parallel\ For$ construct definition by OpenMP 3.0 [1].

This thesis specifically demonstrates how to parallelize on the loop of *cmp_hap()* inside *hap_prior()* and *hap_posterior()* of SNPHAP and also distribute the work load equally at run time.

2.1.2 Task Construct (Task-level parallelism)

OpenMP version 3.0 [1] introduces a $Task$ model to handle irregular and dynamic parallelism in the form of recursive routines. According to OpenMP 3.0, tasks are independent units of work which are executed in parallel by threads. OpenMP 3.0 $Task$ construct can be applied to parallelize recursive algorithms. On the other hand, it is possible to express parallelism using recursion where the amount of hardware/platform parallelism is unknown in advance. The syntax of $Task$ construct is illustrated in Figure 2.2.

```
#pragma omp task [clause [[,] clause] ... ] new-line
        structured-block
```

Figure 2.2 $Task$ construct definition by OpenMP 3.0 [1]

The $Task$ construct defines the statement(s) and its data (variables) associated with it. In addition, threads are assigned to perform the work of each Task. Task may be deferred or executed immediately. When the Master thread encounters a $Task$ construct, any available thread in a Task pool including the Master thread can execute immediately. If the Task is forced to wait due to synchronization, it is placed in the pool that is associated with the current parallel region. The threads in the current team will take all the Tasks out of

the pool and execute them until the pool is empty. This thesis specifically demonstrates how to exploit the $Task$ construct in *qsort()* of SNPHAP. Moreover, how the PPMQSort is implemented with $Task$ construct.

2.1.3 Single Construct (Parallel Synchronization)

The $Single$ construct is used to specify that a specific structure block is executed by only on thread of the team, which is not necessarily the Master thread. This can be useful for Tasks like control message during a parallel execution. The $Single$ construct has the syntax as in Figure 2.3.

```
#pragma omp task [clause [[,] clause] ... ] new-line
#pragma omp single
        structured-block
```

Figure 2.3 $Single$ construct definition by OpenMP 3.0 [1]

The $Single$ construct can be used within a parallel region and also ends with an implicit synchronization unless a nowait clause is specified. This thesis specifically demonstrates how to implemented the PPMQSort and $qsort()$ of SNPHAP with the $Single$ construct.

2.2 Multicore and Parallel Computing Technology

Currently, the trend in high-performance computing (HPC) systems has shifted towards cluster systems with multicore CPUs. Moreover, major chip manufacturers are producing multicore CPUs with/without HyperThreading. Therefore, several demanding programs can be multithreaded to gain higher performance with less expensive hardware and better efficiency [4].

A multicore CPU is a single computing component with two or more independent actual processors (cores) [35]. Moreover, all cores can execute independently and simultaneously. This category is called a Symmetric MultiProcessors (SMP) system (all processors connected to a large shared memory). Fortunately, multithreaded programming techniques can be applied to exploit the resource of multicore CPUs such as *Task-Level Parallelism* as well as *Data-Level Parallelism*. To effectively use multicore CPUs, programmers can write

multithreaded code by using a shared memory parallelism library such as OpenMP [1] or Intel Threading Blocks [36], or using a message passing library such as MPI [37]. For this thesis, OpenMP is considered to use for parallelizing sequential algorithms running on multicore CPUs.

The overview of the multicore CPUs architecture evaluated Intel E5400 Harpertown, Intel E5520 Nehalem-EP, Intel i3-2600 Sandybridge, Intel i7-2600 Sandybridge, AMD Opteron 8356 Barcelona , as well as AMD A6-3650 Llano are presented in the following.

2.2.1 Intel E5405 Harpertown

This family consists of dual and quad core CPUs featuring 1333 MHz to 1600 MHz front-side buses. Each CPU socket contains four processing cores. Each core has private 32 KB L1 cache and 12 MB (6 MB shared among two cores) L2 cache with Intel Advanced Smart Cache architecture. The Harpertown also has the capability of Intel 64 (Intel's x86-64 implementation), the Execute Disable Bit, and Virtualization Technology (VT-x) that help improve application performance.

2.2.2 Intel E5520 Nehalem-EP

Nehalem is the first Intel processor to implement a NUMA architecture incorporating QuickPath Interconnect (QPI) [38] links to communicate with others cores within a socket, and the first to incorporate an integrated memory controller. The E5500 Nehalem-EP machines, the successor to the Xeon Core microarchitecture, is based on the Nehalem microarchitecture. It has four processing cores per socket each with exclusive 32 KB L1 cache and 256 KB L2 cache. All the cores in a socket share an 8 MB L3 cache. It has an integrated memory controller supporting three DDR3 memory channels. In addition, two QPIs allow two processors to be interconnected as well as providing I/O connectivity.

The Xeon E5520 includes several new mechanisms that help improve application performance such as Turbo mode and HyperThreading. Turbo mode enables the processor frequency maximum of 2.53 GHz. For Intel's HyperThreading technology, two threads are enabled to execute on each core to hide memory latencies by switching between the hardware threads on memory stalls.

2.2.3 Intel i3-2100 Sandybridge

In early 2011, Intel introduced a new microarchitecture named Sandy Bridge. It kept all the existing brands from Nehalem, including Core i3/i5/i7. This family includes dual-core and quad-core CPUs variants and implementations targeted a single 32 nm manufacturing process for both the CPU and integrated GPU cores. It has an integrated memory controller supporting DDR3 memory channels.

The Intel i3-2100 also is one of Sandy Bridge microarchitecture. It has two processing cores per socket each with exclusive 32 KB L1 cache and 256 KB L2 cache. Two cores share an 3 MB L3 cache. In addition, DMI at 5GT/s, Smart cache 3MB and HyperThreading technology to improve application performance.

2.2.4 Intel i7-2600 Sandybridge

For Intel i7-2600, it is one of Sandy Bridge family. It has four processing cores per socket each with exclusive 32 KB L1 cache and 256 KB L2 cache. All the cores in a socket share an Smart 8 MB L3 cache. Moreover, DMI at 5GT/s and HyperThreading technology to improve application performance.

2.2.5 AMD Opteron 8356 Barcelona

The AMD Opteron 8356 also is one of the first x86 architecture processors to use a NUMA architecture. The CPU socket contains a single integrated memory controller that supports DDR2 memory channel and uses the Direct Architecture over high speed Hyper-Transport (HT) links between CPU sockets. It is composed of four processing cores per socket. Each core has a private 64 KB L1 cache and private 512 KB L2 cache, and each socket has a shared 2 MB L3 cache.

2.2.6 AMD A6-3650 Llano

The AMD A6-3650 codenamed Llano is one of processors in the A6 Series. The A6-3650 includes a Radeon 6530D GPU and a 2.6GHz four cores per socket each with exclusive 64 KB L1 cache and (16-way) 1024 KB L2 cache. It supports up to 1866 DDR3 memory. The Radeon 6530D has a 443MHz core clock and 320 Radeon cores. Feature support includes PCI express 2.0, AMD Dual Graphics, DX 11, and AMD Accelerated Parallel Processing Technology.

2.3 SNPHAP Algorithm

2.3.1 Haplotype Inference

The sequence of Human Genome contains approximately 3 billion base pairs ACTG (all of the bases in DNA) [39]. Normally, Genome variations are differences in the sequence of DNA from one person to another due to mutations that occur occasionally in a DNA sequence [40]. Therefore, these differences are called Single Nucleotide Polymorphisms (SNPs) as shown in Figure 2.4.

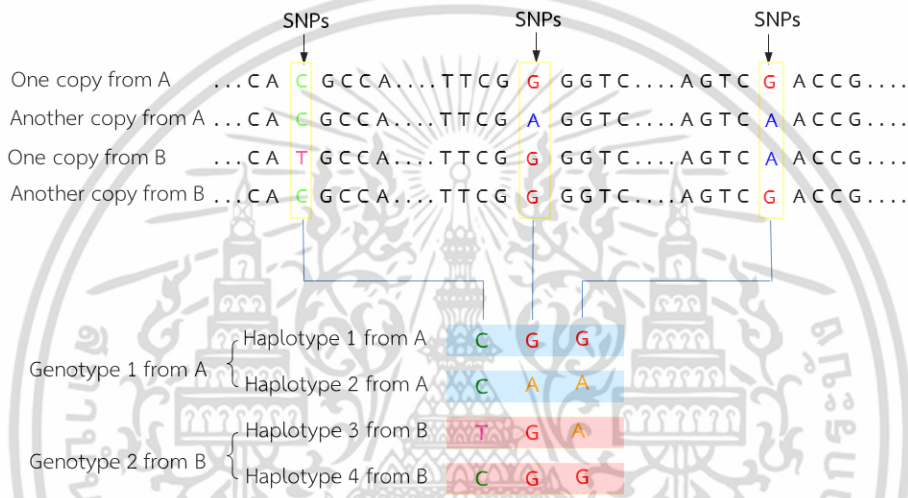


Figure 2.4 3 SNPs of 4 haplotypes and 2 genotypes from individual A's and individual B's chromosome 10.

In diploid organisms such as human beings, each individual has two different copies of each chromosome except sex chromosome; one from an individual's mother and another from the father [41]. Figure 2.4 shows individual A's and individual B's chromosome 10. Notice that each individual has two different copies of chromosome 10. Because individuals differ in SNPs, all other base pairs can be discarded to keep only SNPs. Therefore, haplotype data can be described as information from a single copy of SNP sequences in a chromosome and genotype data as information combined from a pair of haplotypes. In addition to Figure 2.4, two haplotypes are copied from individual A, while other two haplotypes are copied from individual B. Moreover, there are two genotypes; one from individual A, another from individual B.

For computational effectiveness on computer systems, a sequence of SNPs can be

represented as binary string of 0's and 1's for binary coding of haplotype information [42]. Generally, SNPs are bi-allelic that there are only two alleles in single SNP: wild type and mutation type. Figure 2.5 shows a simple coding of haplotypes for two individuals with just twenty SNPs each. As in those four haplotypes, A is wild type and G is the mutation type. Hence, 0 represents "wild type" and 1 represents "mutation".

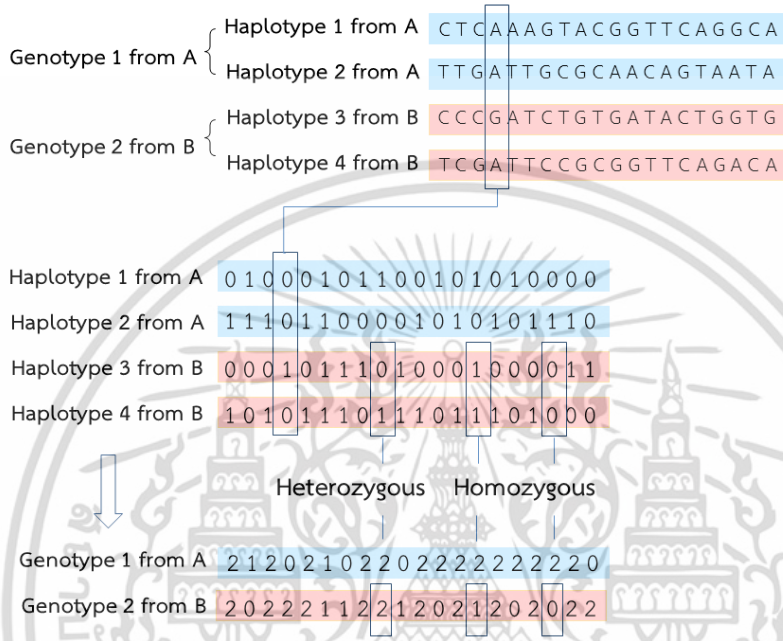


Figure 2.5 Describing SNPs, Haplotypes, Genotypes and a simple coding of haplotypes of two individuals with 20 SNPs each.

The genotype's SNPs of an individual can be encoded as a string of 0 or 1 where two haplotype SNPs are homozygous (either 0,0 or 1,1). On the other hand, a code 2 represents a heterozygous where one haplotype SNP is 1 and another is 0 (in any order). Therefore, a haplotype sequence becomes a pair of binary vectors while genotype becomes a vector of the alphabets (0, 1, 2). The genotypes data are more readily available; however, the haplotype information is more useful in studying human genetics and investigating many population processes. Therefore, this is the haplotype inference problem.

More details are explained in Appendix A.

2.3.2 SNPHAP: A Haplotyping Tool

Haplotyping tools are developed for studying human genetics and also used to investigate many population processes, such as migration and immigration rates, linkage disequi-

librium strength, and the related populations [12].

Several methods for estimating haplotypes and their frequencies have been proposed including EM [43–45], PHASE [46], and Subtraction [47]. Xu *et al.* [48] compared the accuracy and runtime performance of all three methods. They found that these methods are efficient and accurate for haplotype frequencies estimation. For runtime comparison with 5-SNPs (loci), EM is faster than Phase, but it is slower than Subtraction. EM and PHASE, however, provided better estimate compared with Subtraction. From the three studies [12,48,49], furthermore, EM algorithm shows better accuracy for haplotype frequencies estimation. Although the EM is fast with single runs for a small number of loci, it can take long computational with multiple runs and for large number of loci (causing large number of possible haplotype instances) [45, 50]. With the advent of large-scale genotyping platforms that allow researchers to observe more than one million loci the existing tools may not be able to handle a need to speed up existing haplotype inference tools.

There exists some software utilizing EM algorithm to infer haplotype frequencies including Fugue [51], HPlus [52], HelixTree [53] and SNPHAP [2]. The application chosen for the case study is SNPHAP, developed by Clayton. It is a sequential program for inferring haplotype frequencies using genotype data from a number of unrelated individuals, in addition to missing data at some loci. Moreover, it uses an EM algorithm to calculate ML estimation of haplotype frequencies given a genotype. The overall procedure comprises (shown in Figure 2.6) (1) Expectation step (*hap_posterior()* inside SNPHAP), (2) Maximization step (*hap_prior()* inside SNPHAP) and repeat (1)(2) until the haplotype frequencies are stable. More details are explained in Algorithm 3 of Appendix A.

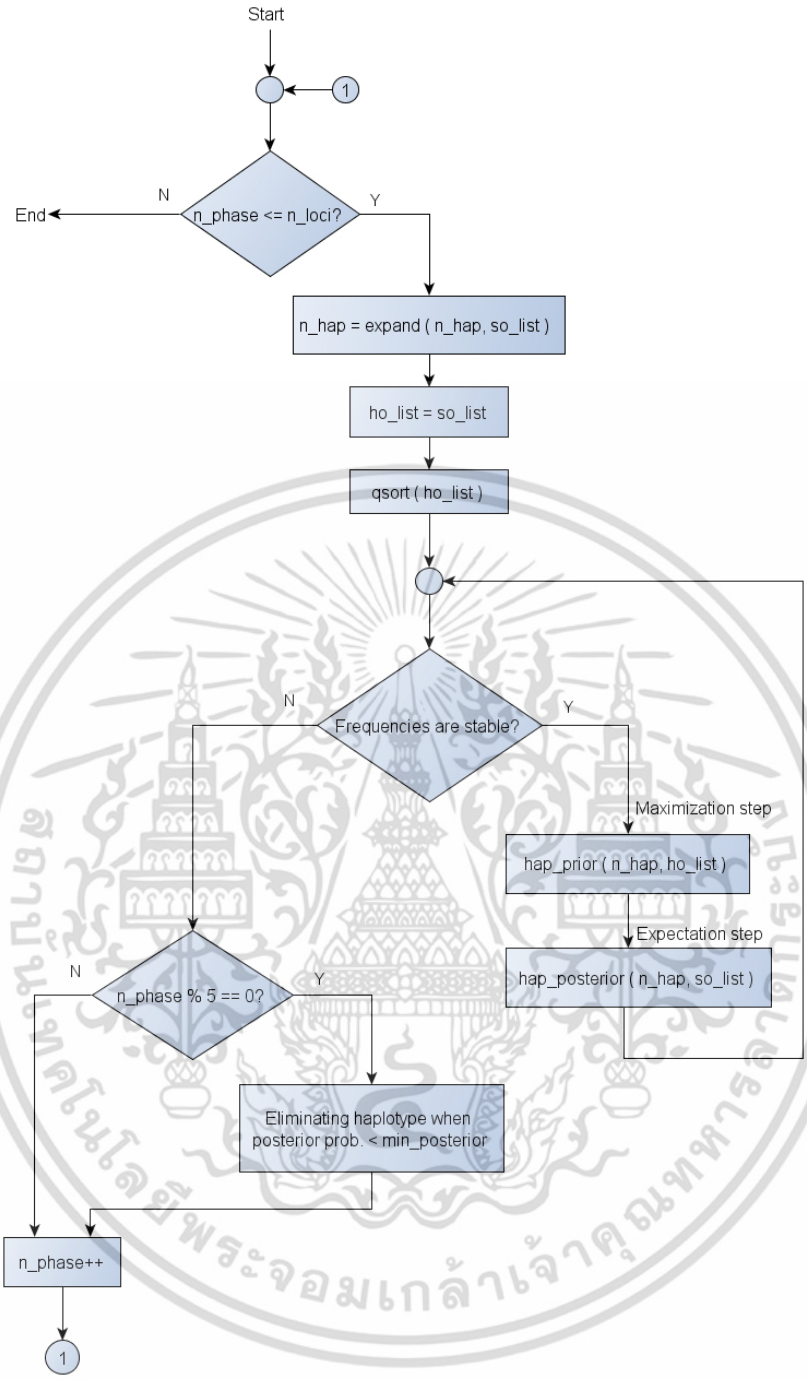


Figure 2.6 Flowchart of original SNPHAP (version 1.3.1) [2] with $hap\_prior()$ and $hap\_posterior()$ that will be multithreaded in Applying OpenMP parallel For construct subsection. and $qsort()$ that will be multithreaded in Applying OpenMP Task construct subsection.

Being a contribution, Eronen *et al.* [3] compare runtime per genotype (seconds) of the SNPHAP with other haplotyping software when varying the number of loci as shown in Figure 2.7.

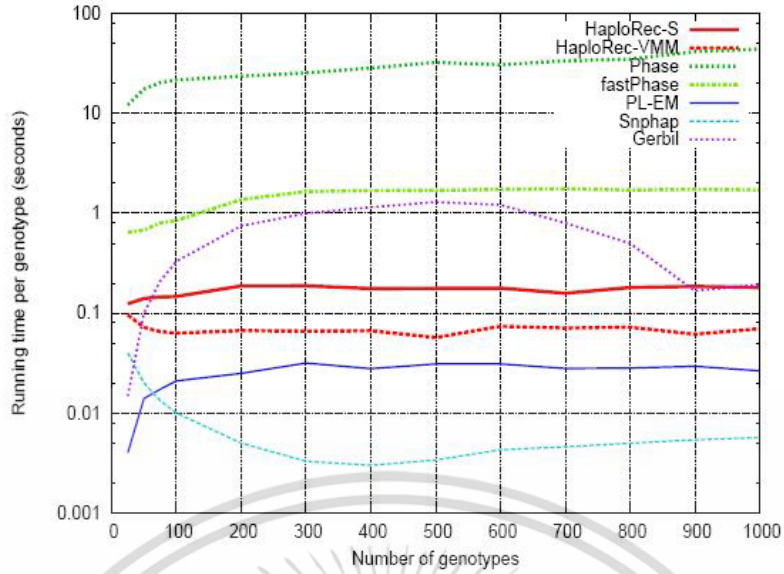


Figure 2.7 Runtime (Seconds) per genotype of various haplotyping software and SNPHAP is the fastest (Loci = 30) [3].

The compared tools include GERBIL [54] (version 1.0), PL-EM [55] (version 1.5, kindly provided by Zhaohui S. Qin), fastPhase [56] (version 1.1.3), Phase [57, 58] (version 2.0), and HaploRec (version 2.3) [59]. From this report, the SNPHAP is the fastest tool among others. Although the SNPHAP has the strong point of runtime, it is still the sequential (single thread) version. The runtime of SHPHAP can be further reduced with multithreading technique. On the other hand, the SNPHAP is not suitable for Clustering because a little bit of time of inferring haplotype in each EM iteration is overcome by communication time between Cluster nodes. The SNPHAP, moreover, is *communication bound* so that multithreading technique on multicore CPUs is the best solution.

2.3.3 Existing Parallel SNPHAPs

Recently, there has been little research to implement the EM algorithm for inferring haplotypes in parallel. Huang and Chen [18] have done experiments as a final project by inferring haplotype using EM algorithm with MPI (Message Passng Interface) library for passing information among the processors. They hierarchically partitioned the data and distributed to each processor for one fragment of the sequence. The final result was assembled progressively. Their program ran on the CTRIS cluster with up to 16 processors with only 250 samples and 130 loci. Finally, the 16-over 2-CPU speedup is 4x. This parallel solution can be unreachable to researchers without large computer clusters. The parallelization should

not be limited to a large cluster only.

2.4 Parallel QuickSort Algorithms

This section reviews the research background associated with Sequential QuickSort Algorithm, Stdlib qsort(), existing Parallel QuickSort Algorithms and Pivot Selection.

2.4.1 Sequential QuickSort Algorithm

QuickSort [15, 16] is the most famous and widely used sorting algorithm. The divide and conquer concept recursively partitions and swaps an input array into two halves: less than or equal (LEQ) half and greater than (GT) half with respect to a selected pivot element at each recursion level. The sequential QuickSort algorithm is presented in Algorithm 1.

The $Seq\_Partition()$ function is called in order to partition the input array into two halves where each half is sorted recursively with $Seq\_QuickSort()$ function and then return the pivot index. In general, the sequential partition algorithm compares elements with the selected pivot, which is located at the first, and swaps the elements that are less than or equal to the pivot to the left half and greater than the pivot to the right half. Then, it finally moves the pivot to the right location and returns the index of the pivot.

The time complexity on average is therefore $O(n \log n)$ although the poorly selected pivot can affect its complexity. Even worse, the worst case input array can make the complexity become $O(n^2)$. In terms of space requirements, QuickSort is considered to be an in-place algorithm using minimal extra memory. During the recursion, extra space for calling stack is proportional to $O(\log n)$. To optimize its performance, selecting good pivot(s) from several candidates has been considered and reviewed in Section 2.4.3. For fully expanded discussion and implementation see in [15, 16]. Moreover, the sequential QuickSort can be enhanced with several parallel techniques to run on any shared memory/multicore systems with multithreading operating system.

2.4.2 Stdlib qsort()

The Standard Library $qsort()$ is a very useful function for sorting an array of any data types with a user-defined comparison function. It is implemented in C/C++ and also provided as a built-in function of several C/C++ compilers. Its function prototype is declared in $Stdlib.h$ as follows.

Algorithm 1 The Sequential QuickSort Algorithm

```
1:  function Seq_QuickSort(a, start, end)
2:      if start < end then                                  ▷ if start < end
3:          p ← Seq_Partition(a, start, end)                 ▷ Partition two halves
4:          Seq_QuickSort(a, start, p − 1)                   ▷ Recursive on Left half
5:          Seq_QuickSort(a, p + 1, end)                     ▷ Recursive on Right half
6:      end if
7:  end function
8:
9:  function Seq_Partition(a,start,end)                      ▷ Sequential Partition Function
10:     pivot ← a[start]                                     ▷ Select the pivot element
11:     i ← start − 1
12:     j ← end + 1
13:     while TRUE do
14:         repeat j ← j − 1
15:         until a[j] <= pivot                              ▷ Move j until a[j] <= pivot
16:         repeat i ← i + 1
17:         until a[i] > pivot                               ▷ Move i until a[i] > pivot
18:         if i < j then
19:             Swap(a[i], a[j]                              ▷ Exchange a[i] <=> a[j]
20:         else
21:             return j                                     ▷ Return the pivot index
22:         end if
23:     end while
24: end function
```

```
void qsort(void  *base, size_t num_elements, size_t element_size,
   int (*compare)(void const *, void  const *));
```

The argument *base* is a pointer to the unsorted array, *num_elements* indicates the number of elements, *element_size* is the size of each element, and *compare* is a pointer to the user-defined function that returns integer values according to the comparison result.

2.4.3 Pivot Selection

PPMQSort is based on Hoare's algorithm [15] that partitions around the first element of subarray. However, selecting the first element is vulnerable to the risk of poor partitioning yielding $O(n^2)$ (worst case). A better technique is the Median of three method, selecting median of three elements. From distribution of the middle element, the pivot selection leans towards the middle and thus $O(n\ log\ n)$.

There are a few publications for pivot selection. Sedgewick [60] had proposed and analyzed a dual pivot approach that number of OS swap is more than classical quicksort

on average. Hennequin [61] analyzed quicksort with k >= 1 pivots. He found that the performance improved slightly and partitioning became too complicated. In 2009, however, Yaroslavskiy [62] proposed a new algorithm using a dual pivot partitioning technique and became the internal sorting in Oracle's java runtime library. The algorithm employs two pivots (p and q) to split the elements into 3 parts: small elements (smaller than p), medium elements (between p and q) and large elements (larger than q). The Yaroslavskiy's algorithm outperforms classical quicksort 10% faster. In [63], Wild and Nebel formulated and analyzed the core of Yaroslavskiy's dual pivot method. They showed that number of key comparisons is $1.9n\ log\ n+O(n)$ on average while $2n\ log\ n+O(n)$ of classical quicksort and $2.13n\ log\ n+O(n)$ of Sedgewick's dual pivot. Because of using two pivot selection, different behaviors are possible. Aumüller and Dietzfelbinger [64] analyzed all dual pivot algorithms and identified a new algorithm that minimizes the key comparisons to $1.8n\ log\ n+O(n)$. Kushagra, *et al.* [65] proposed cache behavior analysis as well as the performance of fast three pivot quicksort algorithm compared with classical quicksort and Yaroslavskiy's dual pivot quicksort. They claimed that the caching behavior of the three pivot quicksort algorithm outperforms single pivot and dual pivot quicksort in experiments.

For this thesis, Median of three technique (selecting three items, sorting them, using the one in the middle as pivot) is employed to partition two subarrays. Extensively studied by [60], obtaining a better partition needs a better pivot. And in [66, 67], Sedgewick practically implemented median of a random sample of the input to be more efficient in terms of performance. Moreover, Erokio [68] experimented to evaluate on worst case. He found that Median of three quicksort uses half the time of classical quicksort. Aumuller and Dietzfelbinger [64] observed that the comparison count of partition step is main cost according with [61] that the partition cost and overall cost of quicksort are more variants than with one pivot. Moreover, the partitioning cost depends on the partitioning procedure, meaning that partitioning should be simple. Therefore, Median of three is chosen for PPMQSort due to practical, simplicity and efficient.

2.4.4 Existing Parallel QuickSort Algorithms

Review of parallel QuickSort are given here.

In 1990, Heidelberger *et al.* [27] presented a parallelization of the Quicksort on a theoretical/ideal Parallel Random Access Machine with average of $O(\log n)$ time complexity.

In practice, the sequential QuickSort can be enhanced with several parallel techniques to run on any shared memory/multicore systems with multithreading operating system. Parallel versions of QuickSort normally start with partitioning data into several chunks to fit any cache level depending on the size. These chunks can be partially or fully sorted and then merged to form bigger chunks. These two steps may be recursive as indicated in the Recursion row of Table 2.1. Some algorithms may use extra space to hold the intermediate results as shown in Ex. Space row. Eventually, they shall be fully sorted again with either the Stdlib $qsort()$ or others. Tsigas and Zhang [28] proposed a fine-grain (block-based) parallel Quicksort algorithm. Subsequently, [29] presented several alternative algorithms of parallel Quicksort based on pthreads and OpenMP 2.0.

Since 2007, Multi-Core Standard Template Library (MCSTL) has been developed by Singler *et al.* [69]. It is integrated into the STL implementation of the GNU libstdc++ library, as called parallel mode. The algorithm is similar to the one by Tsigas and Zhang [28], which uses strategy based on $nthelement$ and $partialsort$. Each thread holds two chunks of size B (threshold) from the end of each side. It then partitions those two chunks until one of them runs empty. If the left chunk runs empty, it reserves a succeeding block using a fetch-and-add primitive, while other is reserved as a preceding block. This process is recursive until the elements between the left and the right boundaries are less than threshold. Finally, this threshold size calls the sequential sort. It can achieve Speedups of 21x on an 8-core 32-thread SUN-T1.

Publications for 2008, Traore *et al.* [70] proposed a provable work-optimal parallelizations of STL algorithms based on recursive range partitioning and work stealing technique. The partition algorithm relies on a list of successful steals, called deque-free. Their algorithm divides data into two subranges and partitions them. The partition follows the STL partition scheme that splits subranges into two halves, the thief partitioning the right-most half of the left block, and the left-most half of the right block. If one half has already been partitioned, the thieves try to steal two blocks of elements. The remaining blocks of elements must be re-ordered. The sorting part of their, algorithm is based on introspective sorting. The sequential partitioning is called when the input array is less than threshold. Otherwise, the previous deque-free partition is performed. It can achieve higher Speedup at 8.1x on an 8 dual-core (16 processors) AMD Opteron at 16 threads. Frias and Petit [71] modified a cleanup algorithm to achieve a minimal number of comparisons. Their algorithm is applied

on the top of Strided, Blocked and F&A algorithms [28, 69]. The implementations follow to the STL partition specifications. From experimental comparison, F&A Algorithm is the best. The cleanup algorithm, however, is very limited due to the number of misplaced elements after the parallel phase is very small. Moreover, I/O limits performance as the number of threads increases.

By 2010, Rashid *et al.* [72] enhanced Tsigas and Zhang's [28] PQuicksort on x86 Multithreaded Architectures. Man *et al.* [30, 31] developed $psort()$ algorithm to be compatible with Stdlib $qsort()$. The input array is divided into groups and $qsort()$ them. Later on, these partitions can be merged using extra space and finally $qsort()$ them again. Their work can achieve Speedup by 11 times faster with up to 24 cores. Kim *et al.* [32] have shown that an embedded dual core OMAP-4430 can achieve 1.47x Speedup from their Introspective Quicksort algorithm.

In 2013, Zurex *et al.* [73] proposed a quick-merge hybrid algorithm that each core sorts partial data by the quicksort algorithm. Then, the results of each core are merged with parallel merge-sort algorithm. The computational complexity is $O(n\ log\ c + \frac{n}{c} \log \frac{n}{c})$, where c is the number of processor cores. Mahafzah [33] spitted the input array with multi-pivot/thread into partitions using extra space and then sort them in parallel up to 8 threads. Saleem *et al.* [74] estimated Speedup for QuickSort and Merge sort algorithms using Intel Cilk Plus.

More recently, as of 2015, Maus [75] divided input array into k equally sized segments. Those segments are assigned to k threads and partitioned within its own segment by the original QuickSort. The algorithm then swaps all small elements with the large elements of the pivot index in parallel. Recursively apply with half of the threads to perform the left segment, and another half to the right segment. It can achieve Speedups 1.7x on 2-core with Hyperthread Core i7 and 2.5x on 4-core with Hyperthread Core i7 for sorting uniform distribution 10M 32-bit integer data and 6.5x on 32-core with Hyperthread Xeon while sorting uniform distribution 100M 32-bit integer data. The comparison of some previous parallel QuickSort algorithms is shown in Table 2.1 in chronological order from left to right.

Table 2.1 Comparison of previous parallel QuickSort algorithms, Par. = Parallel, Seq. = Sequential, NA=Not Available

Year	2003	2004	2011	2011	2013	2014
Reference	[28]	[29]	[31]	[32]	[33]	[74]
Algo. Name	PQuicksort	cv_1.0	psort1	Introspective	QuickSort	Quicksort
Partition	Par. in blocks of L1 size	Seq.	Par. n/c and $qsort()$	Seq. n/c	Par. multiple pivots	Seq. n
Merge	Seq. Swap	No	Seq. Merge and and $qsort()$	No	No	No
Recursion	Yes	Yes	No	No	No	Yes
Time Complexity	$O(\frac{n}{c} + \frac{n}{c}\log\frac{n}{c})$	NA	$O(n + \frac{n}{c}\log\frac{n}{c})$	$O(\frac{n}{2} + \frac{n}{2}\log\frac{n}{2})$	$O(\frac{n}{h}\log\frac{n}{h})$	NA
Extra Space(size)	No	No	Yes(n)	No	Yes(n)	No
Using $qsort()$	Similar	No	Yes	No	No	No
Other Sort	Insertion	No	No	Insertion	No	No
Library	NA	pthreads	OpenMP 3.0	OpenMP 3.0	pthreads	Cilk Plus
Pros	Cache efficient, Fine-grained	Load balance- with Busy Waiting	Qsort() lib. compatible, Good load balance	Limit deep partition, Cache friendly, Good load balance	Utilize SMT architecture, Good load balance	Easily
Cons	Bottleneck- in Seq. Swap, Special Sync. Instruction	Sync. added, Less algorithm details	Difficult to implement, High Overhead	No nested parallelism, Seq. Partition	Sync. added, Extra Space	Unpopular lib.

CHAPTER 3

PARALLEL SNPHAP

This thesis can be useful to parallel program developers as a parallelizing/multithreading methodology especially for SNPHAP on multicore CPUs. As mentioned earlier in Section 2.3.2, SNPHAP is the fastest solution for inferring Haplotype based on Expectation Maximization algorithm. This section is divided into three subsections; The first one is Development of OMP SNPHAP. The second and third are Experiment Results and Discussions, and Theoretical Analysis, respectively. Notations used in this chapter are listed in Table 3.1.

3.1 Development of OMP SNPHAP

To parallelize (multithread) the sequential program, SEQ SNPHAP, to execute on the multicore CPUs in multiple threads, multithreading technique can be applied by inserting OpenMP constructs on hotspot and independent functions. However, not all modules/functions in a sequential program can be converted to multithreaded functions.

The methodology for parallel SNPHAP consists of four major steps as shown in Figure 3.1: 1) Profiling (Gprof), 2) Multithreading (GCC + OpenMP), 3) Verification, and 4) OpenMP Profiling (ompP). This methodology has applied these steps to SEQ SNPHAP in order to create a multithreaded version of SNPHAP called OMP SNPHAP. Use of a profiling tool helps identify the most time-consuming areas in the program. The information from the profiler can be used to appraise performance gain in the OpenMP profiling step. GCC compiler + OpenMP constructs will be used to force multithreading at hotspot functions that are computational intensive according to the GProf profiler [76]. Multithreading and Verification steps must be repeated to ensure that the optimized code produces the same outputs as those of the sequential version.

Finally, OpenMP (ompP) [77] performs overhead analysis (synchronization, load imbalance, thread management, limited parallelism) and performance properties. This information can be used to evaluate and justify the parallel performance. If the achievable Speedup is not satisfactory, the multithreading step can be repeated. The tools for this thesis are free/open source tools as well as flexible and easy to use. Each step will be explained in the following subsections.

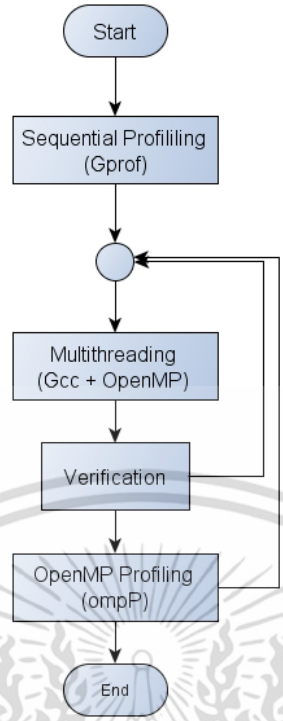


Figure 3.1 Flowchart of applying the multithreading methodology using OpenMP 3.0.

3.1.1 Sequential Profiling: GProf

In this subsection, the most time consuming functions how to be found by the profiling tool and evaluating the parallel performance measurement of the parallel fractions.

First of all, the SEQ SNPHAP is profiled to find all hotspot functions that consume significant portion of execution time by GProf tool [76]. The results are shown in Figure 3.2. Then, the OpenMP constructs are inserted on those functions to convert them as threads just like Fork-Join model.

According to Figure 3.2, it is clear that *cmp_hap()* is called by 874,204,285 times, while the *hap_posterior()* and the *hap_prior()* are called by just 4,080 and 3,930 times, respectively. Among 874,204,285 times, the *cmp_hap()* is called 568,631,912 and 293,886,461 times by *hap_prior()* and *hap_posterior()*, respectively. Therefore, the *cmp_hap()* should be parallelized within *hap_prior()* and *hap_posterior()* and executed by multicore CPUs to significantly reduce the total execution time. Moreover, after the the SEQ SNPHAP has been further profiled, the sorting function ($qsort()$) called by the $cmp\_hap()$ consumes more execution time. Therefore, the $qsort()$ should be parallelized too.

Index	%time	self	children	called	name
[1]	100	0.11	1065.70		name [1]
		37.60	618.16	3930/3930	hap_prior [3]
		65.78	320.99	4080/4080	hap_posterior [4]
.....					
		319.48	0.00	293886461/874204285	hap_posterior [4]
		618.16	0.00	568631912/874204285	hap_prior [3]
[2]	89.2	950.35	0.00	874204285	cmp_hap [2]
.....					

Figure 3.2 Results of GProf v.2.18.50 profiling the SEQ SNPHAP program.

3.1.2 Multithreading with OpenMP 3.0

For the Multithreading step, the SEQ SNPHAP will be introduced how can be the multithreaded program by applying *parallel For* construct and *Task* construct to improve the performance. Each construct will be explained in the following subsections.

3.1.2.1 Applying OpenMP parallel For Construct

The *parallel For* construct is used to split up the loop iterations among the threads and also distribute the workload equally at run time. As referred to profiling in the previous subsection, the loop of the *cmp_hap()* should be optimized within both the *hap_prior()* and the *hap_posterior()* to gain performance. The *cmp_hap()* compares a couple of haplotypes until it reaches the last haplotype. But the loops of *cmp_hap()* are not independent. So, parallelizing the *cmp_hap()* is not appropriate due to its small loop body. In order to run *cmp_hap()* in parallel, both the *hap_prior()* and the *hap_posterior()* have to be modified by moving the *cmp_hap()* to compute in the outer loop. Moreover, the *cmp_temp[]* array is allocated to store temporary output for this technique. In addition, the outputs of comparison are kept in *cmp_temp[]* array for availability for the next $while$ loop.

The OMP SNPHAP version of the *hap_prior()* and *hap_posterior()* are shown in Figure 3.3. As it can be seen in each function, a *parallel region* is defined to contain a *work-sharing* construct. A parallel region is a block of code executed by multiple threads simultaneously, while a work-sharing construct divides the iterations of the loop to distribute over the CPU cores.

A *parallel* construct is used to define a parallel region and a *For* construct is used

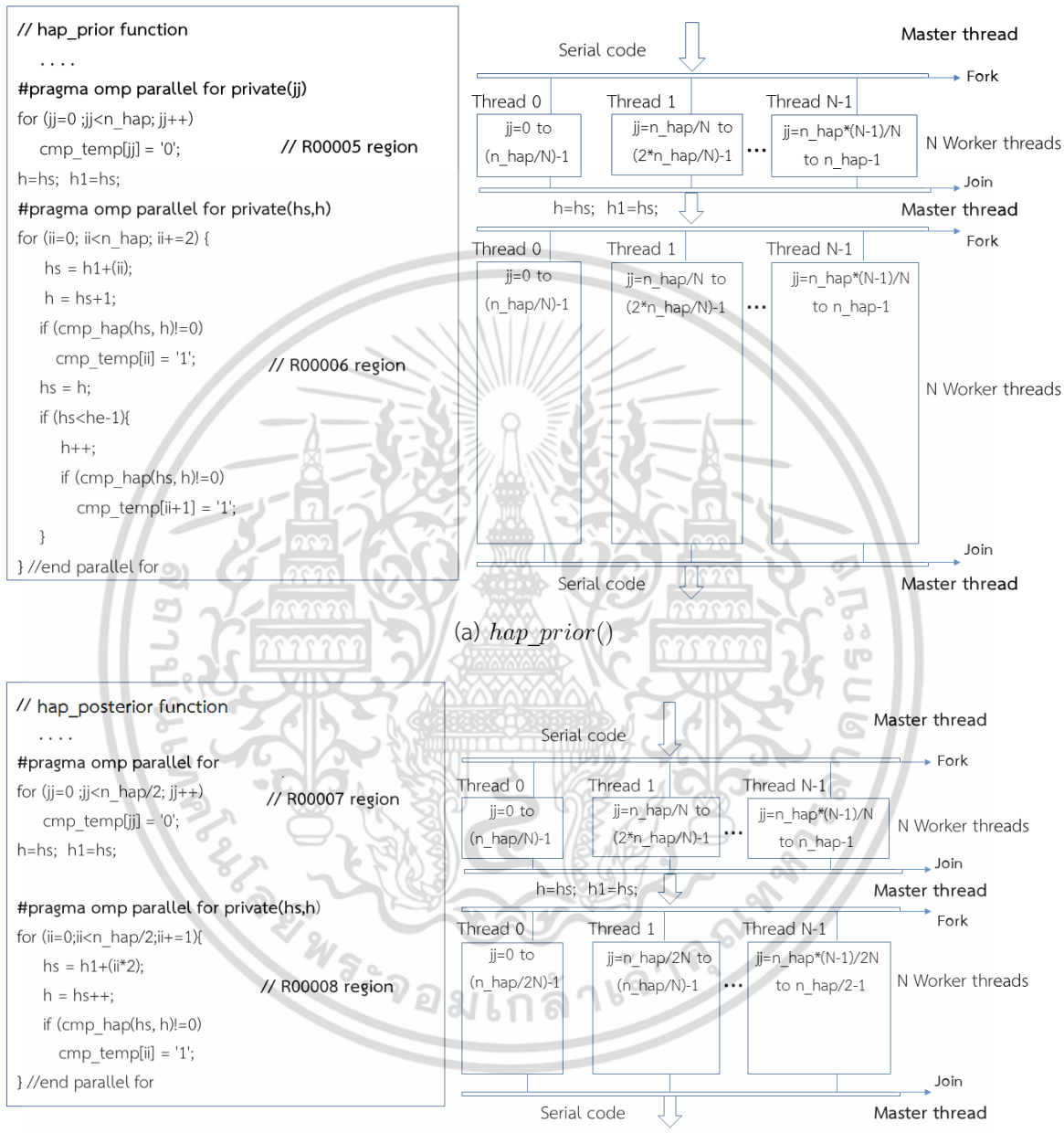


Figure 3.3 Parallel execution of $cmp\_hap()$ and code optimization inside (a) $hap\_prior()$ and (b) $hap\_posterior()$ with OpenMP $parallel\ For$ construct.

to define a work-sharing to split up the loop iterations among the threads. In addition, the *private(hs,h)* clause is added to define *hs* and *h* pointers as a private copy of each thread. When the Master thread encounters the parallel region, the code within the parallel construct is copied to a number of T Worker threads. On the other hand, different sets of haplotypes (data) are distributed to those worker threads for parallel execution. The number of threads for the parallel region is set with the $omp\_set\_num\_threads()$.

3.1.2.2 Applying OpenMP Task Construct

Recently, OpenMP version 3.0 [1] introduced $Task$ construct to express the parallelism in several algorithms whose the amount of hardware/platform parallelism unknown in advance. Some application uses $Task$ constructs to express the parallelism in algorithms such as Floorplan Alignment, SparseLU, Multisort [78] and so on.

In this work, the *Task* construct is used to parallelize recursive algorithms (*qsort()*) inside *main()* as shown in Figure 2.6. As referred to previous subsection, the execution of *cmp_hap()* has been parallelized inside *hap_prior()* and *hap_posterior()*, respectively. From Sequential Profiling step, the sorting function (*qsort()*) is called *cmp_hap()* consumes the most execution time too. So, previous OMP SNPHAP has been extended to include multithread version of *qsort()*. Due to the standard library *qsort()* in SNPHAP to difficultly parallelize. Then, a new *qsort1()* has been developed with OpenMP *Task* construct as shown in Figure 3.4 so that it can be called from $main()$ as the following,

$$qsort1(ho\_list, n_h, sizeof(HAP*), cmp\_hap); \tag{3.1}$$

where ho_list is the haplotype list, n_h is the number of haplotype instances, $sizeof(HAP*)$ is the memory size of one haplotype, and cmp_hap is the haplotype comparison function. From Figure 3.4, it can be seen that $begin$ is the left-most index of the subarray of haplotype list and end is the right-most index of the subarray of haplotype list. Two array indices, ii and jj are instantiated. Let ii is initially "$begin$", and jj is initially "end", so that ii and jj sandwich the items to be sorted. OpenMP *Parallel/Single* construct in $qsort1()$ is incorporated in the first call of $sort()$ function (using *#pragma omp parallel* and *#pragma omp single*). As mentioned earlier, the thesis makes use of $Task$ construct to create tasks recursively (using *#pragma omp task*). A thread executes a task, which may be any thread

```
static void sort(char *array, size_t size, int (*cmp)(const void*, const void*), int begin, int end)
  if(begin == end) return;
  int ii=begin;
  int jj=end;
   void *pivot= array + (((begin+end)/2) *size);
  do {
      while ( cmp(array+(ii*size),pivot) < 0) ij++;
      while ( cmp(array+(jj*size),pivot) > 0) jj--;
      if (ii<=jj) {
                swap(array+(ii*size),array+(jj*size),size);          // Pivot partition
                ii++;
                jj--;
       }
  } while (ii<=jj);
   #pragma omp task
    sort(array,size,cmp,begin,jj);
   #pragma omp task                                                  // Recursive  sort()
   sort(array,size,cmp,ii,end);
}
void qsort1(void *array, size_t nitems, size_t size, int (*cmp)(const void*, const void*,)) {
   #pragma omp parallel
   #pragma omp single   {
      sort(array, size, cmp, 0, (nitems-1));                        // Calling sort()
   }
}
```

Figure 3.4 Parallel $qsort1()$ with OpenMP 3.0 $Task$ construct.

in the thread pool. Moreover, when a thread finishes executing a task, it catches a new task to execute. In this way, all threads can execute tasks without barrier synchronization, thereby improving load balancing. The number of threads for the parallel region is set with the $omp\_set\_num\_threads()$ function.

3.1.3 OMP SNPHAP Verification

The OMP SNPHAP results must be verified again and again until they are close to those of sequential version. As shown in Figure 3.5, a utility program named Beyond Compare matches and differentiates both output files from OMP and SEQ SNPHAP in black and red respectively.

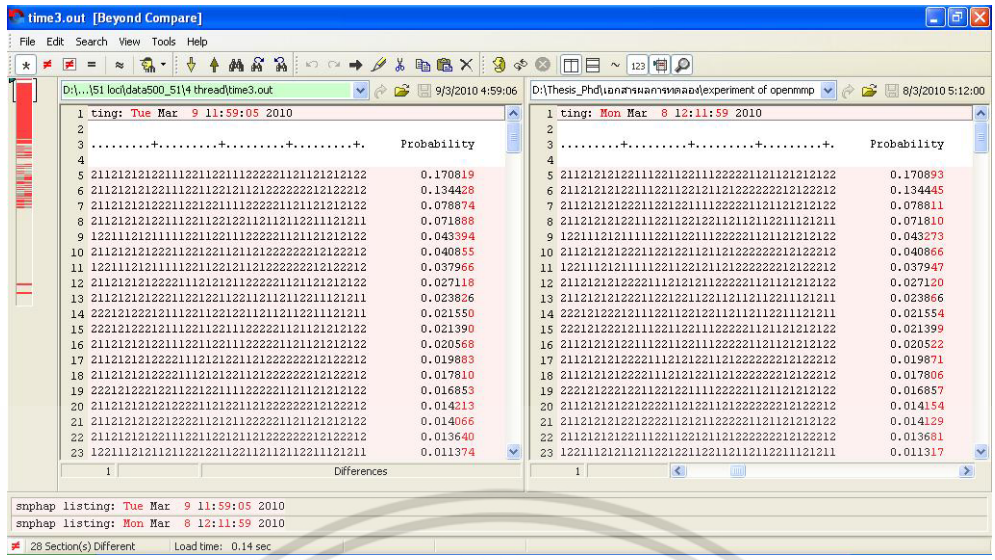


Figure 3.5 Verifying haplotyping results of OMP and SEQ SNPHAP by using Beyond Compare.

3.1.4 OMP SNPHAP Profiling

After the OMP SNPHAP results are completely verified in step 3 (Output Verification, $Task$ and $parallel\ For$ constructs are analyzed to investigate the performance characteristics of multithreading with OpenMP profiling tool named ompP [77, 79]. As shown in Figure 3.6, *ompP* profiling report consists of Header, Region Overview, Flat Region Profile, Call-graph Profile and Overhead Analysis Report. Header contains general information such as data and time of the program. Region Overview shows number of OpenMP regions (constructs) and their source code locations. The *ompP* can examine the effectiveness and efficiency of OpenMP parallel constructs. The profiling report consists of threads distribution and overhead analysis. This thesis shall determine each parallel region whether it can be parallelized and sped up in accordance with the number of threads forked by OpenMP. To instrument with *ompP*, prefix gcc openmp with $kinst - ompp$ is added. The explicit initialization ♯*pragma pomp inst init* should be placed at the beginning of *main()*. The profiles of each individual parallel region (construct) and the entire program are reported. According to Figure 3.3, the parallel construct labelled R00005, R00006, R00007, and R00008 corresponding to $hap\_prior()$ and $hap\_posterior()$ consumes the biggest portion of total execution time.

In summary, the methodology is the use of OpenMP library directives to perform multithreading at functions consuming most execution time according to the profiler. However,

```
------------------------------------------------------------------------
----     ompP Region Overview     ---------------------------------------
------------------------------------------------------------------------
PARALLEL: 1 region:
 * R00003 snphapfun.c (81-85)

PARALLEL LOOP: 4 regions:
 * R00005 snphapfun.c (404-409)
 * R00006 snphapfun.c (414-433)
 * R00007 snphapfun.c (595-599)
 * R00008 snphapfun.c (603-615)

SINGLE: 1 region:
 * R00004 snphapfun.c (82-85)

TASK: 2 regions:
 * R00001 snphapfun.c (69-70)
 * R00002 snphapfun.c (73-74)

TASKEXEC: 2 regions:
 * R00001 snphapfun.c (69-70)
 * R00002 snphapfun.c (73-74)

------------------------------------------------------------------------
----     ompP Callgraph     ---------------------------------------------
------------------------------------------------------------------------

  Inclusive   (%)    Exclusive   (%)
   2.21 (100.0%)     0.63 (28.47%)              [snphap: 32 threads]
   0.56 (25.23%)     0.02 ( 1.01%) PARALLEL  |-R00003 snphapfun.c (81-85)
   0.54 (24.22%)     0.54 (24.22%)   SINGLE  |  +-R00004 snphapfun.c (82-85)
   0.00 (0.001%)     0.00 (0.001%)     TASK  |     |-R00001 snphapfun.c (69-70)
   0.09 ( 4.17%)     0.03 ( 1.46%) TASKEXEC  |     |-R00001 snphapfun.c (69-70)
   0.03 ( 1.49%)     0.03 ( 1.49%)     TASK  |     |  |-R00001 snphapfun.c (69-70)
   0.03 ( 1.22%)     0.03 ( 1.22%)     TASK  |     |  +-R00002 snphapfun.c (73-74)
   0.10 ( 4.54%)     0.04 ( 1.80%) TASKEXEC  |     |-R00002 snphapfun.c (73-74)
   0.03 ( 1.38%)     0.03 ( 1.38%)     TASK  |     |  |-R00001 snphapfun.c (69-70)
   0.03 ( 1.36%)     0.03 ( 1.36%)     TASK  |     |  +-R00002 snphapfun.c (73-74)
   0.00 (0.000%)     0.00 (0.000%)     TASK  |     +-R00002 snphapfun.c (73-74)
   0.20 ( 9.13%)     0.20 ( 9.13%)  PARLOOP  |-R00005 snphapfun.c (404-409)
   0.32 (14.55%)     0.32 (14.55%)  PARLOOP  |-R00006 snphapfun.c (414-433)
   0.22 (10.06%)     0.22 (10.06%)  PARLOOP  |-R00007 snphapfun.c (595-599)
   0.28 (12.56%)     0.28 (12.56%)  PARLOOP  +-R00008 snphapfun.c (603-615)

------------------------------------------------------------------------
```

Figure 3.6 Profiling result of OMP SNPHAP by ompP with 32 threads to examine %run time of $hap\_prior()$ and $hap\_posterior()$, and $qsortl()$.

due to dependencies and sequential behavior of SNPHAP, the source code must be modified to preserve correctness. In addition, parallel behavior must be profiled to further improve performance and efficiency due to OpenMP overhead.

3.2 Experiment Results and Discussions

In this section, the experiment set up and measurement, Runtime and Speedup of the OMP SNPHAP running under various multithreaded configurations will be presented.

3.2.1 Experiment Set Up

The results reported in this thesis are based on experimental measurements on three multicore CPUs systems: Intel Xeon E5405 Harpertown, Intel Xeon E5520 Nehalem-EP (Hy-

Table 3.1 SNPHAP Notations

a	A constant incorporating the multinomial coefficient
c_j	The number of possible haplotype pairs at the j_{th} genotype
f_p	The fraction of a program that can be parallelized
$f_t^{(l+1)}$	"Posterior" haplotype frequencies at l_{th} iteration and t_{th} haplotype
f_t, f_{t+1}	The frequencies of the h_t and the h_{t+1}
$\widetilde{f}_t^{(l)}$	"Prior" haplotype frequencies at l_{th} iteration and t_{th} haplotype
G	Total number of genotypes in a population
h	Individual haplotype
h_{p_i}	Probability of i_{th} haplotype pairs of each c_j
i	The i_{th} position of haplotype pairs
j	The j_{th} position of genotype
k	The number of heterozygous loci sites
k_j	The number of heterozygous loci at the j_{th} genotype
l	EM iteration number
L	Likelihood function of the haplotype frequencies
n	Observed genotype data
n_c	The maximum number of loci () of each genotype
n_h	The maximum number of haplotypes
n_ϕ	The number of current phase ϕ
$o(p)$	Overhead of p processor cores
$o(T)$	Overhead of T threads and OpenMP
p	The number of processor cores
P_j	Probability of the j_{th} genotype
$P(h_t, h_{t+1})$	The probability of i_{th} haplotype pairs made up
t	The t_{th} position of haplotype
T	The number of threads
T_{post}	The number of threads at hap_posterior()
T_{prior}	The number of threads at hap_prior()
Θ	Logarithm Likelihood value
ε	Tolerance of logarithm Likelihood

perThread) and AMD Opteron 8356 Barcelona located at National Center for Genetic Engineering and Biotechnology (BIOTEC), Thailand. The operating systems are Linux Fedora 9 kernel 2.6.25 SMP version for E5405 and E5520 and Linux CentOS 5.2 kernel 2.6.18 SMP version for AMD Opteron. The modified SNPHAP 1.3.1 is compiled with GCC 4.4 together with OpenMP 3.0 library (only OpenMP and no other compiler optimizations) and profiled with GNU GProf v.2.18.50.

Table 3.2 provides a summary of evaluated systems including a dual-socket quad-core (8 cores total), a dual-socket quad-core (HyperThread) and an octal-socket quad-core (32 cores total) systems. These systems have a range of characteristics of particular interest to

multithreading performance. The OMP SNPHAP shall be improved by each OpenMP construct to get the highest performance. Secondly, the systems span a range of cache capacity and cache sharing configurations. The L3 cache capacities range from as little as 512 KB per core (AMD Opteron 8356) to as high as 2 MB per core (Intel Xeon E5520). Thus, the effect of cache capacity shall be explored. Moreover, the Intel Xeon E5405 system has on each socket two L2 caches as 3MB per core, each of which is shared by a pair of cores. Experiment (simulation) genotype data set can be obtained from Snap [80], the open source tool for generating the genotype data, with parameters as shown in Table 3.3.

Table 3.2 Architectural details of multicore systems used in this study. †shared among cores on a socket. ‡shared among 2 cores on a socket.

System Core Architecture	Intel E5405 Harpertown	Intel E5520 Nehalem-EP	AMD Opteron 8356 Barcelona
Socket x cores x threads	2 x 4 x 1	2 x 4 x 2	8 x 4 x 1
# threads	8	16	32
Clock (GHz)	2	2.66	2.3
L1/L2/L3 cache (KB) per core	32/6144‡/-	32/256/8192†	64/512/2048†
DRAM Capacity (GB)	4	12	16
Bandwidth (GB/s)	21.33	51.2	21.33

Table 3.3 Parameter set of the experiments

Parameters	Values
Maximum thread number (T)	1, 2, 4, 8, 16, 32, 48, 64
Number of genotypes (G)	500, 1000, 2000, 5000, 10000
Number of loci in genotypes (n_c)	21, 51, 101, 151

3.2.2 Runtime

From a given set of parameters in Table 3.3, the Runtime of SEQ SNPHAP on AMD Opteron 8356 machines compared with the Runtime of OMP SNPHAP on Intel Xeon E5405, Intel Xeon E5520, and AMD Opteron 8356 machines for Combined Parallelization *(parallel For + Task)* are plotted as shown in Figure 3.7.

Here are the observations in general. It can be observed that the Runtimes tend to grow up along with the number of loci , n_c, increases. Moreover, Intel Xeon E5520 can run OMP SNPHAP and achieves better Runtime than AMD Opteron 8356 and Intel Xeon E5405 due to higher clock frequency (2.6 GHz vs. 2.3 GHz vs. 2 GHz, respectively), larger and deeper cache per 4 cores (8-MB L3 vs. 2-MB L3 vs. 2x6-MB L2, respectively), much higher memory

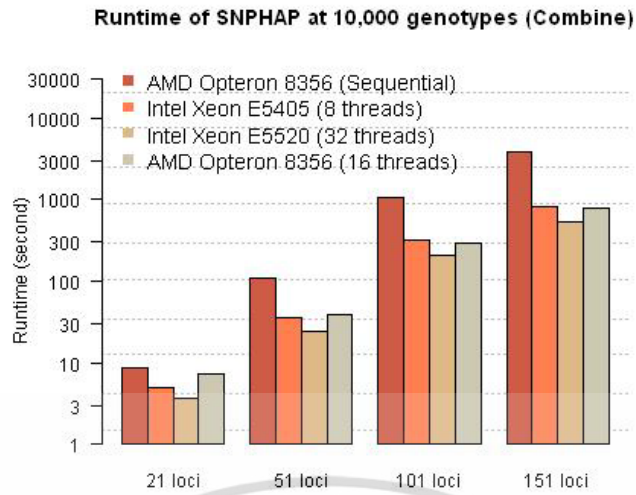


Figure 3.7 Runtime comparison of SEQ SNPHAP on 32-Core AMD Opteron 8356 and OMP SNPHAP on 8-Core Intel Xeon E5405 (8 threads), 8-Core Intel Xeon E5520 (32 threads), and 32-Core AMD Opteron 8356 (16 threads) at 10,000 genotypes, respectively (OpenMP only, No other compiler options).

bandwidth, better memory technology (DDR3 vs. DDR2 vs. DDR2, respectively).

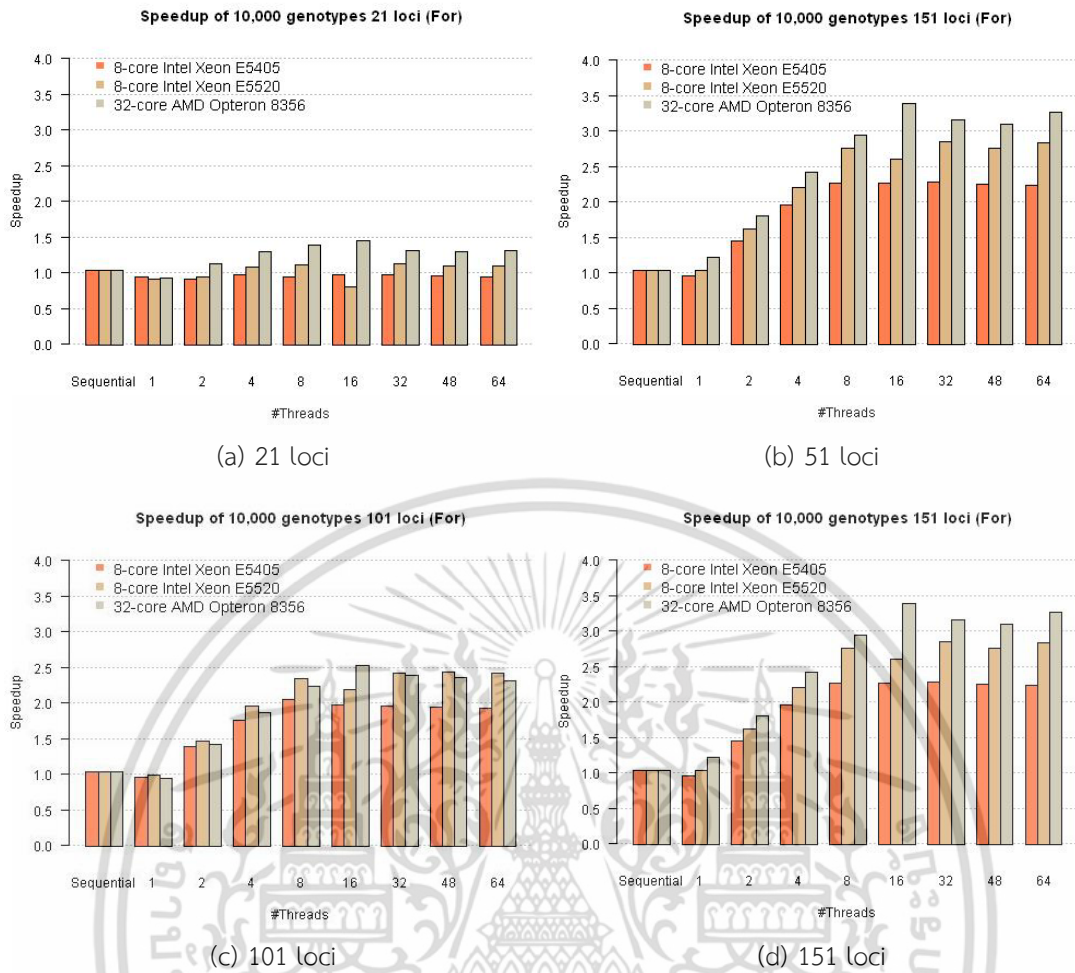
To simply understand and compare the results, the Speedup of Runtimes between SEQ SNPHAP and OMP SNPHAP of Intel Xeon E5405, Intel Xeon E5520 and AMD Opteron 8356 machines are plotted in the next subsections.

3.2.3 SpeedUp

For result comparison, the Speedups are presented in terms of three techniques: applying only *parallel For* construct, applying only $Task$ construct and combining both *parallel For* construct and $Task$ construct. The Speedup of each technique will be shown and described in the following subsections.

3.2.3.1 parallel For Construct

Only *parallel For* construct is applied to improve the performance. To simply understand the experimental results and description, let a long word "*parallel For* construct" as short word "*For* construct". Figure 3.8 shows Speedup of OMP SNPHAP for *For* construct on of three CPUs compared to sequential version of 10,000 genotypes at different numbers of loci: 21, 51, 101 and 151, respectively. It can be observed that the Speedups of all CPUs tend to grow up along with the number of threads increase except in (Figure 3.8 (a) 21 loci), no Speedup from Intel Xeon E5405. The maximum Speedup of AMD Opteron 8356 of



(a) 21 loci (b) 51 loci

(c) 101 loci (d) 151 loci

Figure 3.8 Speedup Comparison of OMP SNPHAP for *For* construct on Intel Xeon E5405, Intel Xeon E5520 and AMD Opteron 8356 of 10,000 genotypes (a) 21 loci (b) 51 loci (c) 101 loci and (d) 151 loci (OpenMP only, No other compiler options).

3.35 can be achieved at larger number of loci, data set, and particularly at 16 threads and decrease slightly later on. For Intel Xeon E5520, the maximum Speedup is 2.73 at 32 threads at larger number of loci and larger data set. Moreover, the Speedup is maximized at 8 threads and slowed down at 16 threads. It turns out at 32 threads and decreases slightly later on. The reasons behind this effect are the system might use only one memory controller, it follows that the SNPHAP does not benefit much from the second memory controller [81], the SNPHAP is memory bounded program for NUMA (E5520) and also L2 cache capacities as little as 256 MB per core. As of Intel Xeon E5405, the Speedup tends to grow up along with the number of threads increase as same as both machines except in (Figure 3.8 (a) 21 loci); that is, as the Speedup tends to slowdown when the number of threads increases. The

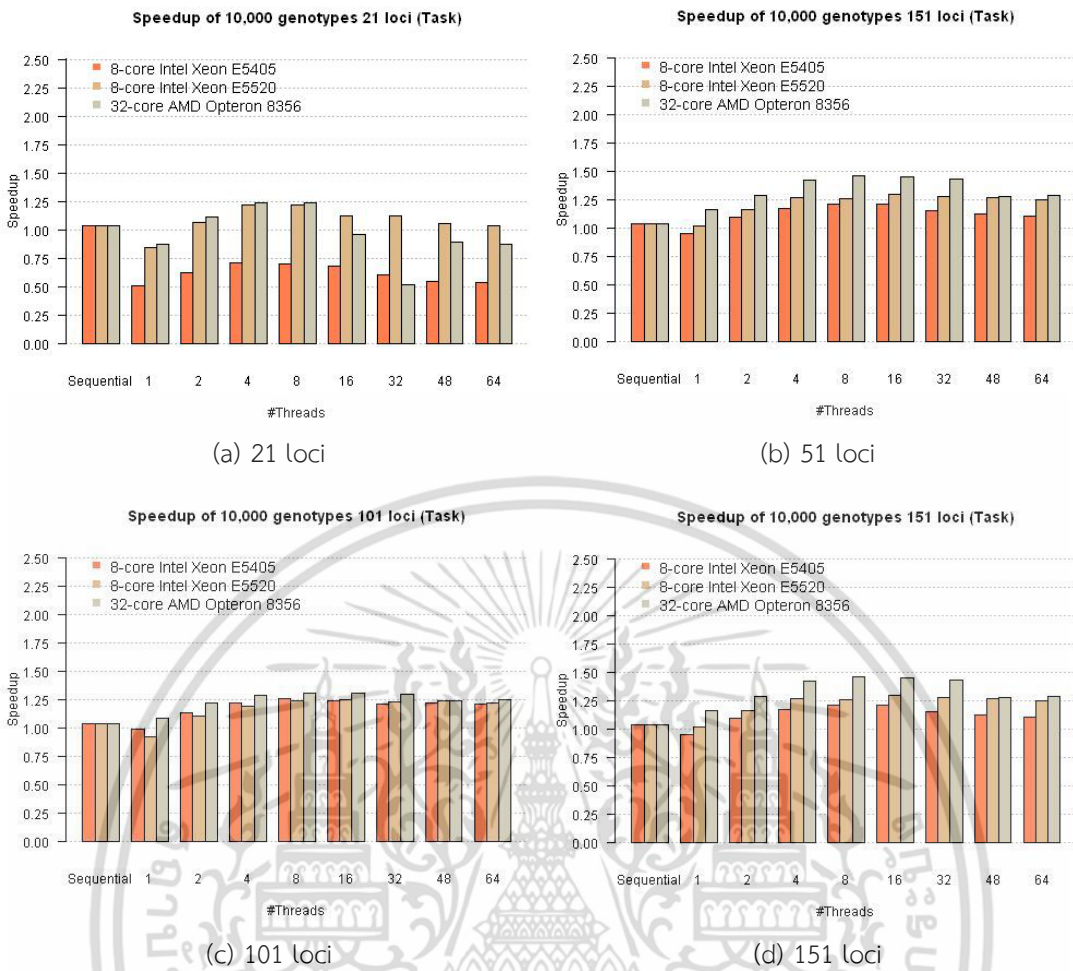
reasons behind this effect are short execution time and overhead cost of For construct. The maximum Speedup is 2.19x at 8 threads and larger number of loci and larger data set and decrease slightly and almost constant later on.

Here are the observations. AMD Opteron 8356 can run OMP SNPHAP and achieve higher Speedup than Intel Xeon E5520 and Intel Xeon E5405 with the same parameter set. Note that Speedup of each CPU is compared with its own SEQ SNPHAP. The higher number of cores, the higher Speedup (32 vs. 8 vs. 8, respectively). Moreover, Intel Xeon E5520 can run OMP SNPHAP faster than Intel Xeon E5405 due to higher clock frequencies (2.66 GHz vs. 2.0 GHz), larger and deeper cache per 4 cores (8-MB L3 vs. 2x3-MB L2), much higher memory bandwidth, better memory technology (DDR3 vs. DDR2) and Hyper-Threading technology (2 threads/core vs. 1 thread/core). In previous work [82], it is reported that each *For* construct can be parallelized and Speedup in accordance with the number of threads forked by OpenMP. Furthermore, the %overhead of OpenMP can significantly affect the total execution time especially at small data sizes and large number of threads (%overhead of 1 thread/core is much lower than that of 2 threads/core).

3.2.3.2 Task Construct

Only *Task* construct is applied to improve the performance. Figure 3.9 shows Speedup of OMP SNPHAP gained on the *Task* construct implementation using OpenMP 3.0, as measured against the sequential performance data given in Table 3.3.

It can be observed that the Speedup of AMD Opteron 8356 tends to grow up along with the number of threads. The maximum Speedup of 1.58x can be achieved at 51 loci, larger data set (10,000 genotypes), and particularly at 8 threads. In addition, the Speedup tends to increase to the maximum values at 8 threads and decrease slightly later on. For Intel Xeon E5520, the Speedup tends to grow up along with the number of threads increase as same as the AMD Opteron 8356. It can be noticed that the Speedup is maximum of 1.41x at 8 threads at 51 loci and larger data set. Moreover, the Speedup tends to peak at 8 threads and decreases slightly later on. While Intel Xeon E5405, the Speedup tends to grow up along with the number of threads on both machines except in (Figure 3.9 (a) 21 loci); that is, as the Speedup of all threads below 0.75. The reason behind this effect is short execution time and $Task$ construct overhead cost. Especially, when the number of threads increases over the number of CPUs cores, its overhead grows up so rapidly. The maximum Speedup is 1.29x at



(a) 21 loci (b) 51 loci

(c) 101 loci (d) 151 loci

Figure 3.9 Speedup Comparison of OMP SNPHAP for $Task$ construct on Intel Xeon E5405, Intel Xeon E5520 and AMD Opteron 8356 at of 10,000 genotypes (a) 21 loci (b) 51 loci (c) 101 loci and (d) 151 loci (OpenMP only, No other compiler options).

8 threads at 51 loci and larger data set. It can achieve the maximum values at 8 threads and decrease slightly later on.

It can also be observed that the Speedups of both AMD Opteron and Xeon E5520 reduce significantly, when $T = p$ (So that the best throughput should be obtained when $T = p$). However, the Speedup tends to grow up when problem size (G and n_c) increase. The reasons behind this effect are problem size and more recursion levels. The small problem sizes lead to Speedup loss due to data dependency. For larger problem sizes, data dependency reduces significantly. Unfortunately, the main source of Speedup loss is the memory overhead, which increases both with the number of cores and with the problem size. For more recursion levels, when many threads are defined available for use, more

recursion levels lead to memory overhead and data dependency. Moreover, it does not benefit much from the second memory controller as previously discussed.

Here are the observations, the $Task$ version scales better on the larger data set at 51 loci. Especially, as the Speedups of Intel Xeon E5520 and Intel E5405 tend to slowdown when the number of loci increases. The reason behind this effect is more haplotype instances (depending on the number of loci, n_c). While AMD Opteron 8356 still scales on 101 loci and 151 loci better than Intel Xeon E5520 and Intel E 5405 because higher number of CPUs cores (32 vs. 8 vs. 8, respectively). Moreover, Intel Xeon E5520 outperforms Intel Xeon E5405 due to higher clock frequencies (2.66 GHz vs. 2.0 GHz), larger and deeper cache per 4 cores (8-MB L3 vs. 2x3-MB L2), much higher memory bandwidth, and better memory technology (DDR3 vs. DDR2). However, both of them achieve less maximum Speedup than AMD Opteron 8356, due to the fact that both have only 8 physical cores.

3.2.3.3 Combined Parallelization

Two OpenMP construct, *For* construct and $Task$ construct, are combined to improve the performance. From Figure 3.10, the Speedup of all CPUs and all graphs tend to grow dramatically when the number of threads increase (peak at 8-16 threads) and decrease slightly later on. The Speedup scales well as the number of threads and the physical cores are available. Moreover, the Speedup grow up along with problem size (G and n_c). Especially, as a larger number of loci (151 loci) and a larger data set (10,000 genotypes) all system show the highest Speedup.

It can be observed that the maximum Speedup of Intel Xeon E5405 is 3.16x at 8 threads and slightly lower and almost constant later on. For Intel Xeon E5520, the maximum Speedup is 4.10x at 32 threads. Moreover, the Speedup tends to peak at 8 threads and slow down at 16 threads. It turns out at 32 threads and decreases slightly almost later on. This effect is due to the For construct. On the other hand, the Speedup of AMD opteron 8356 tends to peak at 4.88x at 16 threads and slow down at 32 threads. It turns out at 48 threads and decreases slightly later on. This effect is due to $Task$ construct. Due to this experiments combine For construct and $Task$ construct to improve the performance. So, two factors leading to poor performance are OpenMP overhead costs and load imbalance. Also, modification to the source code to support OpenMP. However, as the larger data set due to larger number of loci, the OMP SNPHAP version scales effectively. Even when only a

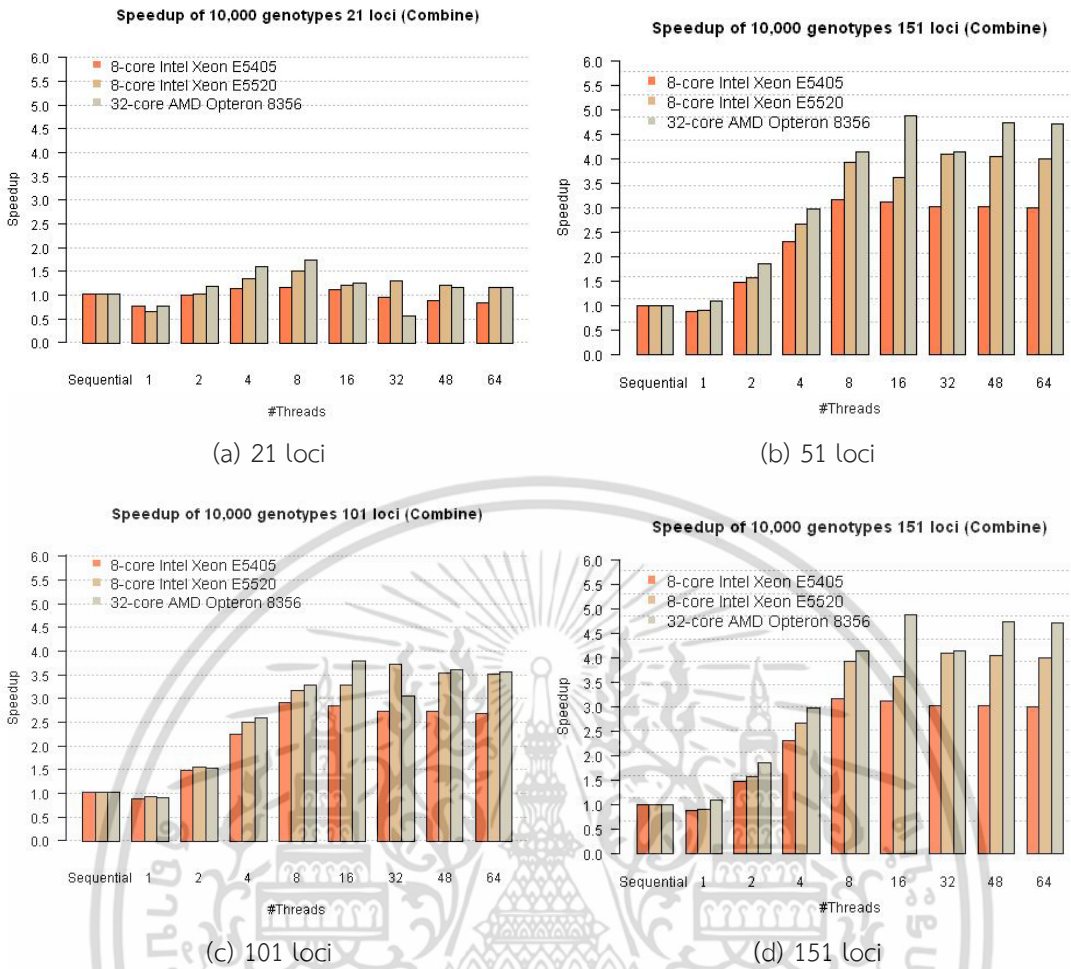


Figure 3.10 Speedup Comparison of OMP SNPHAP for Combined Parallelization (For + $Task$) on Intel Xeon E5405, Intel Xeon E5520 and AMD Opteron 8356 at of 10,000 genotypes (a) 21 loci (b) 51 loci (c) 101 loci and (d) 151 loci (OpenMP only, No other compiler options).

single thread is used, there are some overhead costs incurred by OpenMP and modification of the source code to support OpenMP.

Here are the observations, AMD Opteron 8356 can run OMP SNPHAP faster than Intel Xeon E5520 and Intel Xeon E5405 due to higher number of CPU cores (32 vs. 8 vs. 8, respectively). Moreover, Intel Xeon E5520 can run OMP SNPHAP faster than Intel Xeon E5405 due to higher clock frequencies (2.66 GHz vs. 2.0 GHz), larger and deeper cache per 4 cores (8-MB L3 vs. 2x3-MB L2), much higher memory bandwidth, and better memory technology (DDR3 vs. DDR2). Although the Intel Xeon E5520 shows its architectural features better than AMD Opteron 8356 except for the number of CPU cores, the highest Speedup is lower than that of AMD Opteron 8356. So, more physical CPU cores of AMD Opteron 8356 can achieve

to the highest performance.

3.3 Theoretical Analysis

Even by exploiting the actual/specific parallelizable parts, the suitable OpenMP constructs, the number of physical cores, the sizes of cache memories within/among CPU cores, the clock frequency, the memory technology, the OMP SNPHAP can significantly run faster than SEQ SNPHAP. Thie section will theoretically analyze the performance of OMP SNPHAP.

3.3.1 The Parallelizable Code Fraction

In parallel computing, Amdahl's law [83] is used to predict the theoretical speedup using multiple processors. For the parallelization on multicore systems, the speedup $S(p)$ that can be achieved by using p processor cores is given by Eq. (3.2).

$$S(p) = \frac{1}{(1 - f_p) + \frac{f_p}{p} + o(p)}, \tag{3.2}$$

where f_p is the fraction of a program that can be parallelized and *o(p)* = overhead of p processor cores. This overhead consists of two portions: operating system overhead and threading activities including synchronization, load imbalance, memory contention, cache misses, and also OpenMP overhead (these overhead can be explored by OpenMP profiling step which have introduced in [82]). It is either a strong scaling when the data size remains constant or weak scaling when the data size is proportional to *p* (runtime remains constant).

In practice, if p is fixed and then the maximum speedup can be limited by the fraction $(1 - f_p)$ and the overhead $o(p)$. In this context, the numerical value of fraction f_p determines the theoretical maximum speedup. As referred to the results of GProf profiling [76] in Figure 3.2, the fraction f_p is the *cmp_hap()* which called by *hap_prior()* and *hap_posterior()* as 89.2% of the total execution time. Therefore, the ideal speedup *S(p)* is 4.56 at $p = T = 8$. Unfortunately, the experimental speedup of three systems at G = 10,000 and n_c = 151 (as Figure 3.10) are less than 4.56. These effects occur due to the *o(p)* and some source code modification to preserve the correct output. Moreover, the speedups scale along with the number of data size $(G + n_c)$ increases. This observation could be described by Eq. (3.2) that when f_p grows up, the serial (sequential) part, $(1 - f_p)$, decreases in opposite to f_p. This leads to higher speedup. If p is increased proportionally, nevertheless, the speedup would

be weak scaling and could be further limited by memory contention, cache misses, and communication bottleneck between cores. There are two ways for the solution including more cache capacity and improving memory contention for shared memory resources. The number of cache misses could be reduced with bigger cache capacity and deeper cache per socket (chip). For reducing memory contention, shared data among cores should avoid as many as possible. The best one is choosing the architecture that have not shared cache among cores and/or have to modify the program to avoid shared data.

Next, to demonstrate the complexity analysis of both parts depend on the number of CPU cores, the number of threads, and its overhead.

3.3.2 Applying parallel For construct

The theoretical analysis of *hap_prior()* and *hap_posterior()* shall be investigated and accounted for f_p portion of the total execution time. The OMP SNPHAP was achieved by parallelizing two functions by using *parallel For* construct [82]. In theoretical analysis of the multicore CPUs, T threads are created such that each thread executes a chunk of *cmp_hap()*. In other words, the OpenMP *parallel For* construct splits and assigns the chunk of *cmp_hap()* to each thread.

As referred to the haplotype inference (HI) definition [14, 41], HI is the problem of inferring $2G$ haplotype pairs from G observed genotype data . For strictly biallelic genotype, the number of possible haplotype pairs per genotype is 2^{k_j-1}, where k_j is the number of heterozygous loci site (representing by 2) in an individual genotype data. So, the maximum number of possible haplotypes n_h is calculated by Eq. (3.3).

$$n_h = 2\sum_{j=1}^{G} 2^{k_j-1}, \tag{3.3}$$

where k_j is the number of heterozygous loci site at j_{th} genotype.

The *cmp_hap()* has been called by $O(n_h)$ times to compare two haplotypes at the number of currently phased n_ϕ, in which the complexity of each call is $O(n_\phi)$, Therefore, the computational complexity of the *hap_prior()* for running one EM iteration as shown in Eq. (3.4).

$$T_{prior}(n_\phi, n_h) = O(n_\phi \times n_h) \tag{3.4}$$

Moreover, the computational complexity of the *hap_posterior()* for running one EM iteration is defined in Eq. (3.5).

$$T_{posterior}(n_\phi, n_h) = O(n_\phi \times n_h) \tag{3.5}$$

As referred to the SEQ SNPHAP algorithm in Algorithm 4 of Appendix A, the EM algorithm will repeat *hap_prior()* and *hap_posteior()* until the EM iteration reach to the maximum l_{max} or frequencies are stable. At outer loop inside *main()*, the loop of n_ϕ will repeat until the n_ϕ reach to the maximum of the number of loci (n_c) for each genotypes. The possible haplotype instance of the j_{th} genotype (c_j) can be explained with 2^{k_j}, where k_j is the number of heterozygous loci site (representing 2). In this way the total computational complexity of the *hap_prior()* + *hap_posteior()* as shown in Eq. (3.6).

$$T_{total}(n_c, G, l_{max}) = \sum_{n_\phi=1}^{n_c} \sum_{j=1}^{G} \sum_{l=1}^{l_{max}} ((3n_\phi \times 2^{k_j-1}) + 2^{k_j+1}) \tag{3.6}$$

Let *k* be the average of the number of heterozygous loci site so that the new equation can be evaluated as Eq. (3.7).

$$T_{total}(n_c, G, l_{max}, k) = O(n_c \times G \times l_{max} \times 2^k) \tag{3.7}$$

Since multicore CPUs can execute a number of threads T in parallel, the actual computational complexity as shown in Eq. (3.8).

$$T_{total}(n_c, G, l_{max}, T) = \frac{n_c \times G \times l_{max} \times ((3n_c \times 2^{k-1}) + 2^{k+1})}{T} + o(T) = O(n_c \times G \times l_{max} \times 2^k), \tag{3.8}$$

where T is the number of threads, p is the number of processor cores, $o(T)$ are threading multiplexing, synchronization, load imbalance and also OpenMP overhead, and $T > p$. In the best case, if $T = p$ then $o(T)$ can be eliminated. The n_h depends on the number of heterozygous site (representing 2) in each genotype which normally contains $\approx 3/4$ of n_c (evaluating from SNAP generating data set tool). As the n_c increases, the n_h increases which leads to memory consumption and cache miss problem that cannot be explored by Amdhal's law. This problem causes the speedup limitation.

For performance evaluation, let *p* and *T* be fixed and the l_{max} can be eliminated due

to it is a small constant ($l_{max} = 100$) when compared with others, the complexity depends on n_c, G, $o(T)$, and some overheads that cannot be explored by Amdhal's law.

To improve the performance, the number of processor cores should be increased. Some effects that cannot be explored by Amdhal's law including cache miss, communication between cores, less memory bandwidth, and memory contention problem. The complexity analysis of qsort() part will be investigated in next chapter.

3.3.3 Applying Task construct

$Task$ construct is applied to parallelize the recursive algorithm (*qsort()*) so that all threads can execute tasks without barrier synchronization, leading to improve load balancing. QuickSort, moreover, can operate on the same memory, the processor cores do not have to share data except synchronization among cores. Note that the *qsort()* is located inside the loop of n_ϕ in SEQ SNPHP. A new *qsort1()* is developed with *Task* construct in a parallel/ single paradigm with only one thread creating tasks and then inserting tasks into ready queue (using *#pragma omp single*) and other threads will pickup tasks from the queue for sorting a small haplotype instances list. This technique has introduced in [84].

In theoretical analysis of the multicore CPUs, T threads are created for tasks (using *# pragma omp parallel*). It has been acknowledge that QuickSort complexity is *n log n* (on the average). Therefore, the computational complexity of the *qsort1()* at the number of currently phased n_ϕ is $O(n_h\ log\ n_h)$, where n_h is evaluated from Eq. (3.3). The total computational complexity of the *qsort1()* is shown in Eq. (3.9) when n_ϕ equals n_c and *k* is the average of number of heterozygous loci.

$$T_{qsort}(n_c, G, k) = n_c \times ((G \times 2^k)\ log\ (G \times 2^k)) = O(n_c \times ((G \times 2^k)\ log\ (G \times 2^k))), \quad (3.9)$$

Since multicore CPUs can execute *T* threads in parallel, the actual computational complexity as shown in Eq. (3.10)

$$T_{qsort}(n_c, G, k, T) = \frac{n_c \times ((G \times 2^k)\ log\ (G \times 2^k))}{T} + o(T) = O(n_c \times ((G \times 2^k)\ log\ (G \times 2^k))), \quad (3.10)$$

where T is the number of thread, p is the number of processor core, $o(T)$ are threading

multiplexing, synchronization and also OpenMP overhead, and $T > p$. In the best case, if $T = p$ then $o(T)$ can be eliminated. Note that the n_h depends on the number of heterozygous site of n_c in each genotype so that the n_c proportional to the n_h.

Let *p* and *T* be fixed, the complexity of *qsort1()* depends on n_c, *G*, *o(T)*, and some overheads that cannot be explored by Amdhal's law including cache miss, communication between cores, and memory contention problem. In recursive algorithm, a larger number of haplotypes lead to more recursive levels that can affect multiple tasks or threads on long execution time and also more memory consumption. On the other hand, the scheduler can easily distribute the work among all threads when more tasks are available. To achieve even higher performance, the *qsort1()* should be optimized by determining optimal tasks that the scheduler can spawn besides more processor cores, more cache capacity, and also its overhead should be reduced.

3.3.4 Cache memory and data size

In this section, anomaly relating cache memory and data size (G + n_c) would be investigated in term of the performance analysis.

The number of possible haplotypes (n_h) depends on the data size (G + n_c). A large number of haplotypes cannot fit in cache memory on both the same socket and among sockets. From system architecture summary in Table 3.2, each core has a 32KB L1 data cache totalling 4 x 32KB L1 data cache per socket (chip). From the experiments, the maximum number of haplotypes at currently running phase (n_ϕ) are ≈ 1 million haplotypes so that 18 MBytes of memory space must be allocated. Note that one haplotype data structure uses 18 Bytes.

For working data size within cache memory, only some haplotypes can fit into the L1 data cache on the same socket (chip) while remaining haplotypes must be transferred from lower-level caches (L2 and L3). Unfortunately, these haplotypes might be moved to the neighborhood socket via communication links. If many more processor cores are used in the experiment, high shared-data management cost between cores and sockets (chips) will occur. These can lead to additional memory bandwidth contention and cache coherence problems. The factors of different result for Intel Core i7-2600 are larger/bigger L3 cache, higher bandwidth (three data accesses per cycle) and the new 32-bit ring-based interconnects

between the processor cores.

There are a few solutions. The first one is using larger cache capacity and deeper level cache per socket (chip) to reduce sharing data among sockets. The second is increasing high speed communication links between cores and sockets to increase the bandwidth and reduce the bottlenecks. The third is avoiding shared data between cores and sockets as possible. These solutions could help achieve even higher performance.

CHAPTER 4

PARALLEL PARTITION AND MERGE QUICKSORT (PPMQSORT)

This section is divided into three subsections; The first one is PPMQSort overview. The second and third are Complexity Analysis and Performance Evaluation and Discussion, respectively.

4.1 PPMQSort overview

Previous parallel QuickSort algorithms focus on optimizing either partitioning phase or recursive QuickSort phase. This thesis is focused on both phases. In this section, the Parallel Partition and Merge QuickSort (PPMQSort) is proposed on any multicore CPUs. The concept of PPMQSort is to partition an unsorted array into partially sorted partitions in the Parallel Partition Step. Then, these partitions can be eventually sorted independently using OpenMP $Task$ construct in the Parallel $qsort()$ Step. Those important steps mentioned above can be illustrated in Fig. 4.1.

The PPMQSort is actually developed in C language on top of an open-source/past version of Stdlib $qsort()$ utilizing stack rather than recursion. Due to limited space and ease of understanding, the algorithms are explained in recursion. Notations used in this chapter are listed in Table 4.1. This section first presents the *Parallel Partition Step* consisting of 2 phases: *Partition Phase with 2 threads* and *Merge Phase with 1 or 2 Threads*. Then it shows how to apply OpenMP $Task$ parallelism to call $qsort()$. At last, the time complexity of PPMQSort is analyzed.

4.1.1 Parallel Partition Step

The partitioning operation has been a major bottleneck of QuickSort since it was invented. Previous work has tried to optimize it by both reducing the number of key comparisons and fast swapping code. The key idea of the Partition Phase with 2 Threads is to divide the input data array into two subarrays. Then, they can be partitioned in parallel with 2 threads into 4 sub-subarrays using the same pivot value. Next is the Merge Phase with 1 or 2 Threads swapping the second and third sub-subarrays. Both phases of Parallel Partition

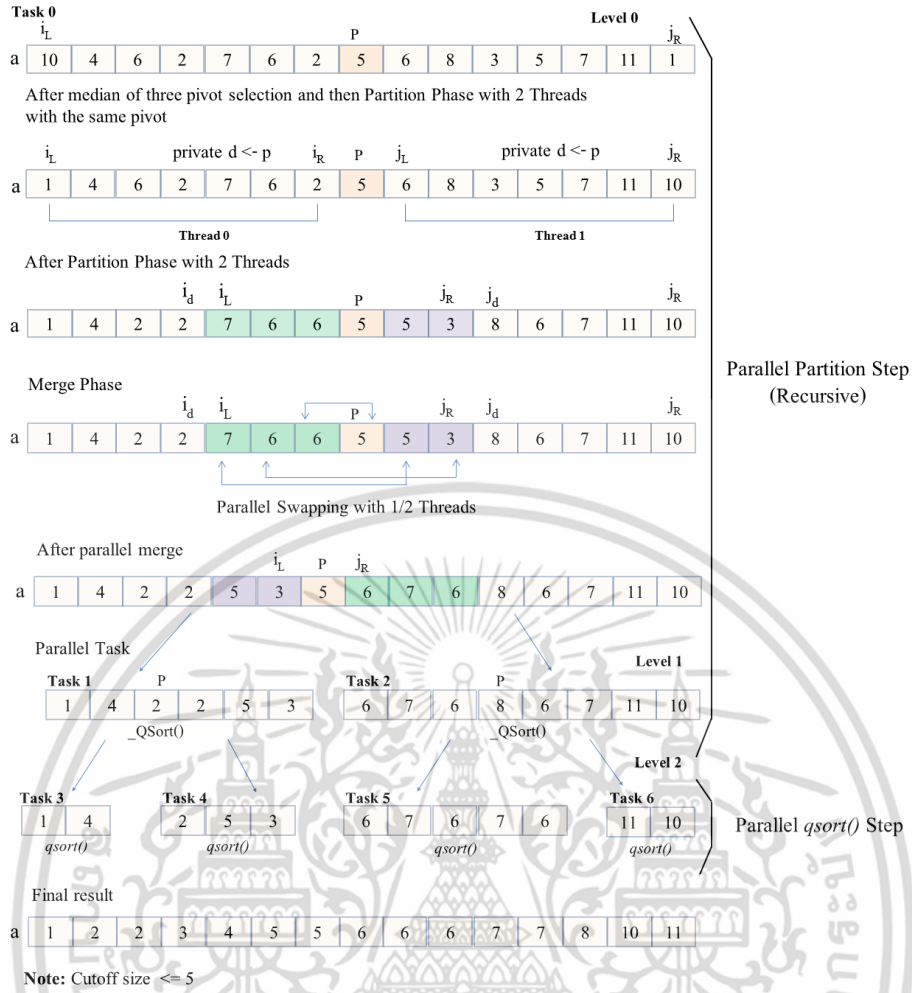


Figure 4.1 Illustration of Parallel Partition and Merge QuickSort (PPMQSort) consisting of Parallel Partition Step and Parallel $qsort()$ Step

Step are explained in details as follows.

4.1.1.1 Partition Phase with 2 Threads

Initially, an unsorted data array, $a = a_0, a_1, ..., a_{n-1}$, is divided into two independent subarrays at the pivot p. Let a_p denotes the pivot element selected by $MedianOfThree()$ function. Let i_L and i_R be left indices and j_L and j_R be right indices of a, respectively. The left subarray of a, $a_0, .., a_{p-1}$ corresponds to $(i_L = 0, i_R = p - 1)$. Similarly, the right subarray, $a_{p+1}, .., a_{n-1}$, corresponds to $(j_L = p + 1, j_R = n - 1)$. In this phase, both subarrays, (a_{i_L}, a_{i_R}) and (a_{j_L}, a_{j_R}) are compared and swapped with the same pivot a_p simultaneously using 2 threads on line 13 and line 15 in $seq\_partition()$ of Algorithm 2. In addition, $seq\_partition()$ returns the partition index as i_d and j_d for the left and right partitions, respectively as shown. As a result, $a_0, .., a_{n-1}$ are splitted into 4 sub-subarrays;

two sub-subarrys on the left, $a_0, .., a_{i_d}$ and $a_{i_d+1}, .., a_{p-1}$, and two sub-subarrys on the right, $a_{p+1}, .., a_{j_d-1}$ and $a_{j_d}, .., a_{n-1}$. Notice that i_d and j_d are the middle indices of the left and right subarrays, respectively.

From a programming perspective, OpenMP Parallel Tasks have been applied without barrier synchronization, leading to improved CPU utilization. To reduce the number of shared memory accesses, $d = p$ is copied to be a local private variable to improve cache locality. Both Phases are listed in Algorithm 2.

In summary, based on i_d, p, and j_d, two independent subarrays can be partitioned into 4 sub-subarrays in parallel with respect to the global a_p pivot in this phase. These 4 sub-subarrays are ordered as follows: less than or equal (LEQ), greater than (GT), LEQ, and GT from left to right.

4.1.1.2 Merge Phase with 1 or 2 Threads

In this subsection, the second (GT) sub-subarray, $a_{i_d+1}, .., a_{p-1}$, and the third (LEQ) sub-subarray, $a_{p+1}, .., a_{j_d-1}$, are swapped and merged together. The idea of this phase is to swap all data in both sub-subarrays in order to rearrange them in the correct order, LEQ and GT. Because this phase needs only to swap a bulk of data between them, no comparisons are necessary. Furthermore, swapping would work with data on the same sub-subarrays so that the method does not use an extra memory.

The Merge Phase with 1 or 2 Threads is shown as function $Merge()$ on line 21 of Algorithm 2 where $len()$ returns the number of elements between two arguments ($len(x, y) = y - x + 1; y >= x$). Let $i_L = i_d + 1$ and $j_R = j_d - 1$ be the left most index and the right most index of the sub-subarray. So, the second (GT) sub-subarray consists of $a_{i_L}, .., a_{p-1}$ and the third (LEQ) sub-subarray consists of $a_{p+1}, .., a_{j_R}$. Both arrays must be swapped to complete the Parallel Partition Step. The swapping will start from this pair (a_{i_L+i}, a_{temp+i}) and incrementally continue for $i = 0$ to $length - 1$ on line 39 of Algorithm 2. After swapping is finished, a_p must be adjusted to the correct position. Both phases of Parallel Partition Step are recursive as shown in function $\_QSort()$ until each partition's size is no greater than Cutoff u on line 1 of Algorithm 3. Although it is associated with OpenMP $Single$ construct on line 15, parallelism can be achieved up to $2h$ threads in reality. That's because the Parallel Partition Step each forks 2 threads internally.

Three important steps need to be considered in this phase. Firstly, the total number of elements swapped between two sub-subarrays is calculated. This number can be determined from the shorter length of either i_d and pivot place p or j_d and p. The variable $length$ can be $\leq \frac{n}{4}$. Then, the direction of swapping sequence is determined. In order to be cache friendly, increasing order is chosen. The last step is to move the pivot to the appropriate position in the array after swapping process is finished to guarantee that the Parallel Partition Step is completed. In the next Parallel $qsort()$ Step, those partitions can be $qsort()$ in parallel up to h threads.

4.1.2 Parallel qsort() Step

The Parallel Partition Step can be cutoff by u elements to avoid over partitioning so that $qsort()$ can efficiently sort in each core's private L2 cache or shared L3 cache depending on the hardware. The Parallel $qsort()$ Step is on the *else* part of function $\_Qsort()$ on line 9 of Algorithm 3. Therefore, Cutoff u should be parameterized in the experiment to achieve the best Speedup. As a result, if the Stdlib. $qsort()$ performance is improved, the performance of PPMQSort will be automatically enhanced. Next, the time complexity of PPMQSort will be analyzed in $O()$ notation.

4.2 Comparison with QSort

PPMQSort is compared with original QSort version (as referred to Algorithm 1). The summary of comparisons as shown in Table 4.2.

Algorithm 2 The Parallel Partition algorithm

1: **function** ParallelPartition(a,$start$,end) ▷ Parallel Partition Step
2: $i_L, j_R, p \leftarrow Partition(a, start, end)$ ▷ call Partition function
3: $p \leftarrow Merge(a, p, i_L, j_R)$ ▷ call Merge function
4: **return** p
5: **end function**

6: **function** Partition(a, i_L, j_R) ▷ Partition Phase with 2 Threads
7: $p \leftarrow MedianOfThree(a, i_L, j_R)$
8: $d \leftarrow p$
9: $i_R \leftarrow p - 1$
10: $j_L \leftarrow p + 1$
11: **begin OpenMP parallel Tasks private(d)**
12: **OpenMP Task**
13: $i_d \leftarrow seq\_partition(a, d, i_L, i_R)$ ▷ Partition the Left Subarray
14: **OpenMP Task**
15: $j_d \leftarrow seq\_partition(a, d, j_L, j_R)$ ▷ Partition the Right Subarray
16: **end parallel Tasks**
17: $i_L \leftarrow i_d + 1$
18: $j_R \leftarrow j_d - 1$
19: **return** (i_L,j_R, p)
20: **end function**

21: **function** Merge(a, p, i_L, j_R) ▷ Merge Phase with 1 or 2 Threads
22: ▷ Three cases for calculating location and moving the pivot p
23: **if** $len(i_L, p-1) < len(p+1, j_R)$ **then** ▷ Left side is shorter.
24: $length \leftarrow len(i_L, p-1)$
25: **Swap**$(a_p, a_{j_R - length})$
26: $p \leftarrow j_R - length$
27: $temp \leftarrow p + 1$
28: **else if** $len(i_L, p-1) > len(p+1, j_R)$ **then** ▷ Right side is shorter.
29: $temp \leftarrow p + 1$
30: $length \leftarrow len(p+1, j_R)$
31: **Swap**$(a_p, a_{i_L + length})$
32: $p \leftarrow i_L + length$
33: **else** ▷ Left side equals right side.
34: $temp \leftarrow p + 1$
35: $length \leftarrow len(p+1, j_R)$
36: **end if**
37: **begin OpenMP parallel For with 1 or 2 Threads**
38: **for** $i \leftarrow 0, length - 1$ **do** ▷ Swapping with 1 Thread or 2 Threads
39: **Swap**$(a_{i_L + i}, a_{temp + i})$
40: **end for**
41: **end parallel For**
42: **return** p
43: **end function**

Algorithm 3 The PPMQSort Algorithm

```
1:  function _QSort(a, i_L, j_R, u)
2:      if i_L + u < j_R then                          ▷ if j_R − i_L > Cutoff u
3:          p ← ParallelPartition(a, i_L, j_R)         ▷ Parallel Partition Step
4:          OpenMP Task
5:          _QSort(a, i_L, p − 1, u)                   ▷ Left Subarray
6:          OpenMP Task
7:          _QSort(a, p + 1, j_R, u)                   ▷ Right Subarray
8:      else                                           ▷ else less than or equal Cutoff u
9:          qsort(a, i_L, j_R)                         ▷ Parallel qsort() Step
10:     end if
11: end function

12: function PPMQSort(a,start,end,h,u)                 ▷ PPMQSort() Function
13:     begin OpenMP parallel with h threads
14:     OpenMP Single
15:     _QSort(a, start, end, u)                       ▷ with Cutoff u
16:     end parallel
17: end function
```

Table 4.1 PPMQSort Notations

a	Input data array
a_j	Data element at index j
a_{i_d}	Data element at the middle index i_d
a_p	Data element at pivot p
B	Branch Loads
B_m	Branch Load Misses
B_m/s	Branch Load Misses Per Second
C	Cache References
C_m	Cache Misses
C_m/s	Cache Misses Per Second
$\|C_{line}\|$	Cache Line Size
c	Number of Processor cores
d	The middle index
$f()$	A function
HT/NHT	HyperThread/Non-HyperThread
i, j	Loop indices
i_L, i_R	Left most and right most indices of the left hand side subarray, respectively
j_L, j_R	Left most and right most indices of the right hand side subarray, respectively
i_d, j_d	The middle indices of left subarray and right subarray, respectively
k	Number of CPU Sockets
K	10^3 or 2^{10}
l	Recursion level l
M	10^6 or 2^{20}
M_{bw}	Total memory bandwidth
n	Data set size
$O()$	BigO Notation
o	Optimization Level
p	The pivot place
$R_{x,y}$	Correlation Coefficient of x and y
S	Speedup
s	Second
T_{qsort}	Run Time of Sequential Stdlib $qsort()$
$T_{ppmqsort}$	Run Time of PPMQSort
T_{seq}	Run Time of Any Sequential QuickSort
T_{par}	Run Time of Any Parallel QuickSort
U	%CPU Utilization
u	Cutoff size

Table 4.2 Comparison between PPMQSort and Sequential QSort

	PPMQSort	Sequential QSort
Partition	**1)** Choose a pivot using Median of three technique. **2)** Divide the input array into two subarrays along with the selected pivot. **3)** Two subarrays are assigned to 2 threads to partition with the same global pivot in parallel. **4)** Copy global pivot to local cache of each thread for good cache locality.	**1)** Pivot is the first element. **2)** Partition the input array with 1 Thread.
Merge	**1)** Sequential/Parallel merge with 1/2 threads by only Swapping (No comparison). **2)** No extra memory needed.	No
Recursive QSort	**1)** Nested Parallel OpenMP Task. **2)** Recursive call PPMQSort until the obtained partitions are up to Cutoff size. **3)** These unsorted partitions can be locally cached. **4)** Use stdlib $qsort()$ to sort independent unsorted partitions in parallel.	Recursive call Seq. QSort until $start \geq end$.
Stdlib qsort() compatible Partition traversal algo. Time Complexity (average)	Yes Bread first search $O(n + \frac{n}{c} \log \frac{n}{2c})$	No Depth first search $O(n \log n)$

4.3 Complexity Analysis

The time complexity of PPMQSort is analyzed assuming that all c cores are 100% utilized by running $h \geq c$ threads. The analysis can be divided into two steps: Parallel Partition Step and Parallel $qsort()$ Step as follows.

Lemma 4.1. *Let n be the size of data array a, where $a = a_0, a_1, ..., a_{n-1}$. Then, the time complexity of* Parallel Partition Step *with h Threads on c cores where $h \geq c$ is $O(n + \frac{n}{c} \log \frac{n}{2uc})$.*

Proof: At the beginning (level 1), the number of comparisons in Partition Phase with 2 Threads is $2 \times \frac{n}{4}$. Due to $c \geq 2$ cores, the time complexity is $\frac{1}{c} \times 2 \times \frac{n}{4} = \frac{2}{2}(\frac{n}{4})$. The number of swappings in Merge Phase with 1 Thread is $\frac{n}{4}$. Due to its sequential operation, its time complexity is $\frac{n}{4}$. In the first recursion level, the time complexity is hence $\frac{2}{2}(\frac{n}{4}) + \frac{n}{4}$. In the second level, there are *two* independent partitions with $c \geq 2$ processor cores. the time complexity of Partition Phase with 2 Threads is $\frac{1}{c} \times 4 \times \frac{n}{8} = \frac{4}{c}(\frac{n}{8})$. The number of swappings in Merge Phase with 1 Thread is $2 \times \frac{n}{8}$. Due to its parallel operation, its time complexity is now $\frac{1}{c} \times 2 \times \frac{n}{8} = \frac{2}{c}(\frac{n}{8})$. The total time complexity of the second level is $\frac{4}{c}(\frac{n}{8}) + \frac{2}{c}(\frac{n}{8})$. The partitioning process is recursive until the condition on line 2 of Algorithm 3 is FALSE. That means the partition size is not larger than Cutoff u elements. Based on the divide and conquer concept, the number of this recursive partitioning is $\log_2 \frac{n}{u}$ levels on average with respect to Cutoff u.

Therefore, the total time complexity of the Parallel Partition Step is

$$= \frac{2}{2}(\frac{n}{4}) + \frac{n}{4} + \frac{4}{c}(\frac{n}{8}) + \frac{2}{c}(\frac{n}{8}) + \frac{8}{c}(\frac{n}{16}) + \frac{4}{c}(\frac{n}{16}) + ... + \frac{2^{\log_2 \frac{n}{u}}}{c}(\frac{n}{2^{\log_2 \frac{n}{u}+1}}) + \frac{2^{\log_2 \frac{n}{u}-1}}{c}(\frac{n}{2^{\log_2 \frac{n}{u}+1}})$$

$$= 3 \times [\frac{2^0}{2}(\frac{n}{2^2}) + \frac{2^1}{c}(\frac{n}{2^3}) + \frac{2^2}{c}(\frac{n}{2^4}) + ... + \frac{2^{\log_2 \frac{n}{u}-1}}{c}(\frac{n}{2^{\log_2 \frac{n}{u}+1}}]$$

$$= 3 \times [n \sum_{l=1}^{\log_2 c} \frac{1}{2^l}(\frac{2^{l-1}}{2^{l+1}}) + \frac{n}{c} \sum_{l=\log_2 c+1}^{\log_2 \frac{n}{u}} (\frac{2^{l-1}}{2^{l+1}})]$$

$$= \frac{3}{4} \times [n \sum_{l=1}^{\log_2 c} \frac{1}{2^l} + \frac{n}{c} \sum_{l=\log_2 c+1}^{\log_2 \frac{n}{u}} 1]$$

$$= \frac{3}{4} \times [n(1 - \frac{1}{c}) + \frac{n}{c} \log_2 \frac{n/u}{c}].$$

$$= \frac{3}{4} \times [n - \frac{n}{c} + \frac{n}{c} \log_2 \frac{n/u}{c}].$$

$$= \frac{3}{4} \times [n + \frac{n}{c} \log_2 \frac{n/u}{2c}].$$

$$= \frac{3}{4} \times [n + \frac{n}{c} \log_2 \frac{n}{2uc}].$$

As a result, the time complexity of Parallel Partition Step is $O(n + \frac{n}{c} \log \frac{n}{2uc})$. ■

Lemma 4.2. *Let c processor cores perform $qsort()$ each partition of size u elements in parallel. Since there are at least $\frac{n}{u}$ partitions, the time complexity of Parallel $qsort()$ Step is $O(\frac{n}{c} \log u)$.*

Proof: From Parallel Partition Step, at least $\frac{n}{u}$ partitions can be obtained. Each partition of up to u elements is sorted by $qsort()$ in parallel up to $h \geq c$ threads. The time complexity of Parallel $qsort()$ Step is therefore, $\frac{1}{c} \times \frac{n}{u} \times u \log_2 u$

$= \frac{n}{c} \log_2 u$

$= O(\frac{n}{c} \log u)$. ■

Theorem 4.3. (PPMQSort's Theorem) *The total time complexity of sorting n elements with the proposed PPMQSort running in parallel on $c \geq 2$ processor cores with Cutoff u elements and $h \geq c$ threads is $O(n + \frac{n}{c} \log \frac{n}{2c})$.*

Proof: The complexities of Parallel Partition Step (see Lemma 4.1) and of Parallel $qsort()$ Step (see Lemma 4.2) are $O(n + \frac{n}{c} \log \frac{n}{2uc})$ and $O(\frac{n}{c} \log u)$, respectively. The total time complexity is $O(n + \frac{n}{c} \log \frac{n}{2uc} + \frac{n}{c} \log u)$. Therefore, the time complexity of PPMQSort is $O(n + \frac{n}{c} \log \frac{n}{2c})$.

■

The time complexity of PPMQSort is similar to that of psort1 algorithm [30] as listed in Time Complexity row of Table 2.1. PPMQSort requires no extra space for intermediate results. As the data size n and number of cores c grow, PPMQSort can eventually outperform other algorithms due to its simplicity, scalability, and efficiency. The next section will show how PPMQSort is evaluated.

4.4 Performance Evaluation and Discussion

This section presents how various performance metrics are measured. The experiment setups and results are discussed later on.

4.4.1 Performance Measurement

In order to investigate how the multicore architectures impact the performance of the algorithm, various performance metrics are measured and analysed.

1. CPU Time (in Seconds)

   To fairly compare T_{qsort} and $T_{ppmqsort}$ in any experimental configurations, the CPU time is measured without data file loading and other overheads and averaged by 5 times.

2. Speedup S(x)

   This metric indicates that how many times our PPMQSort can be executed faster than the sequential Stdlib $qsort()$. Based on the measured T_{qsort} and $T_{ppmqsort}$, Speedup S can be computed as

   $$S = \frac{T_{qsort}}{T_{ppmqsort}} \tag{4.1}$$

   where x denotes *times*.

3. Efficiency: Speedup/Core

   We would like to propose a new metric to measure the efficiency of any parallel QuickSort called Speedup per Core, S/c. $S/c > 1.00$ corresponds to superlinear Speedups. It can be due to cache locality/friendliness of the algorithm [85, 86]. Similarly, [33] proposed a similar metric, Speedup/Thread instead. Higher thread counts h can lead to more opportunities to achieve more parallelism that will be limited by hardware.

4. %CPU Utilization U

   The metric can be obtained from the contents of $/proc/stat$ file which keeps track of statistics of all HyperThread-enabled/disabled CPU cores. This %CPU Utilization is

based on *user-time* only.

5. Cache Refs/Cache Misses

   Perf [87] is a software tool that relies on a number of hardware/software counters to collect statistics of CPU resource usages with minimal overhead [88]. For this paper, Cache Ref, C, Cache Misses, C_m and other performance events are collected and averaged by 5 times to achieve high accuracy. In addition, a new metric called Cache Miss per Second, C_m/s, can be obtained as shown in Eq. (4.2).

$$C_m/s = \frac{C_m}{T_{ppmqsort}} \tag{4.2}$$

   It can be beneficial to measure the number of cache misses per time unit especially for highly multithreaded programs. Larger C_m/s may result in higher demands for memory bandwidth which will be presented next.

6. Branch Loads/Branch Load Misses

   Other important metrics of Perf are Branch Loads, B, and Branch Load Misses, B_m. They can be used to address the algorithm whose performance is limited by branch prediction, i.e., parallel QuickSort. Perf makes use of the hardware counters to measure the branch prediction unit. Similarly, a new metric called Branch Load Misses per Second, B_m/s, can be obtained as shown in Eq. (4.3).

$$B_m/s = \frac{B_m}{T_{ppmqsort}} \tag{4.3}$$

   B_m/s can be regarded as number of branch mispredictions per time unit. Larger B_m/s and C_m/s may result in lower utilization of the long execution pipelines and frequent memory stalls which may affect %CPU Utilization U eventually.

7. Average Memory Bandwidth M_{bw}

   The complex interactions between multicore architecture and characteristics of a parallel algorithm directly and indirectly impact both branch mispredictions and cache misses. In multithreaded programs, off-chip memory bandwidth is one of the im-

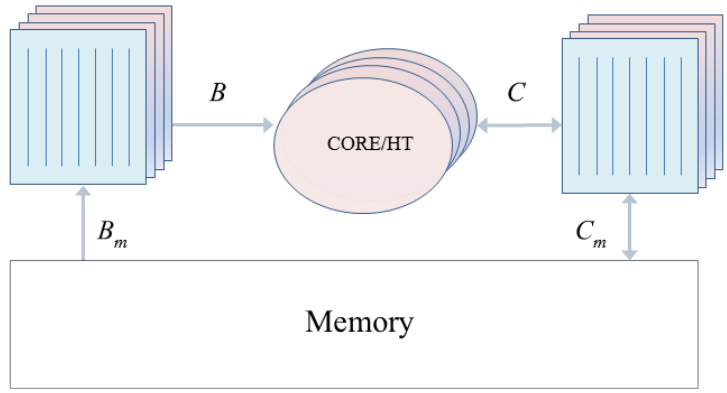


Figure 4.2 Our Shared Memory/Multiprocessor/multicore System Model Measured by Perf (Blue boxes on the left and right hand sides are instruction and data caches, respectively. HT denotes HyperThread.

portant metrics that can be the performance bottleneck due to memory contention, memory saturation and bad allocation among cores.

Many researchers use hardware performance counters to track the amount of consumed memory bandwidth while the multithreaded program is running [89, 90]. The measurement accuracy depends on measurement events, number of counters and the characteristics of memory system including DDR2/DDR3, channels (interleaving), bus clock frequency, etc. However, we cannot directly measure the amount of memory bandwidth consumption. This paper rather proposes a performance model to estimate and evaluate as shown in Fig. 4.2. Our model can utilize a number of available events measured by Perf resulted in Average Memory Bandwidth.

$$M_{bw} = f(B_m/s, C_m/s) \tag{4.4}$$

Assume that B_m/s has negligible effects due to small program size and its recursive nature. The majority of memory bandwidth should be proportional to C_m/s. Therefore, the Average Memory Bandwidth, M_{bw}, can be calculated in terms of cache line size $|C_{line}|$ multiplied by C_m/s as shown in Eq. (4.5).

$$M_{bw} = |C_{line}| \times C_m/s \tag{4.5}$$

4.4.2 Experiment Setup

The results reported in this thesis are based on five multicore CPUs: Intel E5405 Harpertown, Intel E5520 Nehalem-EP, Intel i3-2100 Sandybridge, Intel i7-2600 Sandybridge, and AMD A6-3650 APU. Table 4.3 provides a summary of these multicore systems.

Table 4.3 Architectural details of multicore CPUs in our experiment

System	Xeon E5405	Xeon E5520	A6-3650	i3-2100	i7-2600
Code Name	Harpertown	Nehalem-EP	Llano	Sandybridge	Sandybridge
Clock (GHz)	2.00	2.66	2.6	3.1	3.4
$k \times c$	2x4	2x4	1x4	1x2	1x4
HyperThread	No	Yes	No	Yes	Yes
L1/L2(KB)/core	32/3072	32/256	64/1024	32/256	32/256
L3(KB)/socket	-	8192	-	3072	8192
RAM (GB)	4	12	16	8	16
RAM	DDR2-667	DDR3-800/1066	DDR3-1866	DDR3-1066/1333	DDR3-1066/1333
Others		Smart Cache 8 MB	PCI express 2.0	Smart cache 3MB	Smart cache 8MB
		QPI2 5.86 GT/s	16-way L2	DMI 5GT/s	DMI 5GT/s

In every system listed in Table 4.3, the operating system is 64-bit Ubuntu 14.04 kernel 3.13 LTS. The PPMQSort is compiled with GCC 4.8 and linked with OpenMP 3.0 library under *-fopenmp* option. The measurement tool, Perf version 4.2 is called using *perf stat -r 5 -e* to profile PPMQSort algorithm for 5 times.

Data sets are unsigned 32-bit integer (*Uint32*), unsigned 64-bit (*Uint64*) and 64-bit double precision floating point (*Double*). These are generated using the GCC *random()* function with two distributions: Random and Worst-Case and in different number of elements, n=10M, 20M, 50M, 100M, 200M. The first distribution contains random elements with small number of duplicates. The second distribution is generated such that the sequence seems to be sorted in a descending manner. However, for each distribution, the input sequence once generated is stored as a file. Therefore, both *PPMQSort* and sequential *qsort()* algorithms sort the same input sequences. All parameters are listed in Table 4.4.

Table 4.4 Parameter set of the experiments

Parameters	Values
Data types	Uint32, Uint64, Double
Size n(M)	10, 20, 50, 100, 200
Cases	Random, Worst
Cutoff u(K)	50, 100, 200, 500
Sockets k	1, 2
Cores c	1, 2, 4, 8
Threads h	1, 2, 4, 8, 16, 32
HyperThread	Enable, Disable
Optimization	o2, o3

4.4.3 Results and Discussions

This subsection elaborates various aspects of the PPMQSort algorithm such as best Speedups, tradeoffs between Speedup, Cutoff, and Thread, etc. Finally, the last two subsections are based on statistical analysis of Perf results.

4.4.3.1 The Best Speedups

Table 4.5 - Table 4.6 tabulate the best Speedup, T_{qsort}, and $T_{ppmqsort}$ of all systems based on various data types, cases, and optimizations. The T_{qsort} is obtained with the same experiment configuration as $T_{ppmqsort}$. It can be noticed that the best Speedups of Uint32 are higher than those of Uint64 and Double. Remark that i3-2100, i7-2600 and E5520 systems

are HyperThread enabled. Therefore, their Speedups are higher than the number of physical cores. For a non-HyperThread 8-core Intel Xeon E5405 system, the best Speedup is as high as 7.75x. Due to limited space, best Speedups of Xeon E5404 are omitted. An exceptional case is the 4-core AMD A6-3600 whose Speedups are superlinear at 4.91x and 4.96x in Random and Worst cases, respectively. It can be observed that %CPU Utilizations approach 100% in every Random-case configuration while those of Worst-case are significantly lower.

PPMQSort can achieve high Speedup regardless of the data types and randomness even in the Worst case. It can be obviously noticed that Worst-case T_{qsort} and $T_{ppmqsort}$ are always faster than those of Random-case with the same configuration. Furthermore, their Speedups are almost always higher those of Random case except in 2-socket systems, E5520 and E5405. PPMQSort can exploit the Branch Prediction Unit and caches well, although $seq\_partition()$ must execute a large number of comparisons and swappings on lines 13 and 15 in Algorithm 2. That means the Branch Prediction unit can learn/yield higher prediction rate than the Random-case due to remarkably low B_m/s except those of E5520 cases.

However, the highest memory bandwidth M_{bw} of Worst-case is always greater than Random-case because of its two to three times higher C_m/s. The highest M_{bw} of each system is highlighted in bold face. On the other hand, high B_m/s can be the performance bottlenecks in Random cases as shown in Italic. Despite of 2-3 orders of magnitude lower B_m/s, %CPU Utilization U's of Worst-case are generally lower than those of Random-case in every configuration. It can be due to often memory stalls. This also concurs with Eq.(4.5) that memory bandwidth of PPMQSort depends heavily on C_m/s.

In both Random and Worst cases, Cutoff u should fit the last level cache of each system. It can be noticed that the suitable Cutoff u for Uint32 ranges between 50K-200K elements. For Uint64 and Double cases, Cutoff u ranges between 200K-500K elements or even bigger instead. The best Cutoff u of i3-2100 (Uint32) is 50K by majority vote. It seems like 50K of Uint32 can fit the private L2 cache (256KB) in each core. The rest can almost fit Cutoff in their last level caches except in some cases of u=500K of Uint64 and Double.

Table 4.5 Best Speedup S and other metrics on different data types and corresponding parameters

CPU	Opt.	Uint32				Uint64				Double			
		Random		Worst		Random		Worst		Random		Worst	
		o2	o3	o2	o3	o2	o3	o2	o3	o2	o3	o2	o3
	$n(M)$	200	200	200	200	100	100	200	200	200	200	200	200
	$u(K)$	50	100	50	50	200	500	50	50	200	500	50	50
i3-2100	S(x)	3.19	3.03	**3.79**	**3.67**	2.88	2.79	**3.44**	**3.35**	2.82	2.72	**3.69**	**3.63**
	T_{qsort}(s)	39.80	39.88	14.88	14.89	20.39	20.37	16.83	17.00	45.18	45.21	17.72	17.87
	$T_{ppmqsort}$(s)	12.49	13.14	3.92	4.04	7.07	7.31	4.89	5.06	16.00	16.64	4.79	4.91
	h(threads)	8	8	8	8	8	8	8	8	8	16	8	8
	U(%)	98	98	86	85	96	96	82	80	95	94	84	83
	M_{bw}(MB/s)	150	170	319	300	361	395	**700**	669	331	372	619	608
	B_m/s	1.3e+8	1.4e+8	4.9e+6	4.7e+6	1.5e+8	1.6e+8	3.2e+6	3.1e+6	1.4e+8	1.4e+8	3.2e+6	3.1e+6
	$n(M)$	200	200	200	100	50	20	200	200	20	200	200	50
	$u(K)$	200	200	200	200	200	100	100	100	100	500	100	100
A6-3600	S(x)	4.67	**4.91**	**4.96**	4.74	3.64	3.54	**4.62**	**4.37**	3.72	3.56	**4.60**	**4.43**
	T_{qsort}(s)	59.18	65.03	23.43	11.39	12.53	4.76	26.45	25.32	5.47	60.98	26.66	6.30
	$T_{ppmqsort}$(s)	12.67	13.25	4.71	2.40	3.45	1.35	5.71	5.78	1.47	17.15	5.79	1.42
	h(threads)	4	16	4	4	8	8	8	8	8	8	8	8
	U(%)	96	96	83	80	94	91	76	76	91	92	77	75
	M_{bw}(MB/s)	1.85	2.25	4.39	5.98	9.11	6.53	4.57	3.81	6.85	2.30	5.97	**20.33**
	B_m/s	4.3e+6	6.8e+6	2.4e+4	3.2e+4	1.3e+7	1.7e+7	7.9e+4	5.9e+4	2.1e+7	1.4e+7	6.4e+4	1.6e+5
	$n(M)$	200	200	200	200	100	100	200	200	200	200	200	200
	$u(K)$	100	200	500	500	200	200	500	500	200	500	500	500
i7-2600	S(x)	5.65	5.35	**5.71**	**5.46**	**5.01**	**4.85**	4.66	4.48	4.79	4.61	**5.12**	**4.98**
	T_{qsort}(s)	32.53	32.53	12.36	12.36	16.76	16.78	14.03	14.08	37.05	37.15	14.79	14.85
	$T_{ppmqsort}$(s)	5.76	6.08	2.16	2.26	3.34	3.46	3.00	3.14	7.74	8.06	2.88	2.98
	h(threads)	16	16	8	16	16	16	16	16	16	16	16	16
	U(%)	92	92	78	75	90	90	72	70	87	86	75	73
	M_{bw}(MB/s)	145	148	536	519	266	256	**1,042**	993	249	279	971	958
	B_m/s	1.2e+7	1.5e+7	4.9e+4	1.6e+5	2.2e+7	2.1e+7	9.6e+4	7.7e+4	2.5e+7	4.4e+7	9.5e+4	6.4e+4

Table 4.6 Best Speedup S and other metrics on different data types and corresponding parameters

CPU	Opt.	Uint32				Uint64				Double			
		Random		Worst		Random		Worst		Random		Worst	
		o2	o3	o2	o3	o2	o3	o2	o3	o2	o3	o2	o3
E5405	$n(M)$	50	50	200	200	50	10	100	100	10	10	100	100
	$u(K)$	100	200	500	500	500	200	500	500	200	200	500	500
	S(x)	**7.75**	**7.71**	7.46	7.22	**5.73**	5.42	**5.89**	5.71	4.83	5.20	**6.32**	**6.08**
	T_{qsort}(s)	20.39	20.31	32.27	32.42	22.07	3.90	19.46	19.42	3.27	3.34	20.77	20.53
	$T_{ppmqsort}$(s)	2.63	2.63	4.32	4.49	3.85	0.72	3.30	3.40	0.68	0.64	3.28	3.37
	h(threads)	16	16	16	16	8	8	8	8	8	8	8	8
	U(%)	91	91	77	74	85	78	70	68	79	77	70	67
	M_{bw}(MB/s)	14.74	4.48	57.59	56.21	45.78	60.66	81.74	82.55	62.63	70.20	85.07	**85.26**
	B_m/s	1.39e+7	8.44e+6	2.67e+4	3.28e+5	3.84e+7	7.26e+7	2.11e+4	2.23e+4	4.97e+7	5.57e+7	2.24e+4	1.23e+5
E5520	$n(M)$	200	200	200	100	200	200	50	100	200	200	100	100
	$u(K)$	100	100	500	200	100	200	200	500	200	500	500	500
	S(x)	**12.29**	**11.20**	9.44	8.60	**11.34**	**10.96**	8.16	7.26	**9.43**	**9.06**	8.40	7.90
	T_{qsort}(s)	72.35	70.00	21.05	10.57	80.41	80.53	6.33	12.17	69.40	69.37	12.65	12.70
	$T_{ppmqsort}$(s)	5.89	6.25	2.23	1.23	7.09	7.35	0.78	1.67	7.36	7.66	1.51	1.61
	h(threads)	16	16	16	16	16	16	16	16	16	32	32	16
	U(%)	83	82	67	59	80	80	55	54	73	73	59	56
	M_{bw}(MB/s)	190	172	751	595	306	278	1,537	1,390	303	337	**1,558**	1,278
	B_m/s	7.8e+8	4.0e+8	1.9e+8	1.9e+8	6.6e+8	6.1e+8	2.9e+8	3.8e+8	7.1e+8	1.2e+9	7.2e+8	4.1e+8

4.4.3.2 Speedup S vs. Cutoff u and Thread h

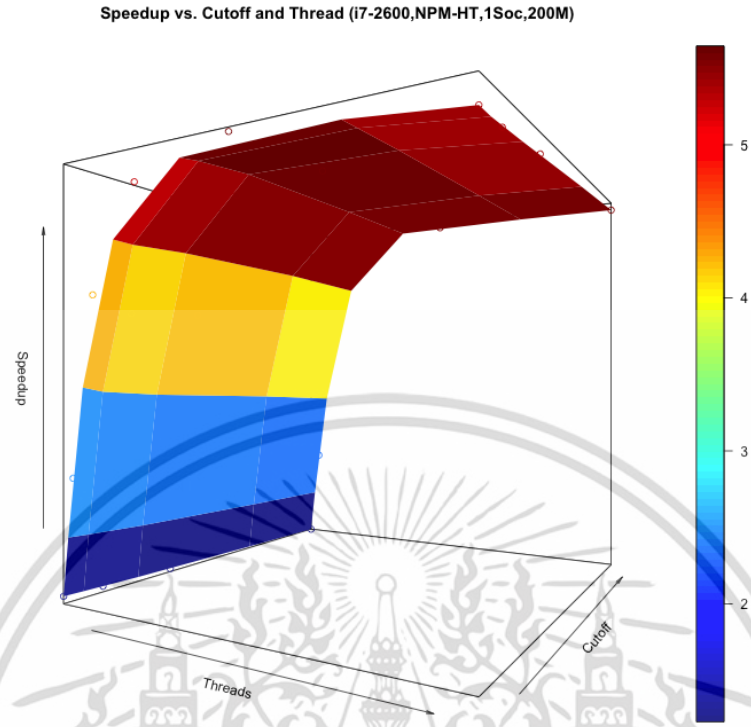


Figure 4.3 Three-D Surface Plot of Speedup, S vs. Cutoff, u and Thread, h of PPMQ-Sort on i7-2600 (Uint32, Random, o2, n=200M)

For a given system and experiment configuration, Speedup S of PPMQSort is a function of Cutoff u and Thread h. As already listed in Table 4.5, the best S of i7-2600 system is 5.65x at n=200M of Uint32, u=100K, and h=16 threads. Figure 4.3 shows a 3-D surface plot of PPMQSort with this configuration. Speedups can be visualized as surface height on the Z axis with colors according to the Color bar on the right hand side. This plot presents the scalability and tradeoffs between Speedups, Cutoffs, and Threads. While increasing thread count h, the Speedup S scales up for all Cutoffs. Therefore, high thread counts enable the PPMQSort to utilize the CPU cores more until S saturates. As discussed earlier in Section 4.4.3.1 Best Speedups, while varying Cutoff u, Speedup changes slightly as darker and lighter colors at the same thread count. This behavior in this 3-D surface plot agrees with the derived time complexity in Theorem 4.3, where u has been canceled out.

4.4.3.3 HyperThread vs. Non-HyperThread CPUs

This subsection will contrast and compare Speedups of PPMQSort on Intel Hyper-Thread and non-HyperThread CPUs with the same experiment configuration. Fig. 4.4 illustrates Speedups (Line) and %CPU Utilization (Bar) of Intel HyperThread and non-HyperThread

of PPMQSort (Uint32, Random case, o2 optimization). The cyan bars and lines are of HT-enabled while the brown ones are HT-disabled. The Speedup differences between HT-enabled and HT-disabled systems are significant due to lower average %CPU Utilization U, despite the fact that other statistics are similar. It can be roughly estimated that HT can boost up the performance by more than 50% which is comparable to [91].

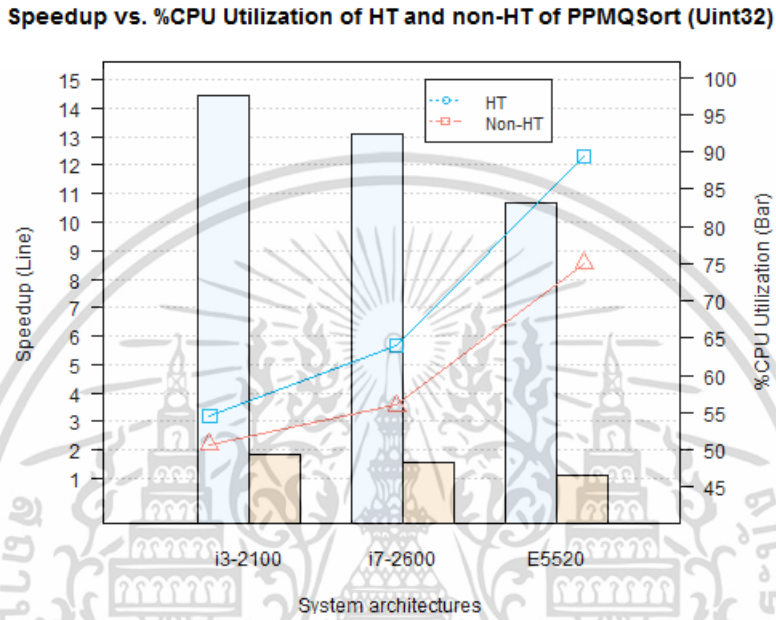


Figure 4.4 Best Speedup, S (Line, Left) vs. %CPU Utilization, U (Bar, Right) of PPMQ-Sort on Intel HyperThread (HT) and non-HyperThread (non-HT) Platforms (Uint32, Random, o2)

4.4.3.4 PPMQSort vs. PPPMQSort

PPPMQSort is a minor variation of PPMQSort where its Merge Phase is parallelized with 2 Threads on line 37 of Algorithm 2. To compare PPMQSort (Cyan) with PPPMQSort (Brown), their Speedups (Line) and Cache Refs (Bar) are plotted on all platforms (Uint32, Random, o2) with the same parameter set. Note that Cache Refs on the left Y axis is in logarithm and scaled by 1 Million. It can be observed in Fig. 4.5 that PPMQSort can achieve better Speedups on the same experiment configurations due to significantly lower C.

Cache Refs are particularly high on AMD A6-3600 compared to other Intel systems. It might be due to fewer general-purpose Integer/Floaing-Point registers thus resulting in more register spills. However, AMD A6-3600 demands M_{bw} up to 20.3 MB/s as listed in Table 4.5 due to both large private L1 data cache (64KB/core) and L2 cache (1MB/core). In addition, its Branch Load Misses/sec B_m/s's are considerably lower than those of Intel

systems. Therefore, its PPMQSort Speedups can be superlinear in some configurations. The rest is comparable on all Intel systems.

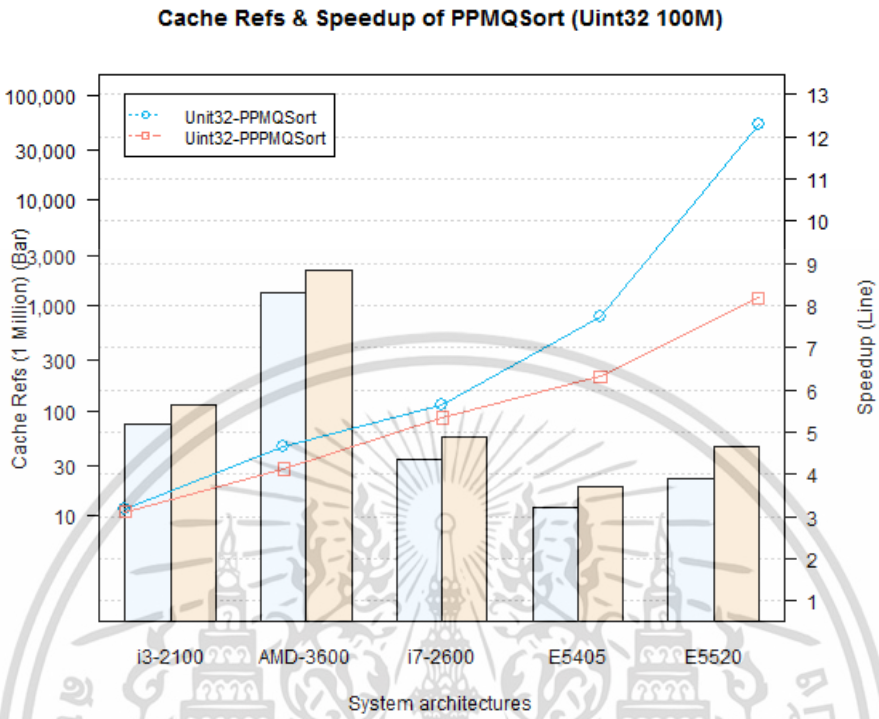


Figure 4.5 Best Speedup (Line, Right) vs. Cache Refs (Bar, Left) of PPMQSort (Cyan) and PPPMQSort (Brown) on all Platforms (Uint32, Random, o2)

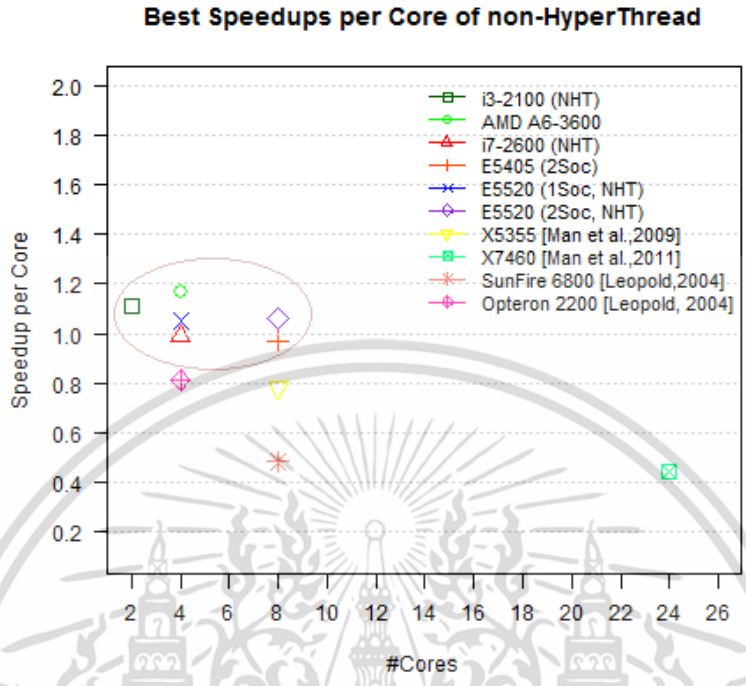
4.4.3.5 Efficiency: Speedup/Core

Figures 4.6(a) and 4.6(b) depict the scatter plot S/c vs. c of non-HT and HT, respectively. It can be observed in Fig. 4.6(a) that PPMQSort can achieve $S/c \simeq 1.00$ or above (inside the oval) while others can only reach up to 0.8. The HyperThread-disabled i3-2100 and non-HT A6-3600 can achieve S/c at 1.11 and 1.17 resulting in superlinear Speedup because of high %CPU Utilization at 98 and 96, respectively. Similarly, Fig. 4.6(b) shows that PPMQSort can achieve $S/c \simeq 1.40$ or above while others can only reach up to 0.33. Some HT systems like i3-2100 and E5520 can achieve S/c at 1.59 and 1.63, respectively because of better %CPU utilization with 3-MB and 8-MB Smart Caches, respectively. It can be concluded that PPMQSort can exploit the CPU cores much better than other algorithms on both non-HT and HT architectures. Moreover, PPMQSort can be scablable on any non-HT/HT/multicore/ multi-socket systems with $S/c \simeq 1.00$/ $S/c \simeq 1.50$ or better.

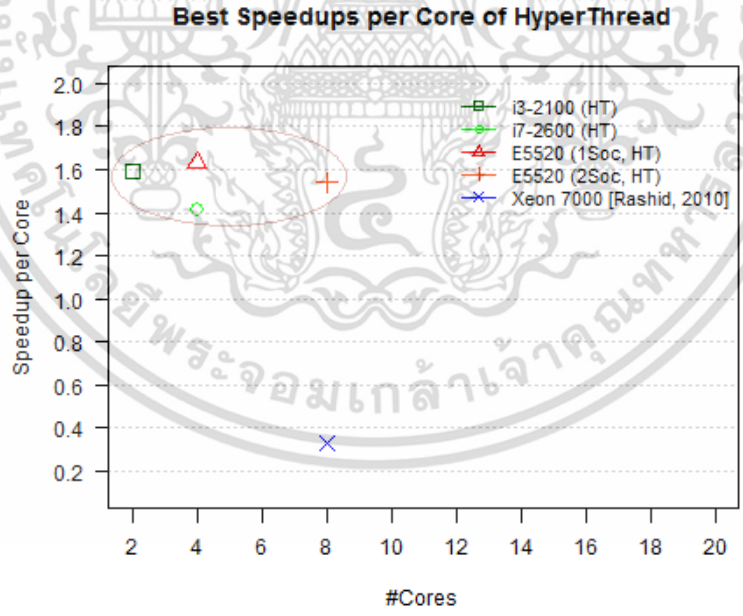
4.4.3.6 Comparison with Previous Implementations

Table 4.7 compares our PPMQSort with previous parallel QuickSort implementations to show that we can achieve the best performance at data size around 100M 32-bit Integers with respect to T_{par} and Efficiency, S/c. [33] reported only the Speedup based on Pthreads Library resulting in higher S/c that may not compare against Stdlib. $qsort()$. In addition, he also did not report the run time. With respect to 11.58x Speedup, our PPMQSort on an 8-core HyperThread E5520 can clearly outperform pqsort1 of [30] on an 8-core Xeon X5355 using the same $qsort()$ benchmark. Although [28] can achieve 25.03x Speedup on a 32-core UltraSPARC, their efficiency is not quite good and the benchmark may not be Stdlib. $qsort()$.

On the Uint64 data, Man *et al.* [31] reported their highest Speedup of 10.47x for 100M random on 24 cores and T_{par} = 2.712 Seconds. With only 8 cores, PPMQSort can achieve S=10.24x at 3.54 Seconds with the same configuration. This can confirm that PPMQSort is more efficient than others.



(a) non-HyperThread



(b) HyperThread

Figure 4.6 Speedups per Core S/c of PPMQSort (inside the oval) vs. Others (Random, Uint32) (a) non-HyperThread (NHT) (b) HyperThread (HT)

Table 4.7 Performance of PPMQSort vs. other parallel QuickSort implementations (Uint32, Random), NA: Not Available

Reference	PPMQSort	[33]	[72]	[30]	[29]	[29]	[28]
Year		2013	2010	2009	2004	2004	2003
$n(10^6)$	100	80	64	100	60	100	100
S(x)	11.58	3.8	2.65	6.22	3.24	3.875	25.03
T_{seq}(s)	34.78	NA	15.9	28.81	24.3	37.2	139.22
T_{par}(s)	3.00	NA	6	4.63	7.5	9.6	5.56
Using qsort()	Yes	No	No	Yes	No	No	No
Architecture	x86	x86	x86	x86	x86	UltraSPARC III	UltraSPARC
GHz	2.66	2.66	NA	2.66	2.2	0.9	0.25
$k \times c$	2x4	1x2	4x2	2x4	1x4	1x8	32x1
HT	Yes	Yes	No	No	Yes	No	No
Cache	L3 8MB	L2 3MB	L2 4x2MB	L2 2x2MB	L2 2x1MB	NA	L2 4MB
Compiler	GCC -o2	G++	Intel C++ 9.1	GCC -o2	Intel C++ 8.1 -o3	Guide	NA
Library	OpenMP 3.0	pthreads	OpenMP 2.5	OpenMP 2.0	OpenMP 2.0	NA	NA
Remarks	Xeon E5520	Intel	Xeon 7000	Xeon X5355	Opteron 2200	Sun Fire 6800	Sun Enterprise
	MAC Pro 2010	Dual-Core	PowerEdge 6800		sort_omp_2.0	sort_pthreads_cv_1.0	10000

4.4.3.7 Statistical Analysis

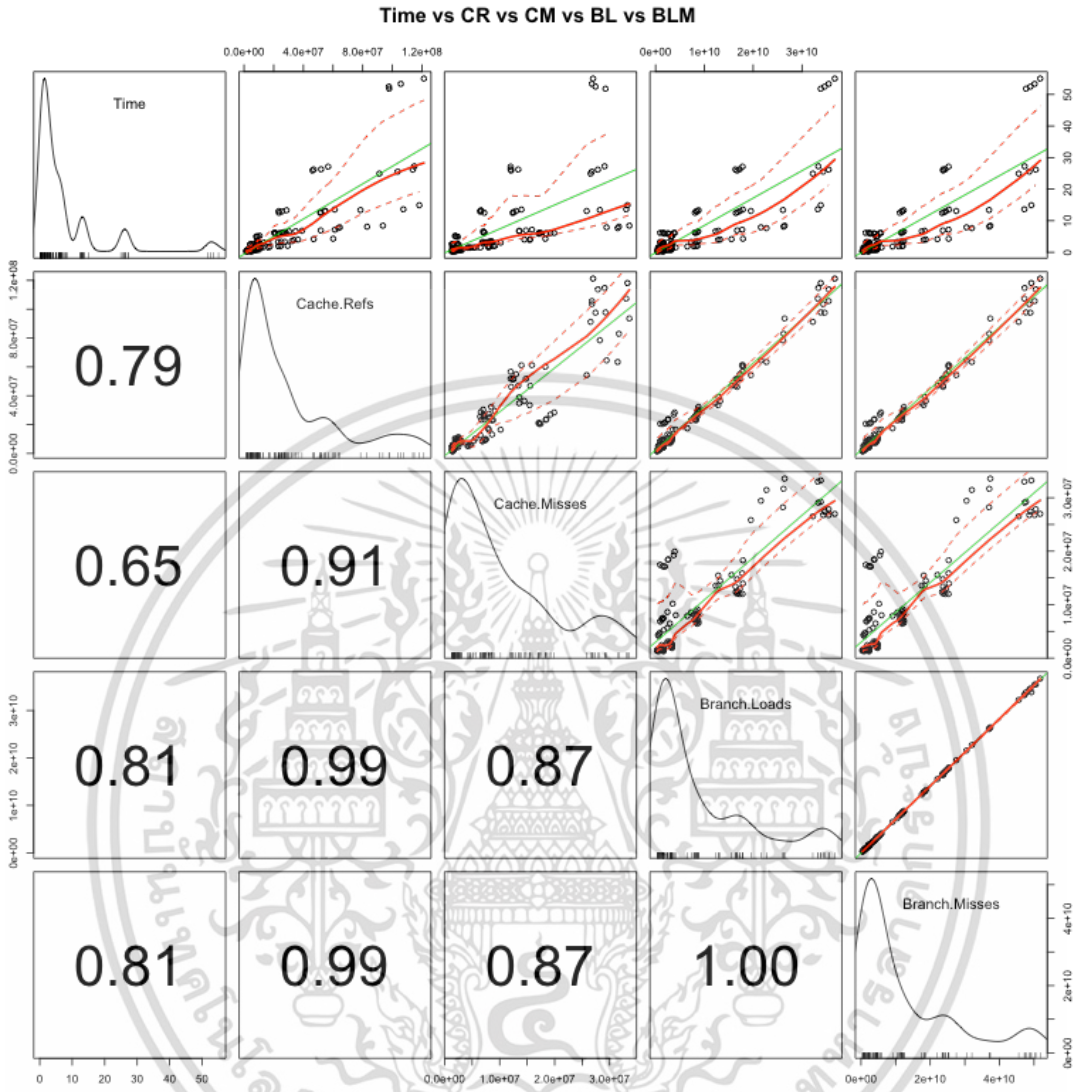


Figure 4.7 Matrix Scatter Plots between Time $T_{ppmqsort}$ vs. Cache Refs C, Cache Misses C_m, Branch Loads B, and Branch Load Misses B_m of PPMQSort on E5520, Uint32, Random, o2, all Cutoffs, Data Sizes, and Threads

Figure 4.7 shows matrix scatter plots between $T_{ppmqsort}$ vs. Cache Refs C, Cache Misses C_m, Branch Loads B, and Branch Load Misses B_m of PPMQSort on E5520, Uint32, o2, all Cutoffs, data sizes and threads. The upper half, the diagonal, and the lower half of the matrix plot illustrate the scatter plots, the density, and the correlation value between/of them, respectively. Each dot in the scatter plot represents an experiment configuration.

The top-row figures show the regression analysis between parameters that Time or $T_{ppmqsort}$ is proportional to Cache Refs C, Cache Misses C_m, Branch Loads B, and Branch

Load Misses B_m, respectively. The green line is a linear regression generated by $lm()$ function in R Project (http://cran.r-project.org). The solid red line is a LOESS (Local Regression Smoothing) mean fit line according to $loess()$ function. The red dotted lines above and below are positive and negative residual squares above and below the LOESS mean fit line, respectively.

C_m is highly correlated with C as indicated by correlation value $R_{C,C_m/s}$=0.91. As expected, they are highly correlated. The higher C, the more C_m, the longer $T_{ppmqsort}$. Similarly, the higher B, the more B_m, the longer $T_{ppmqsort}$. In addition, they are highly correlated with one another. Other systems in the experiment show similar behaviors.

4.4.3.8 Speedup vs. %CPU Utilization vs. Memory Bandwidth

Speedup S vs %CPU Utilization U vs. Cache Misses per Second C_m/s and Branch Load Misses per Second B_m/s of PPMQSort on i7-2600 can be depicted in Fig. 4.8. The configuration of this figure is random n=200M Uint32, o2 and h=4-32 threads. Both C_m/s and B_m/s can be obtained by equations (4.2) and (4.3), respectively.

As plotted, Speedup S is directly proportional to %CPU Utilization U because the correlation coefficient $R_{S,U}$ is 1.00. That means the higher %CPU Utilization, the better Speedup because all the forked threads can effectively execute with fewer memory stalls and pipeline stalls/flushes.

In general, cache misses can be due to cold misses, capacity misses, conflict misses, and coherence misses. Lower C_m/s can be due to better cache locality resulted from suitable Cutoff u and Thread h of PPMQSort as shown in Fig.4.8. On the other hand, lower B_m/s represents infrquent branch mispredictions thus more efficient pipelining. Both frequent cache misses and branch mispredictions per unit time can lead to memory stalls and pipeline stalls and thus lower U. It can be reflected on both $R_{U,C_m/s}$ and $R_{U,B_m/s}$ approaching -1.00. That means U is negatively proportional to C_m/s and B_m/s.

This figure confirms with the basic concept that memory is the bottleneck of the parallel algorithms [92] especially in the Worst-case bounded by C_m/s. However, Random-case Speedups are limited by B_m/s rather than C_m/s. As shown in Table 4.5, Random-case B_m/s's are two to three orders of magnitude higher than those of Worst-case with the same data size n in one-socket systems. For two-socked systems, the gap is not that wide. That

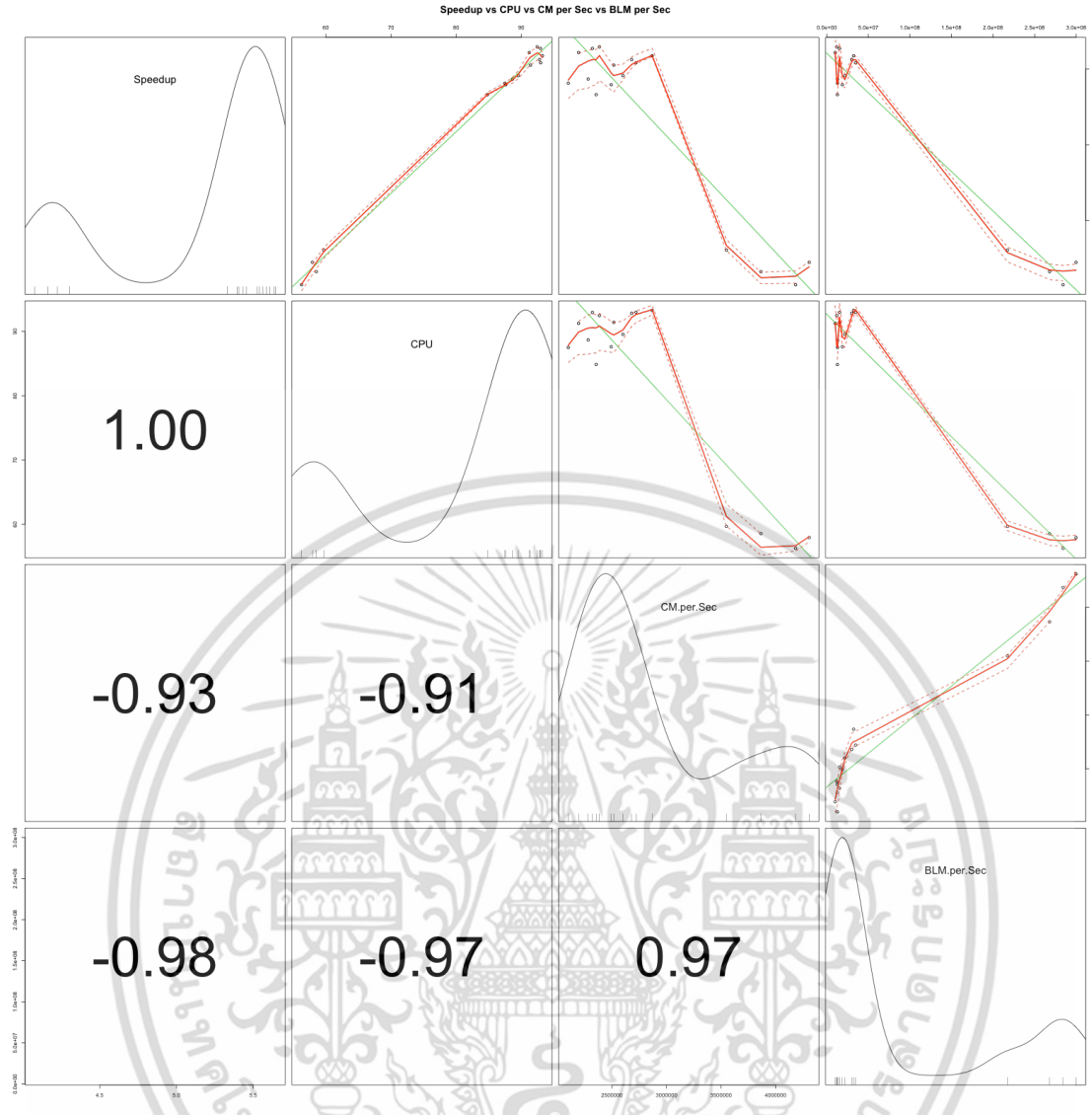


Figure 4.8 Speedup S vs %CPU Utilization U vs. C_m/s and B_m/s of PPMQSort on i7-2600 (Uint32, 200M, Random, o2, 4-32 Threads)

results in almost three times longer of Random-case T_{qsort} and $T_{ppmqsort}$ than those of Worst-case in the same table. As pointed out by Eyerman *et al.* [93], the misprediction penalty of superscalar CPUs with Reorder Buffer and deep pipeline equals to the number of clock cycles to refill the front-end pipeline. Other systems show similar behaviors as the i7-2600 system. It can be concluded that the branch prediction unit is as performance-critical as the memory hierarchy for parallel sorting algorithms due to the randomness of input data in modern multicore CPUs.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

This thesis has introduced a methodology for a parallel programming environment for OpenMP 3.0 library running on any shared memory/multicore/multi-socket systems. As the first case study, SNPHAP, a haplotype inference program based Expectation Maximization algorithm, is profiled and analyzed. A multithreaded parallel version of SNPHAP called OMP SNPHAP is developed. The OMP SNPHAP utilizes $parallelFor$ and $Task$ constructs of the OpenMP 3.0 library to spawn multiple independent threads so that they can be executed on multicore CPUs at the same time. Experimental results reveal that the performance of OMP SNPHAP on Xeon E5405, Xeon E5520 and AMD opteron 8356 running Linux operating system can achieve great Speedups up to 3.16x, 4.10x and 4.88x, respectively, compared with the running time of SEQ SNPHAP. It can be concluded that the Speedup of OMP SNPHAP is proportional to the number of threads forked by OpenMP. However, %overhead of OpenMP can significantly slow down the total execution time especially due to small input data with large number of threads (%overhead of 1 thread/core is much lower than that of 2 threads/ core). Moreover, other factors comprise the number of CPU cores, the parallel fraction part, the OpenMP constructs, the number of processor cores, the size of multilevel cache, the clock frequency, and eventually memory bandwidth/technology.

The second case study is the PPMQSort. Its basic concept is to divide the input data array by half in parallel/recursively until the obtained partitions are up to Cutoff size u. These partitions can be locally cached and $qsort()$ simultaneously by a number of threads equal or greater than number of processor cores. Hence, the performance bottleneck can be eliminated. PPMQSort is compatible with and compared against the Stdlib $qsort()$ as a benchmark. Performance of PPMQSort was evaluated on one AMD and four Intel CPUs running 64-bit Ubuntu Linux 14.4 LTS. In general, PPMQSort can achieve Speedups up to and beyond the number of non-HyperThread CPU cores for Random cases. For HyperThread CPUs, PPMQSort can get up to 50% Speedup increase over HT-disabled ones. In terms of

efficiency, the PPMQSort can get Speedup/Core from 0.97 to 1.17 and from 1.41 to 1.63 on non-HT and HT CPUs, respectively. Based on the statistical analysis, $T_{ppmqsort}$ is proportional to Cache Misses and Branch Load Misses. On the other hand, its Speedup S is proportional to %CPU Utilization U and limited by B_m/s and C_m/s.

5.2 Future Work

As future work, $qsort()$ could be replaced with PPMQSort to speedup SNPHAP even further. The PPMQSort itself should be investigated/optimized to support thread affinity/ cache locality and to minimize cache coherence misses even more. In addition, on-chip and off-chip GPUs (Graphics Processing Unit) should be investigated whether PPMQSort can be applied to exploit a massive number of GPU cores as it has been done on multicore CPUs. Finally, the performance model and memory bandwidth shall be analyzed and fine tuned to support a variety of algorithms/programs. To serve Big Data, Task scheduling and load balancing strategy are investigated by mixed CPU, memory, and I/O-intensive [94].

REFERENCES

[1] A. R. Board. (2015) The openmp api specification for parallel programming. [Online]. Available: http://www.openmp.org

[2] D. Clayton. (2015) Snphap (version 1.3.1). [Online]. Available: https://www-gene.cimr.cam.ac.uk/staff/clayton/software/

[3] L. Eronen, F. Geerts, and H. Toivonen, "Haplorec: efficient and accurate large-scale reconstruction of haplotypes," *BMC Bioinformatics*, vol. 7, p. 542, 2006.

[4] S. Akhter and J. Roberts, *Multi-core programming increasing performance through software multi-threading*. Intel Press, 2006.

[5] M. Evgeniy and C. A. Malchu, "Application of parallel computing technology openmp to search for the generator polynomials," in *International Conference on Mechanical Engineering, Automation and Control Systems (MEACS)*, Tomsk, Russia, October 16-18 2014, pp. 1 – 5.

[6] M. Wu, W. Wu, N. Tai, H. Zhao, J. Fan, and N. Yuan, "Research on openmp model of the parallel programming technology for homogeneous multicore dsp," in *5th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, Beijing, China, June 27-29 2014, pp. 921–924.

[7] M. B., R. Shahana, and W. Ahmed, "Parallel implementation of doolittle algorithm using openmp for multicore machines," in *IEEE International Advance Computing Conference (IACC)*, Banglore, India, June 12-13 2015, pp. 575–578.

[8] W. C. Pi, X. M. Pan, and X. Q. Sheng, "A parallel multilevel fast multipole algorithm based on openmp," in *International Conference on Microwave and Millimeter Wave Technology (ICMMT)*, Chengdu, China, May 8-11 2010, pp. 1356 – 1359.

[9] T. Xionggang and C. Jun, "Parallel image processing with openmp," in *The 2nd IEEE International Conference on Management and Engineering (ICIME)*, Chengdu, China, April 16-18 2010, pp. 20 – 23.

[10] A. M. de Lima, T. Webber, and M. A. S. Netto, "Openmp-based parallel algorithms for solving kronecker descriptors," in *22nd International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, Petropolis, Brazil, October 27-30 2010, pp. 55–60.

[11] I. Park, M. J. Voss, S. W. Kim, and R. Eigenmann, "Parallel programming environment for openmp," in *High Performance Computing in Science and Engineering*. Springer, 2002, pp. 111 – 125.

[12] D. Fallin and N. J. Schork, "Accuracy of haplotype frequency estimation for biallelic loci, via the expectation-maximization algorithm for unphased diploid genotype data," *The American Journal of Human Genetics*, vol. 67, pp. 947 – 959, 2000.

[13] P. Bonizzoni, G. D. Vedova, R. Dondi, and J. Li, "The haplotyping problem: an overview of computational models and solutions," *Journal of Computer Science and Technology (JCST)*, vol. 18, pp. 675 – 688, 2003.

[14] D. Gusfield and S. H. Orzack, *Haplotype Inference*, ser. CRC Handbook on Bioinformatics. Boca Raton, USA: CRC Press, 2005, ch. 1, pp. 1 – 25.

[15] C. A. R. Hoare, "Quicksort," *ACM*, vol. 4, p. 321, 1962.

[16] R. Sedgewick, "Implementing quicksort program," *Communications of the ACM*, vol. 21, no. 10, pp. 847–857, October 1978.

[17] A. D. Mishra, "Selection of best sorting algorithm for a particular problem," Master's thesis, Computer Science and Engineering Department, Thapar University, 2009.

[18] P. Huang and H. Chen, "Parallel algorithm for inferring haplotypes," Dept. of Computer Science, U.C. Berkeley, Tech. Rep., 2005.

[19] S. M. Bhandarkar and H. R. Arabnia, "The hough transform on a reconfigurable multi-ring network," *Journal of Parallel and Distributed Computing*, vol. 24, no. 1, pp. 107–114, 1995.

[20] H. R. Arabnia and S. M. Bhandarkar, "Parallel stereocorrelation on a reconfigurable multi-ring network," *Journal of Supercomputing*, vol. 10, no. 3, pp. 243–269, 1996.

[21] S. M. Bhandarkar and H. R. Arabnia, “Parallel computer vision on a reconfigurable multiprocessor network,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 3, pp. 292–309, Mar. 1997.

[22] D. Koch and J. Torresen, “Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting,” in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’11. New York, NY, USA: ACM, 2011, pp. 45–54.

[23] R. Mueller, J. Teubner, and G. Alonso, “Sorting networks on fpgas,” *The VLDB Journal*, vol. 21, no. 1, pp. 1–23, February 2012.

[24] J. Casper and K. Olukotun, “Hardware acceleration of database operations,” in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA ’14. New York, NY, USA: ACM, 2014, pp. 151–160.

[25] G. Capannini, F. Silvestri, and R. Baraglia, “Sorting on gpus for large scale datasets: A thorough comparison,” *Information Processing and Management*, vol. 48, no. 5, pp. 903–917, 2012.

[26] T. Xiaochen, K. Rocki, and R. Suda, “Register level sort algorithm on multi-core simd processors,” in *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 ’13. New York, NY, USA: ACM, 2013, pp. 9:1–9:8.

[27] P. Heidelberger, A. Norton, and J. T. Robinson, “Parallel quicksort using fetch-and-add,” *IEEE Transactions on Computers*, vol. 39, no. 1, pp. 847–857, January 1990.

[28] P. Tsigas and Y. Zhang, “A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000,” in *11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP 2003)*, Genoa, Italy, February 5th - 7th 2003, pp. 372–381.

[29] M. Sub and C. Leopold, “A user’s experience with parallel sorting and openmp,” in *Proc. of the 6th European Workshop on OpenMP (EWOMP 2004)*, Stockholm, Sweden, October 18-23 2004.

[30] D. Man, Y. Ito, and K. Nakano, “An efficient parallel sorting compatible with the standard qsort,” in *International Conference on Parallel and Distributed Computing, Applications and Technologies*, Hiroshima, Japan, December 8-11 2009, pp. 512 – 517.

[31] D. Men, Y. Ito, and K. Nakano, “An efficient parallel sorting compatible with the standard qsort,” *International Journal of Foundations of Computer Science*, vol. 22, no. 5, pp. 1057–1071, 2011.

[32] K. J. Kim, S. J. Cho, and J.-W. Jeon, “Parallel quick sort algorithms analysis using openmp 3.0 in embedded system,” in *11th International Conference on Control, Automation and Systems*, KINTEX, Gyeonggi-do, Korea, October 26-29 2011, pp. 757–761.

[33] B. A. Mahafzah, “Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture,” *Journal of Supercomputing*, vol. 66, pp. 339–363, 2013.

[34] T. Bingmann, A. Eberle, and P. Sanders, “Engineering parallel string sorting,” *Algorithmica*, pp. 1–52, 2015.

[35] Wikipedia. (2015) Multi-core processor. [Online]. Available: https://en.wikipedia.org/wiki/Multi-core_processor

[36] Intel. (2015) Threading building blocks (tbb). [Online]. Available: http://threadingbuildingblocks.org

[37] MPI-Forum. (2015) Mpi: A message passing interface. [Online]. Available: http://www.mpi-forum.org

[38] Intel. (2015) An introduction to the intel quickpath interconnect. [Online]. Available: http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html

[39] J. M. Butler, *FORENSIC DNA TYPING*, ser. Second edition. Elsevier Academic Press, 2005.

[40] GNN. (2015) Genome variations. [Online]. Available: http://www.genomenewsnetwork.org/resources/whats_a_genome/Chp4_1.shtml

[41] Y. S. Song, Y. Wu, and D. Gusfield, "Algorithms for imperfect phylogeny haplotyping (ipph) with a single homoplasy or recombination event," in *5th Workshop on Algorithms in Bioinformatics (WABI'2005)*. Mallorca, Spain: Lecture Notes in Computer Science 3692, October 3-6 2005, pp. 152–164.

[42] S. Rahman, J. Hein, and T. Melham, "Algorithms for haplotype inference from genotypes in the presence of recombination," Master's thesis, Oxford University, Computing Laboratory and Department of Statistics, 2006.

[43] M. E. Hawley and K. K. Kidd, "Haplo: a program using the em algorithm to estimate the frequencies of multi-site haplotypes," *The Journal of heredity*, vol. 86, no. 5, pp. 409–4011, 1995.

[44] J. C. Long, R. C. Williams, and M. Urbanek, "An e-m algorithm and testing strategy for multiple locus haplotypes," *The American Journal of Human Genetics*, vol. 56, pp. 799–8103, 1995.

[45] L. Excoffier and M. Slatkin, "Maximum-likelihood estimation of molecular haplotype frequencies in a diploid population," *Molecular Biology and Evolution*, vol. 12, pp. 921 – 927, 1995.

[46] M. Stephens, N. J. Smith, and P. Donnelly, "A new statistical method for haplotype reconstruction from population data," *The American Journal of Human Genetics*, vol. 68, pp. 978–989, 2001.

[47] A. G. Clark, "Inference of haplotypes from pcr-amplified samples of diploid populations," *Molecular Biology and Evolution*, vol. 7(2), pp. 111–122, 1990.

[48] C.-F. Xu, K. Lewis, K. L. Cantone, P. C. Donnelly, N. White, N. Crocker, P. R. Boyd, D. V. Zaykin, and I. J. Purvis, "Effectiveness of computational methods in haplotype prediction," *Human Genetics*, vol. 110(2), pp. 148–156, 2002.

[49] S. Zhang, A. J. Pakstis, K. K. Kidd, and H. Zhao, "Comparisons of two methods for haplotype reconstruction and haplotype frequency estimation from population data," *The American Journal of Human Genetics*, vol. 69(4), pp. 906–912, 1990.

[50] X. S. Zhang, R. S. Wang, L. Y. Wu1, and L. Chen, "Models and algorithm for haplotyping problem," *Current Bioinformatics*, vol. 1, pp. 105 – 114, 2006.

[51] G. Abecasis. (2015) Fugue project home page. [Online]. Available: http://www.sph.umich.edu/csg/abecasis/fugue/index.html

[52] S. S. Li and L. P. Zhao. (2015) Hplus haplotype analysis. [Online]. Available: http://cdsweb01.fhcrc.org/HPlus/

[53] C. Lambert and G. Helix. (2015) Helixtree. [Online]. Available: http://www.statistics.com/software-directory/helixtree

[54] G. Kimmel and R. Shamir, "Gerbil: Genotype resolution and block identification using likelihood," in *Proceedings of the National Academy of Sciences*, January 4 2005, pp. 158–162.

[55] Z. S. Qin, T. Niu, and J. S. Liu, "Partition-ligation-expectation-maximization algorithm for haplotype inference with single-nucleotide polymorphisms," *The American Journal of Human Genetics*, vol. 70, pp. 1242–1247, 2002.

[56] S. P and S. M, "A fast and flexible statistical model for large-scale population genotype data: Applications to inferring missing genotypes and haplotypic phase," *The American Journal of Human Genetics*, vol. 78(4), pp. 629–644, 2006.

[57] M. Stephens, N. J. Smith, and P. Donnelly, "A new statistical method for haplotype reconstruction from population data," *The American Journal of Human Genetics*, vol. 68, pp. 978–989, 2001.

[58] M. Stephens and P. Scheet, "Accounting for decay of linkage disequilibrium in haplotype inference and missing-data imputation," *The American Journal of Human Genetics*, vol. 76(3), pp. 449–462, 2005.

[59] L. Eronen, F. Geerts, and H. Toivonen, "Haplorec: Efficient and accurate large-scale reconstruction of haplotypes," *BMC Bioinformatics*, vol. 7, p. 542, 2006.

[60] R. Sedgewick, "Quicksort," Ph.D. dissertation, Stanford University, 1975.

[61] P. Hennequin, "Analyse en moyenne d'algorithmes: tri rapide et arbres de recherche," Ph.D. dissertation, Ecole Politechnique, Palaiseau, 1991.

[62] V. Yaroslavskiy. (2015) Replacement of quicksort in java.util.arrays with new dual-pivot quicksort. [Online]. Available: http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628

[63] S. Wild and M. E. Nebel, "Average case analysis of java 7's dual pivot quicksort," in *Proceedings of the 20th European Symposium on Algorithms (ESA'12)*, 2012, p. 825–836.

[64] M. Aumüller and M. Dietzfelbinger, "Optimal partitioning for dual pivot quicksort," in *Proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP'13)*, ser. Lecture Notes in Computer Science. Springer, 2013, p. 33–34.

[65] J. I. M. Shrinu Kushagra, Alejandro López-Ortiz and A. Qiao, "Multi-pivot quicksort: Theory and experiments," in *Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX 2014)*, Portland, Oregon, USA, January 5 2014, p. 47–60.

[66] R. Sedgewick, "The analysis of quicksort programs," *Acta Informatica*, vol. 7, no. 4, pp. 327–355, 1977.

[67] ——, "Implementing quicksort programs," *Communications of the ACM*, vol. 21, no. 10, pp. 847–857, October 1978.

[68] H. Erkio, "The worst case permutation for median-of-three quicksort," *Computer Journal*, vol. 27, no. 3, pp. 276–277, 1984.

[69] J. Singler, P. Sanders, and F. Putze, "Mcstl: The multi-core standard template library," in *Euro-Par 2007 Parallel Processing*. Springer, 2007, pp. 682–694.

[70] D. Traoré, J.-L. Roch, N. Maillard, T. Gautier, and J. Bernard, "Deque-free work-optimal parallel stl algorithms," in *Euro-Par 2008–Parallel Processing*. Springer, 2008, pp. 887–897.

[71] L. Frias and J. Petit, *Parallel partition revisited*. Springer, 2008.

[72] L. Rashid, W. M.Hassanein, and M. A.Hammad, "Analyzing and enhancing the parallel sort operation on multithreaded architectures," *Journal of Supercomputing*, vol. 53, pp. 293–312, 2010.

[73] D. Zurek, M. Pietron, M. Wielgosz, and K. Wiatr, “The comparison of parallel sorting algorithms implemented on different hardware platforms,” *Computer Science*, vol. 14, no. 4, p. 679, 2013.

[74] S. Saleem, M. I. Lali, M. S. Nawaz, and A. B. Nauman, “Multi-core program optimization: Parallel sorting algorithms in intel cilk plus,” *International Journal of Hybrid Information Technology*, vol. 7, no. 2, pp. 151–164, 2014.

[75] A. Maus, “A full parallel quicksort algorithm for multicore processors,” *Norsk Informatikkonferanse (NIK)*, 2015.

[76] S. L. Graham, P. B. Kessler, and M. K. McKusick, “gprof: A call graph execution profiler,” in *SIGPLAN ’82 Symposium on Compiler Construction*, ser. SIGPLAN Notices, vol. 17, no. 6, June 1982, pp. 120 – 126.

[77] K. Fuerlinger and M. Gerndt, “ompp: A profiling tool for openmp,” in *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, May 2005.

[78] E. Ayguade, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel, “An experimental evaluation of the new openmp tasking model,” in *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC2007)*, October 2007.

[79] K. Fuerlinger. (2015) ompp profiler tool. [Online]. Available: http://www.ompp-tool.com

[80] M. Nothnagel. (2015) Snap: simulation of snp haplotype data and phenotypic traits. [Online]. Available: http://portal.ccg.uni-koeln.de/ccg/index.php?id=61

[81] B. M. Tudor and Y. M. Teo, “A practical approach for performance analysis of shared-memory programs,” in *Proceeding of 25th IEEE International Parallel and Distributed Processing Symposium*, Anchorage, USA, May 16-20 2011.

[82] U. Ranok, S. Kittitornkun, and S. Tongsima, “Multithreading bioinformatics software with openmp: Snphap case study,” in *Proceeding of the IASTED International Conference Parallel and Distributed Computing and Systems (PDCS2010)*, Marina Del Rey, USA, 2010.

[83] A. Grama, A. Gupta, G. Karypis, , and V. Kumar, *Introduction to Parallel Computing*, ser. second edition. Pearson Education Limited, 2003.

[84] U. Ranok, S. Kittitornkun, and S. Tongsima., "A multithreading methodology with openmp on multi-core cpus: Snphap case study," in *8th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON 2011)*, Khon Kaen, Thailand, May 17-19 2011.

[85] J. L. Gustafson, "Fixed time, tiered memory, and superlinear speedup," in *In Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, 1990.

[86] D. P. Helmbold and C. E. Mcowell, "Modeling speedup (n) greater than n," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 250–256, April 1990.

[87] V. M. Weaver, "Linux perf event features and overhead," in *Second International Workshop on Performance Analysis of Workload Optimized Systems (FastPath 2013)*, Austin, TX, USA, April 21 2013.

[88] Y. Zhang, Z.-P. Li, and H.-F. Cao, "System-enforced deterministic streaming for efficient pipeline parallelism," *Journal of Computer Science and Technology*, vol. 30, no. 1, pp. 57–73, January 2015.

[89] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, ser. second edition. Pearson Education Limited, 2003.

[90] S. Akhter and J. Roberts, *Multi-core programming increasing performance through software multi-threading*. Intel Press, 2006.

[91] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. LANG, S. PAKIN, and J. C. SANCHO, "A performance evaluation of the nehalem quad-core processor for scientific computing," *Parallel Processing Letters*, vol. 18, no. 4, pp. 453–469, 2008.

[92] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[93] S. Eyerman, J. E. Smith, and L. Eeckhout, "Characterizing the branch misprediction penalty," in *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS 2006)*, Austin, Texas, USA, March 19-21 2006, pp. 48–58.

[94] K. Qureshi, B. Majeed, J. H. Kazmi, and S. A. Madani, "Task partitioning, scheduling and load balancing strategy for mixed nature of tasks," *J Supercomput*, vol. 59, no. 3, pp. 1348–1359, 2012.

[95] J. Polanska, “The em algorithm and its implementation for the estimation of frequencies of snp-haplotypes,” *International Journal of Applied Mathematics and Computer Science*, vol. 13, no. 3, pp. 419 – 429, 2003.

[96] X. Xie and J. Ott, “Testing linkage disequilibrium between a disease and and a marker locus,” *The American Journal of Human Genetics*, vol. 53, pp. 1107–1113, 1993.

APPENDICES

APPENDIX A

SNPHAP

The haplotype inference (HI) problem is the problem of inferring $2G$ haplotype pairs from G observed individual genotype data in the sample [14, 41]. In practice, each SNP element in genotype is called as loci site. From Figure A.1 , as we can see that the number of possible solutions depends on the number 2 alphabets (heterozygous loci site) in an individual's genotype sequence. Therefore, as an individual with k heterozygous loci sites the number of possible solutions is 2^{k-1}.

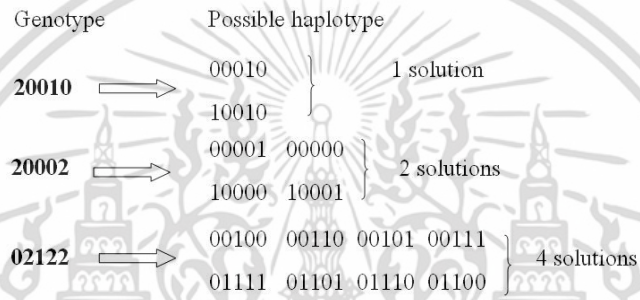


Figure A.1 Haplotype Inference Examples: heterozygous loci site, $k = 1, 2$, and 3. The number of possible solutions is 2^{k-1}.

We begin with the same assumption and notations as Polanska [95]. Given G different genotypes from a population. In order to genotype defined as a combination of two haplotypes, the number of possible haplotype pairs (c_j) compatible for the j-th genotype can be evaluated as Equation A.1.

$$c_j = \begin{cases} 2^{k_j - 1} & \text{if } k_j > 0, \\ \frac{1}{2} & \text{if } k_j = 0. \end{cases} \tag{A.1}$$

where k_j is the number of heterozygous loci of the j-genotype.

Moreover, the maximum number of possible haplotypes n_h is calculated by Equation A.2.

$$n_h = 2 \sum_{j=1}^{G} c_j. \tag{A.2}$$

Haplotypes can be inferred and their frequencies can be estimated via a maximum

likelihood (ML) approach. Under the assumption of the Hardy-Weinberg equilibrium (HWE) and random mating in the population [45], the probability P_j of the j-th genotype is given by a sum of h_{p_i} haplotype pairs with their probability of each the possible c_j haplotype pairs :

$$P_j = \sum_{i=1}^{c_j} P(h_{p_i}) = \sum_{i=1}^{c_j} P(h_t, h_{t+1}), \tag{A.3}$$

where $P(h_t, h_{t+1})$ is the probability that the i-th haplotype pairs made up of haplotype pairs : h_t and h_{t+1}

$$P(h_t, h_{t+1}) = \begin{cases} f_t f_{t+1} & \text{if } h_t = h_{t+1}, \\ 2 f_t f_{t+1} & \text{if } h_t \neq h_{t+1}. \end{cases} \tag{A.4}$$

where f_t and f_{t+1} are the frequencies of the h_t and the h_{t+1} and corresponding to haplotype pairs h_{p_i}.

The (L)ikelihood function of the haplotype frequencies given the genotype probabilities is

$$L(f_1, f_2, ..., f_{n_h} | P_1, P_2, ..., P_m) = a \prod_{j=1}^{G} \Big(\sum_{i=1}^{c_j} P(h_t, h_{t+1}) \Big)^{n_j}, \tag{A.5}$$

where $\sum_{t=1}^{n_h} f_t = 1$ and (h_t, h_{t+1}), $1 \leq i \leq c_j$ are the set of explanations of the j-th genotype that occurs n_j times in the population and a is a constant incorporating the multinomial coefficient.

In ML estimation, the goal is to estimate the parameter values which make the likelihood of observing data as large as possible. The solution can be computed by taking partial derivatives of the log-likelihood function from the n_h partial derivatives equated to 1 for the t-th haplotype.

$$\frac{\partial log L}{\partial f_t} = \sum_{j=1}^{G} \frac{n_j \partial P_j}{P_j \partial f_t}, t = 1, 2, ..., n_h \tag{A.6}$$

However, this procedure is tiresome when n_h is large, and the number n_h is often

unknown. Alternative procedures involving numerical iteration have been developed. Statisticians use a numerical optimization technique called (E)xpectation-(M)aximization (EM) for maximizing the likelihood function [43, 45, 96]. The goal is to find the best estimates of the haplotype frequencies in the population using only the genotype sample data. Fortunately, haplotypes can be obtained from genotype data through experimental software techniques. There exist many methods for inferring haplotypes from genotype data. One of the most widely used methods is to statistically estimate haplotype frequencies from genotypes in unrelated individuals with EM algorithm to overcome the missing or incomplete data [13, 14]. The EM algorithm can take longer computation time [45, 50] caused by the large number of possible haplotype instances. Furthermore, with the advent of large-scale genotyping platforms that allow researchers to observe more than 1 million markers distributed throughout the entire genome, the sequential haplotyping software requires too much computation time to complete the analysis. Then, if the method can be extended into parallel algorithms, the throughput can be increased.

The EM algorithm is an iterative method [14] which is a general technique for finding ML estimation of haplotype frequencies $f_1, f_2, ..., f_{n_h}$, starting with some initial values $f_1^{(0)}, f_2^{(0)}, ..., f_{n_h}^{(0)}$. These initial values are used to estimate of the "prior" haplotype frequencies in the M step. The prior haplotype frequencies are used to estimate the explanation frequencies $P(h_t, h_{t+1})$ and then "posterior" haplotype frequencies as if they were the unknown true frequencies in the E step. These expected frequencies are standardized and used to estimate haplotype frequencies $f_1^{(1)}, f_2^{(1)}, ..., f_{n_h}^{(1)}$ at the next iteration in the M step and repeat M step and E step until haplotype frequencies are stable. The algorithm is elaborated in Algorithm 4.

Algorithm 4 EM Algorithm according to SEQ SNPHAP (version 1.3.1)

1. "Guesstimate" haplotype frequencies:
   initialize $f_t^{(l)}, \Theta^{(l)}$, EM iteration number $(l) = 0$,
   $\forall t : 1 \leq t \leq n_h$.
   Let t is any arbitrary number and $P(h_t, h_{t+1})$ is the probability that h_{p_i} is composed of haplotype pair h_t and h_{t+1}, and also f_t and f_{t+1} are the frequencies of the h_t and h_{t+1}.

2. (M)aximization step corresponds to *hap_prior()* in *SNPHAP*:
   This step can be divided into two parts. First, the total of posterior frequencies is calculated by Equation A.7. Then, the next set of estimates of "prior" haplotype frequencies are calculated by Equation A.8.

$$P_{total}^{(l)} = \sum_{t=1}^{n_h} f_t^{(l)}, \tag{A.7}$$

$$\widetilde{f}_t^{(l)} = \frac{\sum_{\forall h_t \in H_x} f_t^{(l)}}{P_{total}^{(l)}}, (\forall t : 1 \leq t \leq n_h), \tag{A.8}$$

   where $H_x = \{$all elements in group x are the same haplotype$\}$
   (The comparison in this procedure corresponds to *cmp_hap()* inside *hap_prior()* in *SNPHAP*).

3. (E)xpectation step corresponds to *hap_posterior()* in *SNPHAP*:
   Given current estimates of prior frequencies from M step and assuming HWE shown as Equation A.9, Equation A.10 calculates the probability of each phased, assign the frequencies of the haplotype pair with Equation A.11, and then complete the "posterior" haplotype frequencies of the genotype assignments by Equation A.12. Finally, the log-likelihood is calculated with Equation A.13.

   Let P_j as the probability of the j-th genotype that maximizes the likelihood that is the sum of probabilities of haplotype pairs h_{p_i} of each c_j

$$P(h_{p_i})^{(l)} = P(h_t, h_{t+1})^{(l)} = \begin{cases} \widetilde{f}_t^{(l)} \widetilde{f}_{t+1}^{(l)} & \text{if } h_t = h_{t+1}, \\ 2\widetilde{f}_t^{(l)} \widetilde{f}_{t+1}^{(l)} & \text{if } h_t \neq h_{t+1}. \end{cases} \tag{A.9}$$

$$P_j^{(l)} = \sum_{i=1}^{c_j} P(h_{p_i})^{(l)}, \tag{A.10}$$

$$f_t^{(l)} = f_{t+1}^{(l)} = P(h_t, h_{t+1})^{(l)} \tag{A.11}$$

$$f_t^{(l+1)} = \frac{f_t^{(l)}}{P_j^{(l)}}, (\forall j : 1 \leq j \leq G, \forall t : 1 \leq t \leq n_h), \tag{A.12}$$

$$\Theta^{(l+1)} = \log(\sum_{j=1}^{G} \sum_{i=1}^{c_j} P(h_{p_i})^{(l)}) \tag{A.13}$$

4. If $\mid \Theta^{(l+1)} - \Theta^{(l)} \mid > \varepsilon$ (log likelihood tolerance), $l < l_{max}$ then $l = l + 1$ and go to M Step.
   Else $\Theta^* = \Theta^{(l+1)}$

BIOGRAPHY

Personal Information

Name	Ratthaslip Ranokphanuwat
Sex	Male
Nationality	Thai
Date of Birth	June 13, 1975 at Nakhon Ratchasima Province, Thailand
Office Address	Department of Computer Engineering, Faculty of Engineering, Dhurakij Pundit University 110/1-4 Prachachuen Road, Laksi, Bangkok 10210. Phone: +662-954-7300 ext. 591 Mobile: 084-135-3676

Education

*1997-1999, Bachelor of Eng. in Computer Engineering (King Mongkut's Institute of Technology Ladkrabang, Thailand).

*2002-2006, Master of Eng. in Computer Engineering (King Mongkut's Institute of Technology Ladkrabang, Thailand).

Research Interests

Parallel computing, Big Data and Embedded systems.

Work Experience

In 2000 - current: Lecturer at the Faculty of Engineering, Dhurakij Pundit University and taught following subjects:

Parallel computing, Embedded systems, Computer architecture, Operating system, Digital logic circuits and Electronic circuits.

LIST OF PUBLICATIONS

Some parts of this work are published in the following articles.

International Conference Proceedings

1. **Udom Ranok***, Surin Kittitornkun and Sissades Tongsima, "MULTITHREADING BIOINFORMATICS SOFTWAREWITH OPENMP:SNPHAP CASE STUDY", The IASTED International Conference Parallel and Distributed Computing and Systems (PDCS 2010), November 8 - 10, 2010, Marina Del Rey, USA.
2. **Udom Ranok***, Surin Kittitornkun and Sissades Tongsima, "A Multithreading Methodology with OpenMP on Multi-core CPUs: SNPHAP Case Study", The 8th International Conference on Electrical Engineering/Electronics, Computer, Telecommunication and Information Technology (ECTI-CON 2011) May 17-20, 2011, Khon Kaen, Thailand.
3. **Ratthaslip Ranokphanuwat**, Surin Kittitornkun, and Sissades Tongsima, "Performance analysis & improvement of SNPHAP on Multi-core CPUs", The 10th International Conference on Electrical Engineering/Electronics, Computer, Telecommunication and Information Technology (ECTI-CON 2013) May 15-17, 2013, Krabi, Thailand.
   (**Best Paper Award**)

International Journal Paper

1. **Ratthaslip Ranokphanuwat**, and Surin Kittitornkun, "Parallel Partition and Merge QuickSort (PPMQSort) on Multicore CPUs", Journal of Supercomputing, vol. 72, no. 3, pp. 1063-1079, 2016.
   (**ISI impact factor 0.858**).

*Remark of name & surname changes

ECTI-CON 2011

KHON KAEN UNIVERSITY

8th Electrical Engineering/ Electronics, Computer, Telecommunications and Information Technology (ECTI) Association of Thailand - Conference 2011

Khon Kaen, Thailand

May 17-19, 2011

Pullman Khon Kaen Raja Orchid Hotel

ISBN 978-1-4577-0424-6
IEEE Catalog Number: CFP1106E-ART

ECTI Association

KHON KAEN UNIVERSITY

IEEE THAILAND SESSION

Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems (PDCS 2010)
November 8 - 10, 2010 Marina Del Rey, USA

MULTITHREADING BIOINFORMATICS SOFTWARE WITH OPENMP: SNPHAP CASE STUDY

Udom Ranok and Surin Kittitornkun
School of Computer Engineering and Information Science, Faculty of Engineering
King Mongkut's Institute of Technology Lardkrabang
Bangkok 10520, Thailand
email: s100635@kmitl.ac.th, kksurin@kmitl.ac.th
Sissades Tongsima
Genome Institute, National Center for Genetic Engineering and Biotechnology
113 Thailand Science Park, Paholyothin Road, Klong 1, Klong Luang
Pathumtani 12120, Thailand
email: sissades@biotec.or.th

ABSTRACT
This paper presents a parallelization framework for inferring haplotypes using an expectation maximization (EM) algorithm. Our framework utilizes GProf profiling tool, OpenMP library, and ompP profiling tool to parallelize the algorithm by determining the hotspot functions, multithreading, and executing them on the Multi-core CPUs. In our experiments, we choose the SNPHAP program for this case study and run it on an 8-core Xeon Linux machine. The results show that our framework can significantly speedup up to 214% on a large data set with 151 loci of a 10,000 data samples. In addition, deep profiles of multithreaded SNPHAP support our discovery that maximum speedup can be achieved when the number of parallel threads equals to the number of physical cores.

KEY WORDS
Parallel Processing, Multithread, OpenMP, Multi-core CPU, Haplotype, SNPHAP

1 Introduction

Haplotype provides a snapshot of human evolution to estimate the age and location of disease mutations related to a set of multiple linked markers and to investigate many population processes [1]. In other words, an inferred haplotype is not a direct observation but can be obtained from unphased genotype data (resulting in haplotype ambiguity) through experiment in software techniques. There exist many methods for inferring haplotypes from unphased genotype data. One of the most widely used methods is to statistically estimate haplotype frequencies from genotyping markers of a group of unrelated individuals by employing the Expectation Maximization (EM) algorithm to predict the missing or incomplete data [2][3]. This problem can take exponential computation time [4][5] caused by a large number of possible haplotype instances. Furthermore, with the advent of large-scale genotyping platforms that allow researchers to observe more than 1 million markers distributed throughout the entire genome, the sequential haplotyping software requires too much computational time to complete the analysis. Then, if the method can be extended into parallel algorithms, the throughput may be increased.

Recently, there has been little research to implement the EM algorithm for inferring haplotypes in parallel. Huang and Chen [6] use Message Passing Interface (MPI) library to partition the genotype data and distribute them to each processor. They run the parallelized program on the CTRIS cluster with up to 16 processors. This parallel solution can be unreachable to researchers without large computer clusters. However, major chip makers are currently producing multi-core CPUs with Hyper-Threading . Hence, multi-core/Hyper-Threading technologies can enable parallel processing/multithreading with less expensive hardware and better efficiency [8].

In this paper, we present a framework to parallelize SNPHAP [9], which is a haplotype inference bioinformatics program using EM algorithm. The parallel program can be multithreaded by OpenMP library [10] so that it can be executed on multi-core CPUs. Our achievement is the significant 214% speedup over the sequential version.

This paper is organized as follows: Section 2 briefly reviews the background and work related to this paper. Our method is presented in section 3. Then, section 4 discusses the experimental setup and results. Finally, section 5 presents the conclusion of this work and some directions for future work are also provided.

2 Literature Review

2.1 SNPHAP: Haplotype and Expectation Maximization (EM)

Haplotype analyses are tools for studying human genetics and also used to investigate many population processes, such as migration and immigration rates, linkage disequilibrium strength, and the related populations [1]. Haplo-

type can be obtained from unphased genotype data (resulting in haplotype ambiguity) through experimental software techniques. There exist many methods for inferring haplotypes from unphased genotype data. One of the most widely used methods is to statistically estimate haplotype frequencies from maker genotype in unrelated individuals with EM algorithm to overcome the missing or incomplete data [2].

EM algorithm is an iterative method [3] which is a general technique for finding maximum likelihood estimates (MLEs) of parameter on incomplete data. The procedures consist of (E)xpectation step and (M)aximization step as shown in Figure 1 and as follows.

1. Initial parameter values: Guesstimate the frequencies
2. (E)xpectation step corresponding to hap_posterior() function in Figure 1:
   Compute expected values with the current estimate of the distribution for the missing values.
3. (M)aximization step corresponding to *hap_prior()* function in Figure 1:
   Compute MLE found on E step for parameters used to determine the distribution of the missing values in the next E step.
4. Repeat with a new set of parameters until the parameter estimation is stable.

There exists some software utilizing EM algorithm to infer haplotype frequencies. SNPHAP [9] is the chosen bioinformatics software for our work. SNPHAP is a sequential program for inferring haplotype frequencies using unphased genotype data from a number of unrelated individuals, in addition to missing data at some loci. Given a genotype, it uses an EM algorithm to calculate MLEs of haplotype frequencies. For a large number of loci, the number of possible haplotype instances becomes impossibly large. Therefore, one of the solutions for this time consuming haplotype problem is parallel processing (multithreading) on multi-core CPUs.

2.2 OpenMP Library

OpenMP library [10] is an application program interface (API) for thread based parallelism on shared memory multi-core processors. The API consists of a set of compiler directives, library routines, and environmental variables that support FORTRAN and C/C++ on multiple architectures. For programmers, OpenMP provides a portable, scalable model of thread based parallelism application. As a result, OpenMP programmers do not need to specify all the details. They only make strategic decisions and then compiler will figure out the details.

The model of multithreading execution is the fork-join model. The Master thread initially starts the running program. After the program is executed for a while, the Worker threads are spawned (forked) at a parallel region to

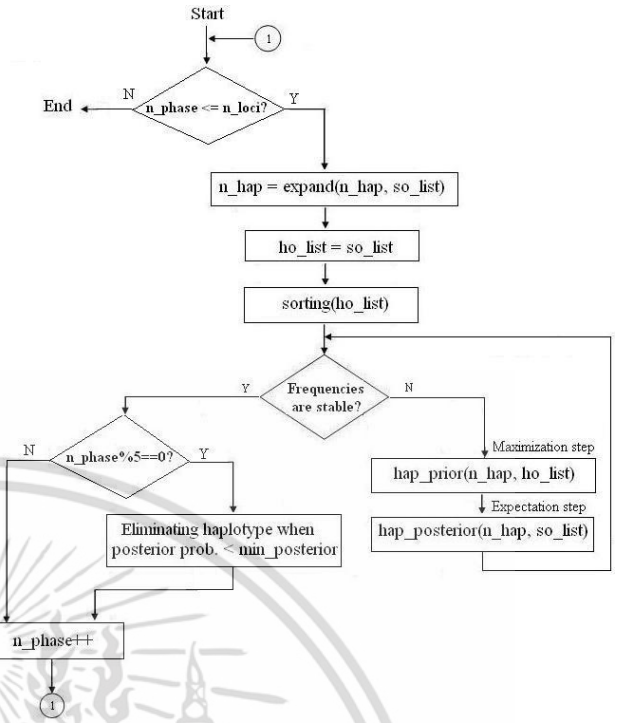


Figure 1: SNPHAP program flowchart

form a team together with the Master thread. At the end of each parallel region, all threads are synchronized (joined) before the Master thread can continue further. A parallel region can also nest with other parallel regions.

The main advantage of using OpenMP is the ability of all CPU cores to share and access the same memory pool (data) with less communication overhead and network latency compared with other parallel computing paradigms such as cluster computing, grid computing, etc.

2.3 Other Parallel Processing Methods for inferring Haplotype using EM

In this section, we review the only existing paper of parallel processing for inferring Haplotype using EM algorithm. Huang and Chen [6] have done experiments as a final project by inferring haplotype using EM algorithm with MPI (Message Passing Interface) library for passing information among the processors. They hierarchically partitioned the data and distributed to each processor for one fragment of the sequence. The final result was assembled progressively. Their program ran on the CTRIS cluster with up to 16 processors with only 250 samples and 130 loci. Finally, the 16-over 2-CPU speedup is 4x.

3 Our Multithreading Method

To parallelize the sequential SNPHAP to execute on the multi-core CPUs in multiple threads, multithreading technique can be applied by inserting OpenMP directives on

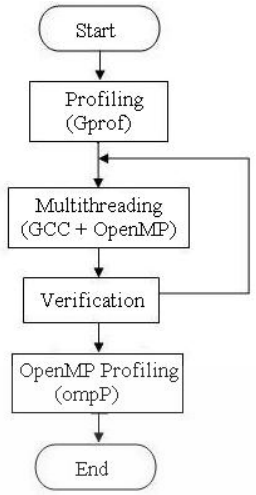


Figure 2: Applying the multithreading directives using OpenMP library flowchart

index	%time	self	children	called	name
[1]	100	0.11	1065.70		main [1]
		37.60	618.16	3930/3930	hap_prior [3]
		65.78	320.99	4080/4080	hap_posterior [4]
		319.48	0.00	293886461/874204285	hap_posterior [4]
		618.16	0.00	568631912/874204285	hap_prior [3]
[2]	89.2	950.35	0.00	874204285	cmp_hap [2]

Figure 3: Results of GProf profiling the SNPHAP program (DataSet = 5000, Loci = 151)

hotspot and independent functions. Our method consists of four major steps as shown in Figure 2. Our method consists of Profiling, Multithreading, Verification, and OpenMP Profiling. Each step will be explained in the following subsections.

3.1 Profiling

First of all, the sequential program, SNPHAP, is profiled to find all hotspot functions that consume significant portion of execution time. Our profiling tool is GProf [11]. The profile of sequential SNPHAP is shown in Figure 3. Then, the OpenMP compiler directives are inserted on those functions to convert them as threads just like Fork-Join model.

According to Figure 3, it is clear that *cmp_hap()* function is called by 874,204,285 times, while the happosterior() and the *hap_prior()* functions are called by just 4,080 and 3,930 times, respectively. Among 874,204,285 times, the *cmp_hap()* function is called 568,631,912 and 293,886,461 times by *hap_prior()* and *hap_posterior()* functions, respectively. Therefore, the *cmp_hap()* function should be parallelized within *hap_prior()* and *hap_posterior()* functions and executed by multi-core CPUs to significantly reduce the total execution time.

3.2 Multithreading with OpenMP library

As referred to profile in the previous section, the loop of the *cmp_hap()* should be optimized within both the *hap_prior()* and the *hap_posterior()* function to gain performance. Function *cmp_hap()* compares a couple of haplotypes until it reaches the last haplotype. But the loops of *cmp_hap()* function are not independent. So parallelizing the *cmp_hap()* is not appropriate due to its small loop body. In order to run *cmp_hap()* function in parallel, both *hap_prior()* and the *hap_posterior()* functions have to be modified by moving the *cmp_hap()* to compute in the outer loop. Moreover, the *cmp_temp[]* array is allocated to store temporary output for this technique. In addition, the outputs of comparison are kept in *cmp_temp[]* array for availability for the next while loop.

The OpenMP multithreaded version of the *hap_prior()* and *hap_posterior()* functions are shown in Figure 4. As it can be seen in each function, a *parallel region* is defined to contain a *work-sharing* construct. A parallel region is a block of code executed by multiple threads simultaneously, while a work-sharing construct divides the iterations of the loop to distribute over the CPU cores. Our study makes use of an OpenMP *parallel for* construct to define a parallel region. In addition, the *private(hs,h)* clause is added to define *hs* and *h* pointers as a private copy of each thread. When the master thread encounters the parallel region, the code within the parallel construct is copied to a number of worker threads. On the other hand, different sets of haplotypes (data) are distributed to those worker threads for parallel execution.

3.3 Output Verification

The parallelized SNPHAP results must be verified again and again until they are close to those of sequential version. As shown in Figure 5, a utility program named Beyond Compare matches and differentiates both output files from OpenMP and sequential SNPHAP in black and red, respectively.

3.4 OpenMP profiling

After the multithreaded SNPHAP results are completely verified in step 3, the *parallel for* constructs are analyzed to investigate the performance characteristics of multithreading with OpenMP profiling tool named ompP [12]. As shown in Figure 6, ompP profiling report consists of Header, Region Overview, Flat Region Profile, Callgraph Profile and Overhead Analysis Report. Header contains general information such as data and time of the program. Region Overview shows number of OpenMP regions (constructs) and their source code locations. The ompP profiler can examine the effectiveness and efficiency of OpenMP parallel constructs. The profiling report consists of threads distribution and overhead analysis. In this paper, we can

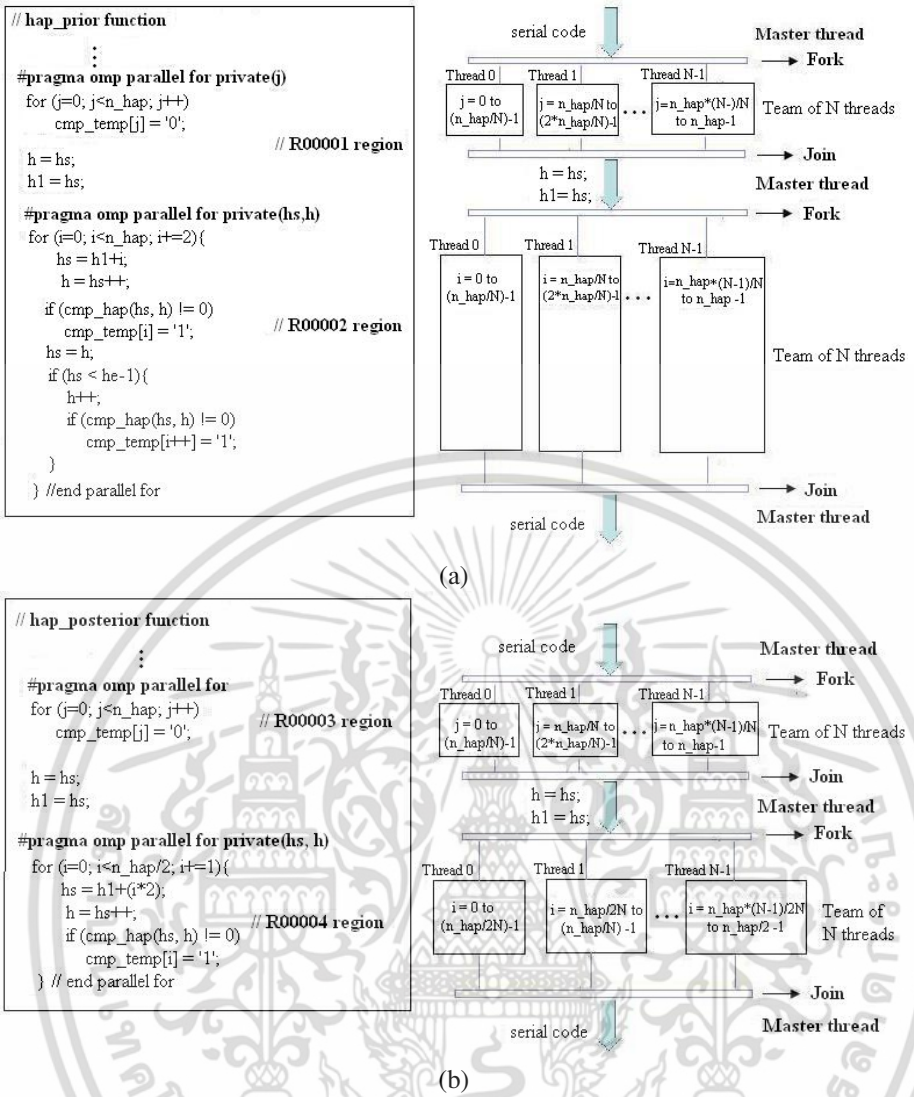


Figure 4: Parallellization of *cmp_hap()* function and code optimization inside (a)*hap-prior()* and (b)*hap-posterior()* functions with OpenMP

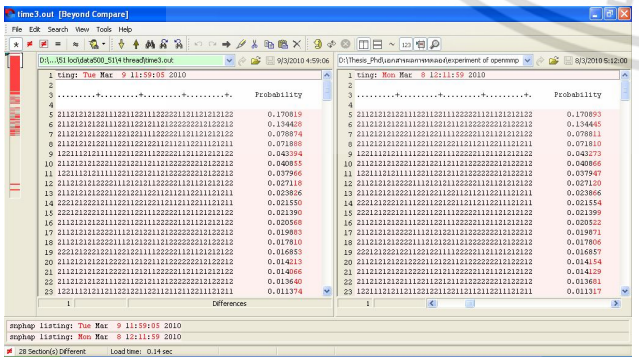


Figure 5: Verifying haplotyping results of OpenMP and sequential SNPHAP by using Beyond Compare.

determine each parallel region whether it can be parallelized and sped up in accordance with the number of threads forked by OpenMP.

In summary, our methodology is the use of OpenMP library directives to perform multithreading at functions consuming most execution time according to the profiler. However, due to dependencies and sequential behavior of SNPHAP, the source code must be modified to preserve correctness. In addition, parallel behavior must be profiled to further improve performance and efficiency due to OpenMP overhead.

4 Experimental Results and Theoretical Discussions

In this section, we run the statically scheduled multithreaded SNPHAP with different parameter sets on an 8-core Intel Xeon E5405 CPUs 2.0 GHz with 4 GB 800-MHz

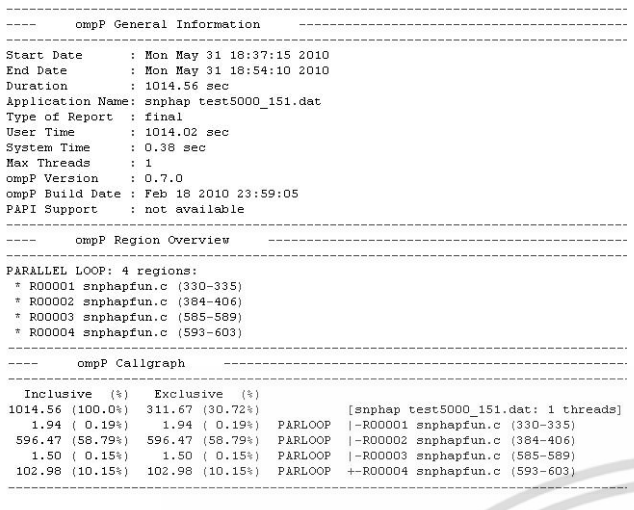


Figure 6: Profiling of OpenMP multithreaded SNPHAP by using ompP with a maximum of 1 thread to examine multithreaded (hotspot) functions, *hap_prior()* and *hap_posterior()*

DDR2 RAM. The operating system is Linux Fedora 9 kernel 2.6.25 SMP version. The SNPHAP 1.3.1 is compiled with GCC 4.4 and profiled with GNU GProf v.2.18.50. The OpenMP multithreaded SNPHAP is profiled with ompP v.0.7.1 [12].

4.1 Multithreading Performance with OpenMP

After the code is modified, we use GCC compiler 4.4 to incorporate the OpenMP library with *–openmp* flag. Experiment (simulation) genotype data set can be obtained from Snap [13], the open source tool for generating the genotype data, with parameters as shown in Table 1.

Table 1: Parameter set of the experiments

Parameters	Values
Maximum thread number	1, 2, 4, 6, 8, 12, 16, 24, 32
Number of genotypes data set	500, 1000, 2000, 5000, 10000
Number of loci in genotypes	21, 51, 101, 151

Table 2 to Table 5 show the averaged execution time of the experiment at different combinations of genotypes and loci. From a given set of parameters, the averaged execution time tends to grow up when the number of genotypes and the number of loci increase. To compare the results, the Speedup of execution times between sequential and multithreaded SNPHAP are plotted as shown in Figure 7.

Figure 7 shows the Speedups of multithreaded SNPHAP compared to sequential version at different numbers of loci: 21, 51, 101 and 151, respectively. It can be observed that the maximum Speedup of 2.14 can be achieved at larger number of loci, data set, and particularly at 8 threads. In addition, the Speedups tend to increase to the maximum values at 8 threads and decrease slightly later on. This behavior shows that the number of threads should match with the number of CPU cores to maximize the performance. Therefore, it can be concluded that the multithreaded SNPHAP program can be useful for practically large data sets with high number of loci.

It can also be observed that the Speedup of 1 thread compared with the sequential version in every experiment is less than 1.00 due to modification to the source code to support OpenMP and the overhead of OpenMP itself. As expected, the Speedup at 12 threads is even lower than those of 6 and 8 threads. This phenomenon shows the imbalance of multithreading due to number of running threads is multiple of number of physical cores. Moreover, the Speedups at 16, 24, and 32 threads are also lower that of 8 threads. This effects can be investigated further in the next subsection by the profiling tool named ompP [12].

4.2 OpenMP Profiling

To instrument with ompP, we simply add prefix *gcc -openmp* with *kinst-ompp*. We make the explicit initialization by placing *#pragma pomp inst init* at the beginning of *main()*. The profiles of each individual parallel region (construct) and the entire program are reported. According to Figure 4, we are interested in the parallel construct labelled R00002 corresponding to *hap_prior()* and consuming the biggest portion of total execution time. Next, we analyze the overhead of OpenMP that affects the total execution time. The profiled results of execution time and Speedup of multithreaded OpenMP (R00002 region) of 21, 51, 101, and 151 loci are plotted and compared as shown in Figure 8.

We can observe from the line graphs with Y-axis on the right hand side in Figure 8 that the highest Speedup in this parallel region R00002 is approaching to 8 times at 8 threads. Therefore, OpenMP can accelerate the execution by multithreading very well. As the number of threads equals to the number of CPU cores (8 cores), the Speedup is the highest. Thus, the multithreaded SNPHAP can be enhanced further as the number of physical cores/HyperThreads per chip is increased in modern generations of multi-core CPU.

Next, we analyze why the highest Speedup of multithreaded SNPHAP of 214% is at 8 threads rather than 16, 24, or 32 threads. The total overhead of OpenMP at 21, 51, 101 and 151 loci are shown as the percentage of total execution time in Figure 9. It is obvious that the %overheads increase as the number of threads is less than the number of CPU cores. However, the %overhead of 16 threads rises dramatically due to imbalance and management overhead. In general, we find that the %overhead of OpenMP increases along with the number of threads. Especially, the %overhead of 16 threads is much higher than that of 8 threads due to thread management and imbalance. For small number of loci (21 and 51 loci), %overhead is much higher than that of large number of loci (101 and 151 loci) because of the shorter execution time. As suspected, the

Table 2: Execution time (seconds) of sequential SNPHAP vs. multithreaded SNPHAP with 21 loci

♯genotypes	Execution Time	Execution Time (seconds) / ♯Threads								
		1	2	4	6	8	12	16	24	32
500	0.29	0.29	0.36	0.34	0.34	0.38	0.38	0.41	0.45	0.48
1000	0.44	0.46	0.53	0.52	0.51	0.59	0.54	0.54	0.55	0.58
2000	0.92	0.97	1.13	1.03	1.03	1.15	1.08	1.09	1.10	1.11
5000	2.48	2.66	3.00	2.72	2.80	2.99	2.84	2.75	2.76	2.85
10000	5.68	6.22	6.42	6.02	6.02	6.24	6.04	6.03	6.04	6.08

Table 3: Execution time (seconds) of sequential SNPHAP vs. multithreaded SNPHAP with 51 loci

♯genotypes	Execution Time	Execution Time (seconds) / ♯Threads								
		1	2	4	6	8	12	16	24	32
500	1.73	1.68	1.60	1.39	1.51	1.52	1.52	1.56	1.62	1.52
1000	3.81	3.91	3.82	3.24	3.40	3.38	3.44	3.43	3.45	3.38
2000	9.75	10.25	9.36	8.00	8.17	8.17	8.30	8.09	8.23	8.17
5000	30.10	32.72	28.04	23.47	23.86	23.81	23.89	23.79	23.76	23.81
10000	67.00	73.51	59.36	50.66	50.25	51.95	51.30	51.32	50.97	51.95

Table 4: Execution time (seconds) of sequential SNPHAP vs. multithreaded SNPHAP with 101 loci

♯genotypes	Execution Time	Execution Time (seconds) / ♯Threads								
		1	2	4	6	8	12	16	24	32
500	12.17	11.74	9.51	7.82	7.04	7.41	7.72	7.97	8.04	7.91
1000	33.18	35.03	27.64	21.03	19.61	19.63	20.59	20.46	20.13	19.97
2000	98.70	107.11	76.42	60.98	57.62	54.96	58.83	58.08	57.95	57.39
5000	366.06	397.30	276.93	219.12	197.92	193.92	205.98	199.55	199.53	199.00
10000	881.05	952.32	656.68	521.01	470.19	448.05	480.30	464.01	472.33	467.54

Table 5: Execution time (seconds) of sequential SNPHAP vs. multithreaded SNPHAP with 151 loci

♯genotypes	Execution Time	Execution Time (seconds) / ♯Threads								
		1	2	4	6	8	12	16	24	32
500	22.71	22.53	17.57	13.88	12.40	12.31	13.53	13.53	13.67	13.85
1000	70.18	72.75	54.08	39.64	36.71	35.37	39.37	38.85	37.32	37.14
2000	245.64	260.28	181.34	136.53	123.73	121.88	127.31	124.17	127.17	125.67
5000	973.15	1048.83	692.77	524.73	479.59	453.86	486.11	464.55	468.71	469.35
10000	2617.50	2778.21	1905.96	1421.66	1271.75	1221.10	1298.07	1244.60	1242.24	1253.27

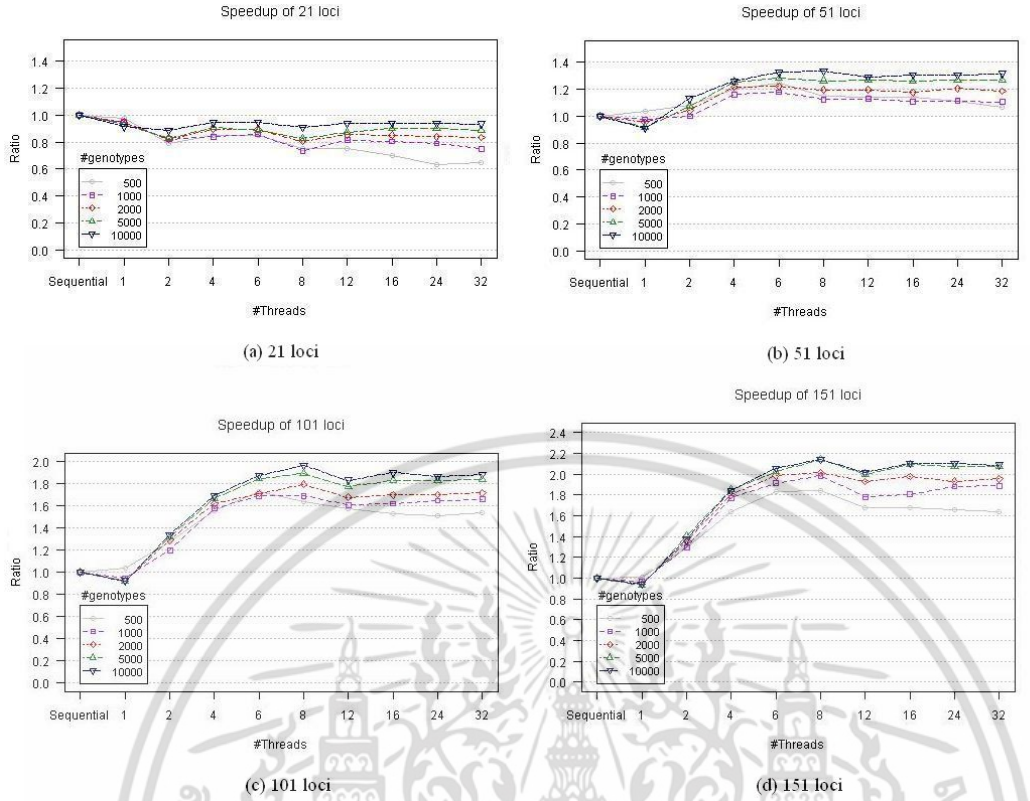
%overhead of OpenMP obtined from the ompP profiling can significantly affect the total execution time especially at small data sizes and large number of threads.

In summary, we make use of *parallel for* construct to distribute the work load equally at run time. Therefore, this can cause high management and imbalancing overhead at more than 1 thread/core. Furthermore, it is obvious that the %overhead of 8 threads/8 cores (1 thread/core) is much lower than that of 16 threads/8 cores (2 threads/core). Other parallel constructs of OpenMP could be applied to compare/minimize the %overhead in order to increase the overall efficiency.

5 Conclusions and Future Works

In this paper, we present the multithreaded version of SNPHAP, which is a haplotype inference bioinformatics program using Expectation Maximization algorithm. To speed up the SNPHAP program which has already been optimized for sequential execution, we study, profile, and modify SNPHAP to run in several threads on an 8-core Intel Xeon machine by applying OpenMP 3.0 library. Our framework can achieve a significant Speedup 214% on a practically large data set with 151 genotyping loci of a 10,000 population samples. Based on our ompP profiles of multithreaded SNPHAP, we also can conclude that the maximum speedup can be reached when the number of parallel threads equal to the number of physical cores.

For future works, different scheduling and synchronization schemes shall be applied to reduce the overhead and hence to increase speedup and efficiency. In addition, more experiments can be conducted on current generation of CPUs, such as AMD Phenom II x6, Intel i7 Core, and Intel Xeon 5600 with up to 6 cores, Intel Xeon 7000 with up to 8 physical cores per CPU while supporting up to 16



(a) 21 loci (b) 51 loci

(c) 101 loci (d) 151 loci

Figure 7: Speedup of sequential vs. multithreaded SNPHAP at (a) 21 loci (b) 51 loci (c) 101 loci and (d) 151 loci.

threads with Hyper-Thread technology. On the other hand, as the CPUs incorporates large Level-3 caches up to 24 MB compared to only 2x6MB L2 cache on Xeon 5405, cache behavior shall be investigated to speed up the execution and to support even larger population size.

References

[1] D. Fallin and N. J. Schork, "Accuracy of haplotype frequency estimation for biallelic loci, via the expectation-maximization algorithm for unphased diploid genotype data," *Am. J. Hum. Genet*, vol. 67, pp. 947 – 959, 2000.

[2] P. Bonizzoni, G. D. Vedova, R. Dondi, and J. Li, "The haplotyping problem: an overview of computational models and solutions," *of Computer Science and Technology (JCST)*, vol. 18, pp. 675 – 688, 2003.

[3] D. Gusfield and S. H. Orzack, *Haplotype Inference*, ser. CRC Handbook on Bioinformatics. Boca Raton, USA: CRC Press, 2005, ch. 1, pp. 1 – 25.

[4] X. S. Zhang, R. S. Wang, L. Y. Wu1, and L. Chen, "Models and algorithm for haplotyping problem," pp. 105 – 114, 2006.

[5] L. Excofier and M. Slatkin, "Maximum-likelihood estimation of molecular haplotype frequencies in a diploid population," *Mol.Bio.Evol*, vol. 12, no. 5, pp. 921 – 927, 1995.

[6] P. Huang and H. Chen, "Parallel algorithm for inferring haplotypes," Dept. of Computer Science, U.C. Berkeley, Tech. Rep., 2005.

[7] "Intel multi-core technology," http://www.intel.com/multi-core/.

[8] S. Akhter and J. Roberts, *Multi-core programming increasing performance through software multi-threading*. Hillsboro: Intel Press, 2006.

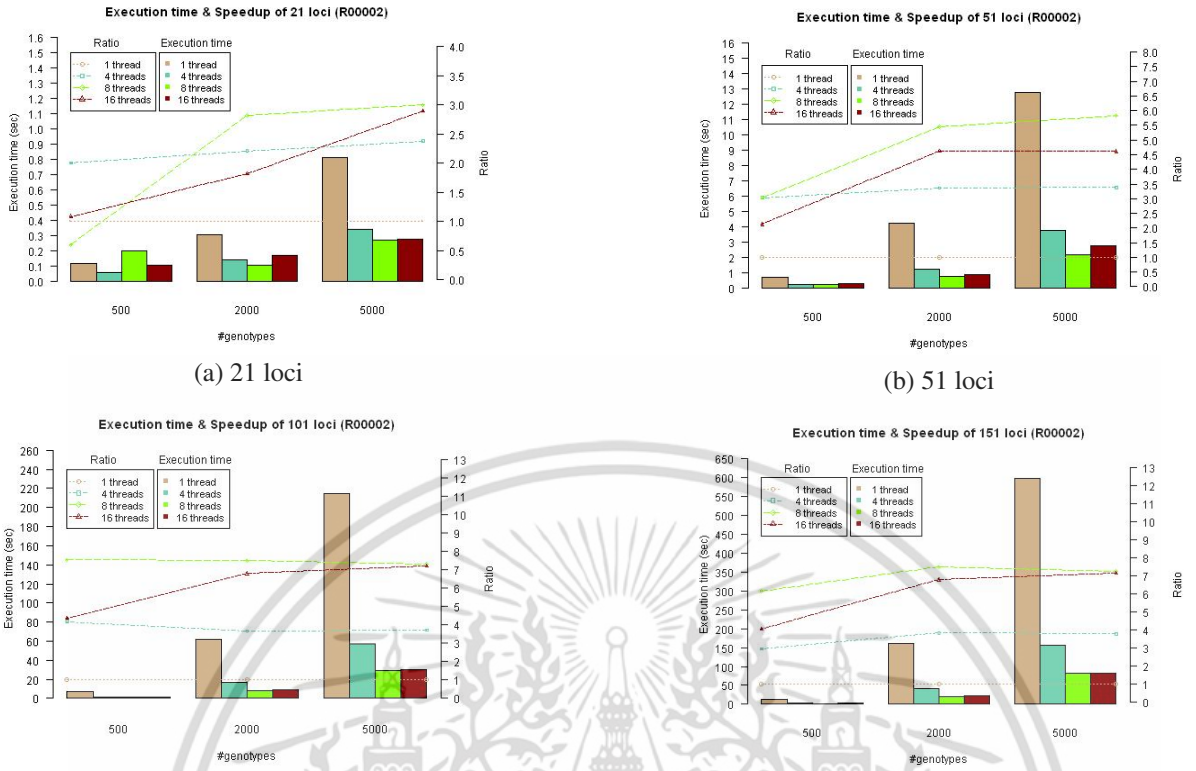
[9] D. Clayton, "Snphap (version 1.3.1)," http://www-gene.cimr.cam.ac.uk/clayton/software/, 2008.

[10] "The openmp api specification for parallel programming," http://www.openmp.org, 2010.

[11] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," in *SIGPLAN '82 Symposium on Compiler Construction*, ser. SIGPLAN Notices, vol. 17, no. 6, June 1982, pp. 120 – 126.

[12] K. Fuerlinger, "ompp profiler tool," http://www.cs.utk.edu/ karl/ompp.html, 2010.

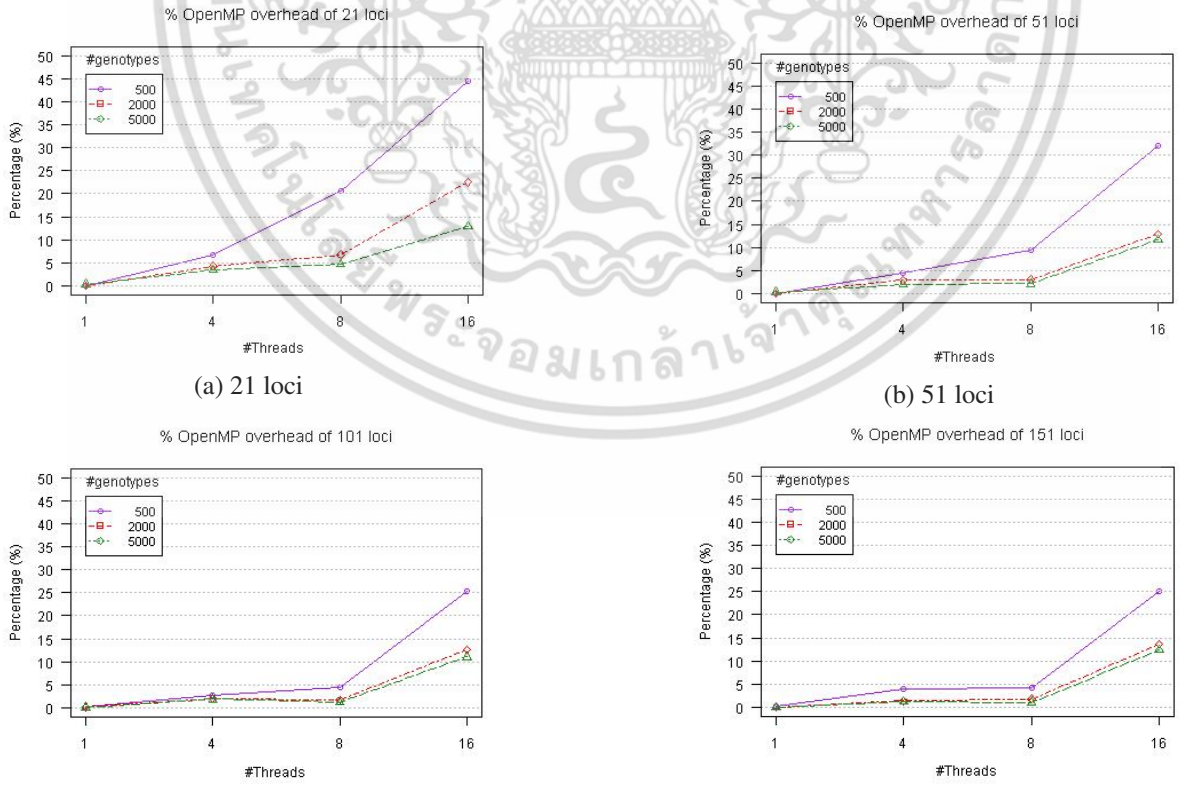
[13] M. Nothnagel, "Snap: simulation of snp haplotype data and phenotypic traits," http://capella.uni-kiel.de/snap/snap.htm, 2005.



(a) 21 loci (b) 51 loci

(c) 101 loci (d) 151 loci

Figure 8: Execution Time vs. Speedup of OpenMP overhead of Region R00002 at (a) 21 loci (b) 51 loci (c) 101 loci and (d) 151 loci.



(a) 21 loci (b) 51 loci

(c) 101 loci (d) 151 loci

Figure 9: Overhead Percentage (%) of Region R00002 at (a) 21 loci (b) 51 loci (c) 101 loci and (d) 151 loci.

A Multithreading Methodology with OpenMP on Multi-core CPUs: SNPHAP Case Study

Udom Ranok and Surin Kittitornkun
School of Computer Engineering, Faculty of Engineering, King Mongkut's Institute of Technology Lardkrabang
Bangkok 10520, Thailand
s100635@kmitl.ac.th, kksurin@kmitl.ac.th

Sissades Tongsima
Genome Institute, National Center for Genetic Engineering and Biotechnology
113 Thailand Science Park, Paholyothin Road, Klong 1, Klong Luang
Pathumtani 12120, Thailand
sissades@biotec.or.th

***Abstract*— This paper presents a multithreading methodology for OpenMP library. The methodology can be applied to convert existing sequential and demanding programs to be multithreaded programs with OpenMP running on the Multi-core CPUs. In our experiments, we apply this methodology to SNPHAP, which is one of the best haplotype inference bioinformatics program in terms of speed. The results show that our significant achievement is the maximum Speedup 316% for Intel Xeon E5405 (8-core 2.0 GHz) and 410% for Intel Xeon E5520 (8-Core with HyperThreading 2.66GHz) faster than its own sequential version.**

I. Introduction

Nowadays, major chip manufacturers are producing multi-core CPUs with/without Hyper-Threading. Therefore, several demanding programs can be multithreaded to gain higher performance with less expensive hardware and better efficiency [1]. The recent parallel language standard for shared memory multiprocessor system (SMP), OpenMP 3.0 is now the most well known library that can be applied successfully to develop parallel programs running on multi-core CPU architecture.

The difficultly of multithreading may depend on the characteristics of each application. Park, et al. [2] create a parallel FORTRAN programming environment using OpenMP. However, they focus on general program development and also create a new tool for supporting each step in the methodology. The methodology is not flexible for other free tools and open source tools. Moreover, it does not support implementation details to get the highest performance, especially, demanding applications such as bioinformatics.

Our proposed methodology can be applied to convert existing sequential and demanding programs to be multithreaded programs with OpenMP. In this paper, we apply it to SNPHAP [3], which is a haplotype inference bioinformatics program using EM algorithm. Run time complexity of SNPHAP is exponential [4][5] due to the large number of possible haplotype instances.

The remainder of the paper is organized as follows: Section 2 briefly reviews the related background and previous work to this paper. The multihtreading methodology is presented in Section 3. Then, Section 4 discusses the experimental setup and results. Finally, Section 5 presents the conclusion of this work and some directions for the future work.

II. Literature Review

A. OpenMP libraries

OpenMP library [6] is an application program interface (API) for thread based parallelism on shared memory multi-core processors. The API consists of a set of compiler directives, library routines, and environmental variables that support FORTRAN and C/C++ on multiple architectures. For programmers, OpenMP provides a portable, scalable model of thread based parallelism application. The model of multithreading execution is the fork-join model. The Master thread initially starts the running program. After the program is executed for a while, the Worker threads are spawned (forked) at a parallel region to form a team together with the Master thread. At the end of each parallel region, all threads are synchronized (joined) before the Master thread can continue further. A parallel region can also nest with other parallel regions. The main advantage of using OpenMP is the ability of all CPU cores to share and access the same memory pool (data) with less communication overhead and network latency compared with other parallel computing paradigms such as cluster computing, grid computing, etc.

B. OpenMP Parallelization Methods

In [2], Park, et al. create a parallel programming environment. They focus on general FORTRAN program development and also create a new tool for supporting each step in the methodology. It is not flexible for other free and open source tools. Programmer shall choose the suitable tools according to the characteristics of each parallel application. Moreover, it does not support deep implementation in details to get the highest performance, especially, for demanding applications such as bioinformatics.

C. SNPHAP: Bioinformatics Application

Haplotype analyses are tools for studying human genetics and also used to investigate many population processes, such as migration and immigration rates, linkage disequilibrium strength, and the related populations [7]. There exist many methods for inferring haplotypes from unphased genotype

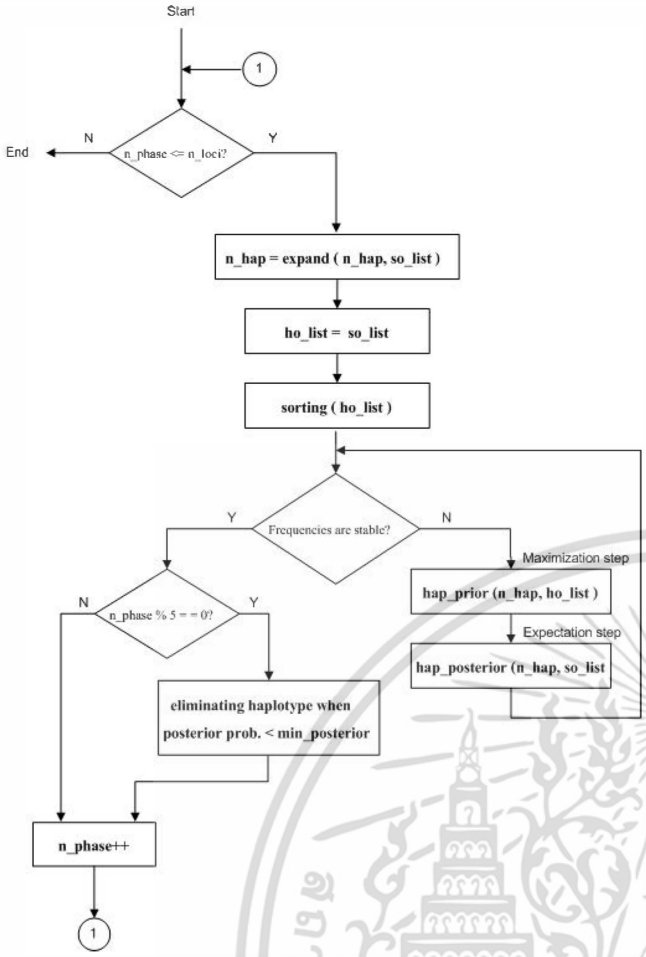


Fig. 1. Main flowchart of SNPHAP [3]

data. One of the most widely used methods is to statistically estimate haplotype frequencies from maker genotype in unrelated individuals with EM algorithm to overcome the missing or incomplete data [8]. This problem can take exponential computation time [4][5] caused by the large number of possible haplotype instances. SNPHAP [3] is the chosen bioinformatics application for our work. EM algorithm is a general technique for finding maximum likelihood estimates (MLEs) of parameter on incomplete data. The procedures consist of (E)xpectation step (*hap_posterior*) and (M)aximization step (*hap_prior*)as shown in Figure 1.

Eronen, et al. compare run time per genotype (seconds) of SNPHAP with other haplotyping software in[9]. From Figure 2, it can be seen that run time per genotype of SNPHAP is the best. This makes parallelization of SNPHAP even more challenging.

In our previous work [10], we have presented the multithreaded version of SNPHAP with OpenMP library on multi-core CPUs. Our method is the use of OpenMP library directives to perform multithreading at functions consuming most execution time according to the profiler. We have found that *cmp_hap* function inside *hap_prior* and *hap_posterior* functions should be parallelized with OpenMP parallel For

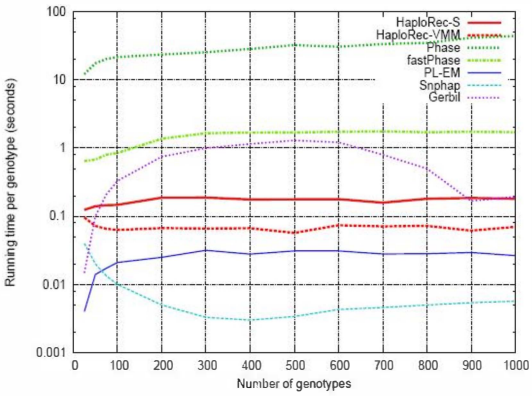


Fig. 2. Run time per genotype (Seconds) of various haplotyping software (Loci = 30) [9]

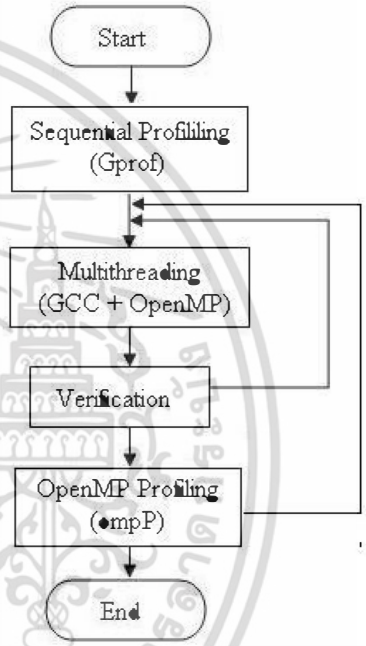


Fig. 3. Flowchart of the proposed multithreading methodology with OpenMP

construct. As such, we can run multithreaded SNPHAP up to 214% faster than the sequential version.

III. Our Multithreading Methodology

Our multithreading methodology consists of four major steps as shown in Figure 3, Sequential Profiling, Multithreading, Verification, and OpenMP Profiling. We have applied this methodology with SNPHAP implementation in details. It can also be applied with other applications easily. Each step will be explained in the following subsections.

A. Sequential Profiling: GProf

First of all, the sequential program, SNPHAP, is profiled to find all hotspot functions that consume significant amount of time. The profiling tool is the Gprof profiler tool [11]. Then, the OpenMP compiler directives are inserted on those

```
static void sort(char *array, size_t size, int (*cmp)(const void*,const void*), int begin, int end) {
    if(begin == end) return;
    int i=begin;
    int j=end,p;
    void *pivot= array + begin;
    do {
        while ( cmp(array+i,pivot) < 0) i+=size;
        while ( cmp(array+j,pivot) > 0) j-=size;
        if (i<=j) {
                swap(array+i,array+j,size);                    // Pivot partition section
                i+=size;
                j-=size;
        }
    } while (i<=j);
    if (begin < j){
      #pragma omp task
        sort(array,size,cmp,begin,j);
    }                                                          // Recursive section
    if (i < end) {
      #pragma omp task
        sort(array,size,cmp,i,end);
    }
}
void qsort1(void *array, size_t nitems, size_t size, int (*cmp)(const void*,const void*)) {
    #pragma omp parallel
    #pragma omp single
    {                                                          // Call sort() section
      sort(array, size, cmp, 0, (nitems-1)*size);
    }
}
```

Fig. 4. Multithreading of qsort1() function with OpenMP parallel Task construct

```
------------------------------------------------------------------------
----      ompP Region Overview      ------------------------------------
------------------------------------------------------------------------
PARALLEL: 1 region:
 * R00003 snphapfun.c (81-85)

PARALLEL LOOP: 4 regions:
 * R00005 snphapfun.c (404-409)
 * R00006 snphapfun.c (414-433)
 * R00007 snphapfun.c (595-599)
 * R00008 snphapfun.c (603-615)

SINGLE: 1 region:
 * R00004 snphapfun.c (82-85)

TASK: 2 regions:
 * R00001 snphapfun.c (69-70)
 * R00002 snphapfun.c (73-74)

TASKEXEC: 2 regions:
 * R00001 snphapfun.c (69-70)
 * R00002 snphapfun.c (73-74)

------------------------------------------------------------------------
----      ompP Callgraph      ------------------------------------------
------------------------------------------------------------------------

  Inclusive   (%)    Exclusive   (%)
   2.21 (100.0%)     0.63 (28.47%)           [snphap: 32 threads]
   0.56 (25.23%)     0.02 ( 1.01%) PARALLEL  |-R00003 snphapfun.c (81-85)
   0.54 (24.22%)     0.54 (24.22%)   SINGLE  |  +-R00004 snphapfun.c (82-85)
   0.00 (0.001%)     0.00 (0.001%)     TASK  |    |-R00001 snphapfun.c (69-70)
   0.09 ( 4.17%)     0.03 ( 1.46%) TASKEXEC  |    |-R00001 snphapfun.c (69-70)
   0.03 ( 1.49%)     0.03 ( 1.49%)     TASK  |    |  |-R00001 snphapfun.c (69-70)
   0.03 ( 1.22%)     0.03 ( 1.22%)     TASK  |    |  +-R00002 snphapfun.c (73-74)
   0.10 ( 4.54%)     0.04 ( 1.80%) TASKEXEC  |    |-R00002 snphapfun.c (73-74)
   0.03 ( 1.36%)     0.03 ( 1.36%)     TASK  |    |  |-R00001 snphapfun.c (69-70)
   0.03 ( 1.36%)     0.03 ( 1.36%)     TASK  |    |  +-R00002 snphapfun.c (73-74)
   0.00 (0.000%)     0.00 (0.000%)     TASK  |    +-R00002 snphapfun.c (73-74)
   0.20 ( 9.13%)     0.20 ( 9.13%)  PARLOOP  |-R00005 snphapfun.c (404-409)
   0.32 (14.55%)     0.32 (14.55%)  PARLOOP  |-R00006 snphapfun.c (414-433)
   0.22 (10.06%)     0.22 (10.06%)  PARLOOP  |-R00007 snphapfun.c (595-599)
   0.28 (12.56%)     0.28 (12.56%)  PARLOOP  +-R00008 snphapfun.c (603-615)

------------------------------------------------------------------------
```

Fig. 5. Profiling result of multithreaded SNPHAP by ompP with 32 threads to examine % run time of *hap_prior()* and *hap_posterior()*, and *qsort1()*

functions to convert them as threads just like Fork-Join model.

B. Multithreading with OpenMP 3.0

In our previous work [10], we have parallelized the execution of *cmp_hap* inside *hap_prior()* and *hap_posterior()*, respectively. From Sequential Profiling step, we have found that the sorting function (qsort()) in Figure 1 calling *cmp_hap* function consumes the most execution time as well. Therefore, it is our target. Unfortunately, qsort() is defined in a standard library and therefore not easy to parallelize. Then, we have to develop a new modified qsort1() with OpenMP as shown in Figure 4.

From Figure 4, it can be seen the qsort1() makes use of OMP Single construct to call sort() function. Because sort() function is actually a quciksort algorithm with recursions. Two OMP Tasks constructs are applied to define an explicit parallel task (thread). The execution model of this mulithreaded qsort1() can be described as a single producer, multiple consumer model. The master thread executing the Single region generates (forks) working tasks (threads). All the working tasks (threads) are guaranteed to complete by the time that all (working) threads exit the Single region (join). When a thread finishes executing a task, it grabs a new task to execute. In this way, all threads can execute available tasks without explicit barrier synchronization, thereby improving load balancing.

C. Verification

The multithreaded SNPHAP results must be verified again and again until the results are similar to those of sequential version as possible. A utility program named Beyond Compare matches and differentiates both output files from both multithreaded and sequential SNPHAP, respectively.

D. OpenMP Profiling: ompP

The OpenMP parallel for and Task constructs can be analyzed by ompP [12] to investigate its multithreading characteristics. We can use the results to analyze the bottleneck and thus increase the performance gain. Our profiling result of mutithreaded SNPHAP shows % run time of each parallel For and Task constructs as shown in Figure 5.

IV. Experiment Results and Discussions

A. Experiment Set Up

In this section, we run the multithreaded SNPHAP with different parameter sets on two Linux systems: an 8-core Intel Xeon E5405 CPUs 2.0 GHz with 4 GB 800-MHz DDR2 RAM and an 8-core Intel Xeon E5520 CPUs (Hyper-Threading) 2.66 GHz with 6 GB 1066-MHz DDR3 RAM. The operating system is Linux Fedora 9 kernel 2.6.25 SMP version. The modified SNPHAP 1.3.1 is compiled with GCC 4.4 together with OpenMP 3.0 library (only OpenMP and no other compiler optimizations) and profiled with GNU GProf v.2.18.50. The OpenMP constructs are profiled with ompP v.0.7.0. The parameters for this experiment are shown in Table I.

B. Runtime vs. Runtime per Genotype

In this section, we would like to show the exponential run time of sequential SNPHAP before applying the proposed methodology.

TABLE I

PARAMETER SET OF THE EXPERIMENTS

Parameters	Values
Maximum thread number	2, 4, 8, 12, 16, 32
Number of genotypes	500, 1000, 2000, 5000, 10000
Number of loci in genotypes	21, 51, 101, 151

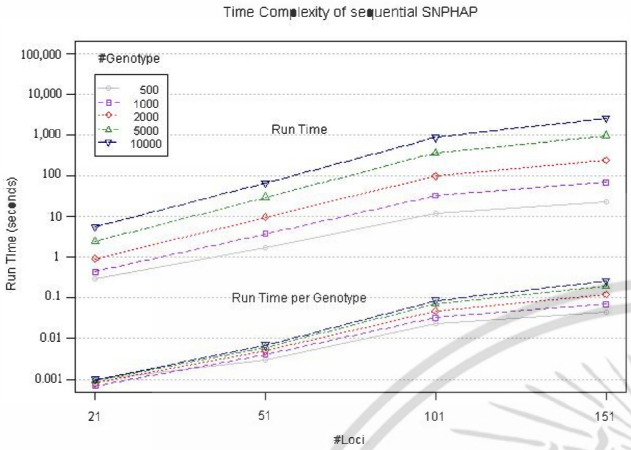


Fig. 6. Run Time and Run Time per Genotype of sequential SNPHAP on 8-Core Intel Xeon E5405

Figure 6 shows Run time and Run time per Genotype in Seconds. From a subset set of parameters in Table I, Run Time and Run Time per Genotype of multithreaded SNPHAP on Intel Xeon E5405 and Intel Xeon E5520 machines are plotted as shown in Figure 7 and Figure 8, respectively.

Here are the observations in general. Intel Xeon E5520 can run multithread SNPHAP faster than E5405 due to higher clock frequencies (2.66 GHz vs. 2.0 Ghz), larger and deeper cache per 4 cores (8-MB L3 vs. 2x4-MB L2), and better memory technology (DDR3 vs. DDR2).

C. Speed Up

To simply understand and compare the results, the Speedup (ratio) of execution times between sequential and multithreaded SNPHAP of Intel Xeon E5405 (without HyperThreading) and Intel Xeon E5520 (with HyperThreading) of 10,000 genotypes and 151 loci are plotted as shown in Figure 9.

Intel Xeon E5520 can run multithread SNPHAP and achieve higher Speedups than E5405 with the same parameter set. Note that Speedups of each CPU are compared with its own sequential runtime. Both CPUs gain higher Speedup when the number of threads increases. The Speedup of E5405 is the maximum of 3.16 at 8 threads and almost constant later on. On the other hand, Speedup of E5520 tends to peak at 8 threads and slow down at 16 threads. It turns out that the maximum Speedup of 4.10 can be achieved at and particularly at 32 threads. That means E5520 is more scalable than E5405.

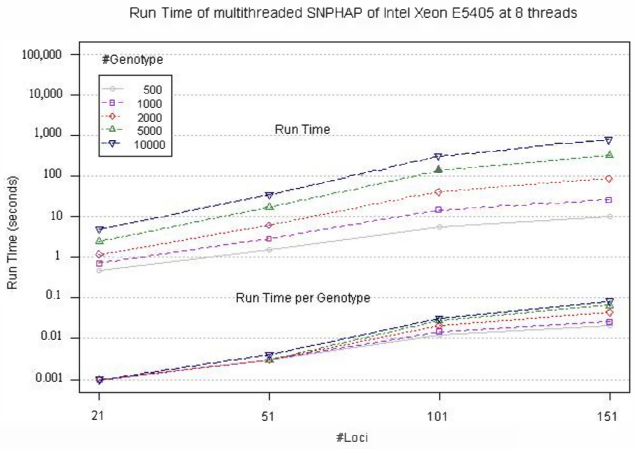


Fig. 7. Run Time and Run Time per Genotype (seconds) of 8-thread SNPHAP on 8-Core Intel Xeon E5405 (OpenMP only, No other optimizations)

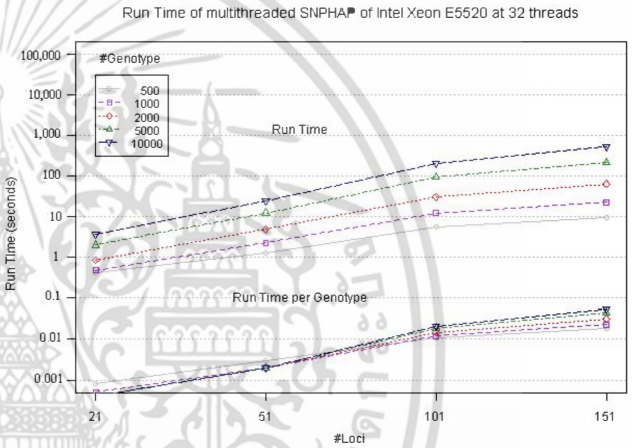


Fig. 8. Run Time and Run Time per Genotype (seconds) of 32-thread SNPHAP on 8-Core Intel Xeon E5520 (OpenMP only, No other optimizations)

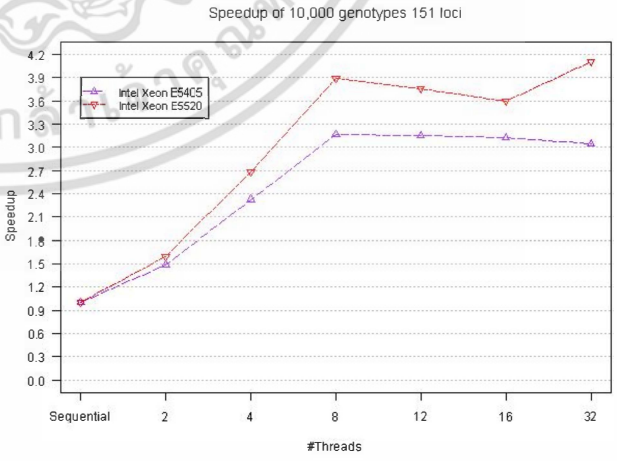


Fig. 9. Speedup Comparison of Multihreaded SNPHAP on 8-Core Intel Xeon E5405 vs. 8-Core Intel Xeon E5520 at 10,000 genotypes & 151 loci (OpenMP only, No other optimizations)

V. Conclusions and Future work

The sequential SNPHAP can be multithreaded by OpenMP library [6] so that it can be executed on multi-core CPUs. We experiment on Intel Xeon E5405 and Intel Xeon E5520 Linux machines. Our achievement is significant in terms of Speedup at up to 316% and 410%, respectively faster than its own sequential version. We shall apply this methodology to other bioinformatics benchmark programs.

References

[1] S. Akhter and J. Roberts, *Multi-core programming increasing performance through software multi-threading*. Hillsboro: Intel Press, 2006.
[2] I. Park, M. J. Voss, S. W. Kim, and R. Eigenmann, "Parallel programming environment for openmp," in *High Performance Computing in Science and Engineering*. Heidelberg: Springer, 2002, pp. 111 – 125.
[3] D. Clayton, "Snphap (version 1.3.1)," http://www-gene.cimr.cam.ac.uk/clayton/software/, 2008.
[4] X. S. Zhang, R. S. Wang, L. Y. Wu1, and L. Chen, "Models and algorithm for haplotyping problem," *Current Bioinformatics*, vol. 1, pp. 105 – 114, 2006.
[5] L. Excofier and M. Slatkin, "Maximum-likelihood estimation of molecular haplotype frequencies in a diploid population," *Mol.Bio.Evol*, vol. 12, no. 5, pp. 921 – 927, 1995.
[6] A. R. Board, "The openmp api specification for parallel programming," http://www.openmp.org, 2010.
[7] D. Fallin and N. J. Schork, "Accuracy of haplotype frequency estimation for biallelic loci, via the expectation-maximization algorithm for unphased diploid genotype data," *Am. J. Hum. Genet*, vol. 67, pp. 947 – 959, 2000.
[8] P. Bonizzoni, G. D. Vedova, R. Dondi, and J. Li, "The haplotyping problem: an overview of computational models and solutions," *Journal of Computer Science and Technology (JCST)*, vol. 18, pp. 675 – 688, 2003.
[9] L. Eronen, F. Geerts, and H. Toivonen1, "Haplorec: efficient and accurate large-scale reconstruction of haplotypes," *BMC Bioinformatics*, vol. 7, p. 542, 2006.
[10] U. Ranok, S. Kittitornkun, and S. Tongsima, "Multithreading bioinformatics software with openmp: Snphap case study," in *Proceeding of the IASTED International Conference Parallel and Distributed Computing and Systems (PDCS2010)*, Marina Del Rey, USA, 2010.
[11] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," in *SIGPLAN '82 Symposium on Compiler Construction*, ser. SIGPLAN Notices, vol. 17, no. 6, June 1982, pp. 120 – 126.
[12] K. Fuerlinger and M. Gerndt, "ompp: A profiling tool for openmp," in *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, May 2005.

CERTIFICATE OF BEST PAPER AWARD

TO

Ratthaslip Ranokphanuwat, Surin Kittitornkun and Sissades Tongsima

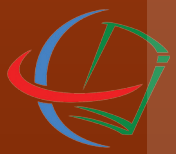
FOR TITLE

Performance analysis & improvement of SNPHAP on Multi-core CPUs

Presented at The 2013 10th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON 2013)
On May 15-17, 2013 in Krabi, Thailand

(Prof. Dr. Prabhas Chongstitvatana)
General Chair

(Prof. Dr. Prayoot Akkaraekthalin)
Technical Program Chair



2013 10th International Conference on
Electrical Engineering/Electronics, Computer,
Telecommunications and Information Technology

ECTI-CON 2013

Krabi, Thailand
May 15-17, 2013

ISBN : 978-1-4799-0545-4
IEEE catalog number : CFP1306E-ART

IEEE
IEEE
THAILAND SECTION

Performance analysis & improvement of SNPHAP on Multi-core CPUs

Ratthaslip Ranokphanuwat
and Surin Kittitornkun
Department of Computer Engineering, Faculty of Engineering
King Mongkut's Institute of Technology Lardkrabang
Bangkok 10520, Thailand
Email: udom.ranok@gmail.com,
kksurin@kmitl.ac.th

Sissades Tongsima
Genome Institute, National Center for Genetic
Engineering and Biotechnology
113 Thailand Science Park, Paholyothin Road,
Klong 1, Klong Luang
Pathumtani 12120, Thailand
Email: sissades@biotec.or.th

***Abstract*—In this paper, we attempt to analyse this highly computational problem by parallelizing a haplotype inference algorithm, called SNPHAP. The analysis is based on both the orignal (sequential) algorithm and its corresponding run time complexity in Big-O notations. Then, we improve its performance using OpenMP 3.0 and test on a 4-core Intel Core i7-2600 (Hyper-Threading), an 8-core Intel XeonE5405, an 8-core Intel Xeon E5520 (Hyper-Threading) and a 32-core AMD Opteron 8356 Linux machines. The achievements in terms of maximum speedups are 260%, 316%, 410% and 488%, respectively. The factors that affect the speedup of SNPHAP are the specific parallelized code fraction, the suitable OpenMP constructs, the number of physical cores, the sizes of cache memories within/among CPU cores, the clock frequency and finally the memory technology.**

I. Introduction

A haplotype provides a snapshot of human evolution to estimate the age and location of disease mutations related to a set of multiple linked markers and to investigate many population processes [1]. In other words, an inferred haplotype is not a direct observation but can be obtained from unphased genotype data (resulting in haplotype ambiguity) through experiment in software techniques. There exist many methods for inferring haplotypes from unphased genotype data. One of the most widely used methods is to statistically estimate haplotype frequencies from genotyping markers of a group of unrelated individuals by employing the Expectation Maximization (EM) algorithm to predict the missing or incomplete data [2][3]. This problem can take longer computation time caused by a large number of possible haplotype instances are widely needed. Therefore, if the method can be extended into parallel algorithms, the throughput may be increased.

In this paper, we analyse this highly computational problem by parallelizing a haplotype inference algorithm, called SNPHAP. The analysis is based on both the orignal (sequential) algorithm and its corresponding run time complexity in Big-O notations. Then, we improve its performance using OpenMP 3.0 and test on a 4-core Intel Core i7-2600 (Hyper-Threading), an 8-core Intel XeonE5405, an 8-core Intel Xeon E5520 (Hyper-Threading) and a 32-core AMD Opteron 8356 Linux machines. The achievements in terms of maximum speedups are 260%, 316%, 410% and 488%, respectively.

978-1-4799-0545-4/13/$31.00 ©2013 IEEE

Due to limited amount of space, the contribution of this paper will mainly focus on the analyses of experimental results with Big-O notations. The remainder of the paper is organized as follows: Section II reviews SNPHAP and OMP SNPHAP that we have done. The experiments and results are demonstrated in Section III. Then, Section IV extensively discusses the analyses of experimental results. Finally, this paper is concluded in Section V.

II. Literature Review

A. SNPHAP

SNPHAP is an EM-based program for inferring haplotype frequencies using genotype data from a number of unrelated individuals. The EM algorithm is an iterative method [3] which is a general technique for finding maximum likelihood (ML) estimation of haplotype frequencies. The overall procedure comprises (1) Expectation step (*hap_posterior()* inside SNPHAP), (2) Maximization step (*hap_prior()* inside SNPHAP) and repeat (1)(2) until the haplotype frequencies are stable. Note that we use the prefixes "SEQ" and "OMP", standing for sequential and OpenMP, to identify the original and our parallelized versions of SNPHAP. The aforementioned steps can be summarized in Algorithm 1 that we derived for Big-O analysis in the subsequent sections.

Eronen et al. [4], compared running time of the SNPHAP with other haplotyping tool when varying the number of loci as shown in Fig. 1.

The compared tools include GERBIL [5] (version 1.0), PL-EM [6] (version 1.5, kindly provided by Zhaohui S. Qin), fastPhase [7] (version 1.1.3), Phase [8], [9] (version 2.0), and HaploRec (version 2.3) [4]. From this report, the SNPHAP is the fastest tool among others. Although the SNPHAP has the strong point of runtime, it is still the sequential (single thread) version. The runtime of SHPHAP can be further reduced with multithreading technique. On the other hand , the SNPHAP is not suitable for Clustering because a little bit of time of inferring haplotype in each EM iteration is overcome by communication time between Cluster nodes. The SNPHAP, moreover, is *communication bound* so that multithreading technique on Multi-core CPUs is the best solution.

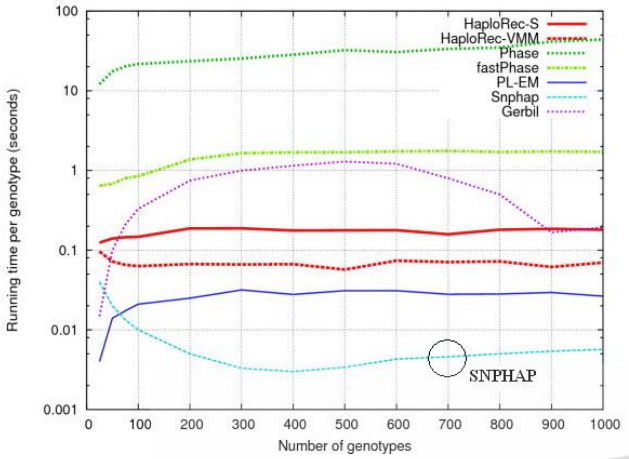


Fig. 1: Runtime (seconds) per genotype of various haplotyping software and SNPHAP is the fastest (*Loci* = 30).

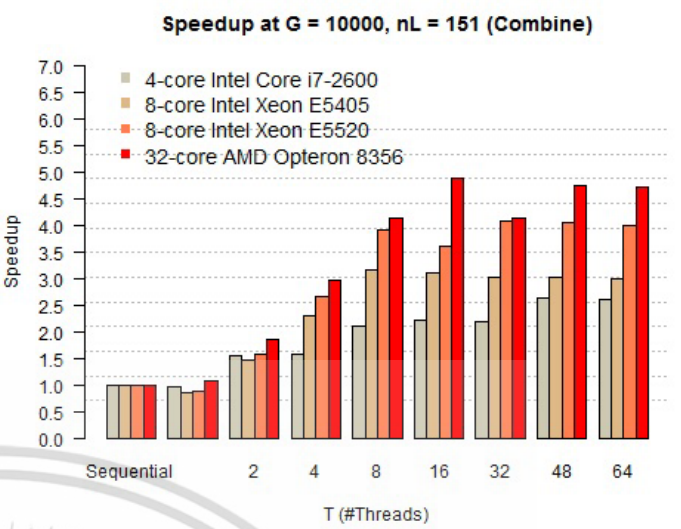


Fig. 2: Speedup comparison of OMP SNPHAP for Combined Parallelization *(parallel For + Task)* on Intel Core i7-2600, Intel Xeon E5405, Intel Xeon E5520 and AMD Opteron 8356 at G = 10,000 and *nL* = 151 (OpenMP only, No other compiler options).

B. Parallel SNPHAP

To parallelize the SEQ SNPHAP program to run on Multi-core CPUs, multiple threads must be generated. OpenMP library offers special instructions, called OpenMP constructs, that can initiate such threads and when executing these threads can be run in parallel on Multi-core CPU. However, not all modules/functions in a sequential program can be converted to multithreaded functions.

Our method consists of four major steps: 1) Profiling (Gprof), 2) Multithreading (GCC + OpenMP), 3) Verification, and 4) OpenMP Profiling (omPP). These steps have been introduced in our previous work [10] that we have parallelized on *cmp_hap()* inside *hap_prior()* and *hap_posterior()* by using OpenMP *parallel For* construct. While latest work [11], we have extended the OpenMP version of SNPHAP from [10] by parallelizing on *qsort()* with $Task$ construct.

III. Experiments and Results

In this section, we present the experimental settings and the speedup of the OMP SNPHAP running under various multithreaded configurations.

A. Experimental Settings

The results reported in this paper are based on experimental measurements on four Multi-core CPUs systems: Intel Core i7-2600 (Hyper-Threading), Intel Xeon E5405 Harpertown, Intel Xeon E5520 Nehalem-EP (Hyper-Threading) and AMD Opteron 8356 Barcelona located at National Center for Genetic Engineering and Biotechnology (BIOTEC), Thailand. The operating systems are Linux Ubuntu 10.04 kernel 2.6.32 SMP version for Intel Core i7-2600, Linux Fedora 9 kernel 2.6.25 SMP version for both Intel E5405 and Intel E5520 and Linux CentOS 5.2 kernel 2.6.18 SMP version for AMD Opteron. The modified SNPHAP 1.3.1 is compiled with GCC 4.4 together with OpenMP 3.0 library (only OpenMP and no other compiler optimizations) and profiled with GNU GProf v.2.18.50.

Table I provides a summary of our evaluation Multi-core systems. We have interested in the multithreading performance of particular characteristics of these systems. Firstly, we consider both one-socket quad-core (Hyper-Threading), dual-socket quad-core (8 cores), dual-socket quad-core (Hyper-Threading) and octal-socket quad-core (32 cores) systems. We study how the OMP SNPHAP scales by each OpenMP construct to get the highest performance. Secondly, the cache capacity and cache sharing configurations of these systems will be investigated on how it effect to the performance. The L3 cache capacity shared among on a socket ranges from 2MB (AMD Opteron 8356) to 8 MB (Intel E5520 and Intel Core i7-2600). Moreover, the Intel E5405 system has two L2 caches as 6MB on a socket (chip), each which is shared among two cores. The experiment (simulation) genotype data set (*G*) can be obtained from SNAP [12], the open source tool for generating the genotype data, with parameters as shown in Table II.

B. Speedup

From a given set of parameters in Table II, the speedups for Combined Parallelization $(parallelFor + Task)$ construct are presented and described in the following.

The experiment combines both For and $Task$ constructs to improve the performance. Two factors leading to poor performance are OpenMP overhead costs and load imbalance. However, as we experiment on the larger *G* at larger *nL*, the OMP SNPHAP version scales much better than SEQ SNPHAP. Even when only a single thread is used, there are some overhead costs incurred by OpenMP and modification of the source code to support OpenMP.

From Figure 2, the speedups of all CPUs tend to grow dramatically when *T* increases (peak at *T* = 8 to 48) and decreases slightly later on. We can see that the speedups scale

TABLE I: Architectural details of Multi-core systems used in our study. †shared among cores. ‡shared between 2 cores.

System	Intel Core i7-2600	Intel E5405	Intel E5520	AMD Opteron 8356
Core Architecture	Sandy Bridge	Harpertown	Nehalem-EP	Barcelona
Socket x cores x threads	1 x 4 x 2	2 x 4 x 1	2 x 4 x 2	8 x 4 x 1
#Threads	8	8	16	32
Clock (GHz)	3.4	2	2.66	2.3
L1/L2/L3 cache (KB) per core	32/256/8192†	32/6144‡/-	32/256/8192†	64/512/2048†
DRAM Capacity (GB)	16	4	12	16
Bandwidth (GB/s)	21	21.33	51.2	21.33

TABLE II: Parameter set of the experiments

Parameters	Values
Maximum thread number (T)	1, 2, 4, 8, 16, 32, 48, 64
Number of genotypes (G)	500, 1000, 2000, 5000, 10,000
Number of loci in genotypes (nL)	51, 101, 151

well as T and the physical cores C are available except Intel Core i7-2600.

We can also observe that the speedup of Intel Core i7-2600 tends to peak at 2.6 at T = 48. While of Intel Xeon E5405, the maximum speedup is 3.16 at T = 8 and slightly lower and almost constant later on. For Intel Xeon E5520, the maximum speedup is 4.10 at T = 32. Moreover, the speedup tends to peak at T = 8 and slow down at T = 16. It turns out at T = 32 and decreases slightly almost later on. This effect is due to the For construct. On the other hand, the speedup of AMD opteron 8356 tends to peak at 4.88 at T = 16 and slow down at T = 32. It turns out at T = 48 and decreases slightly later on. This effect is due to $Task$ construct.

Moreover, the speedups grow up along with problem size *(G + nL)*. Especially, at the largest *nL* (151 loci) and *G* (10,000 genotypes) all systems show their highest speedups. The small problem sizes lead to speedup loss due to multithreading overhead. For larger problem sizes, the overhead reduces significantly. In particular, the main source of speedup loss is the memory overhead (memory contention and cache misses), proportional to the number of cores and the problem size. In our previous work [10], we found that each *For* construct can be parallelized and sped up in accordance with *T* threads forked by OpenMP. Furthermore, the %overhead of OpenMP can significantly affect the total execution time especially at small data sizes *(G + nL)* and large *T* (%overhead of 1 thread/core is much lower than that of 2 threads/core).

In general, AMD Opteron 8356 can run OMP SNPHAP faster than Intel Core i7-2600, Intel Xeon E5520 and Intel Xeon E5405 due to higher number of CPU cores (32 vs. 4 vs. 8 vs. 8, respectively). Moreover, Intel Xeon E5520 can run OMP SNPHAP faster than Intel Xeon E5405 because of higher clock frequencies (2.66 GHz vs. 2.0 GHz), larger and deeper cache per 4 cores (8-MB L3 vs. 2x3-MB L2), much higher memory bandwidth, and better memory technology (DDR3 vs. DDR2) and Hyper-Threading technology (2 threads/core vs. 1 thread/core). Although the Intel Xeon E5520 shows better architectural features than AMD Opteron 8356 except for the number of CPU cores, the highest speedup is still lower than that of AMD Opteron 8356. Therefore, more physical CPU cores of AMD Opteron 8356 can gain the highest speedup. In spite of having more physical CPU of AMD Opteron 8356, its speedup/core is still less than Intel Xeon E5520. The reasons behind this effect are L3 cache size and inefficient communication links (QPI) between cores.

IV. Analyses and Discussions

The factors affecting the speedup of SNPHAP are the specific parallelized code fraction, the suitable OpenMP constructs, the number of physical cores, the sizes of cache memories within/among CPU cores, the clock frequency and finally, the memory technology. They are analysed in the following subsections.

A. The specific parallelized code fraction

In parallel computing, Amdahl's law [13] is used to predict the theoretical speedup using multiple processors. For the parallelization on Multi-core systems, the speedup $S(C)$ that can be achieved by using C processor cores is given by Equation 8.

$$S(C) = \frac{1}{(1-f) + \frac{f}{C} + o(C)}, \tag{8}$$

where f is the fraction of a program that can be parallelized and *o(C)* = overhead of C processor cores. This overhead consists of two portions: operating system overhead and threading activities including synchronization, load imbalance, memory contention, cache miss, and also OpenMP overhead (this overhead can be explored by OpenMP profiling step which have introduced in [10]). We can say that it will be a strong scaling when the data size remains constant and weak scaling when the data size is proportional to *C* (runtime remains constant).

In practice, if C is fixed and then the maximum speedup can be limited by the fraction $(1-f)$ and the overhead $o(C)$. In this context, the numerical value of fraction f determines the theoretical maximum speedup. As referred to the results of GProf profiling [14] in Fig. 3, the fraction *f* is the *cmp_hap()* which called by *hap_prior()* and *hap_posterior()* as 89.2% of the total execution time. Therefore, the ideal speedup *S* is 4.56 at *C* = *T* = 8. Unfortunately, the experimental speedup of three systems at *G* = 10,000 and *nL* = 151 (as Fig. 2) are less than 4.56. These effects occur due to the *o(C)* and some source code modification to preserve the correct output. Moreover, the

speedups scale along with the number of data size (*G* + *nL*) increases. This observation could be described by Equation 8 that when *f* grows up, the serial (sequential) part, *(1-f)*, decreases in opposite to *f*. This leads to higher speedup.

index	%time	self	children	called	name
[1]	100	0.11	1065.70		main [1]
		37.60	618.16	3930/3930	hap_prior [3]
		65.78	320.99	4080/4080	hap_posterior [4]
		319.48	0.00	293886461/874204285	hap_posterior 4]
		618.16	0.00	568631912/874204285	hap_prior [3]
[2]	89.2	950.35	0.00	874204285	cmp_hap [2]

Fig. 3: Results of GProf profiling the SEQ SNPHAP program (*G* = 10000, *nL* = 151)

If *C* is increased proportionally, nevertheless, the speedup would be weak scaling and could be further limited by memory contention, cache misses, and communication bottleneck between cores. There are two ways for the solution including more cache capacity and improving memory contention for shared memory resources. The number of cache misses could be reduced with bigger cache capacity and deeper cache per socket (chip). For improving memory contention, we should avoid shared data among deference cores as possible. The best one is choosing the architecture that have not shared cache among cores and/or have to modify the program to avoid shared data.

Next, we will demonstrate that the complexity analysis of both parts depend on the number of CPU cores, the number of threads, and its Overhead.

B. Applying parallel For construct

We start with the theoretical analysis of *hap_prior()* and *hap_posterior()* of f proportion. The OMP SNPHAP was achieved by parallelizing two functions by using *parallel For* construct [10]. In theoretical analysis of the Multi-core CPUs, T threads are created such that each thread executes a chunk of *cmp_hap()*. In other words, the OpenMP *parallel For* construct splits and assigns the chunk of *cmp_hap()* to each thread.

As referred to the haplotype inference (HI) definition [15], [3], HI is the problem of inferring $2n$ haplotype pairs from n observed genotype data . For strictly biallelic genotype, the number of possible haplotype pairs per genotype is 2^{k-1}, where *k* is the number of heterozygous loci site (representing by 2) in an individual genotype data. So, the maximum number of possible haplotypes H is calculated by Equation 9.

$$H = 2\sum_{j=1}^{G} 2^{k_j - 1}, \tag{9}$$

where k_j is the number of heterozygous loci site at j^{th} genotype.

We need complexity *O(H)* calls the *cmp_hap()* to compare two haplotypes at the number of currently phased nP, in which the complexity of each call is *O(nP)*, Therefore, the computational complexity of the *hap_prior()* for running one EM iteration as shown in Equation 10.

$$T_{prior}(nP, H) = O(nP \times H) \tag{10}$$

Moreover, the computational complexity of the *hap_posterior()* for running one EM iteration, which we define as Equation 11.

$$T_{posterior}(nP, H) = O(nP \times H) \tag{11}$$

As referred to the SEQ SNPHAP algorithm in Algorithm 1, the EM algorithm will repeat *hap_prior()* and *hap_posteior()* until the EM iteration reach to the maximum l_{max} or frequencies are stable. At outer loop inside *main()*, the loop of *nP* will repeat until the *nP* reach to the maximum of the number of loci (*nL*) for each genotypes. The possible haplotype instance of the j^{th} genotype (H_j) can be explained with 2^{k_j}, where k_j is the number of heterozygous loci site (representing 2). In this way the total computational complexity of the *hap_prior()* + *hap_posteior()* as shown in Equation 12.

$$T_{total}(nL, G, l_{max}) = \sum_{nP=1}^{nL}\sum_{j=1}^{G}\sum_{i=1}^{l_{max}}((3nP \times 2^{k_j - 1}) + 2^{k_j + 1}) \tag{12}$$

Let *k* be the average of the number of heterozygous loci site so that the new computational complexity can be evaluated as Equation 13.

$$T_{total}(nL, G, l_{max}, k) = O(nL \times G \times l_{max} \times 2^k) \tag{13}$$

Since Multi-core CPUs can execute a number of threads T in parallel, the actual computational complexity as shown in Equation 14.

$$\begin{aligned} &T_{total\_threads}(nL, G, l_{max}, k, C, T) \\ &= \frac{nL \times G \times l_{max} \times ((3nL \times 2^{k-1}) + 2^{k+1})}{T} + o(T), \\ &= O(nL \times G \times l_{max} \times 2^k), \end{aligned} \tag{14}$$

where T is the number of threads, C is the number of processor cores, $o(T)$ are threading multiplexing, synchronization, load imbalance and also OpenMP overhead, and $T > C$. In the best case, if $T = C$ then $o(T)$ can be eliminated. The H depends on the number of heterozygous site (representing 2) in each genotype which normally contains $\approx 3/4$ of *nL* (evaluating from SNAP generating data set tool). As the nL increased, the H increased which lead to memory consumption and cache miss problem that can not be explored by Amdhal's law. This problem causes the speedup limitation.

For performance evaluation, let *C* and *T* be fixed and the l_{max} can be eliminated due to it is a small constant ($l_{max} = 100$) when compared with others, the complexity depends on nL, G, $o(T)$, and some overheads that can not be explored by Amdhal's law.

To improve the performance, we should increase the number of processor cores and reduce the *o(T)* and also some effects that can not be explored by Amdhal's law including cache miss, communication between cores, less memory bandwidth, and memory contention problem. The complexity analysis of *qsort()* part will be investigated in next subsection.

C. Applying Task construct

We have decided to use of the $Task$ construct to parallelize the recursive algorithm (*qsort()*) due to it is easy to run in parallel and all threads can execute tasks without barrier synchronization, leading to improve load balancing. Quick Sort, moreover, can operate on the same memory, the processor cores do not have to share data except synchronization among cores. Note that the *qsort()* is located inside the loop of *nP* in SEQ SNPHP. Due to the standard library *qsort()* in SEQ SNPHAP to difficultly parallelize, we have developed a new *qsort1()* with *Task* construct in a parallel/single paradigm with only one thread creating tasks and then inserting tasks into ready queue (using *#pragma omp single*) and other threads will pickup tasks from the queue for sorting a small haplotype instances list. This technique have introduced in [11].

In theoretical analysis of the Multi-core CPUs, T threads are created for tasks (using *# pragma omp parallel*). We acknowledge that Quick Sort complexity is *n log n* (on the average). Therefore, the computational complexity of the *qsort1()* at the number of currently phased *nP* is *O(H log H)*, where H is evaluated from Equation 9. The total computational complexity of the *qsort1()* is shown in Equation 15 when nP equals nL and k is the average of number of heterozygous loci.

$$\begin{aligned} T_{qsort}(nL, G, k) &= nL \times ((G \times 2^k)\ log\ (G \times 2^k)), \\ &= O(nL \times ((G \times 2^k)\ log\ (G \times 2^k))), \end{aligned} \tag{15}$$

Since Multi-core CPUs can execute *T* threads in parallel, the actual computational complexity as shown in Equation 16

$$\begin{aligned} &T_{qsort\_threads}(nL, G, k, C, T) \\ &= \frac{nL \times ((G \times 2^k)\ log\ (G \times 2^k))}{T} + o(T), \\ &= O(nL \times ((G \times 2^k)\ log\ (G \times 2^k))), \end{aligned} \tag{16}$$

where T is the number of thread, C is the number of processor core, $o(T)$ are threading multiplexing, synchronization and also OpenMP overhead, and $T > C$. In the best case, if $T = C$ then $o(T)$ can be eliminated. Note that the *H* depends on the number of heterozygous site of *nL* in each genotype so that the *nL* proportional to the *H*.

Let *C* and *T* are fixed, we conclude that the complexity of *qsort1()* depends on *nL*, *G*, *o(T)*, and some overheads that can not be explored by Amdhal's law including cache miss, communication between cores, memory bottleneck, and memory contention problem. In recursive algorithm, a larger number of haplotypes (*H*) lead to more recursive levels that can affect multiple tasks or threads on long execution time and also more memory consumption. On the other hand, the scheduler can easily distribute the work among all threads when more tasks are available. To achieve even higher performance, the *qsort1()* should be optimized by determining optimal tasks that the scheduler can spawn besides more processor cores, more cache memory capacity, and also its overhead should be reduced.

D. Cache memory and data size

In this section, anomaly relating cache memory and data size (*nL + G*) would be investigated in term of the performance analysis.

We have known that the number of possible haplotypes (H) depends on the data size (G and nL). A large number of haplotypes cannot fit in cache memory on both the same socket and among sockets. From system architecture summary in Table I, each core has a 32KB L1 data cache totalling 4 x 32KB L1 data cache per socket (chip). From our experiments, the maximum number of haplotypes at currently running phase (*nP*) are $\approx$ 1 million haplotypes so that 18 MBytes of memory space must be allocated. Note that one haplotype data structure uses 18 Bytes.

For working data size within cache memory, only some haplotypes can fit into the L1 data cache on the same socket (chip) while remaining haplotypes must be transferred from lower-level caches (L2 and L3). Unfortunately, these haplotypes might be moved to the neighborhood socket via communication links. If many more processor cores are used in the experiment, high shared-data management cost between cores and sockets (chips) will occur. These can lead to additional memory bandwidth contention and cache coherence problems. The factors of different result for Intel Core i7-2600 are larger/bigger L3 cache, higher bandwidth (three data accesses per cycle) and the new 32-bit ring-based interconnects between the processor cores.

There are a few solutions. The first one is using lager cache capacity and deeper level cache per socket (chip) to reduce sharing data among sockets. The second is increasing high speed communication links between cores and sockets to increase the bandwidth and reduce the bottlenecks. The third is avoiding shared data between cores and sockets as possible. These solutions could help achieve even higher performance.

V. Conclusion

In this paper, we present the performance analysis & improvement of OMP SNPHAP, which is a multithreaded version of SNPHAP, a haplotype inference program based on Expectation Maximization algorithm. The OMP SNPHAP utilizes *parallel For* and *Task* constructs of the OpenMP 3.0 library to spawn multiple independent threads in two SNPHAP's computational intensive functions so that these threads can be concurrently executed on Multi-core CPUs. Experimental results reveal that the performance of OMP SNPHAP on Intel Core i7-2600, Intel Xeon E5405, Intel Xeon E5520 and AMD Opteron 8356 running Linux operating system can achieve great speedup up to 260%, 316%, 410% and 488%, respectively.

The speedup of OMP SNPHAP running on four CPUs is proportional to the number of threads forked by OpenMP. However, %overhead of OpenMP can significantly slow down the total execution time especially with small input data with large number of threads (%overhead of 1 thread/core is much lower than that of 2 threads/core). Moreover, the factors that affect OMP speedup comprise greater number of CPU cores, the specific parallelized code fraction, the suitable OpenMP constructs for that fraction, the number of processor cores, the

size and organization of cache memories, the clock frequency and the better memory bandwidth/technology.

VI. Acknowledgement(s)

We thank Prof. David Clayto for making SNPHAP source code available for download. We are grateful to the National Center for Genetic Engineering and Biotechnology (BIOTEC) Thailand for 32-core CPUs system supports. This work was supported in part by BIOTEC platform technology grant given to ST and the Thailand Research Fund grant number RSA5480026.

References

[1] D. Fallin and N. J. Schork, "Accuracy of haplotype frequency estimation for biallelic loci, via the expectation-maximization algorithm for unphased diploid genotype data," *Am. J. Hum. Genet*, vol. 67, pp. 947 – 959, 2000.

[2] P. Bonizzoni, G. D. Vedova, R. Dondi, and J. Li, "The haplotyping problem: an overview of computational models and solutions," *Journal of Computer Science and Technology (JCST)*, vol. 18, pp. 675 – 688, 2003.

[3] D. Gusfield and S. H. Orzack, *Haplotype Inference*, ser. CRC Handbook on Bioinformatics. Boca Raton, USA: CRC Press, 2005, ch. 1, pp. 1 – 25.

[4] L. Eronen, F. Geerts, and H. Toivonen, "Haplorec: Efficient and accurate large-scale reconstruction of haplotypes," *BMC Bioinformatics*, vol. 7, p. 542, 2006.

[5] G. Kimmel and R. Shamir, "Gerbil: Genotype resolution and block identification using likelihood," in *Proceedings of the National Academy of Sciences*.

[6] Z. S. Qin, T. Niu, and J. S. Liu, "Partition-ligation-expectation-maximization algorithm for haplotype inference with single-nucleotide polymorphisms," *Am J Hum Genet*, vol. 70, pp. 1242–1247, 2002.

[7] S. P and S. M, "A fast and flexible statistical model for large-scale population genotype data: Applications to inferring missing genotypes and haplotypic phase," *Am J Hum Genet*, vol. 78(4), pp. 629–644, 2006.

[8] M. Stephens, N. J. Smith, and P. Donnelly, "A new statistical method for haplotype reconstruction from population data," *Am J Hum Genet*, vol. 68, pp. 978–989, 2001.

[9] M. Stephens and P. Scheet, "Accounting for decay of linkage disequilibrium in haplotype inference and missing-data imputation," *Am J Hum Genet*, vol. 76(3), pp. 449–462, 2005.

[10] U. Ranok, S. Kittitornkun, and S. Tongsima, "Multithreading bioinformatics software with openmp: Snphap case study," in *Proceeding of the IASTED International Conference Parallel and Distributed Computing and Systems (PDCS2010)*, Marina Del Rey, USA, 2010.

[11] U. Ranok, S. Kittitornkun, and S. Tongsima., "A multithreading methodology with openmp on multi-core cpus: Snphap case study," in *8th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON 2011)*, Khon Kaen, Thailand, May 17-19 2011.

[12] M. Nothnagel. (2005) Snap: simulation of snp haplotype data and phenotypic traits. http://capella.uni-kiel.de/snap/snap.htm.

[13] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, ser. second edition. Pearson Education Limited, 2003.

[14] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," in *SIGPLAN '82 Symposium on Compiler Construction*, ser. SIGPLAN Notices, vol. 17, no. 6, June 1982, pp. 120 – 126.

[15] Y. S. Song, Y. Wu, and D. Gusfield, "Algorithms for imperfect phylogeny haplotyping (ipph) with a single homoplasy or recombination event," in *5th Workshop on Algorithms in Bioinformatics (WABI'2005)*.

Algorithm 1 EM Algorithm according to SEQ SNPHAP (version 1.3.1)

1) "Guesstimate" haplotype frequencies:
initialize $f_t^{(l)}, \Theta^{(l)}$, EM iteration number $(l) = 0$, $\forall t : 1 \leq t \leq H$.
Let t is any arbitrary number and $P(h_t, h_{t+1})$ is the probability that hp_i is composed of haplotype pair h_t and h_{t+1}, and also f_t and f_{t+1} are the frequencies of the h_t and h_{t+1}.

2) (M)aximization step corresponds to *hap_prior()* in *SNPHAP*:
This step can be divided into two parts. First, the total of posterior frequencies is calculated by Equation 1. Then, the next set of estimates of "prior" haplotype frequencies are calculated by Equation 2.

$$P_{total}^{(l)} = \sum_{t=1}^{H} f_t^{(l)}, \tag{1}$$

$$\widetilde{f}_t^{(l)} = \frac{\sum_{\forall h_t \in H_x} f_t^{(l)}}{P_{total}^{(l)}}, (\forall t : 1 \leq t \leq H), \tag{2}$$

where $H_x = \{$all elements in group x are the same haplotype$\}$
(The comparison in this procedure corresponds to *cmp_hap()* inside *hap_prior()* in *SNPHAP*).

3) (E)xpectation step corresponds to *hap_posterior()* in *SNPHAP*:
Given current estimates of prior frequencies from M step and assuming HWE shown as Equation 3, Equation 4 calculates the probability of each phased, assign the frequencies of the haplotype pair with Equation 5, and then complete the "posterior" haplotype frequencies of the genotype assignments by Equation 6. Finally, the log-likelihood is calculated with Equation 7.
Let P_j as the probability of the j^{th} genotype that maximizes the likelihood that is the sum of probabilities of haplotype pairs hp_i of each c_j

$$P(hp_i)^{(l)} = P(h_t, h_{t+1})^{(l)} = \begin{cases} \widetilde{f}_t^{(l)} \widetilde{f}_{t+1}^{(l)} & \text{if } h_t = h_{t+1}, \\ 2\widetilde{f}_t^{(l)} \widetilde{f}_{t+1}^{(l)} & \text{if } h_t \neq h_{t+1}. \end{cases} \tag{3}$$

$$P_j^{(l)} = \sum_{i=1}^{c_j} P(hp_i)^{(l)}, \tag{4}$$

$$f_t^{(l)} = f_{t+1}^{(l)} = P(h_t, h_{t+1})^{(l)} \tag{5}$$

$$f_t^{(l+1)} = \frac{f_t^{(l)}}{P_j^{(l)}}, (\forall j : 1 \leq j \leq G, \forall t : 1 \leq t \leq H), \tag{6}$$

$$\Theta^{(l+1)} = \log(\sum_{j=1}^{G} \sum_{i=1}^{c_j} P(hp_i)^{(l)}) \tag{7}$$

4) If $\mid \Theta^{(l+1)} - \Theta^{(l)} \mid \; > \varepsilon$ (log likelihood tolerance) , $l < l_{max}$ then $l = l + 1$ and go to M Step.
Else $\Theta^* = \Theta^{(l+1)}$

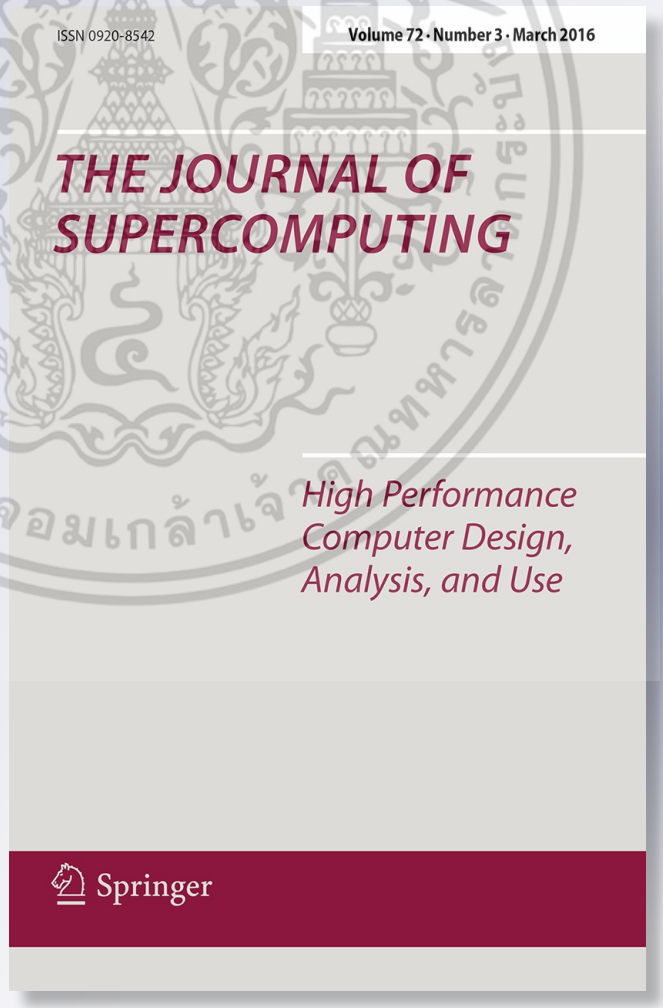
Parallel Partition and Merge QuickSort (PPMQSort) on Multicore CPUs

Ratthaslip Ranokphanuwat & Surin Kittitornkun

The Journal of Supercomputing
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542
Volume 72
Number 3

J Supercomput (2016) 72:1063-1091
DOI 10.1007/s11227-016-1641-y



J Supercomput (2016) 72:1063–1091
DOI 10.1007/s11227-016-1641-y

Parallel Partition and Merge QuickSort (PPMQSort) on Multicore CPUs

Ratthaslip Ranokphanuwat[1] · **Surin Kittitornkun**[1]

Published online: 18 February 2016

Abstract An explosive amount of data has tremendous impacts on sorting, searching, indexing, and so on. Sorting is one of the basic Computer Science problems needed to be fast and efficient to serve Big Data. This paper presents an efficient and scalable algorithm called *Parallel Partition and Merge QuickSort (PPMQSort)* running on any shared memory/multicore/multi-socket systems. Together with OpenMP 3.0 library, the PPMQSort is developed to be compatible and benchmarked with the fastest C/C++ Stdlib *qsort*(). The PPMQSort recursively divides an unsorted input array into partially sorted partitions up to *Cutoff* length using nested multithreading. Finally, those independent partitions are *qsort*() (conquered) such that no synchronizations are needed. The resulting Speedup of 12.29× on a dual-socket 8-core Xeon E5520 can be achieved for sorting random 200 M 32-bit integer data at 16 threads. With the same configuration, a 4-core AMD A6-3600 CPU (non-HyperThread) can reach up to 4.67×, a superlinear Speedup. It has been proved that the proposed PPMQSort can exploit all available cache levels and HyperThread CPU cores well thus utilizing up to 83 % and 96 % of CPU on E5520 and A6-3600, respectively.

Keywords QuickSort · Parallel · OpenMP · Multicore · Multithread · Superlinear

✉ Ratthaslip Ranokphanuwat
udom.ran@dpu.ac.th

Surin Kittitornkun
surin.ki@kmitl.ac.th

[1] Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, No. 1, Soi Chalong Krung 1, Chalong Krung Rd., Ladkrabang, Bangkok 10520, Thailand

1 Introduction

Sorting has become highly important for Big Data analyses especially in social/web mining, large scale scientific, commercial application domains and so on. Among all the sorting algorithms, QuickSort [1,2] is the most well-known and standard sorting algorithm. To compare with any existing sorting algorithms, QuickSort is the fastest one in practice [3].

Numerous parallel architectures can be applied to perform sorting algorithms. Earlier studies have shown [4–6] that sorting can be done at the interconnection level of a particular network of processors named the MultiRing network. Recently, sorting networks have been implemented on FPGAs instead. References [7–9] used FPGAs as sorting kernels for database intensive operations. In addition to FPGAs, hundreds to thousands of processing elements/cores inside the GPUs can be applied as co-processors for sorting [10] based on SIMD parallelism including the Bitonic-Merge Sort on Intel Xeon Phi [11].

A few parallel algorithms have been proposed to enhance the existing QuickSort algorithm. Initially, Heidelberger [12] presented the parallel version on an *ideal* Parallel Random Access Machine. In practice, the sequential QuickSort can be enhanced with several parallel techniques to run on any shared memory/multicore systems with multithreading operating system. In 2003, Tsigas and Zhang [13] proposed a fine-grain parallel QuickSort algorithm to fit data into L1 caches. A year later, [14] presented several alternative algorithms of parallel QuickSort based on pthreads and OpenMP 2.0. Man et al. [15,16] developed *psort*() algorithm to be compatible with Stdlib *qsort*(). Their work can achieve Speedup by only 11 times faster with 24 cores. Meanwhile, Kim et al. [17] have shown that a dual-core OMAP-4430 can achieve only 1.47x Speedup from their Introspective QuickSort algorithm. Mahafzah [18] splitted the input array with multi-pivot/thread into partitions using extra space and then sorted them in parallel up to 8 threads. Very recently, Bingmann et al. [19] proposed multikey QuickSort algorithms for string sorting on NUMA (Non Uniform Memory Access) architectures. Their results show that the Speedup is bounded by memory bandwidth.

However, it is still challenging to enhance parallel QuickSort performance and efficiency at the same time. These challenges are due to sequential data partitioning, latency/bandwidth between memory hierarchy, and sequential and recursive nature of QuickSort. Furthermore, the bottlenecks of parallel QuickSort should be further investigated together with some performance characteristics such as CPU utilization, memory bandwidth and branch misprediction rate.

In this paper, we have proposed and developed a Parallel Partition and Merge QuickSort (PPMQSort) for various multicore CPUs. Our contributions are summarized as follows:

1. The PPMQSort algorithm is compatible and benchmarked with Stdlib *qsort*() while achieving superlinear Speedups in some CPUs.
2. The efficiency called Speedup per Core of PPMQSort and any parallel algorithm on both HyperThread and non-HyperThread CPUs is proposed. Hence, the PPMQSort can achieve higher efficiency than previous algorithms.

3. The time complexity of PPMQSort has been analyzed and presented in big-O notations.
4. Based on the Linux Perf measurement tool, a system performance model of any shared memory/multiprocessor/multicore systems is proposed to estimate memory bandwidth.
5. The Speedups of PPMQSort with Worst-case input data although very rare but can be as high as those of Random cases.

The rest of the paper is organized as follows. Section 2 presents background and related work. Section 3 presents our algorithm and discusses the implementation details. Section 4 describes performance evaluation and discussions. Finally, Sect. 5 concludes and suggests future work.

2 Background and related work

We begin with a brief overview of QuickSort, Stdlib *qsort*(), a number of parallel QuickSort algorithms, and finally OpenMP library.

2.1 QuickSort algorithm [1, 2]

QuickSort is the most famous and widely used sorting algorithm. The divide and conquer concept recursively partitions and swaps an input array into two halves: less than or equal (LEQ) half and greater than (GT) half with respect to a selected pivot element at each recursion level. The time complexity on average is, therefore, $O(n \log n)$ although the poorly selected pivot can affect its complexity. Even worse, the worst-case input array can make the complexity become $O(n^2)$. In terms of space requirements, QuickSort is considered to be an in-place algorithm using minimal extra memory. During the recursion, extra space for calling stack is proportional to $O(\log n)$. To optimize its performance, selecting good pivot(s) from several candidates has been considered.

2.2 Stdlib *qsort*()

The Standard Library *qsort*() is a very useful function for sorting an array of any data types with a user-defined comparison function. It is implemented in C/C++ and also provided as a built-in function for several C/C++ compilers. Its function prototype is declared in *Stdlib.h* as follows.

```
void qsort(void  *base, size_t num_elements,
   size_t element_size,
   int (*compare)(void const *, void  const *));
```

The argument *base* is a pointer to the unsorted array, *num_elements* indicates the number of elements, *element_size* is the size of each element, and *compare* is a pointer to the user-defined function that returns integer values according to the comparison result.

2.3 Parallel QuickSort algorithms

In 1990, Heidelberger et al. [12] presented a parallelization of the Quicksort on a theoretical/ideal Parallel Random Access Machine with average of $O(\log n)$ time complexity. In practice, the sequential QuickSort can be enhanced with several parallel techniques to run on any shared memory/multicore systems with multithreading operating system. Parallel versions of QuickSort normally start with partitioning data into several chunks to fit any cache level depending on the size. These chunks can be partially or fully sorted and then merged to form bigger chunks. These two steps may be recursive as indicated in the Recursion row of Table 1. Some algorithms may use extra space to hold the intermediate results as shown in Ex. Space row. Eventually, they shall be fully sorted again with either the Stdlib $qsort()$ or others. The comparison of previous parallel QuickSort algorithms is shown in Table 1 in chronological order from left to right.

Tsigas and Zhang [13] proposed a fine-grain (block-based) parallel Quicksort algorithm. Subsequently, [14] presented several alternative algorithms of parallel Quicksort based on pthreads and OpenMP 2.0. Rashid et al. [20] enhanced Tsigas and Zhang's [13] PQuicksort on x86 Multithreaded Architectures. Man et al. [15,16] developed $psort()$ algorithm to be compatible with Stdlib $qsort()$. The input array is divided into groups and $qsort()$ them. Later on, these partitions can be merged using extra space and finally $qsort()$ them again. Their work can achieve Speedup by 11 times faster with up to 24 cores. Kim et al. [17] have shown that an embedded dual-core OMAP-4430 can achieve 1.47x Speedup from their Introspective Quicksort algorithm. Mahafzah [18] splitted the input array with multi-pivot/thread into partitions using extra space and then sort them in parallel up to 8 threads. Recently, Saleem et al. [21] estimated Speedup for QuickSort and Merge sort algorithms using Intel Cilk Plus.

2.4 OpenMP library

OpenMP library [22] is the most well-known library that can be applied successfully to develop parallel programs running on multicore CPUs architecture. It provides an application program interface (API) for thread-based parallelism on shared memory multicore processors. The API consists of a set of compiler directives, library routines, and environmental variables that support FORTRAN and C/C++ on multiple architectures. OpenMP uses the fork-join model for multithreading execution model. The main advantage of using OpenMP is the ability of all CPU cores to share and access the same memory pool (data) with less communication overhead and network latency compared with other parallel computing paradigms such as cluster computing, grid computing, etc.

Since OpenMP version 3.0, Task construct has been introduced to handle irregular and dynamic parallelism in the form of recursive routines. Tasks are units of work which can be executed (forked) in parallel as threads. This paper specifically demonstrates how to exploit the Task construct in our parallel QuickSort algorithm.

Table 1 Comparison of previous parallel QuickSort algorithms, *Par.* Parallel, *Seq.* Sequential, *Sync.* Synchronization, *NA* Not Available

Year References	2003 [13]	2004 [14]	2011 [16]	2011 [17]	2013 [18]	2014 [21]
Algo. name	PQuicksort	cv_1.0	psort1	Introspective	QuickSort	Quicksort
Partition	Par. in blocks of L1 size	Seq.	Par. n/c and $qsort()$	Seq. n/c	Par. multiple pivots	Seq. n
Merge	Seq. Swap	No	Seq. Merge and and $qsort()$	No	No	No
Recursion	Yes	Yes	No	No	No	Yes
Time complexity	$O(\frac{n}{c} + \frac{n}{c} \log \frac{n}{c})$	NA	$O(n + \frac{n}{c} \log \frac{n}{c})$	$O(\frac{n}{2} + \frac{n}{2} \log \frac{n}{2})$	$O(\frac{n}{h} \log \frac{n}{h})$	NA
Extra space (size)	No	No	Yes(n)	No	Yes(n)	No
Using $qsort()$	Similar	No	Yes	No	No	No
Other sort	Insertion	No	No	Insertion	No	No
Library	NA	pthreads	OpenMP 3.0	OpenMP 3.0	pthreads	Cilk Plus
Pros	Cache efficient, Fine-grained,	Load balance- With busy waiting	Qsort() lib. compatible, Good load balance	Limit deep partition, Cache friendly, Good load balance	Utilize SMT architecture, Good load balance	Easily
Cons	Bottleneck- In seq. merge, Special Sync.- Instruction	Sync. added, Less algorithm details	Difficult to implement, High overhead	No nested parallelism, Seq. partition	Sync. added, Extra space	Unpopular lib.

3 PPMQSort algorithm

Previous parallel QuickSort algorithms focus on optimizing either partitioning phase or recursive QuickSort phase. Our PPMQSort pays attention on both phases. In this section, we propose the Parallel Partition and Merge QuickSort (PPMQSort) on any multicore CPUs. The concept of PPMQSort is to partition an unsorted array into partially sorted partitions in the Parallel Partition Step. Then, these partitions can be eventually sorted independently using OpenMP Task construct in the Parallel $qsort()$ Step. Those important steps mentioned above can be illustrated in Fig. 1.

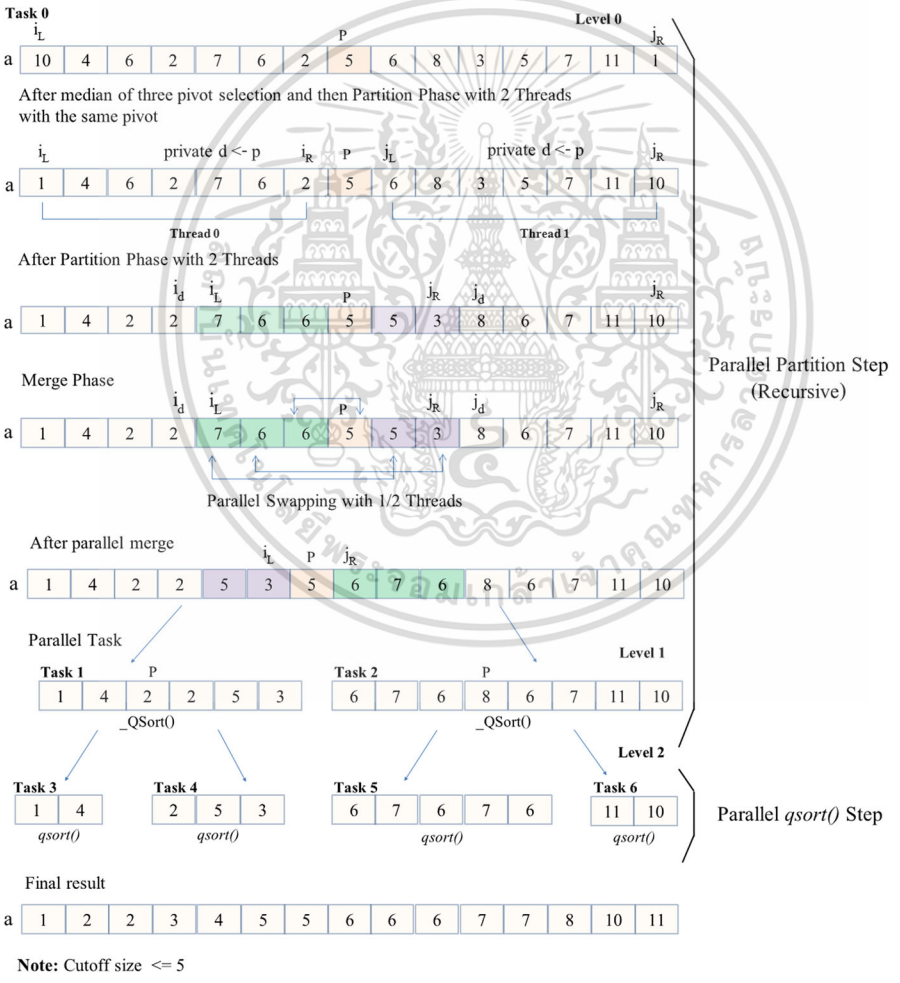


Fig. 1 Illustration of Parallel Partition and Merge QuickSort (PPMQSort) consisting of Parallel Partition Step and Parallel $qsort()$ Step

Table 2 Notations

a	Input data array
a_j	Data element at index j
a_{i_d}	Data element at the middle index i_d
a_p	Data element at pivot p
B	Branch Loads
B_m	Branch Load Misses
$B_{m/s}$	Branch Load Misses Per Second
C	Cache References
C_m	Cache Misses
$C_{m/s}$	Cache Misses Per Second
$\lvert C_{\text{line}} \rvert$	Cache Line Size
c	Number of processor cores
d	The middle index
$f()$	A function
HT/NHT	HyperThread/Non-HyperThread
i, j	Loop indices
i_L, i_R	Left most and right most indices of the left-hand side subarray, respectively
j_L, j_R	Left most and right most indices of the right-hand side subarray, respectively
i_d, j_d	The middle indices of left subarray and right subarray, respectively
k	Number of CPU sockets
K	10^3 or 2^{10}
l	Recursion level l
M	10^6 or 2^{20}
M_{bw}	Total memory bandwidth
n	Number of elements of array a
$O()$	BigO notation
o	Optimization Level
p	The pivot place
$R_{x,y}$	Correlation Coefficient of x and y
S	Speedup
s	Second
T_{qsort}	Run Time of sequential Stdlib $qsort()$
T_{ppmqsort}	Run Time of PPMQSort
T_{seq}	Run Time of any sequential QuickSort
T_{par}	Run Time of any parallel QuickSort
U	%CPU Utilization
u	Cutoff size

The PPMQSort is actually developed in C language on top of an open-source/past version of Stdlib $qsort()$ utilizing stack rather than recursion. Due to limited space and ease of understanding, the algorithms are explained in recursion. Notations used in this paper are listed in Table 2.

We first present the *Partition Phase with 2 Threads and Merge Phase with 1 or 2 Threads*. Then we show how to apply OpenMP Task parallelism to call $qsort()$. At last, the time complexity of our algorithm is analyzed.

3.1 Parallel Partition Step

The partitioning operation has been a major bottleneck of QuickSort since it was invented. Previous work has tried to optimize it by both reducing the number of key comparisons and fast swapping code. The key idea of the Partition Phase with 2 Threads is to divide the input data array into two subarrays. Then, they can be partitioned in parallel with 2 threads into 4 sub-subarrays using the same pivot value. Next is the Merge Phase with 1 or 2 Threads swapping the second and third sub-subarrays. Both phases of Parallel Partition Step are explained in details as follows.

3.1.1 Partition Phase with 2 Threads

Initially, an unsorted data array, $a = a_0, a_1, \ldots, a_{n-1}$, is divided into two independent subarrays at the pivot p. Let a_p denotes the pivot element selected by $MedianOfThree()$ function. Let i_L and i_R be left indices and j_L and j_R be right indices of a, respectively. The left subarray of a, $a_0, \ldots, a_{p-1}$ corresponds to $(i_L = 0, i_R = p - 1)$. Similarly, the right subarray, $a_{p+1}, \ldots, a_{n-1}$, corresponds to $(j_L = p + 1, j_R = n - 1)$. In this phase, both subarrays, (a_{i_L}, a_{i_R}) and (a_{j_L}, a_{j_R}), are compared and swapped with the same pivot a_p simultaneously using 2 threads on line 13 and line 15 in $seq\_partition()$ of Algorithm 1. In addition, $seq\_partition()$ returns the partition index as i_d and j_d for the left and right partitions, respectively, as shown. As a result, $a_0, \ldots, a_{n-1}$ are splitted into 4 sub-subarrays; two sub-subarrays on the left, $a_0, \ldots, a_{i_d}$ and $a_{i_d+1}, \ldots, a_{p-1}$, and two sub-subarrays on the right, $a_{p+1}, \ldots, a_{j_d-1}$ and $a_{j_d}, \ldots, a_{n-1}$. Notice that i_d and j_d are the middle indices of the left and right subarrays, respectively.

From a programming perspective, we have applied OpenMP Parallel Tasks without barrier synchronization, leading to improved CPU utilization. To reduce the number of shared memory accesses, $d = p$ is copied to be a local private variable to improve cache locality. Both Phases are listed in Algorithm 1.

In summary, based on i_d, p, and j_d, two independent subarrays can be partitioned into 4 sub-subarrays in parallel with respect to the global a_p pivot in this phase. These 4 sub-subarrays are ordered as follows: less than or equal (LEQ), greater than (GT), LEQ, and GT from left to right.

3.1.2 Merge Phase with 1 or 2 Threads

In this subsection, we will explain how the second (GT) sub-subarray, $a_{i_d+1}, \ldots, a_{p-1}$, and the third (LEQ) sub-subarray, $a_{p+1}, \ldots, a_{j_d-1}$, are swapped and merged together. The idea of this phase is to swap all data in both sub-subarrays to rearrange them in the correct order, LEQ and GT. Because this phase needs only to swap a bulk of data

Algorithm 1 The Parallel Partition algorithm

```
1: function PARALLELPARTITION(a,start,end)                    ▷ Parallel Partition Step
2:     i_L, j_R, p ← Partition(a, start, end)                 ▷ call Partition function
3:     p ← Merge(a, p, i_L, j_R)                              ▷ call Merge function
4:     return p
5: end function

6: function PARTITION(a, i_L, j_R)                            ▷ Partition Phase with 2 Threads
7:     p ← MedianOfThree(a, i_L, j_R)
8:     d ← p
9:     i_R ← p − 1
10:    j_L ← p + 1
11:    begin OpenMP parallel Tasks private(d)
12:    OpenMP Task
13:    i_d ← seq_partition(a, d, i_L, i_R)                    ▷ Partition the Left Subarray
14:    OpenMP Task
15:    j_d ← seq_partition(a, d, j_L, j_R)                    ▷ Partition the Right Subarray
16:    end parallel Tasks
17:    i_L ← i_d + 1
18:    j_R ← j_d − 1
19:    return (i_L, j_R, p)
20: end function

21: function MERGE(a, p, i_L, j_R)                            ▷ Merge Phase with 1 or 2 Threads
22:                                  ▷ Three cases for calculating location and moving the pivot p
23:    if len(i_L, p − 1) < len(p + 1, j_R) then              ▷ Left side is shorter.
24:        length ← len(i_L, p − 1)
25:        Swap(a_p, a_{j_R − length})
26:        p ← j_R − length
27:        temp ← p + 1
28:    else if len(i_L, p − 1) > len(p + 1, j_R) then         ▷ Right side is shorter.
29:        temp ← p + 1
30:        length ← len(p + 1, j_R)
31:        Swap(a_p, a_{i_L + length})
32:        p ← i_L + length
33:    else                                                   ▷ Left side equals right side.
34:        temp ← p + 1
35:        length ← len(p + 1, j_R)
36:    end if
37:    begin OpenMP parallel For with 1 or 2 Threads
38:    for i ← 0, length − 1 do                               ▷ Swapping with 1 Thread or 2 Threads
39:        Swap(a_{i_L + i}, a_{temp+i})
40:    end for
41:    end parallel For
42:    return p
43: end function
```

between them, no comparisons are necessary. Furthermore, swapping would work with data on the same sub-subarrays so that our method does not use an extra memory.

The Merge Phase with 1 or 2 Threads is shown as function $Merge()$ on line 21 of Algorithm 1 where $len()$ returns the number of elements between two arguments ($len(x, y) = y - x + 1; y \geq x$). Let $i_L = i_d + 1$ and $j_R = j_d - 1$ be the left most index and the right most index of the sub-subarray. So, the second (GT) sub-subarray consists of $a_{i_L}, \ldots, a_{p-1}$ and the third (LEQ) sub-subarray consists of $a_{p+1}, \ldots, a_{j_R}$.

Both arrays must be swapped to complete the Parallel Partition Step. The swapping will start from this pair $(a_{i_L+i}, a_{\text{temp}+i})$ and incrementally continue for $i = 0$ to length -1 on line 39 of Algorithm 1. After swapping is finished, a_p must be adjusted to the correct position. Both phases of Parallel Partiton Step are recursive as shown in function $\_QSort()$ until each partition's size is no greater than Cutoff u on line 1 of Algorithm 2. Although it is associated with OpenMP Single construct on line 15, parallelism can be achieved up to $2h$ threads in reality. That's because the Parallel Partition Step each forks 2 threads internally.

Three important steps need to be considered in this phase. Firstly, the total number of elements swapped between two sub-subarrays is calculated. This number can be determined from the shorter length of either i_d and pivot place p or j_d and p. The variable length can be $\leq \frac{n}{4}$. Then, the direction of swapping sequence is determined. To be cache friendly, increasing order is chosen. The last step is to move the pivot to the appropriate position in the array after swapping process is finished to guarantee that the Parallel Partition Step is completed. In the next Parallel $qsort()$ Step, those partitions can be $qsort()$ in parallel up to h threads.

3.2 Parallel $qsort()$ Step

The Parallel Partition Step can be cutoff by u elements to avoid over partitioning so that $qsort()$ can efficiently sort in each core's private L2 cache or shared L3 cache depending on the hardware. The Parallel $qsort()$ Step is on the else part of function $\_Qsort()$ on line 9 of Algorithm 2. Therefore, Cutoff u should be parameterized in the experiment to achieve the best Speedup. As a result, if the Stdlib $qsort()$ performance is improved, the performance of PPMQSort will be automatically enhanced. Next, the time complexity of PPMQSort will be analyzed in $O()$ notation.

Algorithm 2 The PPMQsort Algorithm

```
1: function _QSORT(a, i_L, j_R, u)
2:     if i_L + u < j_R then                          ▷ if j_R − i_L > Cutoff u
3:         p ← Parallel Partition(a, i_L, j_R)        ▷ Parallel Partition Step
4:         OpenMP Task
5:         _QSort(a, i_L, p − 1, u)                   ▷ Left Subarray
6:         OpenMP Task
7:         _QSort(a, p + 1, j_R, u)                   ▷ Right Subarray
8:     else                                           ▷ else less than or equal Cutoff u
9:         qsort(a, i_L, j_R)                         ▷ Parallel qsort() Step
10:    end if
11: end function

12: function PPMQSORT(a,start,end,h,u)                ▷ PPMQsort() Function
13:     begin OpenMP parallel with h threads
14:     OpenMP Single
15:     _QSort(a, start, end, u)                      ▷ with Cutoff u
16:     end parallel
17: end function
```

3.3 Complexity analysis

The time complexity of PPMQSort is analyzed assuming that all c cores are 100 % utilized by running $h \geq c$ threads. The analysis can be divided into two steps: Parallel Partition Step and Parallel $qsort()$ Step as follows.

Lemma 1 *Let n be the size of data array a, where* $a = a_0, a_1, \ldots, a_{n-1}$. *Then, the time complexity of* Parallel Partition Step *with h Threads on c cores where* $h \geq c$ *is* $O(n + \frac{n}{c} \log \frac{n}{2uc})$.

Proof At the beginning (level 1), the number of comparisons in Partition Phase with 2 Threads is $2 \times \frac{n}{4}$. Due to $c \geq 2$ cores, the time complexity is $\frac{1}{c} \times 2 \times \frac{n}{4} = \frac{2}{2}(\frac{n}{4})$. The number of swappings in Merge Phase with 1 Thread is $\frac{n}{4}$. Due to its sequential operation, its time complexity is $\frac{n}{4}$. In the first recursion level, the time complexity is hence $\frac{2}{2}(\frac{n}{4})+\frac{n}{4}$. In the second level, there are two independent partitions with $c \geq 2$ processor cores. The time complexity of Partition Phase with 2 Threads is $\frac{1}{c} \times 4 \times \frac{n}{8} = \frac{4}{c}(\frac{n}{8})$. The number of swappings in Merge Phase with 1 Thread is $2 \times \frac{n}{8}$. Due to its parallel operation, its time complexity is now $\frac{1}{c} \times 2 \times \frac{n}{8} = \frac{2}{c}(\frac{n}{8})$. The total time complexity of the second level is $\frac{4}{c}(\frac{n}{8}) + \frac{2}{c}(\frac{n}{8})$. The partitioning process is recursive until the condition on line 2 of Algorithm 2 is FALSE. That means the partition size is not larger than Cutoff u elements. Based on the divide and conquer concept, the number of this recursive partitioning is $\log_2 \frac{n}{u}$ levels on average with respect to Cutoff u.

Therefore, the total time complexity of the Parallel Partition Step is

$$
\begin{aligned}
&= \frac{2}{2}\left(\frac{n}{4}\right) + \frac{n}{4} + \frac{4}{c}\left(\frac{n}{8}\right) + \frac{2}{c}\left(\frac{n}{8}\right) + \frac{8}{c}\left(\frac{n}{16}\right) + \frac{4}{c}\left(\frac{n}{16}\right) \\
&\quad + \cdots + \frac{2^{\log_2 \frac{n}{u}}}{c}\left(\frac{n}{2^{\log_2 \frac{n}{u}+1}}\right) + \frac{2^{\log_2 \frac{n}{u}-1}}{c}\left(\frac{n}{2^{\log_2 \frac{n}{u}+1}}\right) \\
&= 3 \times \left[\frac{2^0}{2}\left(\frac{n}{2^2}\right) + \frac{2^1}{c}\left(\frac{n}{2^3}\right) + \frac{2^2}{c}\left(\frac{n}{2^4}\right) + \cdots + \frac{2^{\log_2 \frac{n}{u}-1}}{c}\left(\frac{n}{2^{\log_2 \frac{n}{u}+1}}\right)\right] \\
&= 3 \times \left[n \sum_{l=1}^{\log_2 c} \frac{1}{2^l}\left(\frac{2^{l-1}}{2^{l+1}}\right) + \frac{n}{c} \sum_{l=\log_2 c+1}^{\log_2 \frac{n}{u}} \left(\frac{2^{l-1}}{2^{l+1}}\right)\right] \\
&= \frac{3}{4} \times \left[n \sum_{l=1}^{\log_2 c} \frac{1}{2^l} + \frac{n}{c} \sum_{l=\log_2 c+1}^{\log_2 \frac{n}{u}} 1\right] \\
&= \frac{3}{4} \times \left[n\left(1 - \frac{1}{c}\right) + \frac{n}{c} \log_2 \frac{n/u}{c}\right]. \\
&= \frac{3}{4} \times \left[n - \frac{n}{c} + \frac{n}{c} \log_2 \frac{n/u}{c}\right].
\end{aligned}
$$

$$= \frac{3}{4} \times \left[n + \frac{n}{c} \log_2 \frac{n/u}{2c} \right].$$
$$= \frac{3}{4} \times \left[n + \frac{n}{c} \log_2 \frac{n}{2uc} \right].$$

As a result, the time complexity of Parallel Partition Step is $O(n + \frac{n}{c} \log \frac{n}{2uc})$. □

Lemma 2 *Let c processor cores perform qsort() each partition of size u elements in parallel. Since there are at least* $\frac{n}{u}$ *partitions, the time complexity of Parallel qsort() Step is* $O(\frac{n}{c} \log u)$.

Proof From Parallel Partition Step, at least $\frac{n}{u}$ partitions can be obtained. Each partition of up to u elements is sorted by $qsort()$ in parallel up to $h \geq c$ threads. The time complexity of Parallel $qsort()$ Step is, therefore, $\frac{1}{c} \times \frac{n}{u} \times u \log_2 u$
$= \frac{n}{c} \log_2 u$
$= O(\frac{n}{c} \log u)$. □

Theorem 1 (PPMQSort's Theorem) *The total time complexity of sorting n elements with the proposed PPMQSort running in parallel on* $c \geq 2$ *processor cores with Cutoff u elements and* $h \geq c$ *threads is* $O(n + \frac{n}{c} \log \frac{n}{2c})$.

Proof The complexities of Parallel Partition Step (see Lemma 1) and of Parallel $qsort()$ Step (see Lemma 2) are $O(n + \frac{n}{c} \log \frac{n}{2uc})$ and $O(\frac{n}{c} \log u)$, respectively. The total time complexity is $O(n + \frac{n}{c} \log \frac{n}{2uc} + \frac{n}{c} \log u)$. Therefore, the time complexity of PPMQSort is $O(n + \frac{n}{c} \log \frac{n}{2c})$. □

The time complexity of PPMQSort is similar to that of psort1 algorithm [15] as listed in Time Complexity row of Table 1. PPMQSort requires no extra space for intermediate results. As the data size n and number of cores c grow, PPMQSort can eventually outperform other algorithms due to its simplicity, scalability, and efficiency. The next section will show how PPMQSort is evaluated.

4 Performance evaluation and discussions

This section presents how various performance metrics are measured. The experiment setups and results are discussed later on.

4.1 Performance measurement

To investigate how the multicore architectures impact the performance of the algorithm, various performance metrics are measured and analysed.

1. CPU Time (in Seconds)
   To fairly compare T_{qsort} and T_{ppmqsort} in any experimental configurations, the CPU time is measured without data file loading and other overheads and averaged by 5 times.

2. Speedup S(x)
   This metric indicates that how many times our PPMQSort can be executed faster than the sequential Stdlib $qsort()$. Based on the measured T_{qsort} and T_{ppmqsort}, Speedup S can be computed as

$$S = \frac{T_{\text{qsort}}}{T_{\text{ppmqsort}}} \tag{1}$$

   where $\times$ denotes times.

3. Efficiency: Speedup/Core
   We would like to propose a new metric to measure the efficiency of any parallel QuickSort called Speedup per Core, S/c. $S/c > 1.00$ corresponds to superlinear Speedups. It can be due to cache locality/friendliness of the algorithm [23,24]. Similarly, [18] proposed a similar metric, Speedup/Thread instead. Higher thread counts h can lead to more opportunities to achieve more parallelism that will be limited by hardware.

4. %CPU Utilization U
   The metric can be obtained from the contents of $/proc/stat$ file which keeps track of statistics of all HyperThread-enabled/disabled CPU cores. This %CPU Utilization is based on *user-time* only.

5. Cache Refs/Cache Misses
   Perf [25] is a software tool that relies on a number of hardware/software counters to collect statistics of CPU resource usages with minimal overhead [26]. For this paper, Cache Ref, C, Cache Misses, C_m and other performance events are collected and averaged by 5 times to achieve high accuracy. In addition, a new metric called cache miss per second, $C_{m/s}$, can be obtained as shown in Eq. (2).

$$C_{m/s} = \frac{C_m}{T_{\text{ppmqsort}}} \tag{2}$$

   It can be beneficial to measure the number of cache misses per time unit especially for highly multithreaded programs. Larger $C_{m/s}$ may result in higher demands for memory bandwidth which will be presented next.

6. Branch Loads/Branch Load Misses
   Other important metrics of Perf are Branch Loads, B, and Branch Load Misses, B_m. They can be used to address the algorithm whose performance is limited by branch prediction, i.e., parallel QuickSort. Perf makes use of the hardware counters to measure the branch prediction unit. Similarly, a new metric called branch load misses per second, $B_{m/s}$, can be obtained as shown in Eq. (3).

$$B_{m/s} = \frac{B_m}{T_{\text{ppmqsort}}} \tag{3}$$

   $B_{m/s}$ can be regarded as number of branch mispredictions per time unit. Larger $B_{m/s}$ and $C_{m/s}$ may result in lower utilization of the long execution pipelines and frequent memory stalls which may affect %CPU Utilization U eventually.

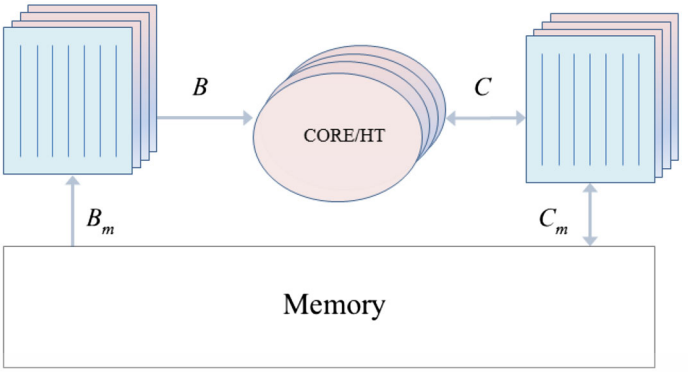


Fig. 2 Our shared memory/multiprocessor/multicore system model measured by Perf (*blue boxes*on the *left*- and *right*-hand sides are instruction and data caches, respectively. *HT* HyperThread (color figure online)

7. Average Memory Bandwidth M_{bw}
   The complex interactions between multicore architecture and characteristics of a parallel algorithm directly and indirectly impact both branch mispredictions and cache misses. In multithreaded programs, off-chip memory bandwidth is one of the important metrics that can be the performance bottleneck due to memory contention, memory saturation and bad allocation among cores.
   Many researchers use hardware performance counters to track the amount of consumed memory bandwidth while the multithreaded program is running [27,28]. The measurement accuracy depends on measurement events, number of counters and the characteristics of memory system including DDR2/DDR3, channels (interleaving), bus clock frequency, etc. However, we cannot directly measure the amount of memory bandwidth consumption. This paper rather proposes a performance model to estimate and evaluate as shown in Fig. 2. Our model can utilize a number of available events measured by Perf resulted in Average Memory Bandwidth.

$$M_{\text{bw}} = f(B_{m/s}, C_{m/s}) \tag{4}$$

   Assume that $B_{m/s}$ has negligible effects due to small program size and its recursive nature. The majority of memory bandwidth should be proportional to $C_{m/s}$. Therefore, the Average Memory Bandwidth, M_{bw}, can be calculated in terms of cache line size $|C_{\text{line}}|$ multiplied by $C_{m/s}$ as shown in Eq. (5).

$$M_{bw} = |C_{\text{line}}| \times C_{m/s}. \tag{5}$$

4.2 Experiment setup

The results reported in this paper are based on five multicore CPUs: Intel E5405 Harpertown, Intel E5520 Nehalem-EP, Intel i3-2100 Sandybridge, Intel i7-2600 Sandybridge, and AMD A6-3650 APU. Table 3 provides a summary of these multicore systems.

Table 3 Architectural details of multicore CPUs in our experiment

System Code name	Xeon E5405 Harpertown	Xeon E5520 Nehalem-EP	A6-3650 Llano	i3-2100 Sandybridge	i7-2600 Sandybridge
Clock (GHz)	2.00	2.66	2.6	3.1	3.4
$k \times c$	2×4	2×4	1×4	1×2	1×4
HyperThread	No	Yes	No	Yes	Yes
L1/L2 (KB)/core	32/3072	32/256	64/1024	32/256	32/256
L3 (KB)/socket	–	8192	–	3072	8192
RAM (GB)	4	12	16	8	16
RAM	DDR2-667	DDR3-800/1066	DDR3-1866	DDR3-1066/1333	DDR3-1066/1333
Others		Smart Cache 8 MB QPI2 5.86 GT/s	PCI express 2.0 16-way L2	Smart cache 3 MB DMI 5GT/s	Smart cache 8 MB DMI 5GT/s

Table 4 Parameter set of the experiments

Parameters	Values
Data types	Uint32, Uint64, Double
Size n (M)	10, 20, 50, 100, 200
Cases	Random, Worst
Cutoff u (K)	50, 100, 200, 500
Sockets k	1, 2
Cores c	1, 2, 4, 8
Threads h	1, 2, 4, 8, 16, 32
HyperThread	Enable, Disable
Optimization	o2, o3

In every system listed in Table 3, the operating system is 64-bit Ubuntu 14.04 kernel 3.13 LTS. The PPMQSort is compiled with GCC 4.8 and linked with OpenMP 3.0 library under *-fopenmp* option. The measurement tool, Perf version 4.2, is called using *perf stat -r 5 -e* to profile PPMQSort algorithm for 5 times.

Data sets are unsigned 32-bit integer (*Uint32*), unsigned 64-bit (*Uint64*) and 64-bit double precision floating point (*Double*). These are generated using the GCC *random()* function with two distributions: Random and Worst-case and in different number of elements, n = 10M, 20M, 50M, 100M, 200M. The first distribution contains random elements with small number of duplicates. The second distribution is generated such that the sequence seems to be sorted in a descending manner. However, for each distribution, the input sequence once generated is stored as a file. Therefore, both *PPMQSort* and sequential *qsort()* algorithms sort the same input sequences. All parameters are listed in Table 4.

4.3 Results and discussions

This subsection elaborates various aspects of the PPMQSort algorithm such as best Speedups, trade-offs between Speedup, Cutoff, and Thread, etc. Finally, the last two subsections are based on statistical analysis of Perf results.

4.3.1 The best Speedups

Table 5 tabulates the best Speedup, T_{qsort}, and T_{ppmqsort} of all systems based on various data types, cases, and optimizations. The T_{qsort} is obtained with the same experiment configuration as T_{ppmqsort}. It can be noticed that the best Speedups of Uint32 are higher than those of Uint64 and Double. Remark that i3-2100, i7-2600 and E5520 systems are HyperThread enabled. Therefore, their Speedups are higher than the number of physical cores. For a non-HyperThread 8-core Intel Xeon E5405 system, the best Speedup is as high as 7.75×. Due to limited space, best Speedups of Xeon E5404 are omitted. An exceptional case is the 4-core AMD A6-3600 whose Speedups are superlinear at 4.91× and 4.96× in Random and Worst cases, respectively. It can be observed that %CPU Utilizations approach 100 % in every Random-case configuration while those of Worst-case are significantly lower.

Table 5 Best Speedup S and other metrics on different data types and corresponding parameters

CPU	Opt.	Uint32				Uint64				Double			
		Random		Worst		Random		Worst		Random		Worst	
		o2	o3	o2	o3	o2	o3	o2	o3	o2	o3	o2	o3
i3-2100	n(M)	200	200	200	200	100	100	200	200	200	200	200	200
	u(K)	50	100	50	50	200	500	50	50	200	500	50	50
	S(x)	3.19	3.03	**3.79**	**3.67**	2.88	2.79	**3.44**	**3.35**	2.82	2.72	**3.69**	**3.63**
	T_{qsort}(s)	39.80	39.88	14.88	14.89	20.39	20.37	16.83	17.00	45.18	45.21	17.72	17.87
	T_{ppmqsort}(s)	12.49	13.14	3.92	4.04	7.07	7.31	4.89	5.06	16.00	16.64	4.79	4.91
	h (threads)	8	8	8	8	8	8	8	8	8	16	8	8
	U(%)	98	98	86	85	96	96	82	80	95	94	84	83
	M_{bw} (MB/s)	150	170	*319*	*300*	361	395	**700**	*669*	331	372	*619*	*608*
	$B_{m/s}$	*1.3e+8*	*1.4e+8*	4.9e+6	4.7e+6	*1.5e+8*	*1.6e+8*	3.2e+6	3.1e+6	*1.4e+8*	*1.4e+8*	3.2e+6	3.1e+6
A6-3600	n(M)	200	200	200	100	50	20	200	200	20	200	200	50
	u(K)	200	200	200	200	200	100	100	100	100	500	100	100
	S(x)	4.67	**4.91**	**4.96**	4.74	3.64	3.54	**4.62**	**4.37**	3.72	3.56	**4.60**	**4.43**
	T_{qsort}(s)	59.18	65.03	23.43	11.39	12.53	4.76	26,45	25.32	5.47	60.98	26.66	6.30
	T_{ppmqsort}(s)	12.67	13.25	4.71	2.40	3.45	1.35	5.71	5.78	1.47	17.15	5.79	1.42
	h (threads)	4	16	4	4	8	8	8	8	8	8	8	8
	U(%)	96	96	83	80	94	91	76	76	91	92	77	75
	M_{bw} (MB/s)	1.85	2.25	*4.39*	*5.98*	9.11	6.53	4.57	3.81	6.85	2.30	5.97	**20.33**
	$B_{m/s}$	*4.3e+6*	*6.8e+6*	2.4e+4	3.2e+4	*1.3e+7*	*1.7e+7*	7.9e+4	5.9e+4	*2.1e+7*	*1.4e+7*	6.4e+4	1.6e+5

Table 5 continued

CPU	Opt.	Uint32				Uint64				Double			
		Random		Worst		Random		Worst		Random		Worst	
		o2	o3	o2	o3	o2	o3	o2	o3	o2	o3	o2	o3
i7-2600	n(M)	200	200	200	200	100	100	200	200	200	200	200	200
	u(K)	100	200	500	500	200	200	500	500	200	500	500	500
	S(x)	5.65	5.35	**5.71**	**5.46**	**5.01**	**4.85**	4.66	4.48	4.79	4.61	**5.12**	**4.98**
	T_{qsort}(s)	32.53	32.53	12.36	12.36	16.76	16.78	14.03	14.08	37.05	37.15	14.79	14.85
	T_{ppmqsort}(s)	5.76	6.08	2.16	2.26	3.34	3.46	3.00	3.14	7.74	8.06	2.88	2.98
	h (threads)	16	16	8	16	16	16	16	16	16	16	16	16
	U(%)	92	92	78	75	90	90	72	70	87	86	75	73
	M_{bw} (MB/s)	145	148	*536*	*519*	266	256	**1042**	*993*	249	279	*971*	*958*
	$B_{m/s}$	*1.2e+7*	*1.5e+7*	4.9e+4	1.6e+5	*2.2e+7*	*2.1e+7*	9.6e+4	7.7e+4	*2.5e+7*	*4.4e+7*	9.5e+4	6.4e+4
E5520	n(M)	200	200	200	100	200	200	50	100	200	200	100	100
	u(K)	100	100	500	200	100	200	200	500	200	500	500	500
	S(x)	**12.29**	**11.20**	9.44	8.60	**11.34**	**10.96**	8.16	7.26	**9.43**	**9.06**	8.40	7.90
	T_{qsort}(s)	72.35	70.00	21.05	10.57	80.41	80.53	6.33	12.17	69.40	69.37	12.65	12.70
	T_{ppmqsort}(s)	5.89	6.25	2.23	1.23	7.09	7.35	0.78	1.67	7.36	7.66	1.51	1.61
	h (threads)	16	16	16	16	16	16	16	16	16	32	32	16
	U(%)	83	82	67	59	80	80	55	54	73	73	59	56
	M_{bw} (MB/s)	190	172	*751*	*595*	306	278	*1537*	*1390*	303	337	**1558**	*1278*
	$B_{m/s}$	*7.8e+8*	*4.0e+8*	1.9e+8	1.9e+8	*6.6e+8*	*6.1e+8*	2.9e+8	3.8e+8	7.1e+8	*1.2e+9*	7.2e+8	4.1e+8

Bold values indicate the maximum results of each CPU
Italics values indicate the maximum results of each Data type

PPMQSort can achieve high Speedup regardless of the data types and randomness even in the Worst case. It can be obviously noticed that Worst-case T_{qsort} and T_{ppmqsort} are always faster than those of Random-case with the same configuration. Furthermore, their Speedups are almost always higher than those of Random-case except in dual-socket systems, E5520 and E5405. PPMQSort can exploit the Branch Prediction Unit and caches well, although $seq\_partition()$ must execute a large number of comparisons and swappings on lines 13 and 15 in Algorithm 1. That means the Branch Prediction unit can learn/yield higher prediction rate than the Random-case due to remarkably low Branch Misprediction Rate $B_{m/s}$ except those of E5520 cases.

However, the highest memory bandwidth M_{bw} of Worst-case is always greater than Random-case because of its two to three times higher $C_{m/s}$. The highest M_{bw} of each system is highlighted in bold face. This also concurs with Eq. (5) that memory bandwidth of PPMQSort depends heavily on $C_{m/s}$. Despite 2–3 orders of magnitude lower $B_{m/s}$, %CPU Utilization U's of Worst-case are generally lower than those of Random-case in every configuration. It can be due to often memory stalls. On the other hand, high $B_{m/s}$ can be the performance bottlenecks in all Random-case as shown in Italic. Much lower M_{bw} can be observed.

In both Random and Worst cases, Cutoff u should fit the last level cache of each system. It can be noticed that the suitable Cutoff u for Uint32 ranges between 50 and 200 K elements. For Uint64 and Double cases, Cutoff u ranges between 200K and 500K elements or even bigger instead. The best Cutoff u of i3-2100 (Uint32) is 50 K by majority vote. It seems like 50 K of Uint32 can fit the private L2 cache (256 KB) in each core. The rest can almost fit Cutoff in their last level caches except in some cases of $u = 500\,\text{K}$ of Uint64 and Double.

4.3.2 Speedup S vs. Cutoff u and Thread h

For a given system and experiment configuration, Speedup S of PPMQSort is a function of Cutoff u and Thread h. As already listed in Table 5, the best S of i7-2600 system is $5.65\times$ at $n = 200\,\text{M}$ of Uint32, $u = 100\,\text{K}$, and $h = 16$ threads. Figure 3 shows a 3-D surface plot of PPMQsort with this configuration. Speedups can be visualized as surface height on the Z axis with colors according to the Color bar on the right-hand side. This plot presents the scalability and trade-offs between Speedups, Cutoffs, and Threads. While increasing thread count h, the Speedup S scales up for all Cutoffs. Therefore, high thread counts enable the PPMQSort to utilize the CPU cores more until S saturates. As discussed earlier in Sect. 4.3.1 Best Speedups, while varying Cutoff u, Speedup changes slightly as darker and lighter colors at the same thread count. This behavior in this 3-D surface plot agrees with the derived time complexity in Theorem 1, where u has been canceled out.

4.3.3 HyperThread vs. non-HyperThread CPUs

This subsection will contrast and compare Speedups of PPMQSort on Intel HyperThread and non-HyperThread CPUs with the same experiment configuration. Figure 4 illustrates Speedups (Line) and %CPU Utilization (Bar) of Intel HyperThread and non-HyperThread of PPMQSort (Uint32, Random-case, o2). The cyan bars and lines

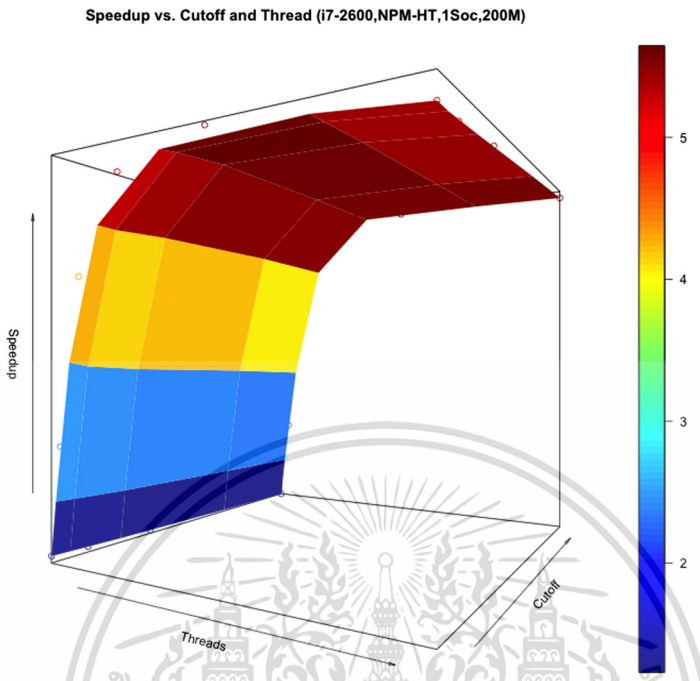


Fig. 3 Three-D Surface Plot of Speedup, S vs. Cutoff, u and Thread, h of PPMQSort on i7-2600 (Uint32, Random, o2, $n = 200$ M)

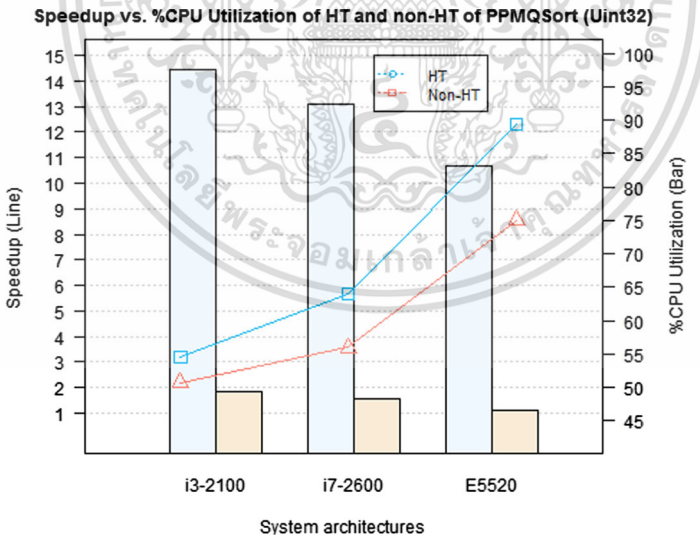


Fig. 4 Best Speedup, S (*Line, Left*) vs. %CPU Utilization, U (*Bar, Right*) of PPMQSort on Intel Hyper-Thread (HT) and non-HyperThread (non-HT) Platforms (Uint32, Random, o2)

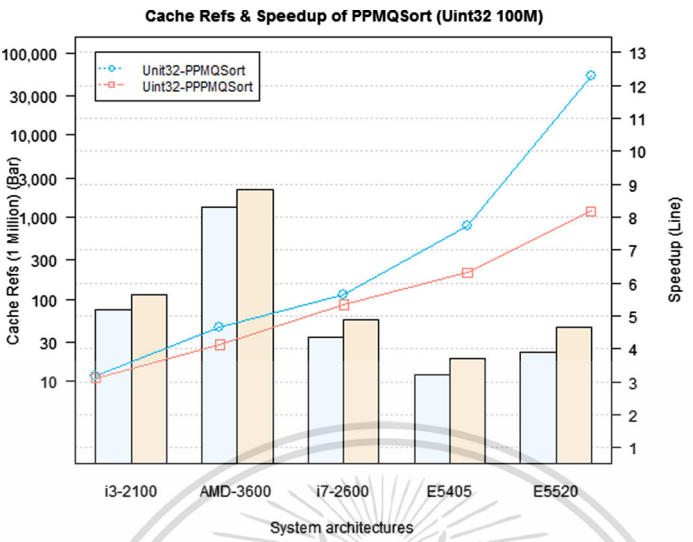


Fig. 5 Best Speedup (*Line, Right*) vs. Cache Refs (*Bar, Left*) of PPMQSort (*Cyan*) and PPPMQSort (*Brown*) on all Platforms (Uint32, Random, o2) (color figure online)

are of HT enabled while the brown ones are HT disabled. The Speedup differences between HT-enabled and HT-disabled systems are significant due to lower average %CPU Utilization U, despite the fact that other statistics are similar. It can be roughly estimated that HT can boost up the performance by more than 50 % which is comparable to [29].

4.3.4 PPMQSort vs. PPPMQSort

PPPMQSort is a minor variation of PPMQSort where its Merge Phase is parallelized with 2 threads on line 37 of Algorithm 1. To compare PPMQSort (Cyan) with PPPMQSort (Brown), their Speedups (Line) and Cache Refs (Bar) are plotted on all platforms (Uint32, Random, o2) with the same parameter set. Note that Cache Refs on the left Y axis are in logarithm and scaled by 1 million. It can be observed in Fig. 5 that PPMQSort can achieve better Speedups on the same experiment configurations due to significantly lower C.

Cache Refs are particularly high on AMD A6-3600 compared to other Intel systems. It might be due to fewer general-purpose Integer/Floaing-Point registers thus resulting in more register spills. However, AMD A6-3600 demands M_{bw} up to 20.3 MB/s as listed in Table 5 due to both large private L1 data cache (64 KB/core) and L2 cache (1 MB/core). In addition, its Branch Load Misses/sec $B_{m/s}$'s are considerably lower than those of Intel systems. Therefore, its PPMQSort Speedups can be superlinear in some configurations. The rest is comparable on all Intel systems.

4.3.5 Efficiency: Speedup/Core

Figure 6a, b depicts the scatter plot of S/c vs. c of non-HT and HT, respectively. It can be observed in Fig. 6a that PPMQSort can achieve $S/c \simeq 1.00$ or above (inside the

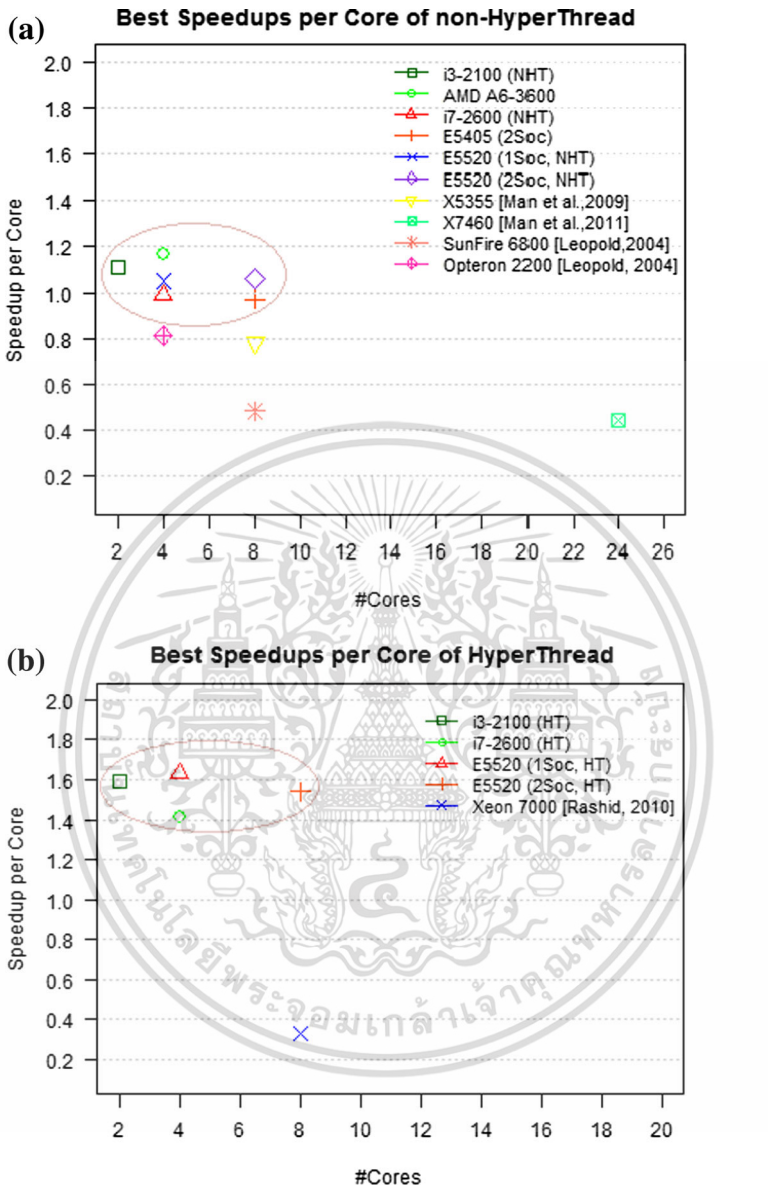


Fig. 6 Speedups per Core S/c of PPMQSort (inside the oval) vs. Others (Random, Uint32) (**a**) non-HyperThread (NHT) (**b**) HyperThread (HT)

oval) while others can only reach up to 0.8. The HyperThread-disabled i3-2100 and non-HT A6-3600 can achieve S/c at 1.11 and 1.17 resulting in superlinear Speedups because of high %CPU Utilization at 98 and 96, respectively. Similarly, Fig. 6b shows that PPMQSort can achieve $S/c \simeq 1.40$ or above while others can only reach up to 0.33. Some HT systems like i3-2100 and E5520 can achieve S/c at 1.59 and 1.63,

respectively, because of better %CPU utilization with 3-MB and 8-MB Smart Caches, respectively. It can be concluded that PPMQSort can exploit the CPU cores much better than other algorithms on both non-HT and HT architectures. Moreover, PPMQSort can be scablable on any non-HT/HT/multicore/multi-socket systems with $S/c \simeq 1.00$ and $S/c \simeq 1.50$ or better.

4.3.6 Comparison with previous implementations

Table 6 compares our PPMQSort with previous parallel QuickSort implementations to show that we can achieve the best performance at data size around 100M 32-bit Integers with respect to T_{par} and Efficiency, S/c. [18] reported only the Speedup based on Pthreads Library resulting in higher S/c that may not compare against Stdlib $qsort()$. In addition, he also did not report the run time. With respect to 11.58× Speedup, our PPMQSort on an 8-core HyperThread E5520 can clearly outperform pqsort1 of [15] on an 8-core Xeon X5355 using the same $qsort()$ benchmark. Although [13] can achieve 25.03x Speedup on a 32-core UltraSPARC, their efficiency is not quite good and the benchmark may not be Stdlib $qsort()$.

On the Uint64 data, Man et al. [16] reported their highest Speedup of 10.47× for 100M random on 24 cores and $T_{\text{par}} = 2.712$ s. With only 8 cores, PPMQSort can achieve $S = 10.24\times$ at 3.54 s with the same configuration. This can confirm that PPMQSort is more efficient than others.

4.3.7 Statistical analysis

Figure 7 shows matrix scatter plots between T_{ppmqsort} vs. Cache Refs C, Cache Misses C_m, Branch Loads B, and Branch Load Misses B_m of PPMQSort on E5520, Uint32, o2, all Cutoffs, data sizes and threads. The upper half, the diagonal, and the lower half of the matrix plot illustrate the scatter plots, the density, and the correlation value between/of them, respectively. Each dot in the scatter plot represents an experiment configuration.

The top-row figures show the regression analysis between parameters that Time or T_{ppmqsort} is proportional to Cache Refs C, Cache Misses C_m, Branch Loads B, and Branch Load Misses B_m, respectively. The green line is a linear regression generated by $lm()$ function in R Project (http://cran.r-project.org). The solid red line is a local regression smoothing (LOESS) mean fit line according to $loess()$ function. The red dotted lines above and below are positive and negative residual squares above and below the LOESS mean fit line, respectively.

C_m is highly correlated with C as indicated by correlation value $R_{C,C_m/s} = 0.91$. As expected, they are highly correlated. The higher C, the more C_m, and the longer the T_{ppmqsort}. Similarly, the higher B, the more B_m, and the longer T_{ppmqsort}. In addition, they are highly correlated with one another. Other systems in our experiment show similar behaviors.

Table 6 Comparison of PPMQSort with other parallel QuickSort implementations (100 M, Uint32, Random), *NA* Not Available

References	PPMQSort	[18]	[20]	[15]	[14]	[14]	[13]
Year		2013	2010	2009	2004	2004	2003
n(M)	100	80	64	100	60	100	100
S(x)	11.58	3.8	2.65	6.22	3.24	3.875	25.03
T_{seq}(s)	34.78	NA	15.9	28.81	24.3	37.2	139.22
T_{par}(s)	3.00	NA	6	4.63	7.5	9.6	5.56
S/c	1.45	1.9	0.33	0.26	0.81	0.48	0.78
Using qsort()	Yes	No	No	Yes	No	No	No
Architecture	×86	×86	×86	×86	×86	UltraSPARC III	UltraSPARC
GHz	2.66	2.66	NA	2.66	2.2	0.9	0.25
$k \times c$	2 × 4	1 × 2	4	2 × 4	1 × 4	1 ×8	32 × 1
HT	Yes	Yes	No	No	Yes	No	No
Cache	L3 8 MB	L2 3 MB	L2 4 × 2 MB	L2 2 × 2 MB	L2 2 × 1 MB	NA	L2 4 MB
Compiler	GCC -o2	G++	Intel C++ 9.1	GCC -o2	Intel C++ 8.1 -o3	Guide	NA
Library	OpenMP 3.0	pthreads	OpenMP 2.5	OpenMP 2.0	OpenMP 2.0	NA	NA
Remarks	Xeon E5520	Intel	Xeon 7000	Xeon X5355	Opteron 2200	Sun Fire 6800	Sun Enterprise
	MAC Pro 2010	Dual-Core	PowerEdge 6800		sort_omp_2.0	sort_pthreads_cv_1.0	10,000

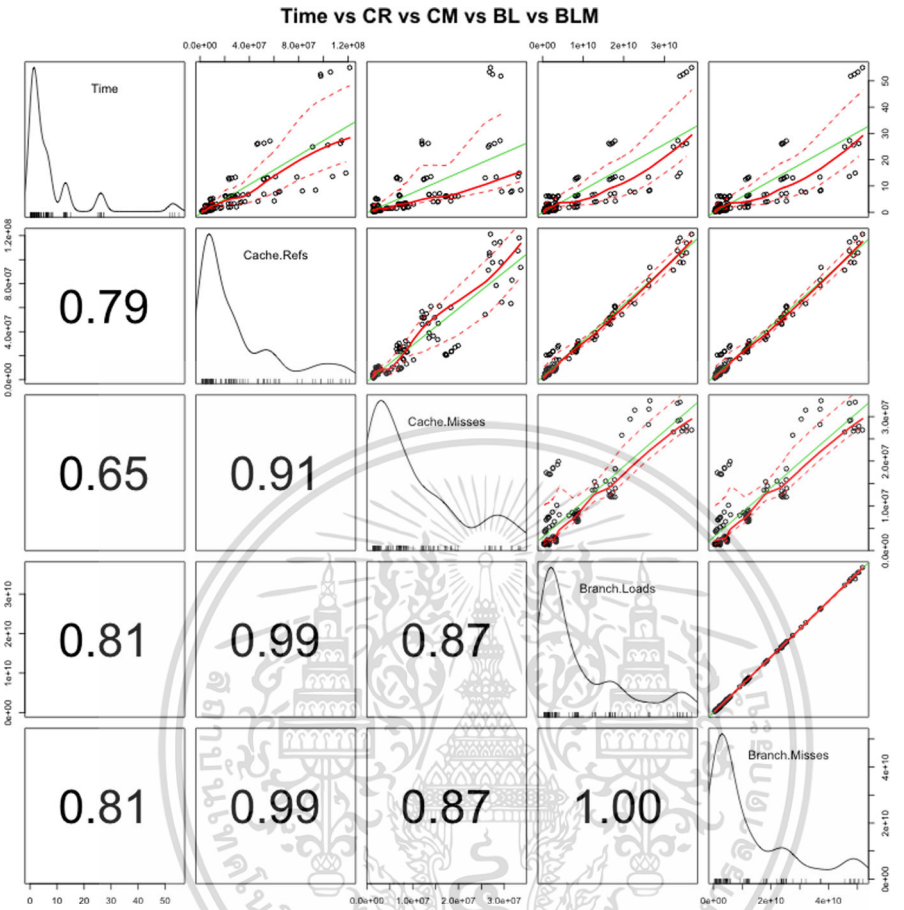


Fig. 7 Matrix Scatter Plots between Time T_{ppmqsort} vs. Cache Refs C, Cache Misses C_m, Branch Loads B, and Branch Load Misses B_m of PPMQSort on E5520, Uint32, Random, o2, all cutoffs, data sizes, and threads

4.3.8 Speedup vs. %CPU Utilization vs. Memory Bandwidth

Speedup S vs %CPU Utilization U vs. Cache Misses per Second $C_{m/s}$ and Branch Load Misses per Second $B_{m/s}$ of PPMQSort on i7-2600 can be depicted in Fig. 8. The configuration of this figure is random n = 200 M Uint32, o2 and h = 4–32 threads. Both C_m/s and B_m/s can be obtained by Eqs. (2) and (3), respectively.

As plotted, Speedup S is directly proportional to %CPU Utilization U because the correlation coefficient $R_{S,U}$ is 1.00. That means the higher %CPU Utilization, the better Speedup because all the forked threads can effectively execute with fewer memory stalls and pipeline stalls/flushes.

In general, cache misses can be due to cold misses, capacity misses, conflict misses, and coherence misses. Lower $C_{m/s}$ can be due to better cache locality resulted from suitable Cutoff u and Thread h of PPMQSort as shown in Fig. 8. On the other hand, lower B_m/s represents infrequent branch mispredictions thus more efficient pipelin-

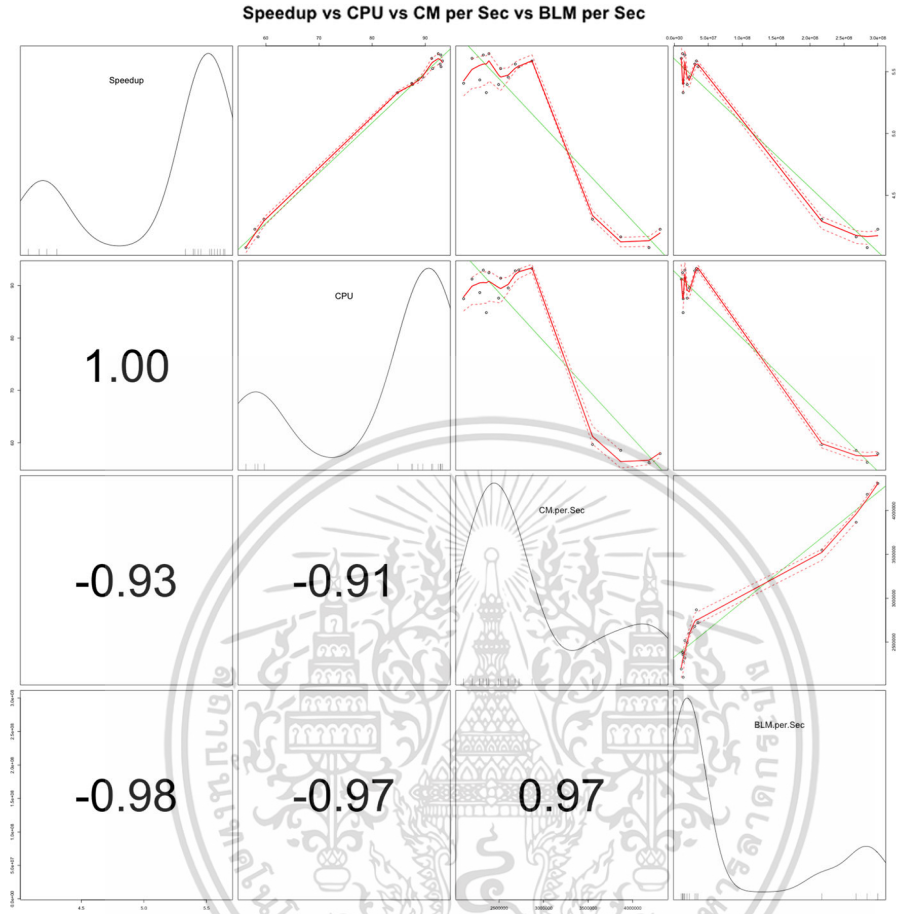


Fig. 8 Speedup S vs %CPU utilization U vs. $C_{m/s}$ and $B_{m/s}$ of PPMQSort on i7-2600 (Uint32, 200M, random, o2, 4–32 threads)

ing. Both frequent cache misses and branch mispredictions per unit time can lead to memory stalls and pipeline stalls and thus lower U. It can be reflected on both $R_{U,C_{m/s}}$ and $R_{U,B_{m/s}}$ approaching -1.00. That means U is negatively proportional to $C_{m/s}$ and $B_{m/s}$.

This figure confirms with the basic concept that memory is the bottleneck of the parallel algorithms [30] especially in the Worst case bounded by C_m/s. However, Random-case Speedups are limited by B_m/s rather than C_m/s. As shown in Table 5, Random-case B_m/s's are two to three orders of magnitude higher than those of Worst case with the same data size n in one-socket systems. For dual-socket systems, the gap is not that wide. This results in almost three times longer Random-case T_{qsort} and T_{ppmqsort} than those of Worst case in the same table. As pointed out by Eyerman et al. [31], the misprediction penalty of superscalar CPUs with Reorder Buffer and deep pipeline equals to the number of clock cycles to refill the front-end pipeline.

Other systems show similar behaviors as the i7-2600 system. We can conclude that the branch prediction unit is as performance critical as the memory hierarchy for parallel sorting algorithms due to the randomness of input data in modern multicore CPUs.

5 Conclusion

The proposed PPMQSort algorithm is different from others as the partitioning process has been simply parallelized since the beginning. The basic concept of the PPMQSort is to divide the input data array by half in parallel/recursively until the obtained partitions are up to Cutoff size u. These partitions can be locally cached and $qsort()$ them simultaneously by $h \geq c$ threads. Hence, the performance bottleneck can be eliminated.

PPMQSort is compatible with the Stdlib $qsort()$ since we use it as a benchmark. Various OpenMP 3.0 parallel constructs are employed and coded in C language. Performance of PPMQSort was evaluated on one AMD and four Intel CPUs running 64-bit Ubuntu Linux 14.4 LTS. In general, PPMQSort can achieve the best Speedup up to and beyond the number of CPU cores. In spite of the Worst cases's fast T_{qsort}, their Speedups are almost always greater than those of Random. For HyperThread CPUs, PPMQSort can get up to 50% Speedup increase over HT-disabled ones. In terms of efficiency, the PPMQSort can get Speedup/Core from 0.97 to 1.17 and from 1.41 to 1.63 on NHT and HT CPUs, respectively, and more superior than previous parallel QucikSort algorithms.

Statistical analysis of PPMQSort shows that T_{ppmqsort} is proportional to Cache Misses and Branch Load Misses. On the other hand, its Speedup S is proportional to %CPU Utilization U and limited by $B_{m/s}$ and $C_{m/s}$. The proposed system performance model can estimate memory bandwidth required by the PPMQSort. In addition, Branch Prediction Units are as performance critical as the memory hierarchy for PPMQSort algorithm due to randomness of input data.

For future work, PPMQSort should be optimized further to support thread affinity/cache locality and minimize cache coherence misses even more. The performance model and average memory bandwidth shall be analyzed and fine-tuned to support a variety of algorithms/programs. To serve big data, task scheduling and load balancing strategy are investigated by mixed CPU, memory, and I/O-intensive [32].

In addition, on-chip and off-chip graphics processing unit (GPUs) should be investigated whether PPMQSort can be applied to exploit a massive number of GPU cores as it has been done on multicore CPUs.

Acknowledgments The authors wish to thank Mr. Apisit Rattanatranurak and Mr. Surapong Towtiamton for experiments and discussions on some of the algorithms in this paper. The authors wish to thank the reviewers for their insightful comments which greatly improved the paper.

References

1. Hoare CAR (1962) Quicksort ACM 4:321
2. Sedgewick R (1978) Implementing quicksort program. Commun ACM 21(10):847–857

3. Mishra AD (2009) Selection of best sorting algorithm for a particular problem. Master's thesis, Thapar University, Computer Science and Engineering Department
4. Bhandarkar SM, Arabnia HR (1995) The hough transform on a reconfigurable multi-ring network. J Parallel Distrib Comput 24(1):107–114
5. Arabnia HR, Bhandarkar SM (1996) Parallel stereocorrelation on a reconfigurable multi-ring network. J Supercomput 10(3):243–269
6. Bhandarkar SM, Arabnia HR (1997) Parallel computer vision on a reconfigurable multiprocessor network. IEEE Trans Parallel Distrib Syst 8(3):292–309
7. Koch D, Torresen J (2011) Fpgasort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '11. ACM, New York, pp 45–54
8. Mueller R, Teubner J, Alonso G (2012) Sorting networks on fpgas. VLDB J 21(1):1–23
9. Casper J, Olukotun K (2014) Hardware acceleration of database operations. In: Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays, FPGA '14. ACM, New York, pp 151–160
10. Capannini G, Silvestri F, Baraglia R (2012) Sorting on gpus for large scale datasets: a thorough comparison. Inf Process Manag 48(5):903–917
11. Xiaochen T, Rocki K, Suda R (2013) Register level sort algorithm on multi-core simd processors. In: Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms, p 9. ACM
12. Heidelberger P, Norton A, Robinson JT (1990) Parallel quicksort using fetch-and-add. IEEE Trans Comput 39(1):847–857
13. Tsigas P, Zhang Y (2003) A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In: 11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP 2003). Genoa, pp 372–381
14. Sub M, Leopold C (2004) A user's experience with parallel sorting and openmp. In: Proc. of the 6th European Workshop on OpenMP (EWOMP 2004). Stockholm
15. Man D, Ito Y, Nakano K (2009) An efficient parallel sorting compatible with the standard qsort. In: International Conference on Parallel and Distributed Computing, Applications and Technologies. Hiroshima, pp 512–517
16. Man D, Ito Y, Nakano K (2011) An efficient parallel sorting compatible with the standard qsort. Int J Found Comput Sci 22(5):1057–1071
17. Kim KJ, Cho SJ, Jeon JW (2011) Parallel quick sort algorithms analysis using openmp 3.0 in embedded system. In: 11th International Conference on Control, Automation and Systems. KINTEX, Gyeonggi-do, pp 757–761
18. Mahafzah BA (2013) Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. J Supercomput 66:339–363
19. Bingmann T (2015) Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. Algorithmica, pp 1–52
20. Rashid L, Hassanein WM, Hammad MA (2010) Analyzing and enhancing the parallel sort operation on multithreaded architectures. J Supercomput 53:293–312
21. Saleem S, Lali MIU, Nawaz MS, Nauman AB (2014) Multi-core program optimization: parallel sorting algorithms in intel cilk plus. Int J Hybrid Inf Technol 7(2):151–164
22. Architecture Review Board (2014) The openmp api specification for parallel programming. http://www.openmp.org
23. Gustafson JL (1990) Fixed time, tiered memory, and superlinear Speedup. In: Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)
24. Helmbold DP, Mcdowell CE (1990) Modeling Speedup (n) greater than n. IEEE Trans Parallel Distrib Syst 1(2):250–256
25. Weaver VM (2013) Linux perf event features and overhead. In: Second International Workshop on Performance Analysis of Workload Optimized Systems (FastPath 2013). Austin
26. Zhang Y, Li ZP, Cao HF (2015) System-enforced deterministic streaming for efficient pipeline parallelism. J Comput Sci Technol 30(1):57–73
27. Grama A, Gupta A, Karypis G, Kumar V (2003) Introduction to parallel computing. 2nd ed. Pearson Education Limited
28. Akhter S, Roberts J (2006) Multi-core programming increasing performance through software multithreading. Intel Press, Hillsboro

29. Barker KJ, Davis K, Hoisie A, Kerbyson DJ, Lang Mike, Pakin Scott, Sancho Jose Carlos (2008) A performance evaluation of the nehalem quad-core processor for scientific computing. Parallel Process Lett 18(4):453–469
30. Wulf WA, McKee SA (1995) Hitting the memory wall: implications of the obvious. SIGARCH Comput Archit News 23(1):20–24
31. Eyerman S, Smith JE, Eeckhout L (2006) Characterizing the branch misprediction penalty. In: IEEE International Symposium on Performance Analysis of Systems Software (ISPASS 2006). Austin, pp 48–58
32. Qureshi K, Majeed B, Kazmi JH, Madani SA (2012) Task partitioning, scheduling and load balancing strategy for mixed nature of tasks. J Supercomput 59(3):1348–1359