KTH Computer Science
and Communication

Impacts of data structures and algorithms on multi-core efficiency

MARWA ABDUL-MONEM AL-SHANDAWELY

Master's Thesis at NADA
Supervisor: Erwin Laure
Examiner: Michael Hanke

TRITA xxx yyyy-nn

Abstract

The advent of multi-core processors provides a massively huge computation power for scientific and business applications. Therefor the need for writing efficient optimized codes and data structures became more essential in order to achieve the expected performance and utilization of the computing resources. The problem with optimization lies not only with the extra effort to improve the coding but also with the dependence on the underlying architecture including memory bandwidth, cache organization and processors topology as well.

The purpose of this work is to study the impact of optimizing data structures and algorithms not only to increase the degree of parallelism but in order to achieve the expected performance and efficiency of the available multi-core system.

Finding potential performance bottlenecks is the first and most important step in the optimization process. The use of cache and memory profiling tools during the code analysis is practical and crucial specially when dealing with complex structures, separated code files and multi-threaded applications. The tool used in this study was Acumem ThreadSpotter.

We studied two different codes as our case studies. The first was the famous simple Gaussian Elimination (GE) with partial pivoting and we were able to increase the speed up from at most two into more than twenty by introducing new loop nesting even for large input sizes. The second case study was optimization for a numerical algorithm based on boundary integral formulation and we were able to increase performance by at least factor of four compared to the original code.

Referat

Påverkan av datastrukturer och algoritmer på effektiviteten hos system med flera kärnor

Processorer med flera kärnor tillhandahåller stor beräkningskraft både för vetenskapliga och företags tillämpningar. Därför finns ett behov av att skriva effektiva och optimerade koder och datastrukturer för att uppnå den förväntade prestandan och utnyttjandet av beräkningsresurserna. Svårigheten med optimeringen ligger inte bara i ansträngningen med att förbättra koden, utan också i beroendet på den använda datorarkitekturen så som minnesbandbredd, cache organisation och processor topologi.

Syftet med det här arbetet är att studera effekten av optimerade datastrukturer och algoritmer, inte bara för att öka graden av parallellism, utan också för att uppnå den förväntade prestandan och effektivitetet hos det tillgängliga multikärnesystemet.

Att hitta möjliga prestandaflaskhalsar är det första, och viktigaste, steget i optimeringsprocessen. Hjälp från verktyg för cache profilering är avgörande, speciellt när man har att göra med komplexa strukturer, uppdelade kodfiler och mångtrådade program. Profileringsverktyget som använts i det här arbetet är Acumem ThreadSpotter.

Vi använde två olika program för våra fallstudier. Det första var Gausselimination (GE) med partiell pivotering och vi kunde snabba upp taktökningen (speed-up) från som mest två till mer än 20 genom att införa nya nästlade slingor (loopar) även för stora indata storlekar. Det andra testfallet var optimering av en numerisk algoritm som bygger på gränsvärdesintegralformuleringen och vi lyckades öka prestandan mer än fyrfaldigt.

To my dear family

Acknowledgements

I would like to express my gratitude to my teacher and supervisor, Dr. Erwin Laure, for his continuous guidance, patience, and encouragement that have been invaluable on both academic and personal levels. His insightful advice and unsurpassed knowledge kept me focused on my goals.

I acknowledge the help and support given to me by Dr. Katarina Gustavsson at NADA/KTH and deeply thank Oana Wiklund, the PhD. student at NADA and the provider of the second case study code, for her support, background and contributions.

I would like to take this opportunity to thank DD3003 teachers at ITC/Uppsala university, Prof. Erik Hagersten, Dr. Jarmo Rantakokko and Dr. David Black-Shaffer. Their insightful experience and deep knowledge of multi-cores provided me with enthusiasm and comprehension.

I want to take the opportunity to thank system support members at PDC/KTH especially to Elisabet Molin and Izhar Ul-Hassan for the help and time they gave to me to facilitate the use of the resources at PDC during the experiments.

I am indebted to all my teachers at NADA, specially to Prof. Jesper Oppelstrup, Dr. Lennart Edsberg, Dr. Michael Hanke, Prof. Axel Ruhe and Ms. Carina Edlund for their deep knowldge and friendliness that made the environment at the class room both constructive and fun. Their fruitful discussions, and energy have been a tremendous source of inspiration.

Finally, I owe my deepest gratitude to my husband Ahmad and to my daughters Yara and Awan for their love and support at all times. I am most grateful to my parents and brothers for helping me to be where I am now.

Contents

Chapter 1

Introduction

Over the past 30 years the speed of processors –in terms of its clock frequency– has tremendously increased. While the original IBM PC processor (that appeared on the market in 1981) had a clock rate of 4.77 MHz, the Intel Pentium 4 model (2002) was introduced as the first CPU with a clock rate of over 3.06 GHz. It was thus in many cases sufficient and indeed cost-effective to upgrade to the latest processor generation in order to increase the speed of applications. However, these days the clock rate of processors is no longer increasing and instead even decreasing to avoid excessive power consumption and associated cooling needs. Thus, buying a new computer does not necessarily speed up programs and give the possibility to gain better performance any longer.

Another issue impacting the performance of applications are limitations in the speed of moving data from main memory to computational units. The Front Side Bus (FSB) is the most important bus to consider when talking about the performance of a computer. The FSB connects the processor (CPU) in the computer to the system memory. The faster the FSB is, the less time needed to get data to the processor. For instance, the Pentium 4 with 3.06 GHz processor has a FSB of 533 MHz.

Due to this huge gap between processor and memory speed, data access latencies, and memory bandwidth issues the number of arithmetic operations alone is no longer an adequate mean of describing the computational complexity of an algorithm. For applications that use large datasets, the memory wall becomes unavoidable and a main performance concern if we can not overlap memory access with computational work.

To improve memory access times modern computers now have small very fast repositories – called caches– that hold copies of data and instructions that will be needed in advance to avoid fetching them from slow main memory. While the average latency to fetch data from local main memory can be 50-70 ns (nano seconds), the average latency of cache is 0.5-2.5 ns. So if the processor uses data that is available in the cache it will take less time to access that data than in the case the data needs to be fetched from the slower main memory. However the size of

caches is practically limited to 8-16 MB so it cannot hold all data we need and the responsibility to fill the cache with useful data and make best use of it lies on the hand of the programmer.

Multi-core systems came to offer a solution between the increasing demands for computational powers and the limitations that prevent the single processor capabilities to be enhanced. Multi-core concept is by multiplying the processors (cores) –that may have less frequency– one will hopefully multiply the processing power with the advance of consuming less power. Applications will then have to exploit the parallelism available in multi-core processors.

Writing cache-aware applications is not an auxiliary choice but a necessity when we start using a multi-core system. With several processors sharing computer resources, it is easy to lose rather than gain performance if caches are not wisely used. Cache coherence, threads interactions, and synchronizations are among many of the performance bottlenecks that programmers of multi-core applications face.

Cache optimization includes finding critical parts of the code that is responsible for most of cache misses and try to enhance it to run more efficiently. It is not always an obvious or intuitive process and in most cases includes trade-off between other resources. Although optimizing compilers can do a good job of arranging instructions and generating high quality code, unfortunately, current compilers can not perform highly sophisticated cache-based transformations.

The aim of this study is to show how applications can benefit from cache optimization and impacts of data layout and access mechanism on the efficiency of multi-core system.

The rest of this document is divided into two parts. The first part contains three chapters. Chapter 2 will show brief background discussions about the multi-core architecture, the motivation for it, elements and programming concepts, the key issues involving migration from a single processor- to a multi-core system and explain the performance factors including speed-up and efficiency. It will also mention the hardware specifications, compiler version and library modules of the system that we used in the experiments. In Chapter 3 we will discuss the major optimization techniques with a focus on techniques to enhance data access and layout for efficient cache usage. Chapter 4 will discuss the need for optimization tools and give a brief description about **Acumem ThreadSpotter**$^{(TM)}$, the tool that we used in this study. The second part of this document will contain two case studies we used to show the effect of applying cache optimization techniques. The first case study is the famous Gaussian Elimination Algorithm for factorizing a dense matrix into lower and upper triangular matrices, know as LU factorization, which is discussed in Chapter 5. The second case study as described in Chapter 6 is a numerical algorithm that applies the boundary integral method to solve Stokelet flow and uses the iterative method GMRES to find solution of the inverse problem. Finally we will end up with the general conclusions and suggested future work in Chapter 7.

Part I

Literature Review

Chapter 2

Multi Core Architecture

During the last decades, computer systems gained more performance due to the steady increase of the processor clock rates and/or due to memory size, bandwidth and speed. So in that time only by upgrading the hardware to more powerful system, the performance gain in terms of speedup was increased without any programming effort. Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004. The runtime of a program is equal to the number of instructions multiplied by the average time per instruction. Maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction. An increase in frequency thus decreases runtime for all computation-bounded programs. The gain was not only to increase performance of existing software but also to expand the expectations, resolutions and complexity of the problems needed to be solved.

However due to physical limitations of semiconductor-based microelectronics and power dissipation, it became more difficult to increase processors speed. The power consumption by a chip is given by the equation [1]:

$$P = C \times V^2 \times f$$

where P is power, C is the capacitance being switched per clock cycle (proportional to the number of transistors whose inputs change), V is voltage which is proportional to the processor frequency, and f is the processor frequency (cycles per second). So we can conclude that:

$$P \propto f^3$$

Increasing processor power consumption led ultimately to Intel's May 2004 cancellation of its Tejas and Jayhawk processors [2], which is generally cited as the end of frequency scaling as the dominant computer architecture paradigm.

Yet the demand for faster applications is continuously increasing. The solution for that is going parallel. The principle concept was that large problems can often be divided into smaller independent ones, which are then solved concurrently or in parallel. Then we need multiple workers or processing elements to execute those

independent jobs. Multi-core architecture is what we mean by multiple processing elements - *processors* - within a single machine. These processors differ from super-scalar processors, which can issue multiple instructions per cycle from one instruction stream (thread); by contrast, a multi-core processor can issue multiple instructions per cycle from multiple instruction streams. The difference between current implementations of multi-core systems can be illustrated in means of complexity of the cores, their communication and coupling, the design of memory hierarchy and capacity, and the associated processing model.

The rest of this chapter will talk about the elements of a multi-core system, give overview about its recent technology and programming paradigms. It will also mention the hardware specifications we used in the experiments and ends up with metrics of measuring multi-core performance.

2.1 Multi-Core systems

2.1.1 Processors

Always referred to as *Central Processing unit* (**CPU**) and it is the part of the computer system that executes tasks. The functionality of the processor is to *fetch* instructions and data, *decode* the instruction, *execute* it and then *writeback* result. These steps are called the **instruction pipeline**. The performance of the processor is determined by the clock rate and the so called MIPS rate. The clock rate referring to the frequency of the processor or the number of cycles generated by the clock per second. Clock rate - measured in mega hertz (MHz= $10^6 cycles/second$) or giga hertz ($GHz = 10^9 cycles/second$)- gained enormous rates during the last 40 years. From approximately 1 MHz in the late 70s to up 6 GHz nowadays in the IBM POWER processors.

Another term to measure the processor performance is the MIPS rate or *Million Instructions Per Second*. This term is dependent on the clock rate, the instruction set given by the processor, CPU implementation and control , and cache and memory hierarchy [3, 4].

$$MIPS = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6} = \frac{f \times I_c}{C \times 10^6}$$

Where I_c is the instruction count, T is the execution time, f is the clock rate, CPI is the cycles per instructions, and C is the total number of cycles needed to execute the program.

Some instructions can require several cycles to be executed and/or some instructions can be *independent* and thus can be executed *in parallel*. Often, it is possible to overlap different instructions' execution stages if they do not conflict with the requested unit of execution or operands in a pipelined fashion on the instruction pipeline. For example a processor with two *adders* can execute two independent addition operations and load/store operation in same time. Time needed to execute a whole instruction can be different for the same instruction depending on where

the manipulated data are. Fetching operands from caches is much faster compared to fetching data faraway in the memory or even further away, e.g. in the disk. We will come to that in more details in the discussion on caches and memory.

A multi-core processor is a processing system composed of two or more independent cores. The cores are typically integrated onto a single integrated circuit die (known as a chip multiprocessor or CMP), or they may be integrated onto multiple dies in a single chip package. Replication of the cores can be done by two mechanisms, either by using few complex cores or less complex several ones. Homogeneous multi-core systems include only identical cores, unlike heterogeneous multi-core systems.

The processors are interconnected using common network topologies like bus, ring, 2-dimensional mesh, or crossbar. Just as with single-processor systems, cores in a multi-core systems may implement architectures like super-scalar, vector processing, SIMD, or multi-threading. Furthermore, the cores (or groups of them) share some circuitry, like the L2 cache and the interface to the front side bus (FSB). More details about the those components will be presented later in the discussion about memory and cache hierarchy.

2.1.2 Memory Hierarchy

In a shared-memory multi-processor system the way the memory is organized - and so accessed by the processors- can be either *uniform memory access* (UMA) , *non-uniform memory access* (NUMA) or hyprid between the two models [3, 4, 5]. UMA gets its name from the fact that each processor must use the same shared bus to access memory, resulting in a memory access time that is uniform across all processors. Note that access time is also independent of data location within memory. That is, access time remains the same regardless of which shared memory module contains the data to be retrieved. Diagram of UMA architecture can be seen in Figure 2.1.

The problem with the UMA model that it is not scalable. As the number of processors increase, the interconnection bus becomes a hot spot and with several requests to the memory the network traffic becomes congested.

Instead, in NUMA each processor has its *own local* memory module that it can access directly and with a distinctive performance advantage. At the same time, it can also access any memory module belonging to another processor using a shared bus (or some other type of interconnect). A diagram of NUMA architecture can be seen in Figure 2.2.

If data resides in local memory, access is *fast*. If data resides in remote memory, access is *slower*. So the advantage of the NUMA architecture as a hierarchical shared memory scheme is its potential to improve average case access time through the introduction of fast, local memory. By providing each node with its own local memory, memory accesses can take place in parallel and avoid throughput limitations and contention issues associated with a shared memory bus that happened in the UMA.

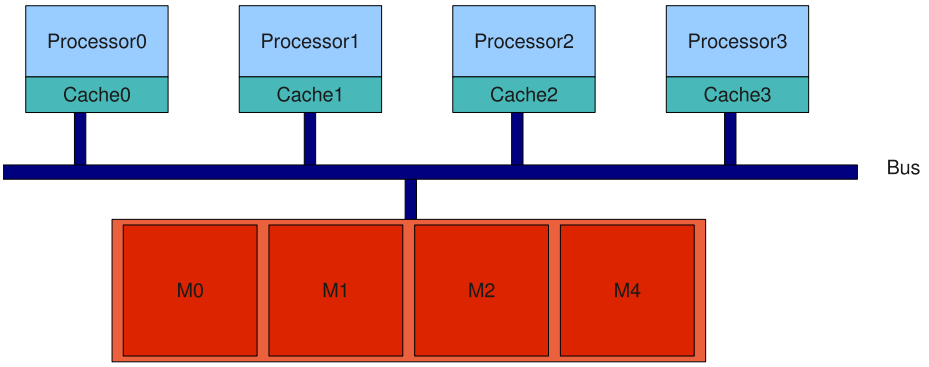

Figure 2.1. UMA memory model

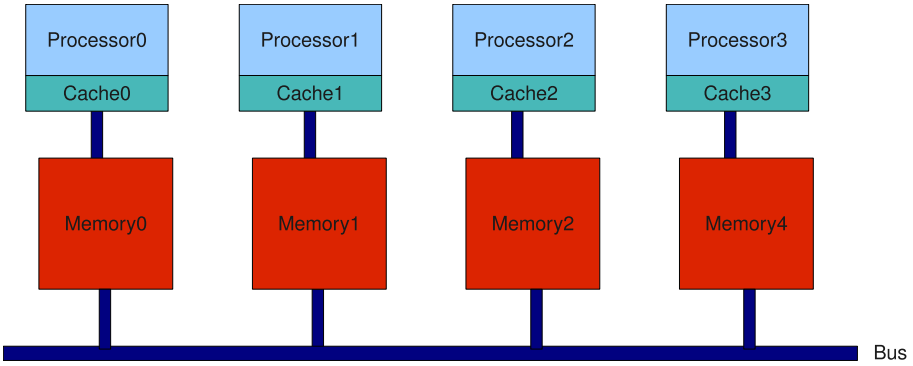

Figure 2.2. NUMA memory model

The downside of the NUMA architecture, however, is the cost associated when data is not local to the processor. In the NUMA model, the time required to retrieve data from an adjacent node within the NUMA model will be significantly higher than that required to access local memory. Furthermore, the time required to retrieve data from a non-adjacent node may be even higher, complicating memory performance and generating a hierarchy of access time possibilities. In general, as the distance from a processor increases, the cost of accessing memory increases.

Modern multiprocessor systems mix these basic architectures as seen in Figure 2.3. In this complex hierarchical scheme, processors are grouped by their physical location on one or the other multi-core CPU package or "node". Processors within a node share access to memory modules as per the UMA shared memory architecture. At the same time, they may also access memory from the remote node using a shared interconnect, but with slower performance as per the NUMA shared memory

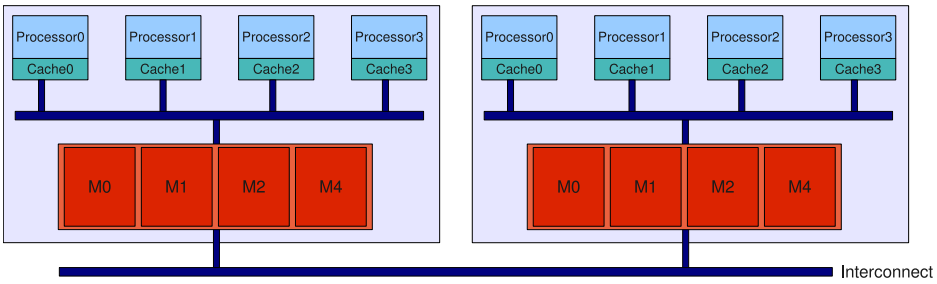

Figure 2.3. Multicore diagram with 8-cores divided into two modules

architecture.

Memory speed

The Front Side Bus (FSB) is the most important bus to consider when you are talking about the performance of a computer. The FSB connects the processor (CPU) in your computer to the system memory. The faster the FSB is, the faster you can get data to your processor. The speed of the front side bus depends on the processor and motherboard chipset you are using as well as the system clock. The Pentium4 with 3.06 GHz processor has a FSB of 533 MHz. Due to technology limitation memory speed can not get to same speed as processors. From 1986 to 2000, CPU speed improved at an annual rate of 55% while memory speed only improved at 10%. Given these trends, it was expected that memory latency would become an overwhelming bottleneck in computer performance [6]. While a processor is able to execute several instructions per ns, an access to RAM memory may take 50-70 ns.

Fetching data is even slower when the requested memory belongs to another processor that lies on another module. The term "memory wall" is introduced to describe the growing disparity of speed between CPU and memory outside the CPU chip. To solve the problem computer designers have introduced *cache* memories.

Caches

Caches are small, extremely fast memories between the processor and the quite slow main memory. The aim of the caches is to hide memory latency and increase availability of data and instructions. Most computer systems nowadays have three independent caches: data cache, instruction cache, and TLB (*translation lookaside buffer*). The average latency of cache is 0.5-2.5 ns. So the more data available in the cache, the less time needed for memory access. In order to increase availability and avoid overwriting data that will be needed in future; more than one level cache can be used. Hardware prefetching of data is another concept by which instead of just fetch the needed data, data that are *close* is also fetched as it is more likely be needed later on to increase availability of data and overlap data loading with processing.

Often not just a single cache is used, but a hierarchy of caches of increasing size and decreasing speed. The top level which is smallest, fastest and most expensive is called L1. Usually it is 64 KB with latency of 3 cycles. The next level -called L2 - may have a 1 MB cache with a latency of 15 cycles. Some computers have also third level, L3 cache.

When introducing multi-core, it was convenient to dedicate first level cache (L1) locally to each core and then second level cache (L2) shared between two cores. Further discussion on the impact of caches on performance will be presented in Chapter 3.

Taking into account that caches works as small expensive repository, we can not expect all the data we need to be in cache all the time. The size of the cache -measured in MB- is divided into what so called *cache lines*. Common sizes for cache lines are 32,64, and 128 bytes.

There are three types of caches when we consider the way of mapping a memory address space into cache entry [3]:

- **Direct mapped** caches: Using the least significant bits in the address as index, each address is placed in specific one location in the cache. Each memory address that share the same index will be mapped to the same location in the cache.

- **Fully associative** caches: Any memory address can be mapped to any entry in the cache. It require mechanism to search the whole cache entries to determine if the requested address exists in cache or not. This mapping is only suitable for small caches.

- **N-way set associative** caches: It is a compromise between the direct mapped and fully associative designs. Cache entries are divided into sets with size N, typically N is either 2,4,8, etc. Using tags in the address, each memory address is assigned a set. Within each set the cache is associative.

Conceptually, we can say that the direct mapped and fully associative caches are just "special cases" of the N-way set associative cache. Direct mapped cache is a "1-way" set associative cache. On the other hand, suppose we make "N" to be equal to the number of lines in the cache, then we only have one set, containing all of the cache lines, and every memory location points to that huge set. This means that any memory address can be in any line, and we are back to a fully associative cache.

In order to determine which is best performing we need to define two factors:

- **Hit time**: Time needed to determine if a memory address exists in a cache entry or not.

- **Hit ratio**: The likelihood of the cache containing the memory addresses that the processor wants.

When we consider the direct mapped caches, Hit time is best since no search is needed. Each address is found in exactly one location in the cache. Hit ratio will be zero, because with every access we need to replace that entry. Although we have *empty* entries but they will not be used.

On the other hand, fully associative caches need specialized hardware to do the searching and a performance penalty is incurred. And this penalty occurs must be added to determine which of the various lines to use when a new entry must be added (usually some form of a *least recently used* LRU algorithm is employed to decide which cache line to use next). All this overhead adds cost, complexity and execution time. But this type of caches has the best hit ratio because any entry in the cache can hold any address that needs to be cached. This means the problem seen in the direct mapped cache disappears, because there is no dedicated single entry that an address must use. Until the cache is *full* we can use the entries without replacement.

The set associative cache is a good compromise between the direct mapped and set associative caches. Let's consider the 8-way set associative cache. Here, each address can be cached in any of 8 places. This can raise the hit ratio from 0% to near 100%! As for searching, since the set only has 4 lines to examine this is not very complicated to deal with, although it does have to do this small search, Again, some form of LRU (least recently used) algorithm is typically used.

When a cache line (entry) requested by the processor is found in the cache, we say that is *cache hit*, otherwise it is a *cache miss* when we need to go next cache level or worst further in the memory hierarchy to fetch the request address. Cache misses can be classified into three categories (known as the Three Cs):

- **Compulsory misses** - also **Cold misses**- : are those misses caused by the first reference to a datum. Cache size and associativity make no difference to the number of compulsory misses. Prefetching can help here, as can larger cache block sizes (which are a form of prefetching).
- **Capacity misses**: are those misses that occur regardless of associativity or block size, solely due to the finite size of the cache. Note that there is no useful notion of a cache being "full" or "empty" or "near capacity": CPU caches almost always have nearly every line filled with a copy of some line in main memory, and nearly every allocation of a new line requires the eviction of an old line.
- **Conflict misses**: are those misses that could have been avoided, had the cache not evicted an entry earlier. Conflict misses can be due to the particular amount of cache associativity, and/or due to the particular victim choice of the replacement policy.

In a multi-threaded application, new type of misses can arise due to the sharing between different threads. When multiple processors with separate caches share a

common data, it is necessary to keep the caches in a state of *coherence* by ensuring that any shared operand that is changed in any cache is changed throughout the entire system. This situation causes what is known as **Communication misses**.

Cache coherence is done in either of two protocols: through a *directory-based* or a *snooping protocol*.

In the *directory-based* protocol, the data being shared is placed in a common directory that maintains the coherence between caches . The directory acts as a filter through which the processor must ask permission to load an entry from the primary memory to its cache. When an entry is changed the directory either updates or invalidates the other caches with that entry.

In the *snooping* protocol, all caches on the bus monitor (or snoop) the bus to determine if they have a copy of the block of data that is requested on the bus. Every cache has a copy of the sharing status of every block of physical memory it has.

Cache misses and memory traffic due to shared data blocks limit the performance of parallel computing in multi-cores. Since a single cache line can contain several data, if two processors operate on independent data in the same memory address region they might end up in a single line, the cache coherency mechanisms in the system may force the whole line across the bus or interconnect with every data write, forcing memory stalls in addition to wasting system bandwidth, causing what is known as **false sharing**. False sharing is difficult to detect and can dangerously cause a performance degradation.

To increase availability and try to lower the *miss ratio*, instead of fetching specific data in a certain address, a chunk of data (cache lines) near that address is loaded into the cache as it is more probable to be needed in the near future. In addition, for the types of applications where constant-stride accesses are dominant, the compiler is quite successful at understanding the access patterns and hence able to predict what data will be needed and prefetch it in advance. Meanwhile, in order to avoid eviction of data in use, LRU is the common cache lines replacements policy. This kind of behavior is known as *data locality*. The term refers to two kinds of locality:

- **Temporal locality**: the program uses the same data (*cache lines*) that are recently used and most likely still in the cache.

- **Spatial locality**: the program uses data close to recently accessed locations that are most likely fetched to the cache within the chunk of cache lines loaded.

So a program that exhibits data locality will then have a low miss rate and then better performance. The aim of cache optimization is to help increase performance by avoiding the unnecessary misses, increase data reusability and/or avoid pulling into cache data that will not be used or removing data that is needed later on.

2.2 Technology

Although there are many different products or chips containing multi-processors including Cell processors, massively parallel GPUs, user configurable Field Programmable Gate Arrays (FPGAs), and other systems with hundreds and may be thousands of processors; the aim of this study in the focus of the general purpose multi-core chips.

The IBM POWER4 released in 2001 was the first non-embedded multicore microprocessor, with two cores on a single die. The original POWER4 had a clock speed of 1.1 and 1.3 GHz, while an enhanced version, the POWER4+, reached a clock speed of 1.9 GHz.

In June 2004, Intel announced its Celeron D (dual-core) processor which was –as said– the first dual-core processor for the budget/entry-level market. The clock speeds were of 2.13 GHz to 3.33 GHz. It had two level caches. L1 cache was 16 KB and L2 cache was 256 KB.

In the same year 2004, IBM announced its dual-core POWER5 microprocessor as upgrade to POWER4. POWER5 is dual-core with hyper-threading technology. With each core supporting one physical thread and two logical threads, it gave a total of two physical threads and four logical threads. The floating-point instruction cache was increased in capacity to 24 entries from 20 in POWER4. The capacity of the L2 unified cache was also increased to 1.875 MB and the set-associativity to 10-way. The unified L3 cache was brought on-package instead of located externally in separate chips. Its capacity was increased to 36 MB.

In May 2005, AMD introduced its first dual-core Opterons. Each core running on top speed 2.2 GHz with 64KB L1 data cache, 64KB Instruction cache and 512KB L2 cache.

In order to win the multi-core race, starting in September 2007, AMD launched its Quad-core Opetron chips that can be viewed as two dual-core chips `glued` together on one chip and used third level cache L3; AMD Barcelona with L3 2MB followed by Shanghai with 6MB L3 in 2008.

On Sept. 15, 2008, Intel has announced it first hexa-core chip `Dunnington` that use 96 KB L1 cache (Data) , with three unified 3 MB L2 caches and 16 MB of L3 shared cache. Shortly afterwards in November 2008, Intel released it core i7 chip known also as Nehalem. Nehalem is Quad-core chip with hyper-threading technology and uses two level private caches and 8MB shared L3 cache.

In June 2009, AMD launched its newest and fastest Opteron hexa-core chip with shared 6MB L3 cache and support of HyperTransport.

On 8^{th} of February 2010, IBM announced its powerful efficient POWER7 microprocessor with 4, 6 or 8 cores per chip. Each core can run four threads giving ability to run 32 different tasks simultaneously. It has top clock speed of 4.14 GHz with 12 execution units per core: 2 fixed-point units, 2 load/store units, 4 double-precision floating-point units, 1 vector unit, 1 decimal floating-point unit, 1 branch unit and 1 condition register unit. It has 32 kB L1 instruction and data cache per core, 256 kB L2 Cache per core and 32 MB L3 cache. The cache is implemented in embedded

DRAM technology (eDRAM), which does not require as many transistors per cell as a standard (SRAM) so it allows for a larger cache while using the same area as SRAM.

With this POWER7 microprocessor, IBM is –*at the time of writing*– the top winner in the multi-core manufacturing race.

2.2.1 Ferlin: our multicore system

This study and optimization was done w.r.t. one node of Ferlin [7, 8] as our multi-core system. The CPU's in Ferlin is Intel Xeon E5430 -known as *Harpertown*- with clock frequency 2.66 GHz and a 1333 MHz front-side bus, and 8 GB memory. The node consists of two quad-core CPU's. It uses 64kB for L1 case divided as 32kB 8-way set associative L1 data cache and 32kB L1 instruction cache. the cache line size is 64 bytes. for L2 cache it uses 2×6MB unified 24-way cache. This study is done with respect to L1 cache. Ferlin node specifications are summerized in table (2.1).

Processor model	Intel Xeon E5430
clock frequency	2.66 GHz
FSB RAM	1333 MHz 8 GB
no. cores	8 cores (2 quad-core CPU's)
L1 cache L1 Associativity	32kB data + 32kB instruction 8-way
L2 cache L2 Associativity	2×6 MB 24-way
cache line size	64 byte

Table 2.1. Ferlin node specifications

For compilation we used the intel compilers module (i-compilers/11.1) that is available on ferlin as the latest and default i-compilers module. For MKL (*Intel Math kernel library*) functions we used version 9.1.023 which is available as the default mkl module on Ferlin.

2.3 Programming for Multi-cores

Parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors. Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically one of the greatest obstacles to getting good parallel program performance.

Parallel programming techniques can benefit from multiple cores directly. Some existing parallel programming models such as Cilk++, OpenMP, PThreads, and MPI can be used on multi-core platforms. Intel introduced a new abstraction for C++ parallelism called TBB. Other research efforts include the Codeplay Sieve System, Cray's Chapel, Sun's Fortress, and IBM's X10.

Amoung those **OpenMP** is attractive to be used to transfer sequential codes –specially numerical algorithms– into parallel programs on shared memory systems, because of its flexibility [9]. It consists of a set of compiler directives, library routines, and environment variables that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It is a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer. Data layout and decomposition is handled automatically by directives. It can work on one portion of the program at one time, no dramatic change to code is needed. OpenMP constructs are treated as comments when sequential compilers are used. But on other hand scalability when using OpnMP is limited by memory architecture.

Multi-core processing has also affected the ability of modern day computational software development. Developers programming in newer languages might find that their modern languages do not support multi-core functionality. This then requires the use of numerical libraries to access code written in languages like C and Fortran, which perform math computations faster than newer languages like C#. Intel's MKL - *math kernel libraray* - and AMD's ACML - *AMD Core Math Library* - are written in these native languages and take advantage of multi-core processing.

MKL is a library of highly optimized with respect to the Intel^{TM} processors and offers extensively threaded math routines for science, engineering, and financial applications that require maximum performance. Core math functions include BLAS, LAPACK, ScaLAPACK, Sparse Solvers, Fast Fourier Transforms, Vector Math, and more.

ACML consists of five main components: a full implementation of Level 1, 2 and 3 Basic Linear Algebra Subroutines (BLAS), with key routines optimized for high performance on AMD Opteron^{TM} processors. , a full suite of Linear Algebra highly tuned LAPACK routines, a comprehensive suite of Fast Fourier Transforms (FFTs) in both single-, double-, single-complex and double-complex data types, fast scalar, vector, and array math transcendental library routines optimized for high performance on AMD Opteron processors and finally random Number Generators in both single- and double-precision.

2.4 Performance measurement

2.4.1 Speedup

The speedup factor indicate how much we gain after enhancement [3, 4, 10]. If we define T_{old} as the time needed to execute the original code and t_{new} the time needed

to execute the enhanced code; then the speedup S_p can be defined as:

$$S_p = \frac{T_{new}}{T_{old}}$$

Speed-up can also measure the gain we get in performance due to parallelization. In that case T_s will be the sequential time and T_p will be the parallel time on p processors. For a perfectly parallel code $S_p \approx p$.

$$S_p = \frac{T_s}{T_p}$$

If we start parallelize poorly optimized sequential code, we will not only notice poor speed up but also –in many cases– decreasing one as the number of cores increase, a term so called - *super slow down*. If the program lacks locality then a single processor will spend most of the execution time loading data that is not efficiently used. That effect will be magnified on the parallel version where the processor will spend most of the time fighting on the shared limited resources. So before running into adding parallel directives into the sequential code, a deeper optimization of that code will not only improve the sequential code but also -taking into account the cache locality and optimization- will allow the parallel code to speed up nicely and so increase performance. So what should be done in order to optimize the code and get the speed we aim for is what we will be discussed in more details in Chapter 3.

2.4.2 Efficiency

The efficiency is defined as the ratio between the speed up and number of processors p:

$$\eta_p = \frac{S_p}{p}$$

Keeping η_P as close as possible to 100% is the most optimistic desire and that will indicate that the parallelism is efficient.

Chapter 3

Optimization Techniques

The goal of optimization is to increase performance by either decreasing the execution time or increasing the number of completed tasks in a given time. Usually it needs trade-off between several resources and can only be done w.r.t one or two aspects of performance like execution time, memory usage, disk space, bandwidth, power consumption or some other resource.

The difficulty of the optimization process lies on the fact that it is - in most cases- machine dependent. If a code is optimized w.r.t. certain hardware sittings, it may not perform well on another machine.

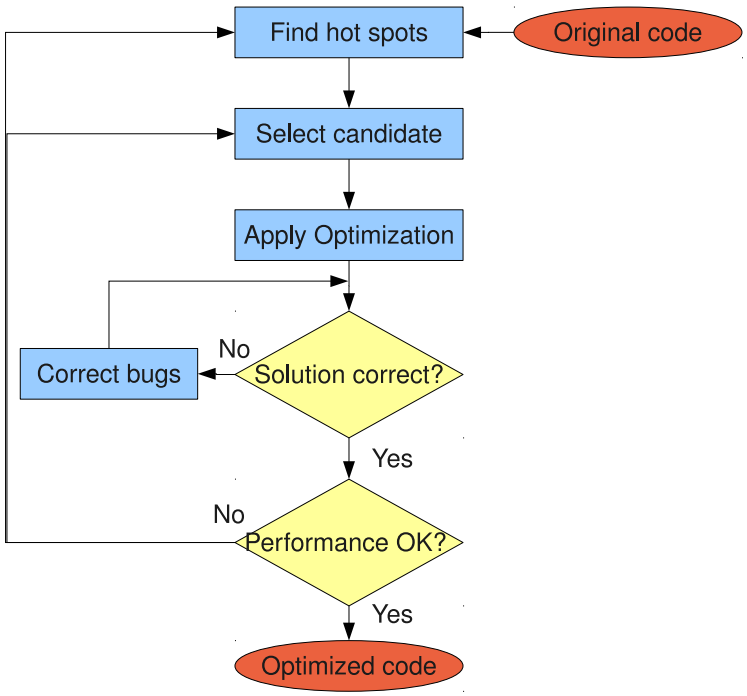

Figure 3.1. Optimization process

Optimization can add extra code that is used only to improve the performance and hence may reduce portability readability. This may complicate programs or systems, making them harder to maintain and debug. That is why it is more convenient to do the optimization as a fine-tuning step after the code is built and tested.

Figure 3.1 shows the steps involved in the optimization step after a goal is set. Our goal in this study is to reduce the execution time by introducing better utilized usage of cache hierarchy. Fining hot spots in the code - or is also known as the bottlenecks- is the most important step in the optimization process. The aid of tools for cache profiling is crucial specially when dealing with complex structures, separated code files and multi-threaded applications. The next step is to select the slowest parts of the program that cause the most significant hot spot and try to enhance it by the suitable technique. Finally the optimized version of the code is tested to ensure it gives the same results as the original code. Note in some numerical intensive algorithms, due to the change of instructions order and/or complexity of the code, rounding off errors can be different. In some cases we have to further apply the process over and over again to reach a satisfing performance.

3.1 Automatic Optimizing

First rule of optimization is to not re-invent the wheel; use optimized libraries and compiler optimization flags.

Current compilers are good enough to do great job in optimizing the code [12]. The most common requirement taken into account by the compiler is to minimize the time taken to execute a program. A less common one is to minimize the amount of memory occupied by the program. Compilers generate codes that are much better than what the human-programmers wrote and definitely can improve highly hand-optimized code even further. Optimization by the compiler is made on the compilation step. Thus, Without any change in the code itself.

The flags to enable optimization are -O[LEVEL]. The flag **-O1** is used to produce smallest memory space. The flag **-O2** considers the best combination for compilation speed and runtime performance. As stated in the IBM manual [11], this flag will perform these actions:

- Value numbering - folding several instructions into a single instruction.
- Branch straightening - rearranging program code to minimize branch logic and combining physically separate blocks of code.
- Common expression elimination - eliminating duplicate expressions.
- Code motion - performing calculations outside a loop (if the variables in the loop are not altered within it) and using those results within the loop.
- Re-association and strength reduction - rearranging calculation sequences in loops in order to replace less efficient instructions with more efficient ones.

- Global constant propagation - combining constants used in an expression and generating new ones.
- Store motion - moving store operations out of loops.
- Dead store elimination - eliminating stores when the value stored is never referred to again.
- Dead code elimination - eliminating code for calculations that are not required and portions of the code that can never be reached.
- Global register allocation - keeping variables and expressions in registers instead of memory.
- Instruction scheduling - reordering instructions to minimize program execution time.

The flag **-O3** is the compiler's highest and most aggressive level of optimization. It performs optimizations that have the potential to slightly alter the semantics of the program. In addition to the work of **-O2** it will allow as said in [11] several action such as :

- Rewriting floating-point expressions - it may cost of potentially different numeric results.
- Aggressive code motion and scheduling of computations that have the potential to raise a conditional exception during the program execution might be definitely scheduled at this level if this might lead to improvements in the performance. In other words, load and floating-point computations may be placed onto execution paths where they will be executed even though, according to the actual semantics of the program, they might not have been.
- Incorrect sign for zero - For example, the expression "x + 0.0" would not be replaced with "x" at the -O2 level. A redundant add of 0.0 to x will be done because x might be equal to -0.0 and, under IEEE rules, -0.0 + 0.0 = 0.0 which would be -x in this case. Since, in the overwhelming majority of cases, this has no significant impact on a program's results, -O3 will substitute "x" for "x + 0.0".

But usually to get better performance relaying only on the compiler optimization is not enough and the such the generated code needs more human enhancement because of several issues:

- Compilers optimize the current code and will not change it with improved algorithmic complexity. As example it will not replace sorting phase with a better one.

- Compilers usually have to support a variety of conflicting objectives so it is not possible to satisfy them all.
- Compilers typically only deal with a part of a program at a time, often the code contained within a single file or module so they do not support the global view of the solution.
- Special knowledge about the data values and execution is not visible in the compilation phase.
- Compilers can not perform complex cache optimization techniques.

So in most cases to allow the compiler do good optimization and until smarter compilers appear, we have to enhance the code manually. Better-written code will lead to better compiler optimization. Through the next sections; we will go quickly on most helpful techniques for optimization.

3.2 Arithmetic Expressions

Most of arithmetic expressions can be enhanced with the compilers level **-O2**. Compilers are good at cominbing, overlapping and splitting even computations in most cases. But as we said helping the compilers will allow it to focus on different path of the compilation and so more enhancements.

Remove loop-invariant calculation to avoid the re-computations and re-accessing of the operands in each operation. See Figure 3.2.

Listing 3.1. Original code

```
1 for i=1 to n
2     A(i)=A(i)+A(j)*A(j)
3 end for
4 ...
```

Listing 3.2. Modified code

```
1 c=A(j)*A(j)
2 for i=1 to n
3     A(i)=A(i)+c
4 end for
```

Figure 3.2. Example of avoiding loop invariant calculation

Division and power is very expensive operations. **Avoid divisions** in the loop and transform it -if possible- into multiplication. That will make use of the build-in multiplier and save many cycles needed to evaluate the division. See Figures 3.3 and 3.2.

Also **Avoid jumps** because if-statements in the loop prevent the compiler optimization. See Figure 3.5

Anther useful technique is what so called **loop unrolling** by which we mean is to increase the work per iterations by explicitly add steps in the loop. In order to enforce consistency the complier would make sure that each loop iteration will have

Listing 3.3. Original code

```
1 c=3.14
2 for i=1 to n
3     A(i)=A(i)/c
4 end for
```

Listing 3.4. Modified code

```
1 c=1/3.14
2 for i=1 to n
3     A(i)=A(i)*c
4 end for
```

Figure 3.3. Example to void division in the loop

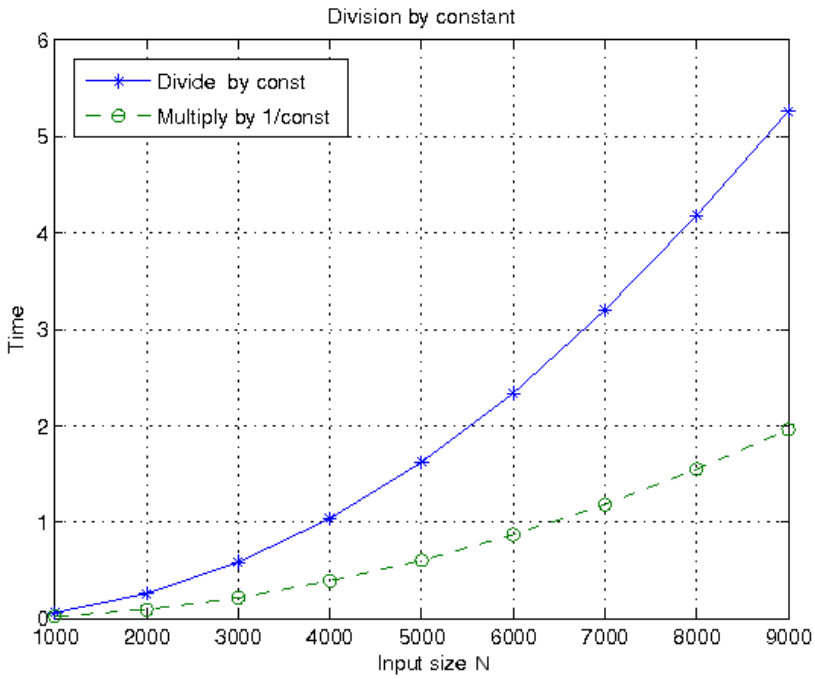

Figure 3.4. Effect of avoiding division. Time in seconds

Listing 3.5. Original code

```
1 for i=1 to 1000
2     if ( mod(i,2)==0)
3                 A(i)=x
4         else
5             A(i)=y
6         end if
7 end for
```

Listing 3.6. Modified code

```
1 for i=1 to 1000, step 2
2     A(i)=y
3     A(i+1)=x
4 end for
```

Figure 3.5. Example to avoid branching

dependency with the previous iterations. Of course although it depends on the work done in the loop but this take advantage of data locality and decrease number of comparisons and loop variable increments and increase degree of parallelism within each loop iteration. A smart compiler will then allow overlap more operations in each loop iteration. More tricks that work with loops will be discussed in the cache optimization techniques. Figures 3.6 and 3.7 show examples of optimization by loop unrolling and its effect on time decreasing.

Listing 3.7. Original code

```
1 for i=1 to n
2     A(i)=f(i)
3 end for
4 ...
```

Listing 3.8. 1 unrolling

```
1 for i=1 to n, step 2
2     A(i)=f(i)
3     A(i+1)=f(i+1)
4 end for
```

Listing 3.9. 4 unrolling

```
1 for i=1 to n, step 4
2    A(i)=f(i)
3    A(i+1)=f(i+1)
4    A(i+2)=f(i+2)
5    A(i+7)=f(i+3)
6 end for
```

Listing 3.10. 8 unrolling

```
1 for i=1 to n, step 8
2    A(i)=f(i)
3    A(i+1)=f(i+1)
4     ...
5    A(i+7)=f(i+7)
6 end for
```

Figure 3.6. Example of loop unrolling

Note that fortran 90 is provided with new selection notation for regularly spaced array entries using (:). Loops that need to update array enteries can be replaced with the efficient notation that allows the compiler to repace it with automatically vectorized calculations.

3.3 Cache Optimization

As we said before, due to the hardware prefetching and fact that accessing data from the main memory is way slower than accessing data from cache, we need to make best usage of what already loaded to the cache or what can interpreted as increasing *data locality*. *Temporal locality* is what we mean by making best use of the data available in the cache.

All techniques that will be discussed in this section are aiming increase cache performance by:

- Avoid irregular data access patterns.
- Avoid fetching data cache lines that are partially used.

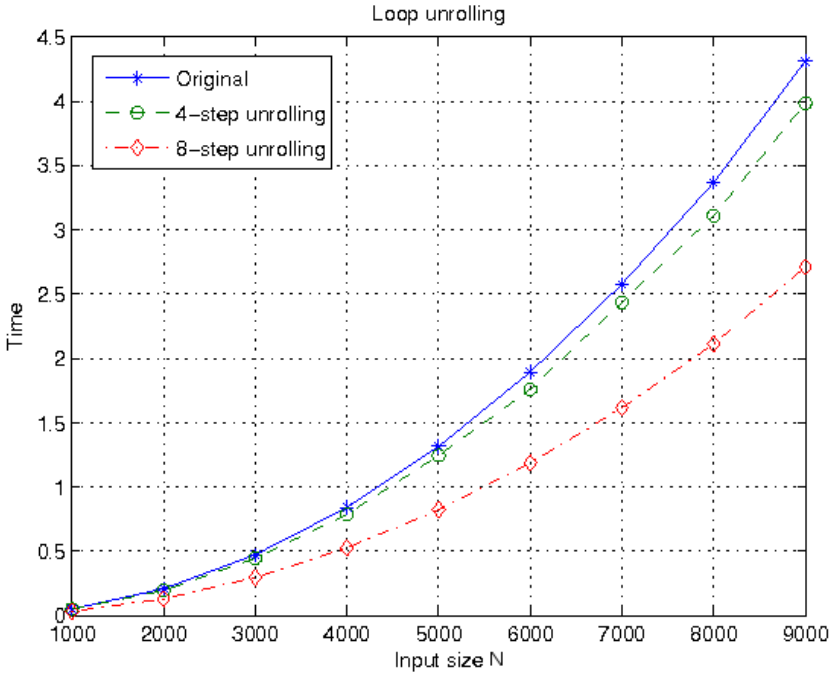

Figure 3.7. Loop unrolling effect. Time in seconds

- Avoid writing back in memory cache lines that are partially modified.
- Avoid eviction of cache lines that will be accessed in future.
- Avoid different threads to update same cache lines.

The first recommendation is to **avoid array sizes that are multiples of the cache line size**. Having a power of 2 array sizes may lead to poor replacement scheme as different segments will be mapped to same cache entry if you use direct or set-associative mapping causing conflict misses. That will lead to poor hit ratio specially if data are reused [12]. **Array padding** is a technique to solve this problem as in Figure 3.8.

Another useful technique is **Array merging**. As in Figure 3.9, merging two arrays in one multidimensional array or structure array will decrease the number of cross interference misses.

Depending on the memory layout; use adjacent contiguous memory locations or what so called as **stride-1 accesses** and **loop re-ordering**. That will avoid pulling into cache data that will be partially used, increase data locality ,and make good use of hardware pre-fetching. Figure 3.3 shows how time is dramatically increase if we use the *bad* loop nesting order. Two dimension arrays in C are stored row-wise. So accessing elements in each row per main iteration is more efficient, See Figure 3.12.

Listing 3.11. Original code

```
1 double arr1[1024];
2 double arr2[1024];
3 for i=1 to 1024
4     sum+= arr1[i]+arr2[i]
5 end for
```

Listing 3.12. Modified code

```
1 double arr1[1024];
2 double padarr[3];
3 double arr2[1024];
4 for i=1 to 1024
5     sum+= arr1[i]+arr2[i]
6 end for
```

Figure 3.8. Example of loop padding optimization

Listing 3.13. Modifiedcode1

```
1 double arr1[1024][2];
2 for i=1 to 1024
3     sum+= arr1(i,1)+arr2(i,2)
4 end for
```

Listing 3.14. Modifiedcode2

```
1 struct{ double a;
2         double b;
3         }ab[1024];
4 for i=1 to 1024
5     sum+= ab[i].a+ab[i].b
6 end for
```

Figure 3.9. Example of using array merging to enhance original code in figure(3.8)

In fortran the arrays are stored column-wise, so the solution would be inverted. For obvious cases, compiler optimization level **-O2** can do loop interchanging

Listing 3.15. Original code

```
1 for j=1 to n
2     for i=1 to n
3                 A(i,j)=f(A(i,j))
4     end for
5 end for
```

Listing 3.16. Modified code

```
1 for i=1 to n
2     for j=1 to n
3                 A(i,j)=f(A(i,j))
4     end for
5 end for
```

Figure 3.10. Example of loop nesting optimization. Time in seconds

Loop fusion is technique by which we combine adjacent loops into one single loop to increase the work per loop iteration and give better re-usability of data and get rid of half the loop comparisons. Figure 3.13 shows example of merging two independent computations loops into one loop.

Although that might sounds opposite to what we just said but it is not; **loop fission** is also useful. It is meant to split a complex fat loop into two or more loops when no common data between these parts. The aim to reduce register pressure

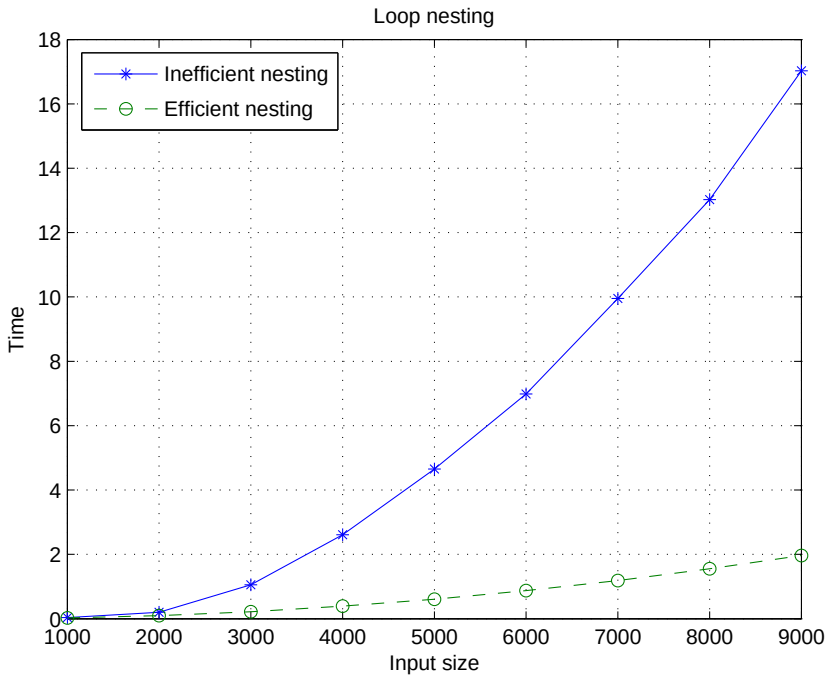

Figure 3.11. Loop order effect. Time in seconds

Listing 3.17. Original code

```
1 for i=1 to n
2     A(i)=f(i)
3 end for
4 for i=1 to n
5     B(i)=A(i)+g(i)
6 end
```

Listing 3.18. Modified code

```
1 for i=1 to n
2     A(i)=f(i)
3     B(i)=A(i)+g(i)
4 end for
```

Figure 3.12. Example of loop fusion optimization

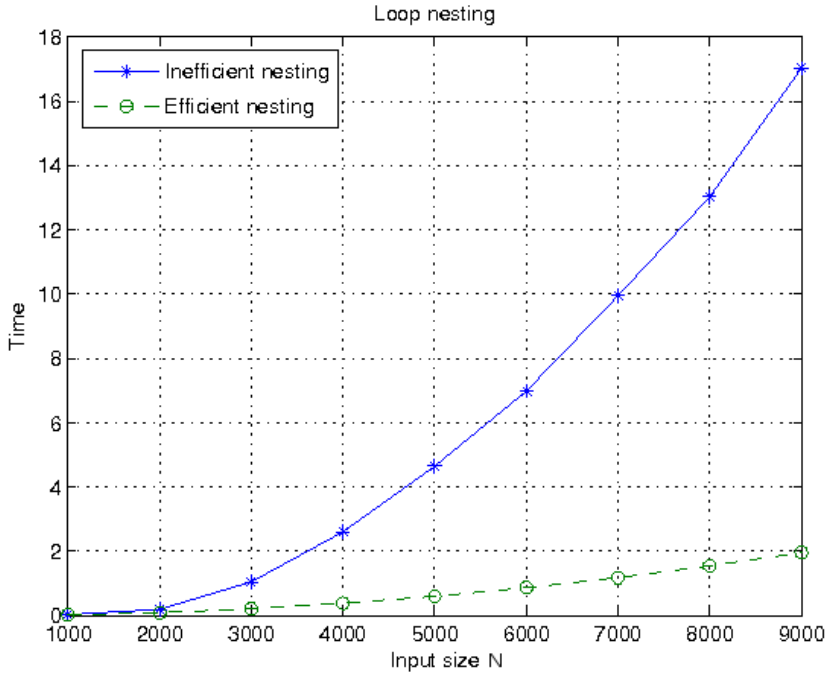

Figure 3.13. Loop fusion effect. Time in seconds

and help compiler to optimize the loops.

The most effective technique that can be applied is what so called **cache blocking** [12, 13]. The aim of blocking is to transfer nested loops that works on large datasets – that will not fit into the cache– to iterate on small blocks of dataset at a time instead. The size of the block depends on the cache as well as the operation to be performed. That technique helps when data alignment is not in the direction of computation. Famous example in matrices multiplication and transpose.

In Figures 3.14 and 3.15 we gave a matrix multiplication computation as an example and so we can see how the program behaves before and after optimization. Note that pseudo-code in C-like access. If the same operation was to be implemented in fortran; first and second loop should be exchanged.

3.3.1 Data Structures and Object-oriented programming

Object oriented concepts such as data encapsulation and dynamically expanding data structure often will cause poor cache performance. If data fields within a structure object are not used in the calculations, it will cause poor cache performance since the cache line is partially used. The solution for that is to split the structure so we only use needed data. Example case as in Figure 3.16.

Dynamically expanding arrays, linked lists, and other forms of data structures that are not contiguous in memory locations and hence need jumping through mem-

Listing 3.19. Original code

```
1  for i=1 to n
2    for j=1 to n
3      c(i,j)=0
4    end for
5  end for
6  for i=1 to n
7    for j=1 to n
8      for k=1 to n
9        c(i,j)+=a(i,k)*b(k,j)
10     end for k
11   end for j
12 end for i
```

Listing 3.20. Modified code

```
1  for i=1 to n
2    for j=1 to n
3      c(i,j)=0
4    end for j
5  end for i
6  for ii=1 to n ,step=Bi
7    for kk=1 to n, step=Bk
8      for jj=1 to n, step=Bj
9        for i=ii to min(n,ii+Bi)
10         for k=kk to min(n,kk+Bk)
11           for j=jj to min(n,jj+Bj)
12             c(i,j)+=a(i,k)*b(k,j)
13           end for j
14         end for k
15       end for i
16     end for jj
17   end for kk
18 end for ii
```

Figure 3.14. Example of matrix multiplication loop blocking optimization

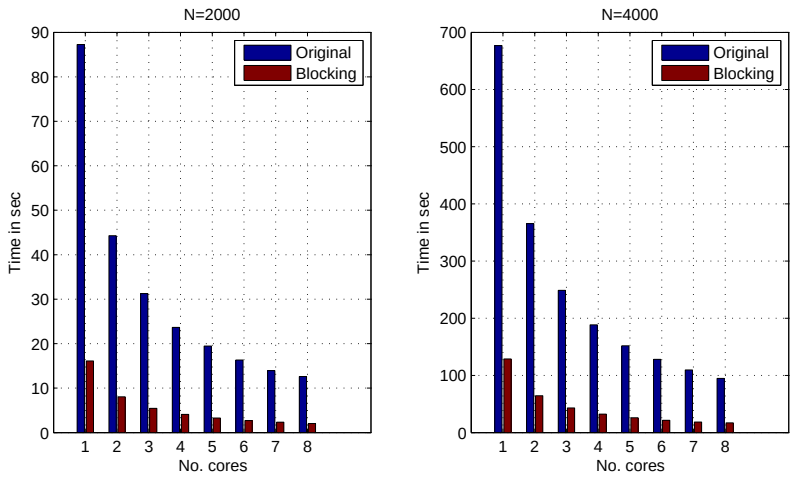

Figure 3.15. Blocking the matrix multiplication

Listing 3.21. Original code

```
1 struct{
2 int d;
3 int a;
4 int b;
5 int c;
6 }arr[n];
7 for i=1 to n
8     sum=arr[i].a+arr[i].c
9 end for
```

Listing 3.22. Modified code

```
1 struct{
2 int d;
3 int b;}arr_db[n];
4 struct{
5 int a;
6 int c;}arr_ac[n];
7 for i=1 to n
8     sum=arr_ac[i].a+arr_ac[i].c
9 end for
```

Figure 3.16. Example of data structure optimization

ory locations are definitely not supported to perform well on caches. The hardware prefetcher will then do the worst job of pulling data to cache that are hardly useful. It is expected in the nearest future to find solutions for this problem by hardware and sofware means [14].

Chapter 4

Optimization Tool: **Acumem ThreadSpotter**^{TM}

The process of optimization starts with identifying the parts of the code that causing problems then based on that analysis modifications are made where it is relevant. This process can get really hard specially for large codes that are splitting into many files and/or the cache and memory access patterns is clearly dependent on the architecture in hand. Even if the aim to study the performance on different architecture is hard if it considered to be done manually. The tool we used during this study is called **Acumem ThreadSpotter**^{TM}.

The code analysis –using **Acumem ThreadSpotter**^{TM}– aims to find the performance problems in the execution and map it to the source code. The typical issues that cause performance problems and can be spotted by **Acumem ThreadSpotter**^{TM} -as stated in the manual [15] - are:

- Inefficient data layout.
- Inefficient data access patterns.
- Unexploited data reuse opportunities.
- Prefetching problems.
- Thread interaction problems.

The analysis of the code is performed through two steps: Application Sampling and Report Generation.

4.1 Analysis Commands

4.1.1 Sampling an Application

Sampling an application is made by collecting data about the application structure and its memory access behavior ,and saving these data to a sample file. This can

be done by either launch and sample an application or sampling a running one. by default it uses the standard 64-byte cache line, however that can be modified to include different cache line settings. A typical command used in this work can be written as:

`sample -l` *`cachelines`* `-s` *`Interval`* `-o` *`samplefile`* `-r` *`application and arguments`*

Description of those command options are discribed in the next table

Option	Description	Example	Comment
-r	run *application and arguments*	`sample -r ./myApp arg1`	mandatory and must come last
-l	specify cache line size(s) in bytes	`sample -l 64,128 -r ./myApp`	optional, default 64
-s	specify sample interval. Good to increase if sampling terminates before application does	`sample -s 100000 -r ./myApp`	optional
-o	specify the sample file name. It will override file with same name.		optional, default `sample.smp`

For more information and other command options please check [15]. In order to be able to locate the code files and be able to see it from the report; the compilation of the program should made with switching the debug mode on.

4.1.2 Report Generation

Using the sample file (`samplefile.smp`), **Acumem ThreadSpotter**^{TM} is used to generate different reports in HTML format. The default processor model will be the one where the package is installed on and L1 cache is the default cache level. However; different settings can be used. A typical command used in this work can be written as:

`report -p` *`percenage`* `-cpu` *`cpumodel`* `-level` *`cachelevel`* `-i` *`samplefile`* `-o` *`reportfilename`*

The next table describes the options used in the reporting command.

Option	Description	Example	Comment
-i	generate report from the specified `.smp` sample file	`report -i sample.smp`	mandatory
-o	specify the report HTML file and folder with same name containing additional files	`report -i s.smp -o Rep`	optional, default acumem-report.html
–cpu	specify cup model	`report -cpu intel/tulsa_2_2_16 -i s.smp`	optional
–level	select cache level to analyze	`report -level 1 -i s.smp`	optional, default highest
-p	limit the report to issues that contribute to at least as much percentage of the total cache line fetches as specified.	`report -p 5 -i s.smp`	optional, default 1

If the source is moved from the location where the application is compiled at, the option `-s` *source directory* can be used to specify the new location. More information about the command options can be found in [15]. `report -cpu help` command is used to show a list of available CPU models.

4.2 Main Report Issues

4.2.1 Utilization issues

- **Fetch utilization**: cache lines fetched from memory are partially used. This is indication to poor spatial locality and can be as a result of structures with unused fields, inefficient loop nesting, irregular access patterns and/or dynamically allocated data.
- **Write-back utilization**: cache lines that are sent back to memory or partially updated. This is indicator of a wasted bandwidth.
- **Communication utilization**: Communication between different threads are mapped to different caches. Poor communication utilization that caused by different data updating the same cache line. This will be as result of bad data partitioning between diffreent threads. A godd partitioning will be in such ways that threads do not update data in same cache lines.

4.2.2 Loops issues

- **inefficient loop nesting**: indicates that loops in multidimensional array is accessed in inefficient way. Spatial blocking might be good solution for such

issues.

- **Random access pattern**: indication of reduced hardware prefetching efficiency and cache performance.
- **Loop fusion**: indicates that two loops are iterating over same datasets but – due to capacity of the cache or being far from each other– the accessed data is evicted from cache between the loops. The solution for that is merging loops together in one loop.
- **Blocking**: indicating good chance to decrease cache misses or fetching by applying blocking.

4.2.3 Hot-Spots

the issue is fired when certain part of the code causes large number of cache misses or fetches but there is no clear way to improve it. In such cases software prefetching mechanisms may be useful.

4.3 Snap shots from the report

The first page of the report contains useful links to the manual and web site. The metrics in the left of the report give the user indicators of the behavior of the application w.r.t. Memory bandwidth, latency, data locality and threads communication/ Interaction. See Figure 4.1.

By clicking on the (Open report) button, we can open the report. As in Figure 4.2, the page is split into three parts. On the top-left we find tabs with the issues and information about the application. By selecting an issue, the right part of the page will show the corresponding lines in the source code responsible for that issue. One code line can have several issues depending on the variable causing them and the way they are manipulated. On the bottom left we find information regarding that issue.

By clicking on Statistics for that issue we can see on that part of the page more information about the issue as in Figure 4.3. In the figure we can see from the plot in the statistics that we are using 6MB cache and if that is increased to 96MB we will not get that issue alarm.

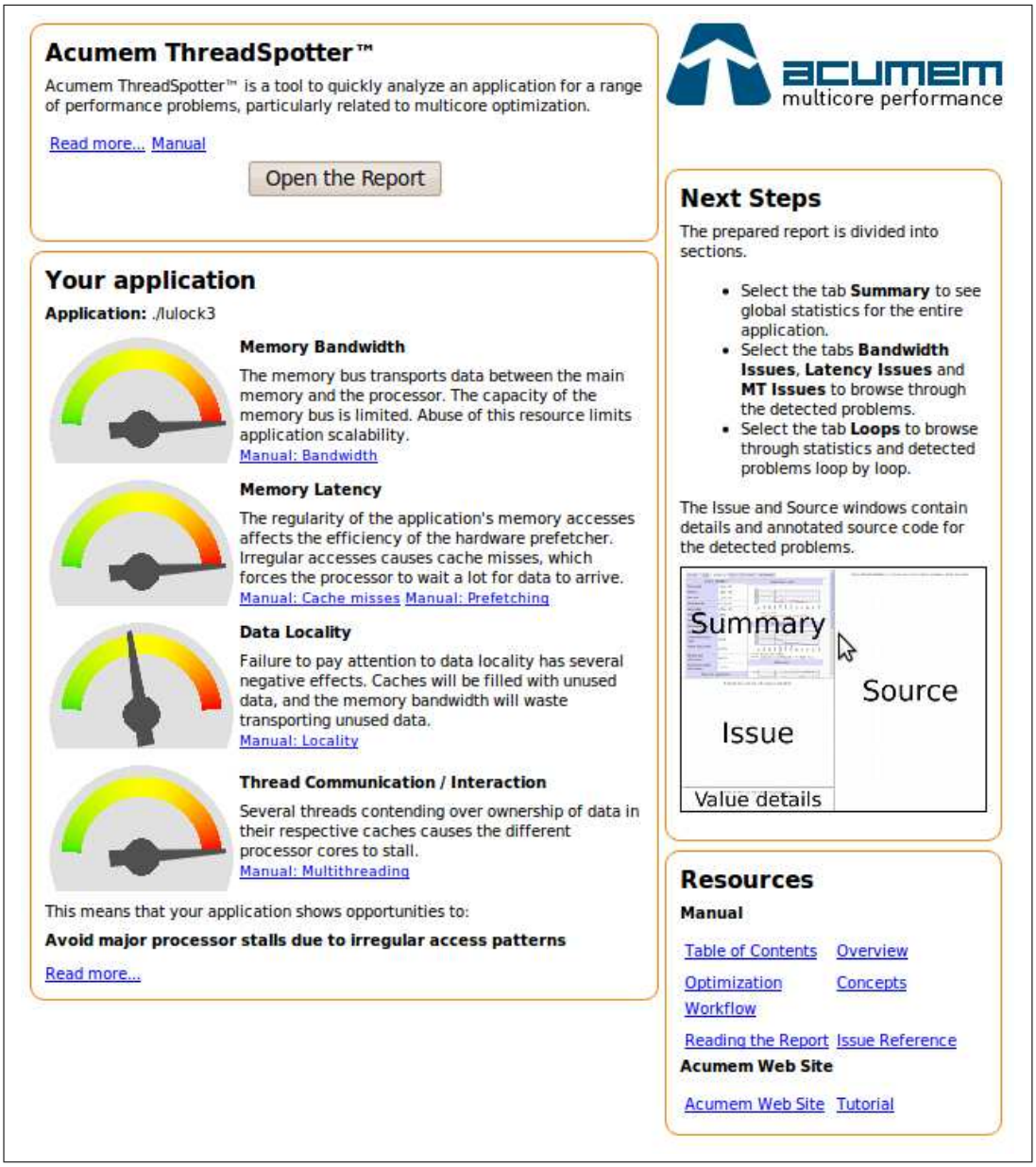

Figure 4.1. The first page in the report

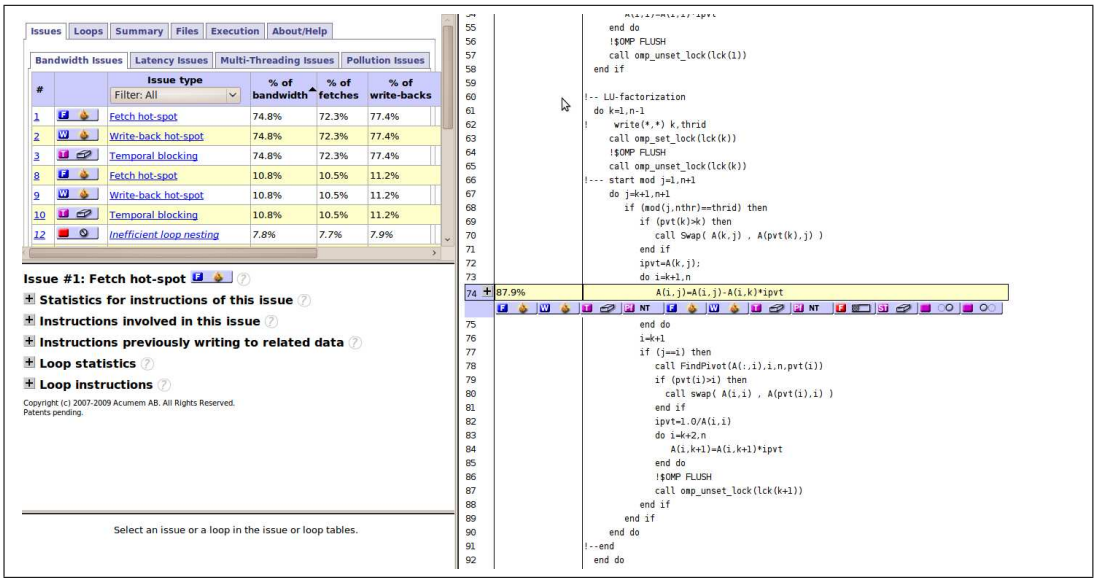

Figure 4.2. Report issues

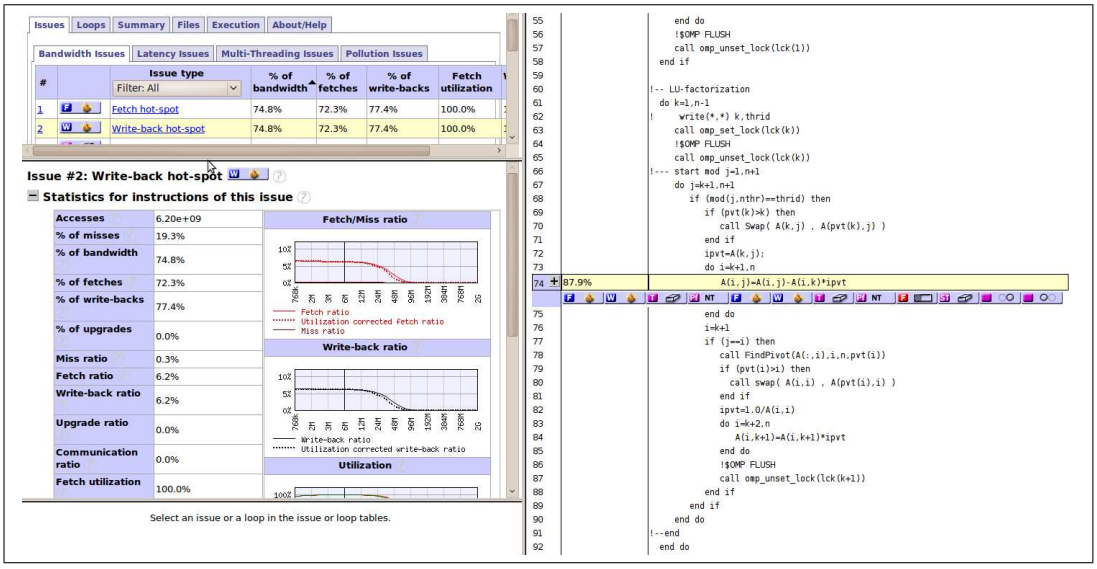

Figure 4.3. Issue statistics

Part II

Case Studies

Chapter 5

Case study: Gaussian Elimination GE

5.1 Introduction

One of the most popular techniques for solving simultaneous linear equations is the Gaussian Elimination method (GE). It is named after German mathematician and scientist Carl Friedrich Gauss. The approach is designed to solve a general set of n equations and n unknowns and can be also used to find the rank of a matrix, generate the LU decomposition and calculate the inverse of an invertible square matrix. In this work we aimed to solve the system:

$$\begin{aligned} Ax &= b \equiv LUx = b \\ Ux &= L^{-1}b = y \\ \Rightarrow x &= U^{-1}y = U^{-1}L^{-1}b = A^{-1}b \end{aligned}$$

Where $A_{N\times N}$ is a full non-singular matrix, $x_{N\times 1}$ the vector of the unknowns, $b_{N\times 1}$ the right hand side, $L_{N\times N}$ is a lower diagonal matrix and $U_{N\times N}$ is an upper diagonal matrix. The process of GE has two steps. The first step; *Forward Elimination* reduces a given system to either triangular or echelon form, or results in a degenerate equation with no solution, indicating the system has no solution. This is accomplished through the use of elementary row operations. The second step uses *Backward Substitution* to find the solution of the system above. It can be written as finding the solution of $Ux = y$

$$\begin{aligned} x_N &= y_N/u_{N,N} \\ x_i &= \frac{y_i - \sum_{j=i+1}^{N} u_{i,j}.x_j}{u_{i,i}} \quad i = N-1, N-2, \ldots 1 \end{aligned}$$

This part is sequential in nature. This work is done to speed up the forward elimination part as it is the dominant factor of the computation. The algorithm as expressed earlier is numerically stable if the original matrix working on is *diagonally dominant* or *positive-definite*. For general matrices, GE is usually considered to be

stable in practice if we use partial pivoting. It is meant to keep the diagonal element ($pivot$=a_{ii}) to be larger in magnitude (absolute value) than all elements below it in the i^{th} iteration of the elimination step. This is done by row exchanges. Note also in this implementation instead of having the elements under diagonal of A set to zero, they are used to hold the multipliers of L. The right hand side b is augmented to the end of the matrix A. So now A is $n \times (n+1)$ and in general the pseudo-code of the algorithm for the full non-singular matrix should be as:

```
for i=1 to n-1
    Find Pivot in column i
    if Pivot not at position i then
        Exchange Pivot row with Row i
    end if
    for j=i+1 to n
        A(i,j)=A(i,j)/A(i,i)
    end for
    for j=i+1 to n
        for k=i+1 to n+1
            A(j,k)=A(j,k)-A(i,j)*A(i,k)
        end for
    end for
end for
```

Since the pivoting is done column-wise, Fortran will have advantage as the matrix is stored column-wise. That operation will be much slower in C since the matrix is stored row-wise, causing a cache miss at each step in the pivoting search. One solution is to implement row pivoting instead. However, the experiments we did was using Fortran implementation.

5.1.1 Matrix Generation

For simplicity, the matrix A was generated using the random number generator RAND in fortran 90. This function generates random numbers between 0 and 1. The norm of the matrix –as tested in matlab– is ≈ 500. The right hand side is augmented to the matrix taken to be the ones vector. The fortran code segment for initialization looked as following:

```
USE IFPORT
integer :: i,j,seed=7654321,n
real, dimension(:,:), ALLOCATABLE :: A
....
read *,n
 ALLOCATE(A(n,n+1))
i=rand(seed)
!-- Initiate matrix
 do j=1,n
```

```
      do i=1,n
         A(i,j)= rand(0)
      end do
      A(j,n+1)=1
      ...
end do
```

5.1.2 Correctness

While applying optimization it is easy to get bugs. In order to check correctness of the results obtained, we implemented the *backward substitution* to obtain the solution x^* we used the residual function to calculate the error:

$$|e| = |Ax^* - b|$$

The residual error –as for the case of our matrix and right hand side– should be tending to zero, but may not be exactly zero due to rounding errors bu it was sufficient to indicate correct result. Having a correct result meaning that $|e| \approx O(10^{-9}) - O(10^{-12})$. Any larger $|e|$ –as will get very big – indicates that there is something wrong.

5.2 Experiments

5.2.1 Forward Elimination: Experiment 1

The first naive parallel treatment for the algorithm in the forward elimination step is done by inserting OpenMP [9] directives into the independent task of updating the sub-matrix at each iteration i once the pivot row is known.

```
for i=1 to n-1
{
  GetPivot(i);
  if(pivotPos!=i)
      XchangeRows(A,i,pivotPos);
  for j=i+1 to n
      A(j,i)=A(j,i)/A(i,i)
  end for
  !$omp parallel for private(i,j,k)
      for j=i+1 to n+1
         for k=i+1 to n
            A(k,j)=A(k,j)-A(k,j)*A(i,j);
             end for
      end for
  !$omp end parallel for
}
```

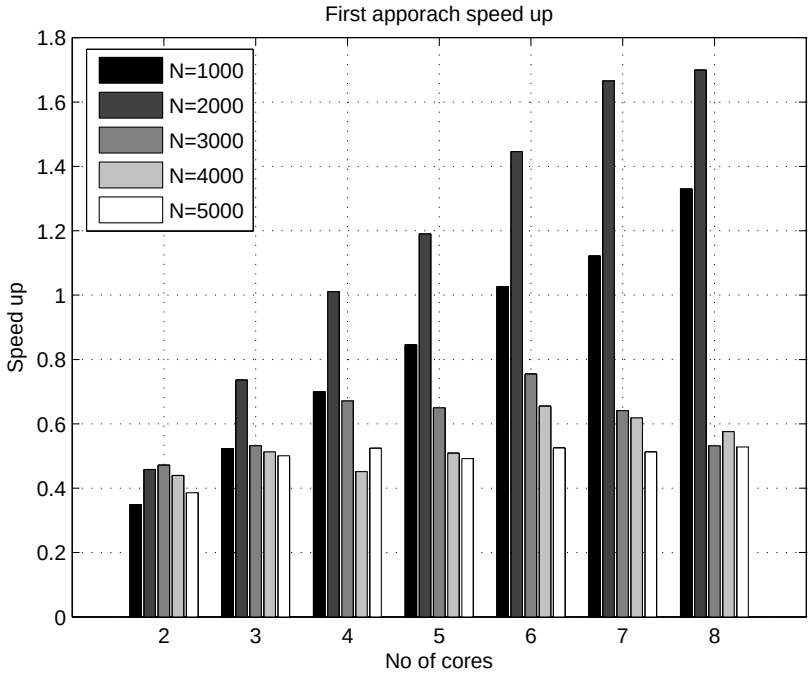

Figure 5.1. Speed up for the unoptimized GE code: Experiment 1

However the speed up was very poor as shown in (5.1).
According the *Acumem ThreadSpotter* report; the program has so many issues:

- **Memory Bandwidth**: The memory bus transports data between the main memory and the processor. The capacity of the memory bus is limited. Abuse of this resource limits application scalability.
- **Memory Latency**: The regularity of the application's memory accesses affects the efficiency of the hardware prefetcher. Irregular accesses causes cache misses, which forces the processor to wait a lot for data to arrive.
- **Data Locality**: Failure to pay attention to data locality has several negative effects. Caches will be filled with unused data, and the memory bandwidth will waste transporting unused data.
- **Thread Communication / Interaction**: Several threads contending over ownership of data in their respective caches causes the different processor cores to stall.

In other words we had many sources of latencies: overheads of creating and destroying the threads with every iteration, the scope of each thread is different at each iteration causing the threads to fight all time for data owned by other threads,

the delay of synchronization at each iteration step and the failure to keep temporal and spatial locality. As from the Figure 5.1 we can see that these effects increase dramatically as we increase the number of threads and the input size. According to the report for N=4000 we need at least 48 MB private cache to overcome the locality problems. For small input sizes as N=1000 we did not get any locality problems as the matrix would fit in the cache. Note also that pivoting and column elimination is done on the master process but parallelizing these parts will cause worse results as it fires false sharing alerts.

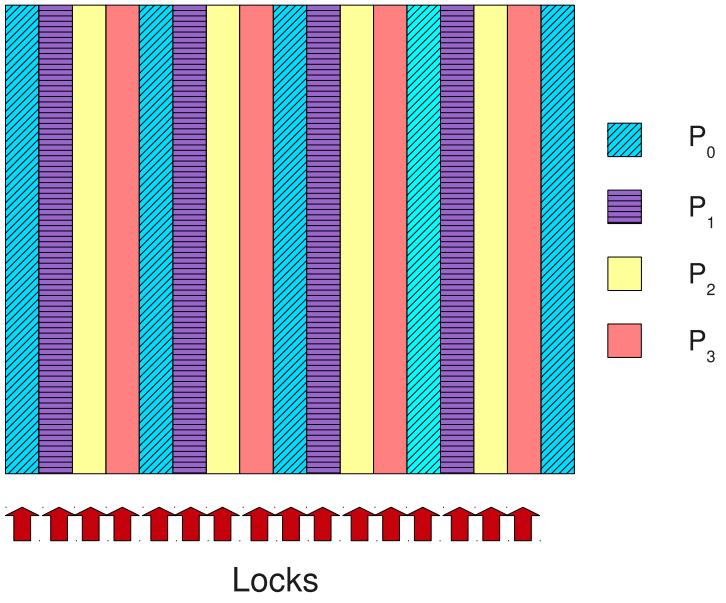

Figure 5.2. Cyclic Column Distribution on 4 processes

5.2.2 Forward Elimination: Experiment 2

The second approach aims to generate the threads just once and allow each thread to have access to the same data elements at each iteration in order to avoid overheads and increase locality. The sub-matrix eliminations (the j-loop) will run from 1 to $n+1$ so the matrix is divided by columns using cyclic distribution of size 1 and the iteration will be done if $j > i$ meaning $j = i+1$ to n. See Figure 5.2.

Pivoting and Row exchange

We introduced an array of size n that holds the default pivot positions which is simply $ptv(k) = k$. Setting this array is done in the initialization. At iteration k, only pivot holder of the next iteration search for that pivot location, update the array at the corresponding position, divide the pivot coumn by the new pivot. At the beginning of the elimination of each column then, the switch of two elements at

positions k and $pvt(k)$ is done. So the exchange of rows is done implicitly after all threads finish that specific iteration.

Synchronization

The synchronization between threads – so they wait until the column i is eliminated before executing the iteration i – is done using locks[9]. We introduced a new array for the locks. Size of the locks array is n. The locks are set during the initialization of the matrix which is also done which the parallel section.

```
do i=1,n
      call omp_init_lock(lck(i))
end do
C=1
...
!OMP PARALLEL PRIVATE(i,j,k,thrid,ipvt)
  id=omp_get_thread_num();
!$OMP DO SCHEDULE(STATIC,C)
  do j=1,n
      do i=1,n
          A(i,j)= rand(0)
          B(i,j)=A(i,j)
      end do
      pvt(j)=j
      LHS(i)=0
      A(j,n+1)=1
      call omp_set_lock(lck(j))
  end do
!$OMP END DO
!Matrix elimination
...
!OMP END PARALLEL
```

At beginning of each iteration each thread has to test the lock. So it will not succeed to enter the iteration unless that lock is unset by the pivot holder of that iteration. The code should look like the next listing:

```
!$OMP PARALLEL ...
 id=omp_get_thread_num();
 !Initializations
 if (id.eq.0)
     PIVOTINGÂ and ELIMINATINGÂ Column 1
     call omp_unset_lock(lck(1))
 end if
```

```
 do k=1,n-1
      call omp_set_lock(lck(k))
      !$OMP FLUSH
      call omp_unset_lock(lck(k))
 !$OMP DO SCHEDULE(STATIC,1)
      do j=1,n+1
          if (j>k) then
            DOÂ ELIMINATIONS
            if( j.eq. k) !Next pivot
                PIVOTINGÂ and ELIMINATING column j
                !$OMP FLUSH
                omp_unset_lock(lck(j))
            end if
          end if
      end do
 !$OMP END DO
!$OMP ENDÂ PARALLEL
```

Note that the threads can start iterations independently if their locks are free. So they only have compulsory stall if the next iteration column is not set yet which is the default natural constraint of the algorithm.

Listing 5.1. Original code

```
1 Do k=j+1, n
2      A(k,j)=A(k,j)/A(j,j)
3 end Do
```

Listing 5.2. Modified code

```
1 c=1/A(j,j)
2 A(j+1:n,j)=A(j+1:n,j)*c
```

Figure 5.3. Enhanced column elimination using fortran (:) notation

Enhancements

Note also that the code has four places that can be enhanced:

- Replace the division by pivot – which is constant at each iteration once it is known – with a multiply by a constant equals to inverse of pivot. See modification in Figure 5.3.
- Avoid loop invariant access in the k-loop.
- Eliminate the check for changing pivot position. If we allow the exchange to be done by default even if the pivot did not change position, it will simply replace the pivot row by itself. Although that might sound extra unnecessary work

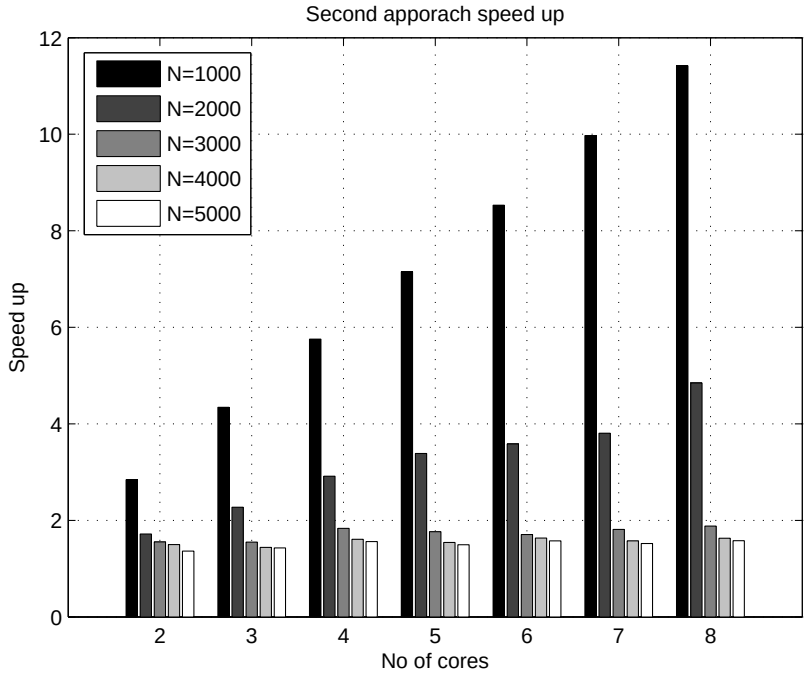

Figure 5.4. Speed up for the modified GE code: Experiment 2

in the case that pivot did not change position but by avoiding this condition, we allow the compiler to optimize the code better.

- Instead of using loops we replaced it by Fortran notation (:) for array updates. It works better with the compiler as in Figure 5.3.

So After modifications, the new version will be as:

```
!$omp parallel private(i,j,k,tmp)
if id==0 then
    newPvtpos=GetPivot(1);
    exchange( A(newPvtpos,1) , A(1,1))
    tmp=1/A(1,1)
    !Note new notation
    A(2:n,1)=A(2:n,1)*tmp
    unlock column 1
end if
for k=1 to n-1
{
    lock column k
    flush cache
    unlock column k
```

```
!$omp for schedule(static, 1)
    for j=1 to n+1
      if(j>k) then
            exchange( A(newPvtpos,j) , A(i,j))
         tmp=A(k,j)
         !Note new notation
         A(k+1:n,j)=A(k+1:n,j)-A(k+1:n,k)*tmp;
         if mod(k,NThreads)==id i.e next pivot holder
            newPvtpos=GetPivot(j);
            exchange(A(newPvtpos,j) , A(j,j))
            tmp=1/A(j,j)
            !Note new notation
            A(j+1:n,j)=A((j+1:n,j)*tmp
            unlock column j
         end if
      end if
    end for
!$omp end for
!$omp end parallel
```

We can see that the speed up in Figure 5.4 is improved when the matrix size is up till 2000. But as the matrix size gets larger than 2000, speed up is very low. We are getting a cache problem and the reason – as known from the Acumem report – is low temporal locality. Again the way that we handled the fixed partitioning of the the matrix caused massively unnecessary loop counter checking and incrementings to get the loop in the desired position.

5.2.3 Blocking failure: Experiment 3

Listing 5.3. Original code

```
1 !$omp for schedule(static, 1)
2     for j=1 to n+1
3        if(j>i) then
4             .....
5        end if
6     end for
7 !$omp end for
```

Listing 5.4. Modified code

```
1 !Remove the omp for
2 for j in my scope of the matrix
3    .....
4 end for
```

Figure 5.5. Remove OMP for and use fixed cyclic column distribution

In order to overcome the problems of the previous experiments. We build new version in which manually divide the matrix into cyclic distribution of size 1. Such division is virtual so just depending on the thread id, the scope of the matrix column

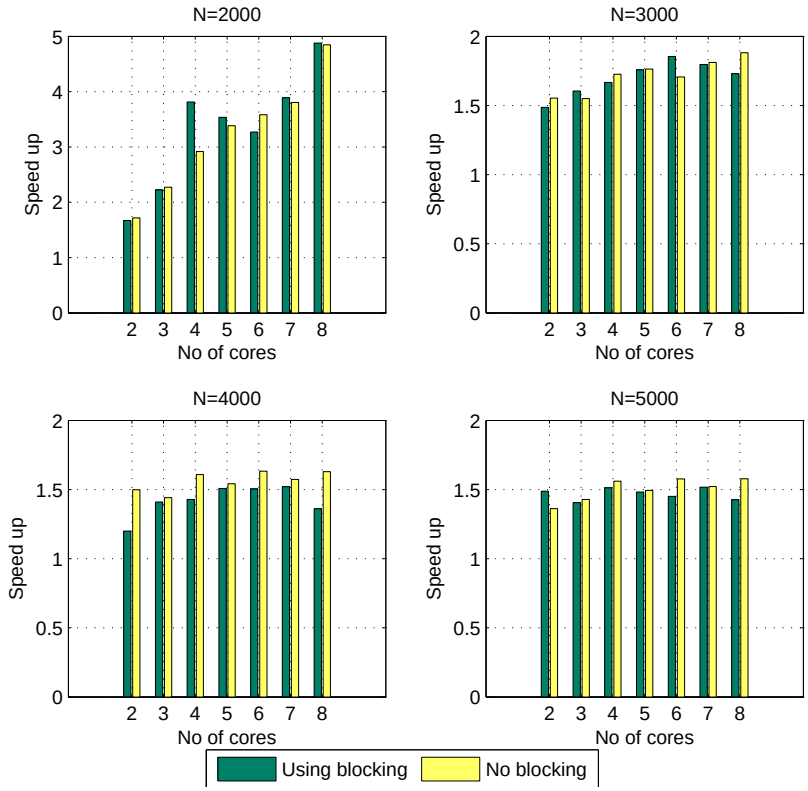

Figure 5.6. Speed up with blocking of GE code: Experiment 3

is known. Hence, we can remove OMP loop. This can be determined by different ways. We just simply stored first and last indeces of the thread's scope and jump with number of threads. The scope is adjusted every time it –*the thread*– happened to be the pivot holder. See the modification in the Figure 5.5

That gain something when the matrix was relatively small (till n=2000) then it made no difference. So we implemented blocking for the scope of the matrix but as in Figure 5.6 it was not improving in fact it caused many more problems including false sharing, inefficient loop nesting and more problems with locality and stalling threads waiting for pivot columns.

5.2.4 Double Elimination: Experiment 4

In this experiment we used the same way for partitioning the matrix manually by columns as in experiment 3. Just with a small observation that the next pivot column is done first thing in the main loop – or it has to be done at first in order to minimize the waiting time for other threads – that make the next pivot in hand for the pivot holder in the same iteration. So instead of waiting until next iteration to do it, the pivot-holder firstly eliminates its column $j > (i + 1)$ with the column

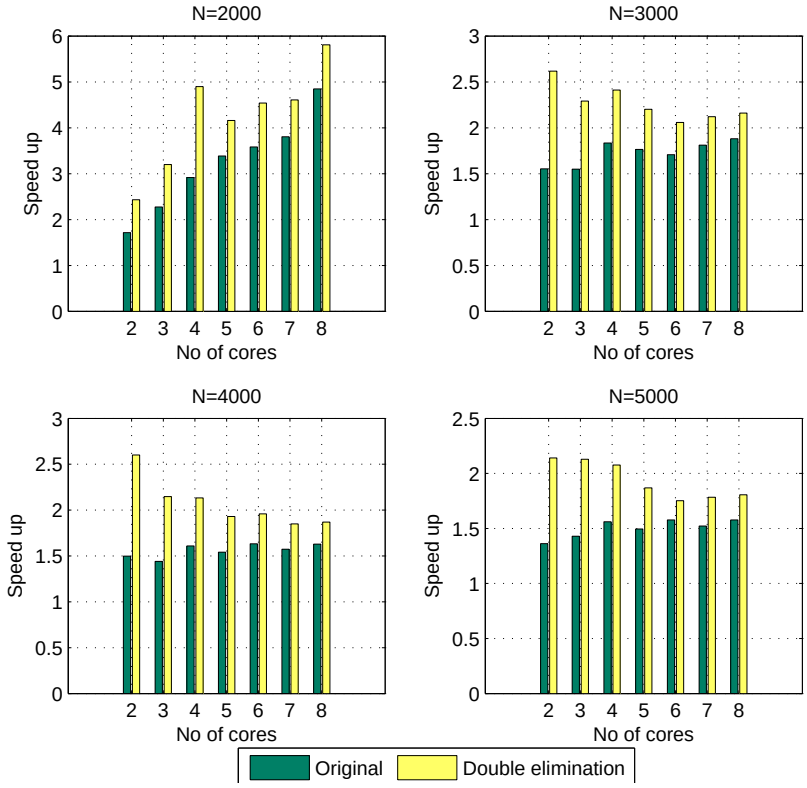

Figure 5.7. Speed up using double elimination for pivot holders: Experiment 4

i then with the next-iteration pivot-column which is definitely $(i+1)$ and skip the next elimination. Now as in Figure 5.7 we can get speed up.

But we can see that if the threads are 2, it was perfect. More than two almost was the same. That is the effect of that double elimination will only cause the threads to skip one iteration every *Nthreads* iterations.

5.2.5 Chunk distribution with double elimination : Experiment 5

The inefficient temporal locality in the prevoius experiments came from the nature of the algorithm to sequentially sweep the whole $j > i$ columns of the matrix at each iteration i. Using the inspiration of experiment 4, it would be great if there are more than one pivot available in each iteration so we can skip iterations somehow.

In this experiment, we used cyclic distribution with size greater than 1 as in Figure 5.8. Although that might give worse load balancing and increase the sequential part of the program – of eliminating first chunk – but it will enable the pivot holder to produce as much pivots as the chunk size **C**, allowing others to do as much as **C** eliminations and skip those **C** iterations –that were done in sequence before– and further more the pivot-holder to do twice. Again we also used locks array for

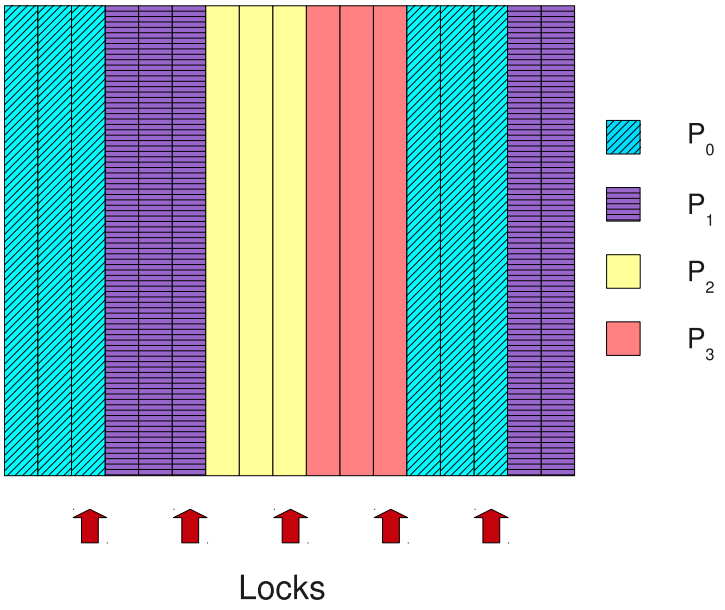

Figure 5.8. Cyclic Chunk Distribution with C=3 on 4 processes

the synchronization but the size of the locks array is $\lfloor n/C \rfloor$. Each lock is unlocked by the pivot holder after the chuck of pivots is computed. In this technique we are aiming to reduce number of times we sweep the whole matrix which caused the eviction of matrix columns from the cache at each iteration.

Comparing the speedup with what we had before was great. In Figure 5.9 we can see the perfect speedup we had. Figure 5.10 show how much gain we get by increasing the chunk size $C = 25$ w.r.t the previous speedup we gained with double elimination only (i.e. chunk size $C = 1$) and the speedup we gained earlier in Experiment 2. Figure 5.11 shows how the efficiency in the three cases showing the best with using chunks with double elimination.

However by increasing the block size more than 25, the performance started to drop due to the fact that the chunk does not fit into the cache any more specially as the input size increases.

5.3 Comparison to LAPACK

LAPACK and BLAS are the software libraries that are often used by numerical applications [12, 16, 17]. We used the intel MKL module that is available on Ferlin for the LAPACK usage as mentioned before in Chapter 2. DGETRF is the LAPACK routine that computes the LU factorization and for that it uses a block algorithm. The non-singular matrix A is partitioned into four sub-matrices $A_{1,1}$, $A_{1,2}$, $A_{2,1}$ and

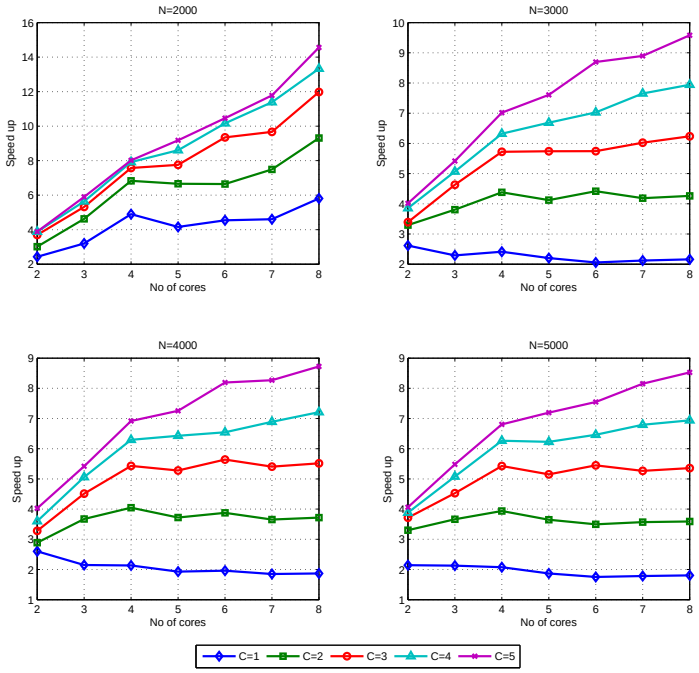

Figure 5.9. Speed up for the chunk GE code: Experiment 5

$A_{2,2}$ and the factorization is written as:

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} = \begin{bmatrix} L_{1,1} & 0 \\ L_{2,1} & L_{2,2} \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} \\ 0 & U_{2,2} \end{bmatrix}$$

And from that the following equations can be obtained:

$$\begin{aligned} A_{1,1} &= L_{1,1}U_{1,1}, \\ A_{1,2} &= L_{1,1}U_{1,2}, \\ A_{2,1} &= L_{2,1}U_{1,1}, \\ A_{2,2} &= L_{2,1}U_{1,2} + L_{2,2}U_{2,2} \end{aligned}$$

So $L_{1,1}$ and $U_{1,1}$ are computed using the standard LU factorization routine. Using the previous relations $L_{2,1}$ and $U_{1,2}$ are determined using Level 3 BLAS solvers for triangular systems. Recursively applying the block algorithm to the last relation, the matrices $L_{2,2}$ and $U_{2,2}$ are then calculated.

We compared the results and execution times that we got from DGETRF and our improved code in Figure 5.12. The source of errors in both methods are due to the rounding errors. In DGETRF the search for the pivots is done only in the block $A_{1,1}$ of each recursive step which may lead to different choice of pivots and corresponding to different round-off error due to finite precision arithmetic.

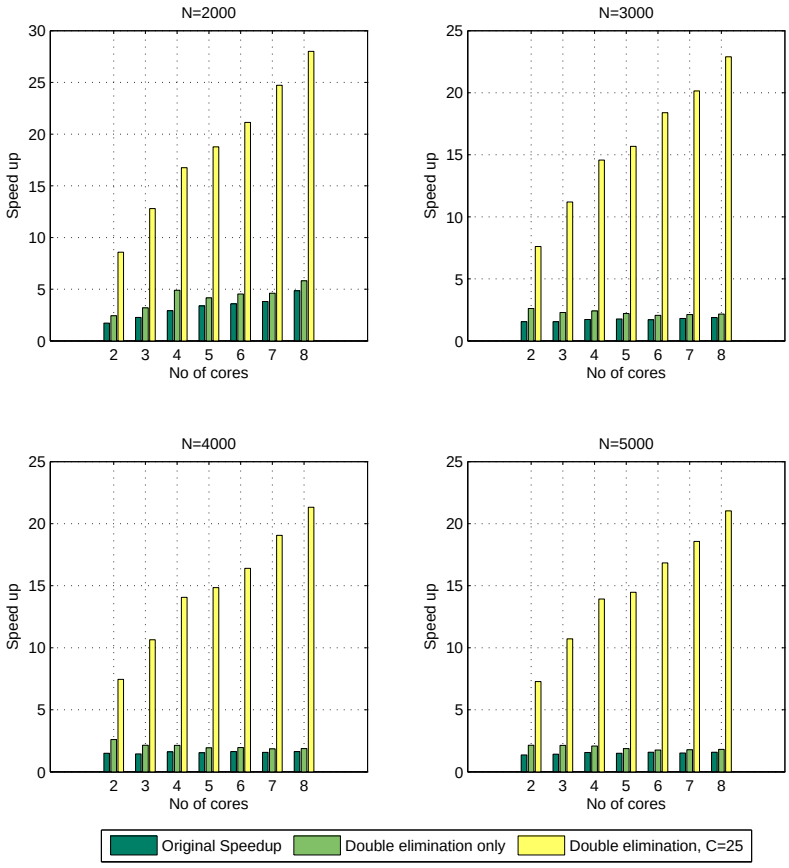

Figure 5.10. Comparison between Speedup before and after using chunks and double elimination

Comparing the time used in both method one will conclude it is quite efficient to use the optimized libraries whenever available. We were not interested to write a code that is faster from the library but to show how a simple code can be improved by giving a cache-aware solution and in meanwhile add new complex code statements.

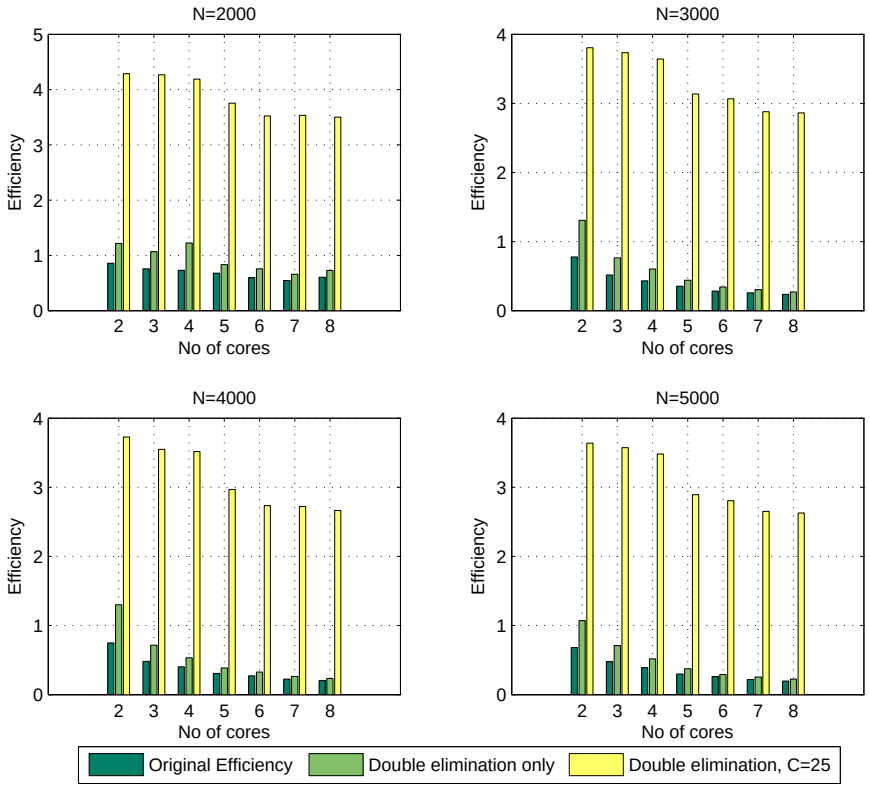

Figure 5.11. Comparison between efficiency before and after using chunks and double elimination

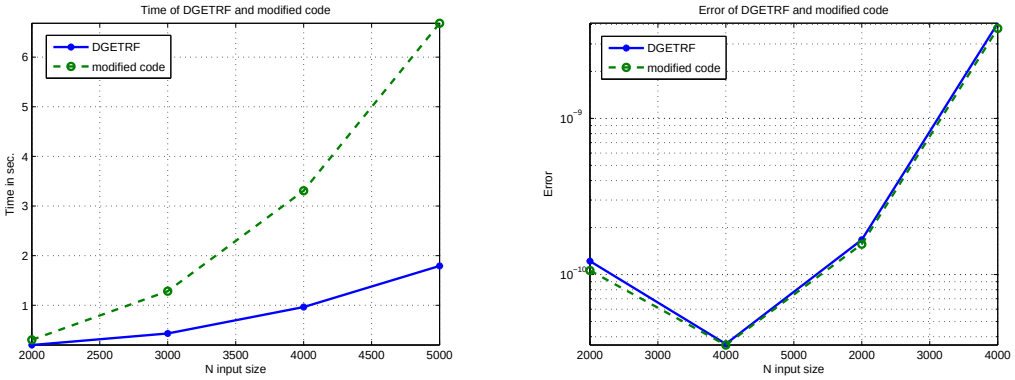

Figure 5.12. Comparison between DGETRF and the modified code using chunks and double elimination

Chapter 6

Case study: Periodic Stokeslet PS

The second case study was optimization for a numerical algorithm based on boundary integral formulation [18, 19]. If we consider a flat plate horizontally placed in our computational domain, orthogonal to the z direction, and call it Γ. Given the known free-space Green's function for the Stokes flow (Stokeslet) we can rewrite the flow equations as an inverse problem. This means that for a known velocity at the wall we retrieve the forces acting on the wall and consequently will obtain the velocity everywhere in the domain.

The integral form of the Stokes equations is given by:

$$U(x) = \int_{x \in \Gamma} S(x, y) f(y) dy$$

$$S(x, y) = S_0(x, y) + S_p(x, y)$$

where S is the total fundamental solution for the Stokes flow, S_p is the periodic remainder that comes from the periodic boundary conditions and S_0 given by:

$$S_0(x, y) = \frac{I}{|R|} + \frac{RR}{|R|^3}$$

where $R = x - y$ and RR is the dyadic product.

The discretization leads to a linear system of equation:

$$Af = u$$

The matrix A is block-Toeplitz-symmetric with circulant sub blocks, and the specific structure is preserved for A^{-1}. Keeping in mind that actually the matrix A corresponds to the discretization of two walls one must note that six columns (or rows) are enough to generate this matrix.

Figure(6.1) shows the structure of A. Recall that at each direction of the matrix, we have 6 parts that contain the same elements but in different order. The digit 6 came from the fact that we have 3D problem and two walls. Despite that the matrix is $(6N^2) \times (6N^2)$ it is sufficient to store $(6N)^2$ elements that we will refer to as *the mask of* A which will be used to reconstruct the full matrix A.

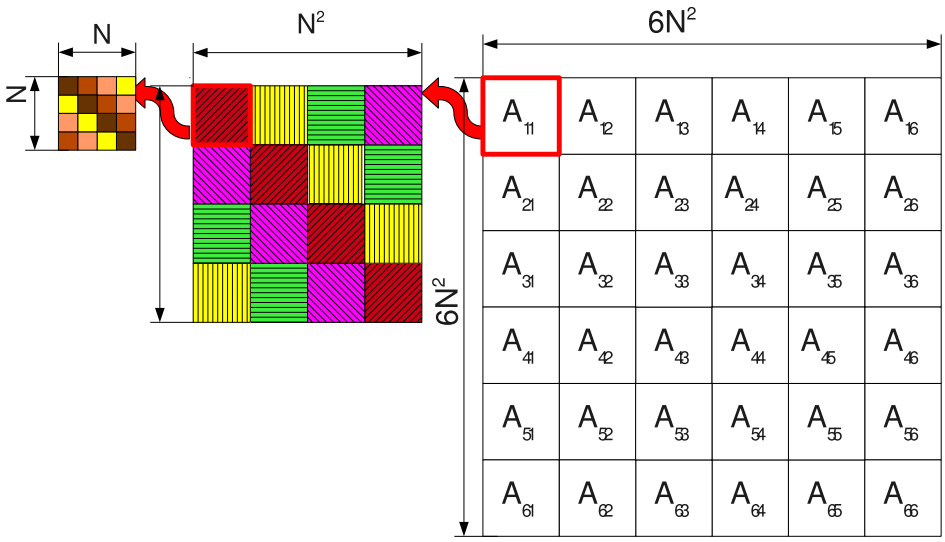

Figure 6.1. Matrix A structure At each dimension 6 Blocks of size $((N^2)^2)$, each of those blocks has (N^2) sub-blocks that are circular toeplitz, further more each of those sub-blocks has (N^2) sub-sub-blocks that are also circular toeplitz. The figure shows example if N=4, A has $(6N^2)^2 = 9216$ elements

As mentioned earlier that the inverse matrix A^{-1} shares the same structure as A, A^{-1} can be constructed from a corresponding *inverse mask*.

Generalized minimal residual method ,*GMRES* [20, 21], is a good option for this since does not require the storage of the matrix A but it suffices to be able to compute the matrix vector product Ax:

$$Ac_i = e_i, i = 1, N^2 + 1, 2N^2 + 1, .., 5N^2 + 1$$

where e_i is the $(6N^2)$-unit vector with zeros everywhere except one at position $((i-1)N^2+1)$ and that would return the generating columns c_i of the inverse A^{-1}. Same e_i was taken as initial guess c_i^0 for the GMRES.

6.1 Code Analysis

The code for this case was originally built at NADA/KTH –*by Oana Wiklund*. She used the GMRES code [22] with special adaptation to compute A^{-1} using the mask matrix. She also used OpenMP directives to parallelize the matrix-vector multiplication inside the GMRES iteration. From the acumem analysis report of the code, we received several issues at the call of the routine that computes the matrix vector product $y = Ax$. The routine MATVECPROD as shown in figure(6.2) uses the $(6N)^2$ mask to generate the elements of the matrix A and perform the matrix vector product. The inefficient order of executing the product was responsible for 87.6% of cache misses.

```
subroutine matvecprod(x,y,mask)
!this computes a matrix vector product Ax needed by GMRES
!x       (input)  vector
!mask    (input)  mask of the invariant matrix elements
!y       (output) result of Ax

use parameters, only: Nx,Ny,syssize
real :: x(syssize), y(syssize)
integer :: i,j,ii,jj,k,kk,indr,indc,indx,indy,kww,kw
real ::mask(6*Nx,6*Ny)
y=0.0d0

!$omp parallel do private(kww,kw,kk,k,i,j,ii,jj,indr,indc,indx,indy)
do jj=1,Ny
  do ii=1,Nx
    do j=1,Ny
      do i=1,Nx
        do k=1,3
          do kk=1,3
            do kww=1,2
              do kw=1,2
                indr=(kww-1)*Nx+(kk-1)*Nx*2+modulo((Nx/2+i-ii),Nx)+1
                indc=(kw-1) *Ny+(k-1)*Ny*2+modulo((Ny/2+j-jj),Ny)+1
                indy= Nx*Ny*(kk-1+(kww-1)*3)+(jj-1)*Nx+ii
                indx= Nx*Ny*(k-1 +(kw-1)*3)+(j-1)*Nx+i
                y(indy)=y(indy)+mask(indr,indc)*x(indx);
              end do
            end do
          end do
        end do
      end do
    end do
  end do
end do
!$omp end parallel do
end subroutine matvecprod
```

Figure 6.2. Original code for Matrix-Vector multiplication

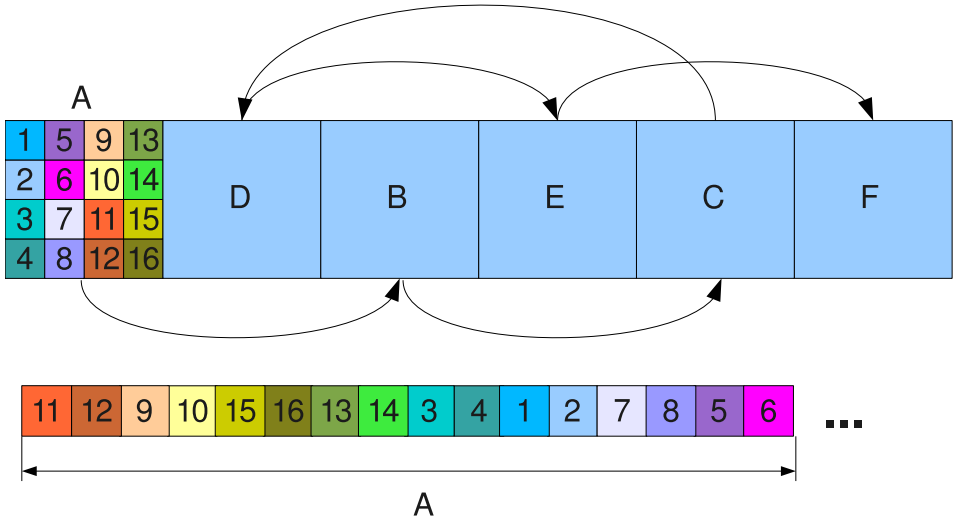

Figure 6.3. Building first row of the full matrix. At top 1-of-6 chunks in the mask. The first row is built by accessing the blocks ABCDEF. Each block is accessed in same way as the colored A to get the corresponding N^2 elements of the row

The slow performance of the code arises from the fact that way we access the vectors and mask is not structured and that what shown from the problems in the report such as the inefficient loop- nesting, poor temporal and spacial locality and low fetch utilization for each variable we access. The formula used to get the correct indices causes jumps that are hard for the compiler to predict and lacks locality. The aim to improve this routine will be to find means to access the data structures in an ordered way.

6.2 Optimization

6.2.1 Ordered Mask

As we said before the matrix A can be generated from the mask, and so it was interesting to see how to use the mask to build a row of A. If we divide the mask into 6 rows-chunks, each one is $N \times 6N$, we can use this mask to find the first row of each block $N^2 \times 6N^2$ of the matrix A in exactly the same fashion as in figure (6.3). If we futher more divided that chunk into 6 blocks, each block will contain the N-elements of the original desired row.

For first row at each row-chunk, this is not entirely inefficient since in order to do this arrangement we access the mask column-wise but still we jump a lot. going to next rows will lead to inefficient access mechanism. The way the mask is built still convenient for visualization and numerical purpose. However by arranging the mask this way into $6N^2$-vectors representing the major rows of A, we can perform

the matrix vector multiplication in better way. The idea is to use those vectors with the correct permutation to perform the routine MATVECPROD2 as in (6.4).

6.2.2 Performance Gain

The comparison is made between three ways to perform $y = Ax$. The old method was as described in `matvecprod` using the generated mask. The new one was using the ordered mask rows as in `matvecprod2`.

It was also interesting to also include the traditional method to perform $y = Ax$ by using the mask and function `matindex` that returns the value of the matrix at given position using the mask as if we have A stored in memory:

```
do i=1,syssize
   do j=1,syssize
      y(i)=y(i)+b(j)*matindex(i,j,mask,Nx,Ny)
   end do
end do
```

Tables (6.1) and (6.2) show how the new method using the ordered mask is superior and takes less time. However the degree of parallelism in the upper level of the `matvecprod2` is only 6 meaning we can only able to divide it over at maximum 6 threads , it still much efficient to use that routine.

		Time		
N	System size	New	Old	Traditional
16	1536	1.79e-3	9.399e-3	6.789e-2
32	6144	2.159e-2	0.1578	0.759
64	24576	0.3498	2. 497	17.242
128	98304	6.223	54.591	190. 935

Table 6.1. Performance Comparison w.r.t. sequential time

	Time		
nThreads	New	Old	Traditional
8	1.356	8.02	24.07
6	1.357	11.107	31.847

Table 6.2. Performance gain due parallelization in case of N=128

6.2.3 Overall performance gain

In order to test the overall performance gain we compared time of the old code and the modified code using residual $r = 10^{-13}$. Note that such residual is used to determine the accuracy of the inverse mask columns and hence the total number of iterations used by GMRES.

```
subroutine matvecprod2(x,y,Rows)
!this computes a matrix vector product Ax needed by GMRES
!I assumed Nx=Ny otherwise some conditions have to be changed
!x       (input)  vector
!Rows    (input)  Rows at positions (i-1)*Nx*Ny+1, i=1:6 as columns
!y       (output) result of Ax
use parameters, only: Nx,Ny,syssize

real , intent(IN):: x(syssize),Rows(syssize,6)
real ,intent(OUT):: y(syssize)
real :: buf(Nx), element
integer :: i,j,k,l,m,N=Nx
integer :: indBB, indB, s, indYs,indY, Bend

y=0.0d0
!$omp parallel
!$omp do private(i,j,k,l,m,s,indBB,Bend,indB,buf,indY,indYs,element)
do i=1,6  !big rows indY=(i-1)*N*N+1
  do j=1,6  !big columns
    indBB=(j-1)*N*N+1
    Bend=j*N*N
    do k=1,N !inside big block
       indB=indBB+(k-1)*N
       buf=Rows(indB:indB+N-1,i)
       indYs=indY
       do  m=1,N !inside small block
          s=indB
          do l=1,N
             y(indYs)=y(indYs)+sum(x(s:s+N-1)*buf(1:N)) ! N multiplications
             indYs=indYs+N
             s=s+N
             if(s >= Bend) then
                s=indBB
             end if
          end do
          element=buf(N)
          buf(2:N)=buf(1:N-1)
          buf(1)=element
          indYs=indY+m
       end do
    end do
  end do
end do
!$omp end do
!$omp end parallel
end subroutine matvecprod2
```


Figure 6.4. The new optimized code for Matrix-Vector multiplication

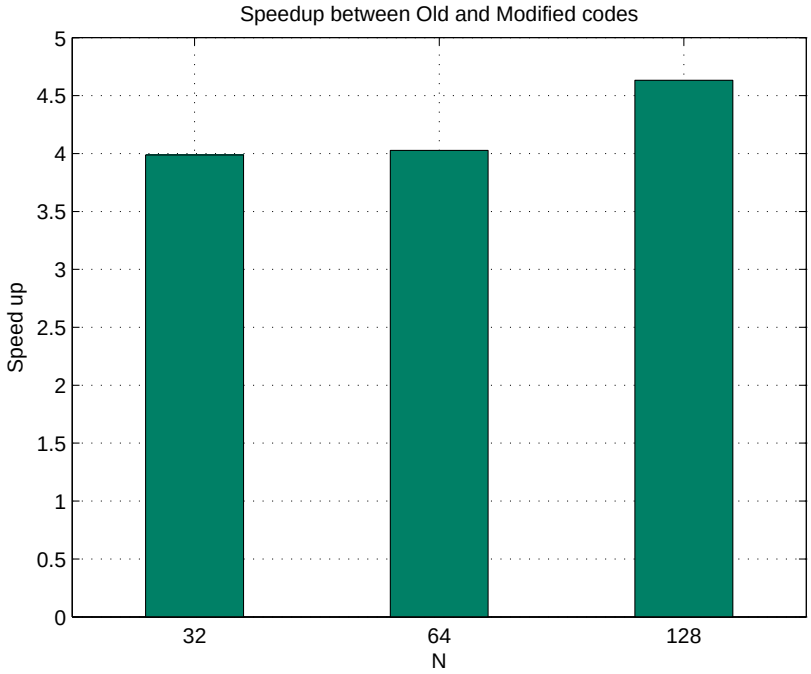

Figure 6.5. Speedup between old and modified code

N	New	Old
32	3.4357	13.6959
64	63.1757	250.3774
128	1498.6391	6941.5962

Table 6.3. Average execution time in sec. using residual $r = 10^{-13}$.

Table (6.3) shows the average time between the old and modified code that uses the ordered mask. Figure (6.5) shows the speed up we got given by:

$$S_p = \frac{T_{old}}{T_{new}}$$

where T_{new} is total time needed to reorder the mask and find the solution with same residual settings. We got speed up at least 4 and the reason that it grew as we reach N=128 is due to the fact that the mask -and similarly the ordered one - does not fit into the cache any more and so we can get more intuition of the importance of using ordered way to access its elements. However, we still have overheads from constructing and destroying the threads at each iteration but the GMRES routine is not thread-safe [21, 22].

Chapter 7

Conclusions

The demands for computational power and application to run faster or be able to serve expanding users requests are continuously increasing. Since single-processors can not get any faster due to technical hardware and power consumptions issues, parallelism needs to be exploited, for instance on multi-core systems wich provide significantly increased computational power, but also new challenges and restrictions. Getting the expected performance gain from migrating to multi-core is highly dependent on the application implementation, data layout and access patterns. In shortly, multi-core programming does not only require parallelizing a sequential code but –in order to get speed-up in the performance– one has to consider utilization of cache usage. Although this is also an issue on single core sytems, multi-core processors have even more demands on proper cache utilization. Immature task and data partitioning will lead to waste performance as cores will be always fighting over shared resources. If developers are unable to design software to fully exploit the resources provided by multiple cores, then they will ultimately reach a frustrating performance ceiling.

In our study we studied two types of algorithms to solve linear systems of equations as important step in most of numerical methods to solve partial differential equations. The experiments and results obtained on the 8-cores system. We were able to show that data locality has a great influence on the performance of the algorithms and how the application of performance optimization steps can help overcome these limitations.

The first case study was *Gaussian Elimination algorithm* (GE) to perform the LU factorization of a full non-singular matrix. The naive parallelization using OpenMP directives only show a poor speed-up (less than 1.8) for array sizes less than 2000 and a slow-down for large array sizes (speed-up less than 1). Using cyclic column-chunk distributions to split the matrix between different threads and the parallel technique we called *Chunk GE with double elimination*, we were able to decrease execution time dramatically. For chunk size 25 we got speed-up over than 30 for array size less than or equal 2000 and about 25 for large matrix sizes in

comparison with the sequential algorithm. We also compared the results with the LAPACK library routine DGETRF that computes the LU factorization. For small array sizes we got the same order of execution time but for array sizes larger than 2000, DGETRF was still about 3 times faster.

The second case study was a numerical algorithm based on boundary integral method. The algorithm uses the *Generalized minimal residual method* (GMRES) as iterative method to solve linear system of equations. The matrix A in that algorithm has a special structure which is block-Toeplitz-symmetric with circulant sub blocks, and the specific structure is preserved for A^{-1}. At each dimension, A has 6 Blocks of size $((N^2)^2)$, each of those blocks has (N^2) sub-blocks that are circular toeplitz, further more each of those sub-blocks has (N^2) sub-sub-blocks that are also circular toeplitz. The total number of elements in A has $(6N^2)^2$ elements. However instead of storing the whole matrix, it was sufficient to store a smaller $6N \times 6N$ array –called the mask– and generate the matrix elements from it. GMRES required an implementation of the matrix-vector product which is implemented using the mask. By replacing the mask with an ordered mask that is $N^2 \times 6$ representing the starting row of the 6 row-blocks of matrix A, we provided a new mechanism to perform the matrix-vector product. The new method was based on the concept of exploiting data reusability and regular access patterns. We were able to speed-up overall computations of the algorithm by factor of 4 for sizes 32 and 64 and by 4.5 for 128.

In our experiments we also exploited cache and memory profiling tools, specially for the analysis of complex structures, separated data files and/or multi-threaded applications. The use of smart tools becomes vital when we migrate from single- to multi-threaded applications. It is not always easy to spot bottlenecks and their causes in such codes. Moreover, a deeper understanding of the code and expected results is really necessary and case dependent. In general, Optimization is not always an obvious or intuitive process.

Through optimization, codes can lose readability and portability and this makes maintenance of the code very hard. The recommendation is to delay optimization and keep the development of the solution as simple as possible in the early stages where stability, robustness and convergence behavior are most of the concerns. Although compilers nowadays are able to perform many optimizations, the results are still far from being optimal and thus it is still the programmer's responsibility to applying optimization techniques in order to help the compiler producing efficient code. The reason is that many performance bottlenecks cannot be detected and addressed by static compiler analysis. One way to overcome these problems is to use dynamic runtime analysis and optimization, as for instance aimed at in the latest developments of the Acumem tool, that was used in this work for static analysis. Until such techniques are able to produce stable and efficient results, producing performing code is as hard as finding a correct solution to any problem.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3 ed., 2002.

[2] L. J. FLYNN, "Intel halts development of 2 new microprocessors." `http://www.nytimes.com/2004/05/08/business/08chip.html?ex=1399348800&en=98cc44ca97b1a562&ei=5007`, May 8 2004. [June, 2010].

[3] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill Science/Engineering/Math, 1 ed., 1992.

[4] D. A. Patterson and J. L. Hen, *Computer organization and design : the hardware/software interface*. Morgan Kaufmann Publisher Inc., 1997.

[5] I. O. Articles, "Optimizing software applications for numa." `http://software.intel.com/en-us/articles/optimizing-software-applications-for-numa/`, June 2010.

[6] S. A. McKee, "Reflections on the memory wall," in *CF '04: Proceedings of the 1st conference on Computing frontiers*, (New York, NY, USA), p. 162, ACM, 2004.

[7] "Pdc resources, ferlin specifications." `http://www.pdc.kth.se/resources/computers/ferlin/`, June 2010.

[8] Intel, "Intel xeon processors." `http://ark.intel.com/ProductCollection.aspx?codeName=26555`, June 2010.

[9] "Official openmp manual specifications." `http://openmp.org/wp/openmp-specifications/`, June 2010.

[10] K. Hwang and Z. Xu, *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill Science/Engineering/Math, 1 ed., 1998.

[11] "Ibm compiler optimization flags." `http://www.nersc.gov/nusers/resources/software/ibm/opt_options/`. [June 2010].

[12] M. Kowarschik and C. Weiß, "An overview of cache optimization techniques and cache-aware numerical algorithms," in *Algorithms for Memory Hierarchies*, pp. 213–232, Springer Berlin / Heidelberg, 2003.

[13] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 63–74, ACM, 1991.

[14] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, (New York, NY, USA), pp. 245–254, ACM, 2009.

[15] ACUMEM, *Acumem ThreadSpotter^{TM} Manual*, 2010.0 ed., Nov. 2009.

[16] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "Lapack: a portable linear algebra library for high-performance computers," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, (Los Alamitos, CA, USA), pp. 2–11, IEEE Computer Society Press, 1990.

[17] "An updated set of basic linear algebra subprograms (blas)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, 2002.

[18] A.-K. Tornberg and K. Gustavsson, "A numerical method for simulations of rigid fiber suspensions," *J. Comput. Phys.*, vol. 215, no. 1, pp. 172–196, 2006.

[19] C. Pozrikidis, *Boundary Integral and Singularity Methods for Linearized Viscous Flow*. Cambridge University Press, 1992.

[20] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, 1986.

[21] S. Maria, W. Layne T., and K. Rakesh K., "A new adaptive gmres algorithm for achieving high accuracy," tech. rep., Blacksburg, VA, USA, 1996.

[22] J. Claerbout. http://sepwww.stanford.edu/sep/prof/geelib/gmres.f90. Stanford University.