Lappeenranta University of Technology
Department of Information Technology

Tuning Performance of Multi-threaded programs

Master's thesis

The topic of the thesis has been confirmed by the Departmental Council of the Department of Information Technology on 5th May, 2004.

Supervisor: Prof. Jari Porras
Examiner: Prof. Jari Porras

Lappeenranta, 15th May, 2004

Ivo Zatloukal
Karankokatu 4B6/1
53810 Lappeenranta
FINLAND
+358408685457

Abstract

Lappeenranta University of Technology
Department of Information Technology

Ivo Zatloukal

Tuning Performance of Multi-threaded programs

Thesis for the degree of Master of Science in Technology
2004
91 pages, 60 figures, 18 tables
Supervisor: Professor Jari Porras
Keywords: parallel programming, multi-threading, hyper-threading, performance tuning

The thesis discusses the multithreaded programming as the upper level in parallel programming hierarchy, with focus on the hyper-threading technology. It discusses its pros and cons and its effects on various parallel algorithms. The motivation of this thesis is to understand hyper-threading as is implemented in Intel Pentium 4 processor and to enable its utilization where it brings performance advantage.

A large set of benchmarking programs was executed in varying conditions (memory access pattern, compiler settings, environment variables…) to collect and analyze performance metrics. Two sets of algorithms were evaluated: matrix operations and sorting. These applications have regular data access pattern, which is a double sided weapon. While this is an advantage in arithmetic-logic processing, memory performance suffers. The reason is that the raw performance of today's processors in processing regular data is very good, but memory architecture is limited by the size of caches and various buffers and once the problem size crosses certain limit, the actual performance can drop to a fraction of the top performance.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto

Tietotekniikan osasto

Ivo Zatloukal

Säikeistettyjen ohjelmien suorituskyvyn parantaminen

Diplomityö

2004

91 sivua, 60 kuvaa ja 18 taulukkoa

Tarkastaja: Professori Jari Porras

Hakusanat: rinnakkaisohjelmointi, monisäikeistys, hypersäikeistys, suorituskyvyn parantaminen

Diplomityö tarkastelee säikeistettyä ohjelmointia rinnakkaisohjelmoinnin ylemmällä hierarkiatasolla tarkastellen erityisesti hypersäikeistysteknologiaa. Työssä tarkastellaan hypersäikeistyksen hyviä ja huonoja puolia sekä sen vaikutuksia rinnakkaisalgoritmeihin. Työn tavoitteena oli ymmärtää Intel Pentium 4 prosessorin hypersäikeistyksen toteutus ja mahdollistaa sen hyödyntäminen, missä se tuo suorituskyvyllistä etua.

Työssä kerättiin ja analysoitiin suorituskykytietoa ajamalla suuri joukko suorituskykytestejä eri olosuhteissa (muistin käsittely, kääntäjän asetukset, ympäristömuuttujat...). Työssä tarkasteltiin kahdentyyppisiä algoritmeja: matriisioperaatioita ja lajittelua. Näissä sovelluksissa on säännöllinen muistinkäyttökuvio, mikä on kaksiteräinen miekka. Se on etu aritmeettis-loogisissa prosessoinnissa, mutta toisaalta huonontaa muistin suorituskykyä. Syynä siihen on nykyaikaisten prosessorien erittäin hyvä raaka suorituskyky säännöllistä dataa käsiteltäessä, mutta muistiarkkitehtuuria rajoittaa välimuistien koko ja useat puskurit. Kun ongelman koko ylittää tietyn rajan, todellinen suorituskyky voi pudota murto-osaan huippusuorituskyvystä.

Contents

1 Introduction **6**

2 Parallel programming **7**

2.1 SIMD processing *8*

2.2 Instruction level parallelism *9*

2.3 Multithreaded execution *10*

2.4 Problem statement *17*

3 Pentium 4 processor **23**

3.1 The Front end *24*

3.2 The Execution core *24*

3.3 Execution units *25*

3.4 Memory architecture *26*

3.5 Hyper-threading and parallel execution *28*

3.6 Performance counters *30*

4 Software environment **32**

4.1 Programming guides *32*

4.2 Execution environment *34*

5 Matrix operations **41**

5.1 Vector-vector multiplication *41*

5.2 Matrix-vector multiplication *48*

5.3 Matrix-matrix multiplication *62*

6 Sorting **73**

6.1 Quicksort ... 73

6.2 Radix sort ... 82

Conclusions ... 87

References ... 89

List of Figures

Figure 1 The pyramid of parallel processing 7
Figure 2 SIMD execution 8
Figure 4 Pentium processor pipeline 10
Figure 5 Video watermark detection 11
Figure 6 OpenMP Architecture 13
Figure 7 A processor with support for multithreading 15
Figure 8 On chip multithreaded execution 17
Figure 9 Problem focus 17
Figure 10 Parallelism choices 19
Figure 11 Memory distribution choices 20
Figure 12 Block partitioning between 2 threads 20
Figure 13 Cyclic partitioning between 2 threads 20
Figure 14 Block-cyclic partitioning between 2 threads 21
Figure 15 Thread scheduling choices 21
Figure 16 Static thread scheduling example 21
Figure 17 Dynamic thread scheduling example 22
Figure 18 The Pentium 4 processor architecture 23
Figure 19 Out-of-order execution engine detailed pipeline 25
Figure 20 Pentium 4 Execution units and Ports in the out-of-order core 26
Figure 21 Processors without hyper-threading technology 28
Figure 22 Processors with hyper-threading technology 29
Figure 23 Execution environment 34
Figure 24 Cyclic partitioning of a vector 42
Figure 25 Block partitioning of a vector 42
Figure 26 Block-cyclic partitioning of a vector 43
Figure 27 Vector-vector multiplication: Comparison of threading approaches (float data type) 46
Figure 28 Vector-vector multiplication: Comparison of threading approaches (double data type) 47
Figure 29 Row-wise matrix partitioning 50
Figure 30 Row-wise matrix-vector multiplication comparison (double data type) 51
Figure 31 Column-wise cyclic partitioning 52

Figure 32 Column-wise matrix-vector multiplication comparison (double data type) 52
Figure 33 Column-wise matrix-vector multiplication: FSB data activity metric (double data type)....53
Figure 34 Column-wise matrix-vector multiplication: Instructions retired metric (double data type) 54
Figure 35 Column-wise matrix-vector multiplication: L1 load miss retired metric (double data type) 54
Figure 36 Column-wise matrix-vector multiplication: 64kB alias conflicts metric (double data type) 55
Figure 37 Row-column-wise partitioning 55
Figure 38 Row-column-wise matrix-vector multiplication comparison (double data type) 56
Figure 39 Comparison of matrix-vector multiply algorithms (float data type)...................................58
Figure 40 Matrix-vector multiplication: Comparison of threading approaches (float data type) 59
Figure 41 Comparison of matrix-vector multiply algorithms (double data type) 60
Figure 42 Matrix-vector multiplication: Comparison of threading approaches (double data type)......61
Figure 43 Matrix-matrix multiplication loop parallelization: execution time (double data type).........64
Figure 44 Matrix-matrix multiplication loop parallelization: memory metrics (double data type, matrix size=1024) 65
Figure 45 Matrix-matrix multiplication block parallelization: execution time (double data type).......66
Figure 46 Matrix-matrix multiplication block parallelization: memory metrics (double data type, matrix size=1024) 67
Figure 47 Comparison of matrix-vector multiply algorithms (float data type)...................................68
Figure 48 Matrix-matrix multiplication: Comparison of threading approaches (float data type).........69
Figure 49 Comparison of matrix-matrix multiply algorithms (double data type)................................70
Figure 50 Matrix-matrix multiplication: Comparison of threading approaches (double data type) 72
Figure 51 Comparison of quick sort algorithms..77
Figure 52 Quick-sort: Normalized execution time metric 78
Figure 53 Quick-sort: Normalized FSB data activity metric 79
Figure 54 Quick-sort: Normalized instructions retired metric 79
Figure 55 Quick-sort: Normalized L1 load cache miss metric 80
Figure 56 Quick-sort: Normalized 64kB alias conflicts metric 80
Figure 57 Quick sort: Comparison of threading approaches...81
Figure 58 Comparison of radix sort algorithms 85
Figure 59 Radix sort: Comparison of threading approaches...86
Figure 60 Parallelism pyramid revisited 87

List of Tables

Table 1 Parallel models compared [38] ... 14
Table 2 Pentium 4 processor resources division ... 30
Table 3 Performance metrics ... 31
Table 4 Metrics gathered in experiments ... 38
Table 5 Vector-vector multiplication: Comparison of threading approaches (float data type)... 45
Table 6 Vector-vector multiplication: Comparison of threading approaches (double data type) .. 47
Table 7 Summary of matrix-vector multiply algorithms (float data type) ... 57
Table 8 Matrix-vector multiplication: Comparison of threading approaches (float data type)... 59
Table 9 Summary of matrix-vector multiply algorithms (double data type) ... 60
Table 10 Matrix-vector multiplication: Comparison of threading approaches (double data type) 61
Table 11 Summary of matrix-matrix multiply algorithms (float data type)... 68
Table 12 Matrix-matrix multiplication: Comparison of threading approaches (float data type) ... 69
Table 13 Summary of matrix-matrix multiply algorithms (double data type) ... 70
Table 14 Matrix-matrix multiplication: Comparison of threading approaches (double data type) 71
Table 15 Summary of quick sort algorithms ... 76
Table 16 Quick sort: Comparison of threading approaches ... 81
Table 17 Summary of radix sort algorithms ... 84
Table 18 Radix sort: Comparison of threading approaches ... 86

1 Introduction

This thesis presents the use of hyper-threading technology [1, 4, 5, 6, 10, 15] in the Intel Pentium 4 [1, 4, 9, 11] processor. The hyper-threading technology is a new architecture used in Pentium 4 and following generations of Intel's processors that makes a single processor look like two processors to the operation system and brings better performance.

The first part of this document presents the concept of parallelism available in the current computers, together with example use cases. The final section of this part also states the main objectives of the thesis. The next part introduces the hyper-threading technology and the Pentium 4 processor. It describes its architecture and functional units, their limitations and how they affect performance. This part is finished with a programming guide to write efficient code. The third part presents software packages related to hyper-threading and describes experiments executed to show the effect of hyper-threading on common problems in parallel computing. The evaluated algorithms are matrix operations and sorting. The thesis is finished with conclusion and literature survey.

The evaluation of this new technology is a complex task and an extra care must be taken to make the proper measurements. A lot of papers are available on this topic [12, 13, 16, 17], some listed in literature survey and the others can be easily found on the Internet. The existing results show that not always the use of hyper-threading technology brings performance advantage and this is also the topic of this thesis work.

2 Parallel programming

Traditional approach to achieve better performance is to increase machine clock rate and enable processing more instruction in one clock period. These techniques include speculative execution, deeper pipelines, larger caches and so on. It also makes the current processors very complex, with many millions transistors and huge power consumption. On the instruction level, new instructions are added to take advantage from regular data pattern that often appear in multimedia applications.

Except of speeding up the clock rate, there is another set of methods to increase performance. This package includes various sets of parallel execution, the basic method of parallel processing is SIMD processing. There, multiple data elements are processed with one instruction. Another step is instruction level parallelism (ILP), which executes multiple instructions concurrently in several execution units. A necessary condition to enable ILP is that all executed instructions are independent and thus there is no risk of data corruption. Generally the compiler exploits both SIMD and ILP parallelism and if the programmer follows usual coding standards there is no need for any special care. The top level of parallel processing is thread level parallelism (TLP), where the problem is split into tasks that are scheduled and executed independently by the operating system. Although there are attempts to automate parallelization [8, 13] by the compiler, the results are not satisfactory and generally the parallelization is still done by the programmer. Threads are executed concurrently on a machine with multiple processors. This can be a machine with several independent (typically 2 to 32) processors (symmetric multiprocessing, SMP), a machine with multiple processors on a single chip (chip multiprocessing, CMP) or a thread enabled processor with symmetric multithreading technology (SMT).

Disregarding large-scale parallel machines, these methods form a pyramid (Figure 1). The methods on the lower levels are more fine grained and machine dependent; top level methods are more general and applicable to large scale problems.

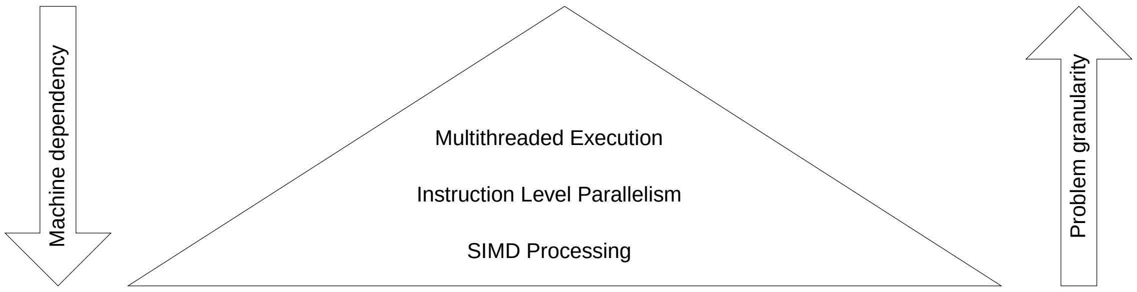

Figure 1 The pyramid of parallel processing

2.1 SIMD processing

A simple method to achieve parallel processing found in most of today’s microprocessors is support for parallel processing of multiple data elements in one instruction. This extension is called s*ingle instruction, multiple data processing* (SIMD). Using smaller data types saves memory and speeds up execution. For image processing it is often enough to represent a data element only with a single byte, sound processing often requires only 16 bit data samples. SIMD enabled processor can process typically 8×8-bit or 4×16-bit data elements. Figure 2 shows instruction that operates on 4 elements. All 4 elements are computed concurrently using input data from Source 1 and Source 2.

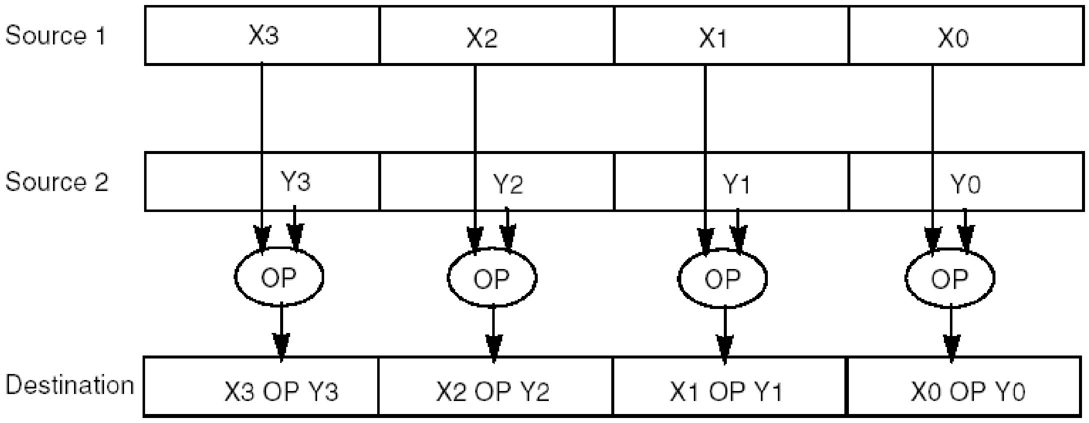

Figure 2 SIMD execution [4]

Intel uses SIMD technology in the MMX extension of the x86 instruction set. Many other processor architectures also use SIMD instruction set extensions (Alpha MAX, Sparc VIS or MIPS MDMX) [20]. SIMD operations usually include arithmetic, comparing, logical operations and packing/unpacking data. Generally, the arithmetic is performed using saturating operations.

Vector processing is a similar concept to SIMD processing. This technique is also targeted to scientific applications (mainly matrix and vector computations), the idea is to replace a program loop by one vector instruction and pipeline its execution. This reduces the overhead of loop termination tests and pointer arithmetic. Vector execution units also employs data forwarding, so that multiple vector operations can be in execution at the same time. Figure 3 shows vectorized execution of $D(i) = C(i) \times [A(i) + B(i)]$. Note the difference from SIMD processing: each vector execution unit operates only on one element at a time.

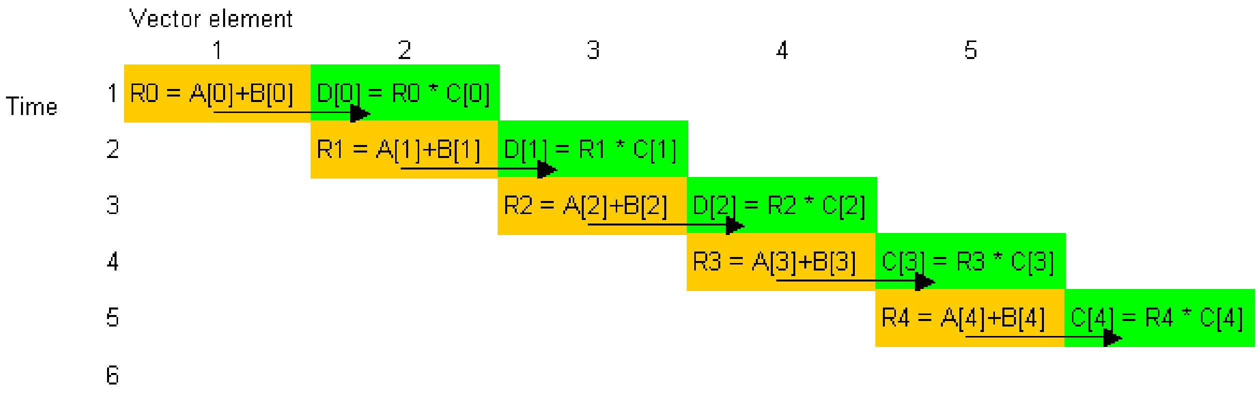

Figure 3 Vector processing [18]

2.2 Instruction level parallelism

In the past time a lot of effort was spent on increasing instruction-processing speed. There are several parameters that measure processor speed. One of them – Clocks per instruction (CPI) - defines how many instructions can be executed on average in one cycle. There are basically two ways to increase CPI:

Increase pipelining

It attempts to achieve higher performance by dividing instruction execution into simple steps (typically 5 to 7) that are executed sequentially. In the ideal case the performance of pipelined processor is *n* times higher than non-pipelined processor, where *n* is the number of pipe stages. Further fragmentation of the pipeline (to more than 10 stages) leads to super pipelining, where each stage is divided into sub-stages.

Multiple instruction issue

Multiple instruction issue processor must replicate some execution resources as execution units and data paths, but many other resources as register file, branch prediction unit and caches are shared. The instruction scheduler must handle access to shared resources

The most important limitation for exploring parallel processing is instruction dependencies and today's compilers schedule code into blocks of independent instructions to enable quick execution.

Figure 4 shows Pentium's pipeline. The Intel Pentium processor has two 5-stage integer pipelines and 7-stage floating-point pipeline, allowing executing multiple instructions in parallel in different pipe stages.

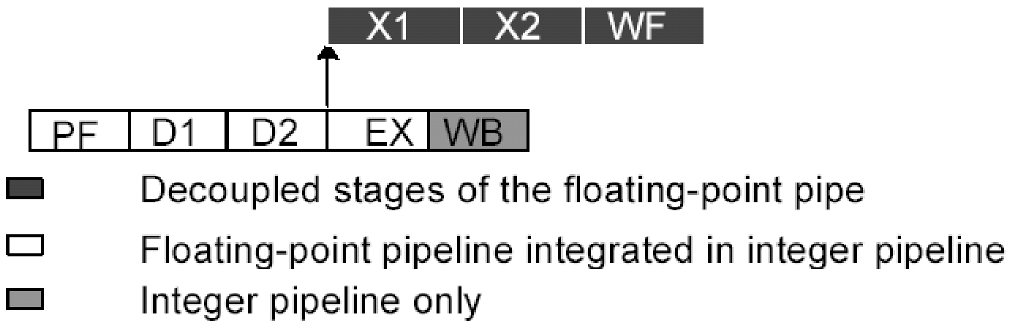
X1
X2
WF
PF
D1
D2
EX
WB
Decoupled stages of the floating-point pipe
Floating-point pipeline integrated in integer pipeline
Integer pipeline only

Figure 4 Pentium processor pipeline [4]

2.3 Multithreaded execution

In many modern operating systems the execution model is as follows: each program that executes is represented as a process. Process is created when program execution is requested. A process is a container for various resources and attributes of the program. Process also owns threads.

Process

- Owns address space and open resources (files, sockets...) that all threads share
- Holds information about user and the environment (User/Group ID, rights, current directory)
- Contains at least one thread

Thread is a path of execution of a program. It is the smallest unit of execution and scheduling. A thread consists of a stack, the state of the CPU registers, and an entry in the execution list of the system scheduler. Each thread shares all of the process's resources.

Thread

- Exists within a process
- Has it's own independent control and scheduling
- Use process' resources
- Is terminated when a process terminates
- Share memory space, but each has it's own stack

- Has it's own priority level and time quantum

An example of multithreaded execution can be video watermark detection (Figure 5). In this example one thread does video frame decoding and the second watermark detection. For different frames these tasks are independent and can be executed in parallel on different processors.

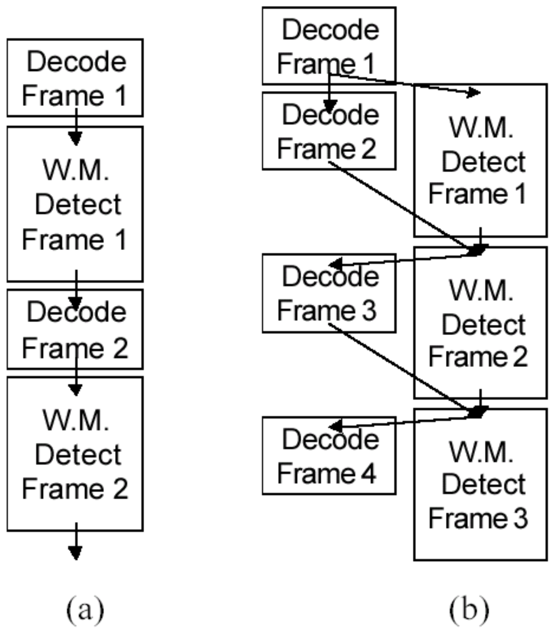

Figure 5 Video watermark detection (a) single threaded, (b) multithreaded [5]

2.3.1 Software threads

Multithreaded environment is available on many modern operating systems including Linux and Windows NT. There are also many thread libraries (POSIX pthread, Win32, OpenMP) and programming languages with direct support for threading in the language (OCCAM, Java); they support commands for creating, destroying and synchronizing threads. OpenMP [28] is of particular importance, because it supports simple, multi-platform parallel programming in C/C++ and Fortran. OpenMP offers easy-use, scalable and portable shared-memory environment with good performance. OpenMP is developed by OpenMP forum that includes mayor hardware and software vendors.

Another two concepts relates to software threads: scheduling and dispatching:

Scheduling

Scheduling is the process of determining which thread to execute next on a given processor. It is one of the most difficult parts in operating system design.

Dispatching

Dispatching is the process of switching the execution from one thread to another. Dispatching includes saving current thread's context and loading next scheduled thread's context.

Software threads are described in the operating system by a task context structure that contains the thread state, the state of processor registers, address thread's stack and pointer to description of memory address space of the parent process. Generally a thread state can be one of following:

- **Wait**. A thread is blocked from execution until some event occurs.
- **Ready**. A thread is ready for execution, but it must wait until the scheduler decides to execute it.
- **Running**. A thread is active on some CPU.

Every thread programming API must support thread creation, destruction and synchronization. The next paragraphs briefly describe OpenMP environment.

OpenMP is a collection of compiler directives, library functions and environment variables that can be used to specify shared-memory parallelism in C/C++ and FORTRAN programs. The goal is to provide a model for parallel programming that allows a program to be portable across shared-memory architectures from different vendors. The OpenMP C/C++ API will be supported by compilers from numerous vendors [28].

The directives, library functions, and environment variables defined in OpenMP allow users to create and manage parallel programs while permitting portability. The directives extend the C and C++ sequential programming model with single program multiple data (SPMD) constructs, work-sharing constructs, and synchronization constructs, and they provide support for the sharing and privatization of data. Compilers that support the OpenMP C and C++ API will include a command-line option to the compiler that activates and allows interpretation of all OpenMP compiler directives. Figure 6 presents OpenMP architecture. The OpenMP runtime library layer contains all the functionality described in OpenMP interface [28]. There are 3 ways how to access the library:

- **Compiler directives**. Most of the OpenMP functionality can be specified through compiler's directives. In case of C/C++ these are `#pragma omp` directives). OpenMP includes directives for loop parallelization, creating parallel sections, data distribution and thread synchronization. This makes it easy to parallelize sequential program just by adding several directives to the right places.
- **Library functions**. The runtime library also exports several function that allow the programmer to get information on runtime environment. There are functions for determining number of executing threads, scheduling method, locking, timing and many other.
- **Environment variables**. The end user can change the executing environment by setting OpenMP environment variables. This include number of threads, scheduling policy, enabling dynamic adjusting of executing threads and enabling nested parallel processing. All these parameters are also accessible from the runtime library functions.

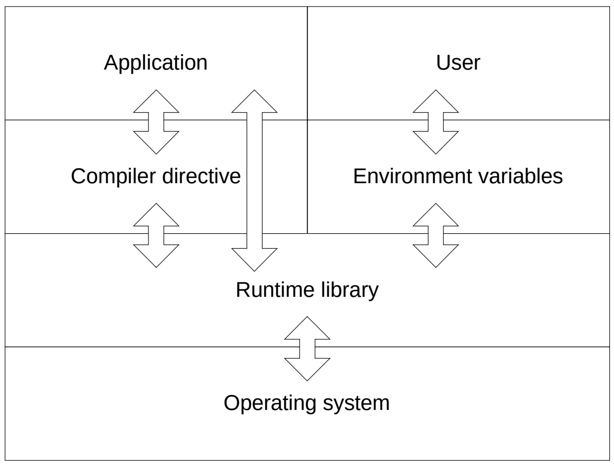

Figure 6 OpenMP Architecture [35]

OpenMP is not the only environment for parallel programming. Another environments include MPI and native threads. Native threads include for example POSIX threads, Win32 threads or Java threads. These thread libraries are not designed for scientific computation and their usage is more complicated than OpenMP. OpenMP is a high level library that includes directives for easy thread management and data distribution. Furthermore the source code can compile under a compiler that doesn't include OpenMP support. Different threading approaches are compared in table 1.

	MPI	Threads	OpenMP
Portable	✓	mostly	✓
Scalable	✓	✓	✓
Performance Oriented	✓		✓
Supports Data Parallel	✓		✓
Incremental Parallelism			✓
High Level			✓
Serially Code Intact			✓
Verifiable Correctness			✓

Table 1 Parallel models compared [38]

2.3.2 Hardware multithreading support

Generally, there is no need for any support in the processor in order to run multithreaded programs. Each thread is described by task structure in the operating system kernel and from the processor point of view there is a little difference between thread and process context switch.

Except of overlapping CPU work with I/O or executing parallel tasks, the processor designer can use threads to overlap latency in memory accesses. In multithreaded execution, instructions from one thread are issued until an instruction with long latency is found (e.g. cache miss). In this case the execution is switched to another thread and the original thread is suspended. The thread status must be saved in the processor by hardware. Hardware support for multithreading is very flexible method for hiding instruction latency, it's main advantages include [18, 24]:

- No need for special software support (except for threads)
- Covers both predictable and unpredictable latencies (branch miss prediction, cache miss, resource conflicts)
- Doesn’t change instruction execution order within one thread

Multithreading support requires keeping separate register file and instruction counter for each active thread. Active thread is the one that is currently assigned to hardware. A block diagram of a multithreaded processor is on figure 7.

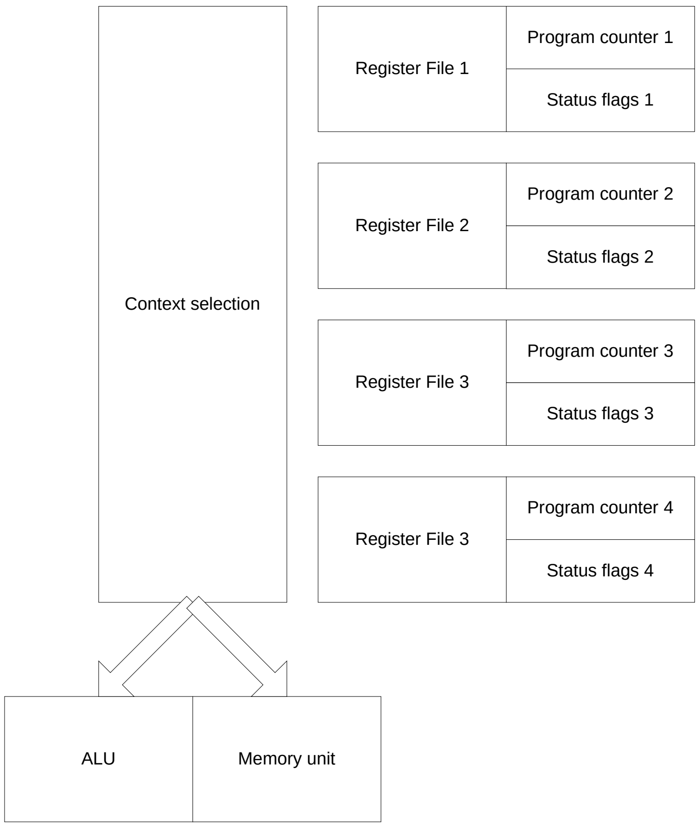

Figure 7 A processor with support for multithreading [18]

The objective of multithreaded execution is maximizing the function [18]:

Equation 1 Efficiency of multithreaded execution

$$E = \frac{work}{(work + switch + wait)}$$

Where:

work	is the time spent on doing useful processing for the user
switch	is the time spent during context switches
wait	is the time spent on waiting for some reason

While multithreaded execution minimizes waiting, context switch time remains critical. The problem in thread context switch is to decide what to do with instructions already in instruction pipeline. There are several options:

- Finish all instruction in the pipeline
- Kill all instruction remaining in the pipeline
- Overlap execution from both threads (share the pipeline)

The last option is clearly the best, but its implementation is complex.

There are two basic principles in multithreaded execution

- Vertical multithreading and
- Horizontal multithreading

Processor utilizing vertical multithreading issues instruction from only one thread at a time. Two variants of vertical multithreading are defined by the time the context switch occurs. If a context switch occurs only when issuing thread is suspended or blocked by long latency operation we call it blocked multithreading. The implementation cost of blocked multithreading is relatively low and single threaded performance is the same as on standard microprocessor. In the case of Interleaved multithreading then context is switched on every cycle. Well-known examples of such a processors are MAJC [21, 22, 23], TERA [24, 25] and Transputer [18].

Until now we assumed instruction issue only from one thread. A processor that issues instructions from multiple threads at one time utilizes *horizontal* (or *simultaneous*) *multithreading*. The main advantage of SMT approach is wider range of independent instructions that are ready to issue. Intel Hyper-threading technology is an implementation of horizontal multithreading.

Figure 8 compares three options: (a) single threaded execution, (b) vertical multithreading and (c) horizontal multithreading. In single threaded execution the processor executes only one thread at a time and context switch to another thread occurs only when the execution thread is blocked or its quota expires. The processor is not required to contain any support for multithreaded execution and the context switch is always initiated by the operating system. In the case of vertical multithreading the processor contains several thread context descriptors in hardware and periodically switches among ready threads (a thread is omitted from running for example when it is handling a long time memory operation or branch misprediction). When a thread is blocked by the operating system or its quota expires the operating system removes its context descriptor from the processor hardware. Thus context switches are of two kinds: processor initiated (every cycle) and operating system initiated (thread is blocked or consumed its quota). Horizontal multithreaded processors can issue instructions from multiple threads at the same time. This means that all threads presented in hardware descriptors are executed truly concurrently and the processor can achieve high resource utilization.

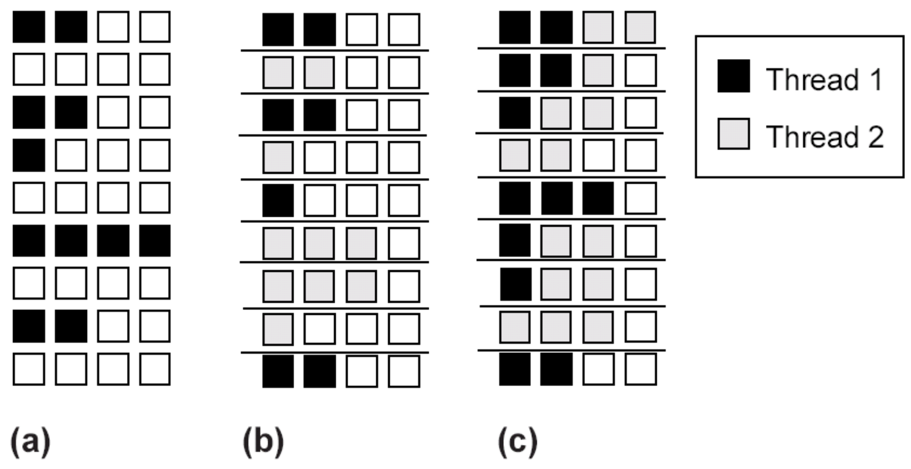

Figure 8 On chip multithreaded execution (a) Single threaded, (b) Vertical multithreading (c) Horizontal multithreading [6]

2.4 Problem statement

The focus of this thesis is the top of the pyramid of parallel processing (Figure 9). The two lower levels (SIMD processing and Instruction level parallelism) are usually handled by the compiler that generate optimized code well suited for the computer and memory architecture. On the contrary, although there is a lot of effort on automatic parallelization on large scale, today’s compilers can’t discover parallelism on thread level in a satisfactory manner.

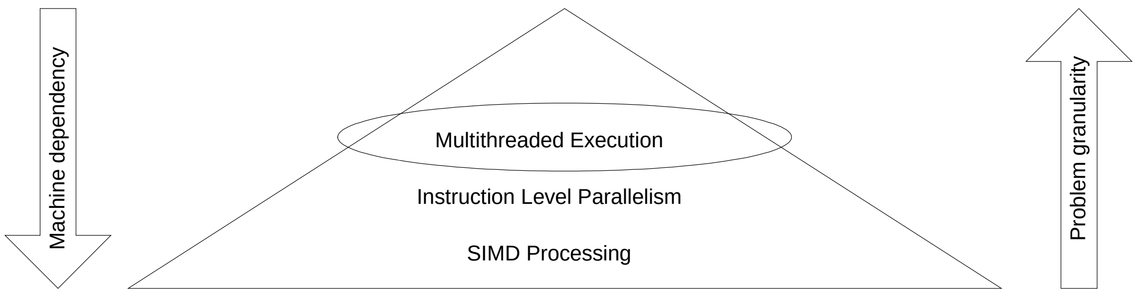

Figure 9 Problem focus

Given a problem, there are several options that must be considered before the implementation phase. These options are interdependent and must be considered as a whole.

- Parallelism
- Memory distribution
- Thread scheduling

Parallelism

There are three basic choices how to implement an algorithm (Figure 10). The most simple way is to use sequential algorithm. The main advantages of sequential algorithm is that it has minimal overhead and can be very well understood and optimized. The disadvantage is that sequential algorithm can’t utilize system resources to the same extent that parallel algorithms can. Parallel algorithm can be implemented using software threads (without hyper-threading) or hardware threads (with hyper-threading enabled). When using hyper-threading the threads are running truly concurrently, allowing maximal system utilization, but also creating higher contention for system resources. Another thing to consider is dependency on hardware; even if the program is binary compatible whether using software or hardware threads, performance can differ significantly.

The method of using parallelism is an inherent property of the algorithm, while the number of working threads can be varied depending on execution environment. For example OpenMP allows to change the number of working threads from environment variables.

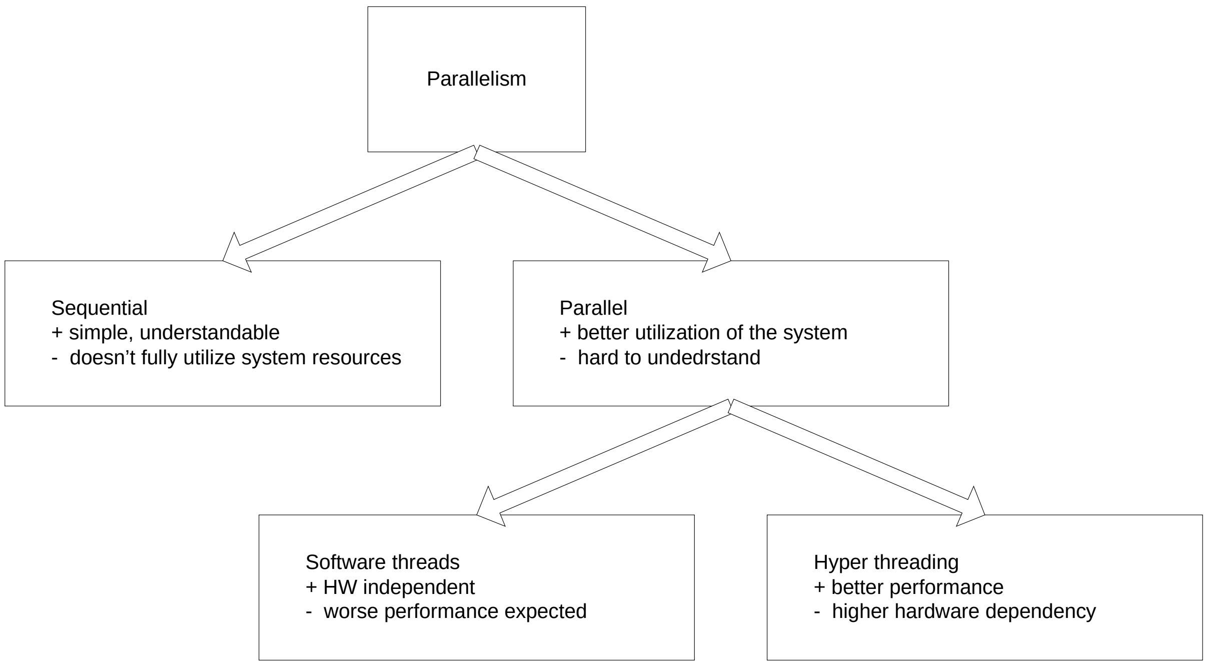

Figure 10 Parallelism choices

Memory distribution

There are three basic choices to memory partitioning among threads (Figure 11). In block partitioning each thread is assigned a continuous block of memory to process. This requires only a small overhead, but there are two major problems. If one thread finishes its work faster than others it must wait for until all threads finish - block partitioning doesn’t support load balancing. The other problem is that if the block size exceeds cache size, the cache performance can be poor (depending on the problem). In cyclic partitioning the problem is divided into single units and each unit is assigned to threads one after another. Cyclic partitioning combined with dynamic thread scheduling is excellent for load balancing. Block-cyclic partitioning combines advantages of both block and cyclic partitioning methods. The user can choose the right block size to allow efficient processing and enable load balancing.

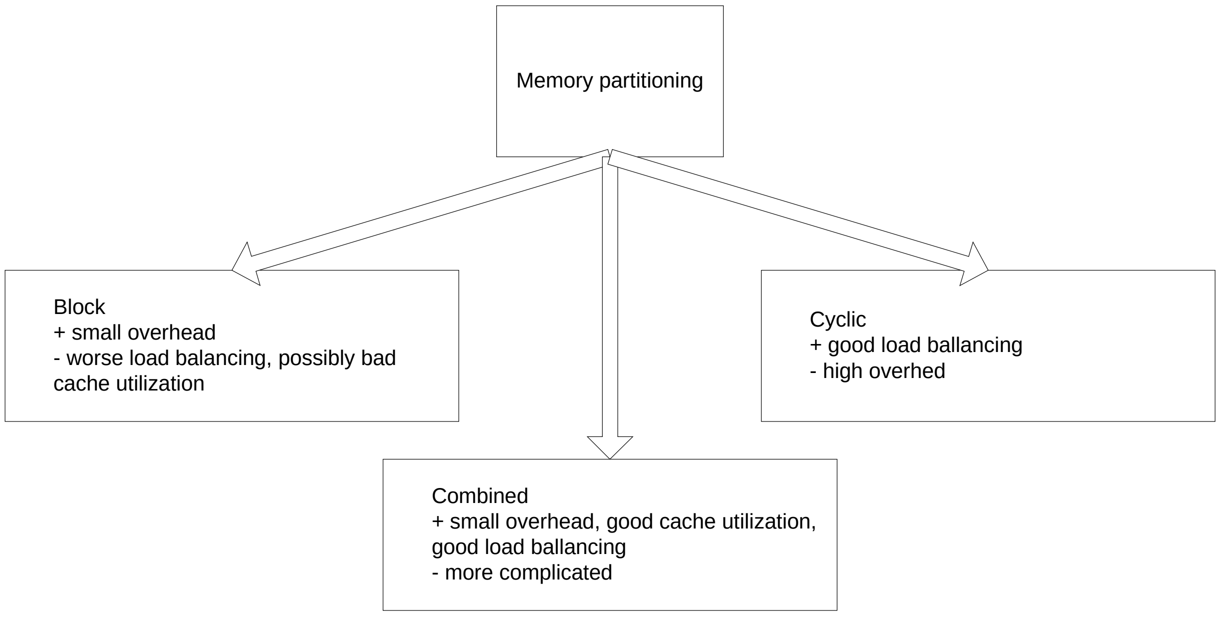
Memory partitioning
Block
+ small overhead
- worse load balancing, possibly bad cache utilization
Cyclic
+ good load ballancing
- high overhed
Combined
+ small overhead, good cache utilization, good load ballancing
- more complicated

Figure 11 Memory partitioning choices

Figures 12, 13 and 14 present the available methods for memory partitioning. The problem is to partition 12 element vector among 2 threads assuming static thread scheduling. The block partitioning assigns the first half of the vector to thread 0 and the other half to thread 1. The cyclic scheduling assigns every even element to thread 0 and every odd element to thread 1. Figure 14 shows block-cyclic partitioning with block size of 3. The blocks are cyclic distributed between the two threads.

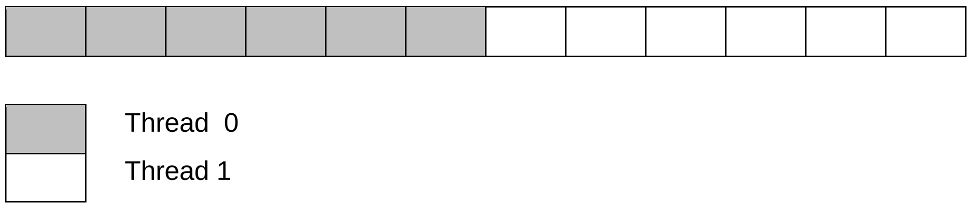

Figure 12 Block partitioning between 2 threads

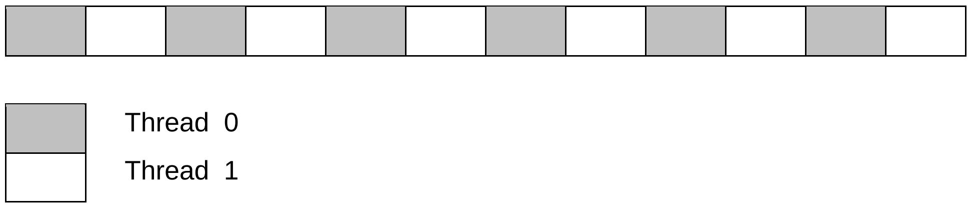

Figure 13 Cyclic partitioning between 2 threads

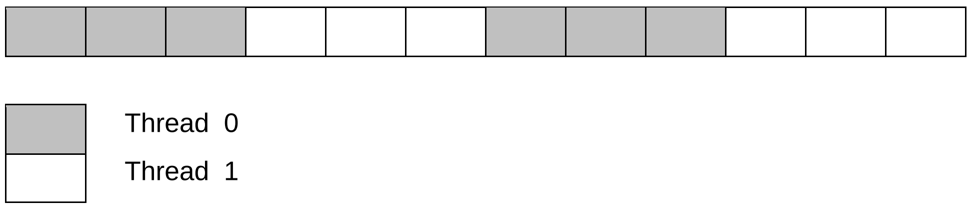

Figure 14 Block-cyclic partitioning between 2 threads

Thread scheduling

Thread scheduling is tightly related to parallelization and memory partitioning. Thread scheduling is in question only when using parallel algorithm with cyclic or block-cyclic memory partitioning. Figure 15 shows the available thread scheduling choices.

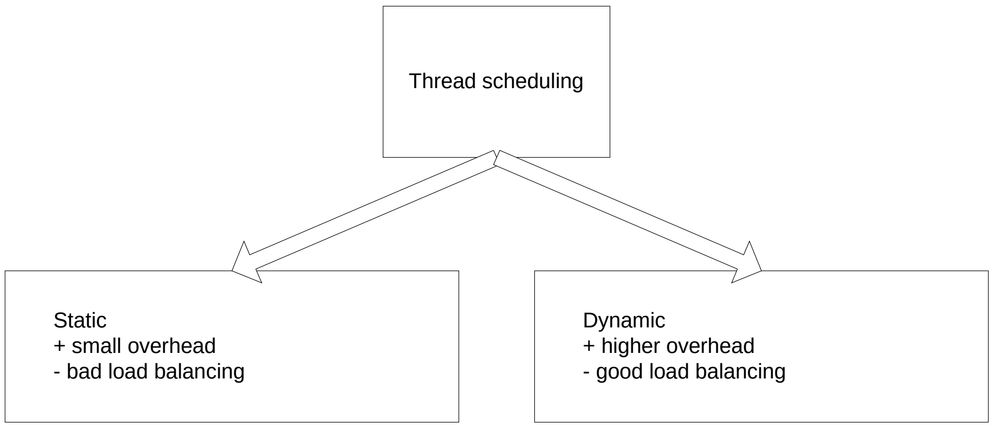

Figure 15 Thread scheduling choices

Static scheduling means that each thread is assigned its data to process right in the partitioning phase (Figure 16). Each thread must process all data that it has been assigned with; the algorithm performs no load balancing.

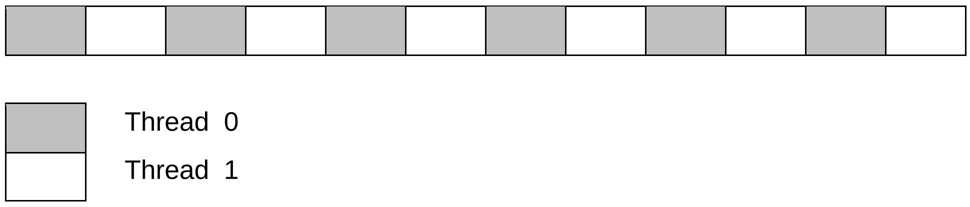

Figure 16 Static thread scheduling example

In dynamic scheduling (Figure 17) the distribution of data partitions is not defined in the partitioning phase, but during computation. Dynamic scheduling adds some overhead because a thread must ask for new element to process at each iteration. In the example on figure 16 the processing time of data elements is not uniform, the time to process the first element is much shorter than time required to process the second element. During the time when thread 1 is processing element 2 the thread 0 can process element 1, 3, 4 and also part of element 5. The work is load balanced at the cost of additional overhead connected with elements distribution.

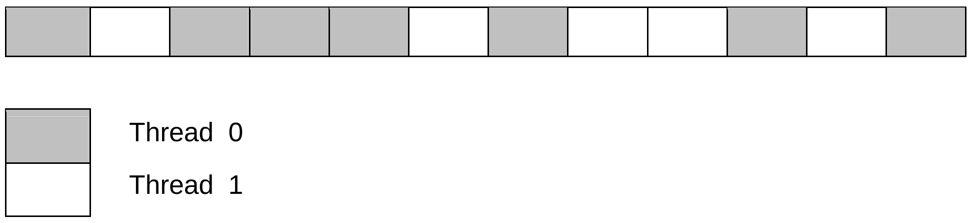

Figure 17 Dynamic thread scheduling example

3 Pentium 4 processor

Pentium 4 is the newest processor that comes out from the x86 architecture to achieve the top speed ratings. The Pentium 4 processor was introduced in the year 2000 at 1.5GHz clock rate. It uses new NetBurst technology that combines deep pipelining with aggressive execution optimizations and high speed, low latency caches. Together with new SSE2 instruction set, the processor is significantly faster than any other Intel's previous processors.

The architecture of Pentium 4 consists of 4 main building blocks (Figure 18).

- The Front End
- The Out of Order Engine
- The Integer and Floating Point Execution Units
- The Memory Subsystem

Each of these units is in brief described in later sections. For more detailed information please refer to [1, 4, 11].

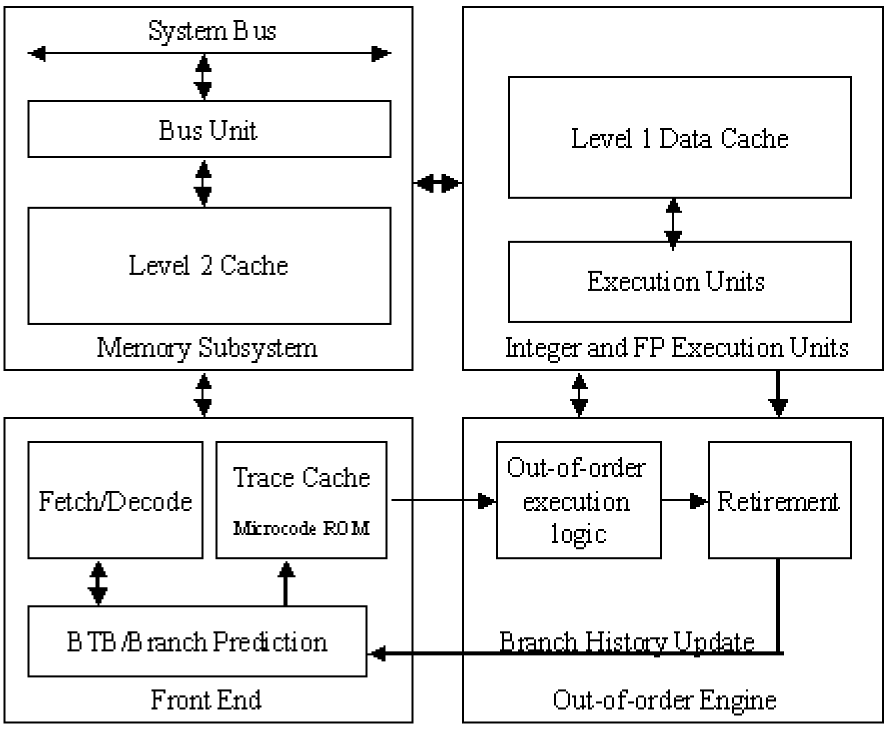

Figure 18 The Pentium 4 processor architecture [11]

3.1 The Front end

The front end is composed from Fetch/Decode unit, Execution trace cache and Branch prediction unit. The task for front-end part is to

- Prefetch IA-32 instructions
- Decode IA-32 instructions into micro-operations
- Store decoded micro-operations into trace cache
- Perform branch prediction

The IA32 architecture native instruction can vary in length (1 to 16 bytes), number of registers used and memory access modes. To simplify instruction execution, these native instructions are translated into small, simple and fixed length micro-operations. Each micro-operation has 2 source operands and one destination and is executed in one execution unit. Typically simple IA32 instructions are translated to one micro-operation, but complex floating-point instructions can generate hundreds of them. The front end must guarantee steady stream on micro-operation to the execution units. There are two main sources of stalls:

- Time to decode the instruction is too big
- Cycles are lost with branch misprediction

The Pentium 4‘s front-end unit solves these problems by using trace cache. The trace cache is placed between decode logic and execution logic. The trace cache stores decoded micro-operations, so that an instruction doesn’t need to be decoded every time it’s about to be executed. The micro-operations are stored in the trace cache in the order of current execution path; an execution path can go across many branches. The trace cache can forward up to 3 micro-operations to the execution core. In hyper-threaded execution mode the two logical processors cannot use the trace cache at the same time; the trace cache issues micro-operations of only one logical processor each cycle.

3.2 The Execution core

The need for high performance orders the execution core to execute many instructions in parallel. To allow high parallelism, the instructions are scheduled for execution in out of order fashion. The execution core contains reorder buffer that stores micro-operations on the current execution path and the micro-operations that are ready for execution sends to proper execution unit. To maximize performance the micro-operations execution order is given only by their dependences, this keeps the execution units and cache as busy as possible.

Figure 19 presents the path that every micro-operation follows. The execution engine starts with micro-operations queue that guarantees steady supply of micro-operations to the execution engine, even if the front end is stalled. Later the micro-operations are examined for data dependences and if possible dependences are removed by register renaming. Then micro-operations are scheduled to the appropriate execution units and executed. Finally results are written to destination memory or registers and micro-operation is retired.

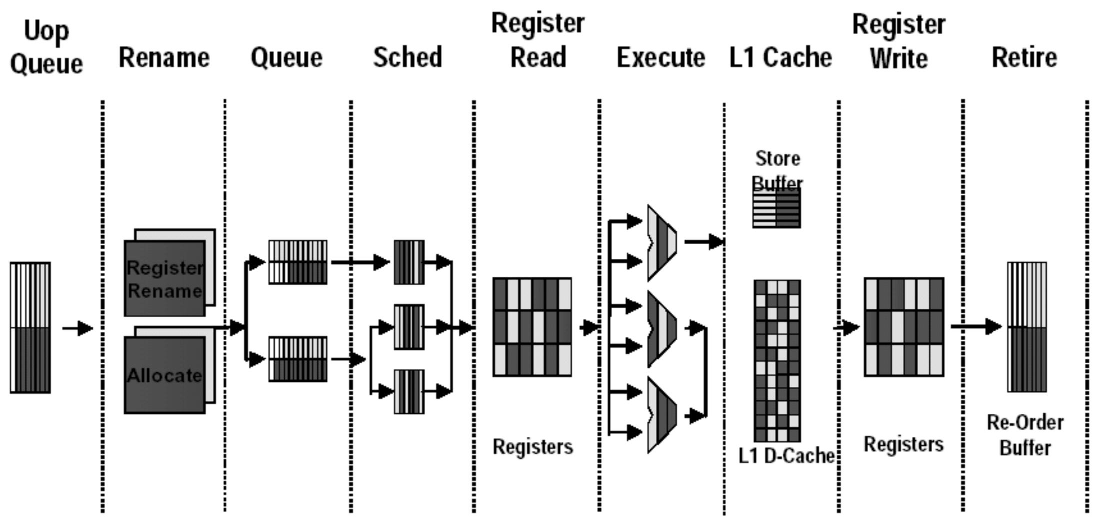

Figure 19 Out-of-order execution engine detailed pipeline [5]

Completed micro-operations are returned from the execution unit back to the reorder buffer. The retirement unit scans the reorder buffer and looks for such micro-operations. Retirement is made in-order and makes final changes into visible machine state (machine state registers, memory writes, exception reporting…). The retirement unit can retire up to 3 micro-operations at every cycle.

3.3 Execution units

The execution units actually execute the micro-operations. The scheduler sends a micro-operation to be executed to the proper execution unit as soon as its operands are available and there is no other micro-operation with higher priority. This section also stores register files for both integer and floating point data. Pentium 4 has 7 execution units connected to 4 ports (Figure 20). These are integer, floating point and MMX and memory execution unit. Two of the integer ALUs are double speed, meaning that they can execute two micro-operations in each cycle.

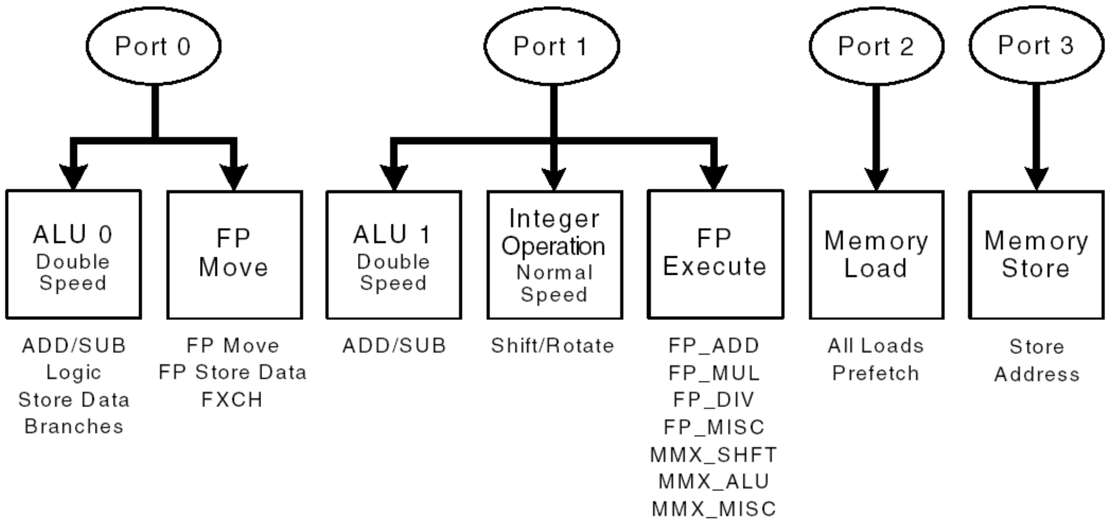

Figure 20 Pentium 4 Execution units and Ports in the out-of-order core [4]

3.4 Memory architecture

Memory architecture significantly influences the overall performance of the machine. Pentium 4 has sophisticated memory architecture and this chapter describes it in quite a detailed manner. The memory subsystem consists of:

- **Caches**. A small and fast memories used to quick access to frequently needed data
- **Translation lookaside buffers**. Buffers used to speed up translation between logical and physical address.
- **Prefetchers**. Prefetchers are used to fetch data or instructions that are expected to be used in the near future. There are two kinds of prefetchers: software and hardware.

Every access to memory utilizes all resources of memory subsystem; over utilizing a single part of the system can downgrade the performance significantly. Except of capacity problems, the memory subsystem also imposes several other limits, mainly load/store ordering (3.4.2) and aliasing (2.3.4).

3.4.1 Caches

Cache is a fast memory used to store frequently accessed data. Pentium 4 uses 2-level cache hierarchy.

L1 cache

The first level is L1 cache, located near the execution resources in order to achieve high speed. To keep the access time short, the size of the cache is limited to only 8kB. It is 4-

way set associative with 64B cache line size and allows one load and one store operation to be performed at the same time.

The level 1 instruction cache found on previous processors was replaced by execution trace cache. Unlike the instruction cache, this memory doesn't store instructions, but rather stores traces of micro-operations. A micro-operations trace means an execution path that can go over several branches and thus non-contiguous area in memory.

L2 cache

Data that doesn't fit into level 1 cache are searched in level 2 cache. The 2nd level cache is 256kB or 512kB big, 8-way set-associative. This cache stores both code and data.

3.4.2 Loads and stores

Pentium 4 can use many techniques to speed up memory operations. Among these, there are operations that reorder or speculate memory accesses. Pentium 4 offers:

- Speculative memory loads
- Reordering loads with respect to loads and stores
- Multiple outstanding cache misses
- Buffering of writes
- Data forwarding

The memory unit can handle at most 1 load and 1 store operation each cycle. Loads can be reordered so that other independent loads and stores do not block them and enable higher parallelism. Speculative loads cannot cause exception. Store buffers handle write operations, and can delay bus access to more suitable time. The data from any store operation can be forwarded to dependent load operation without delay and using the system bus.

3.4.3 Data aliasing

The architecture poses several aliasing cases. An aliasing case is a situation when accesses to memory address A and different memory address B causes conflict in memory resources.

Pentium 4's aliasing cases are listed here [4]:

- 2kB for L1 data cache access. L1 data cache is 4-way set-associative with overall size of 8kB. Two memory accesses with 2kB distance use the same entry in the L1 cache.
- 16kB for data is the granularity for store forwarding logic. Load and store with 16kB distance appears to be at the same address.
- 32kB for data uses the same L2 cache set.
- 64kB. Only one of these can be in L1 cache. Any other reference that has this alias with an address that is already in cache cannot be processed until the blocked cache line is freed. Pentium 4's optimization manual strongly encourages avoiding this kind of aliases.

3.5 Hyper-threading and parallel execution

Hyper-threading is an implementation of symmetric multithreading. It is designed to take advantage of thread level parallelism. A processor with hyper-threading technology contains two logical processors inside one physical package (Figures 21 and 22). Thus a Pentium 4 machine with two physical processors and enabled hyper-threading appears to have four processors.

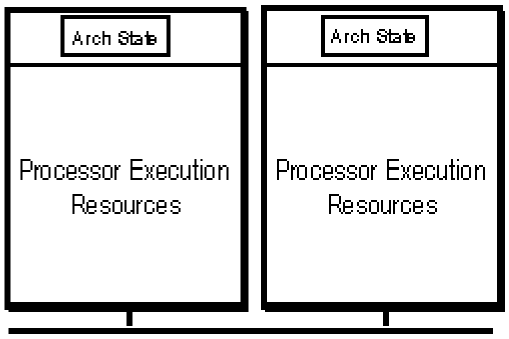

Figure 21 Processors without hyper-threading technology [5]

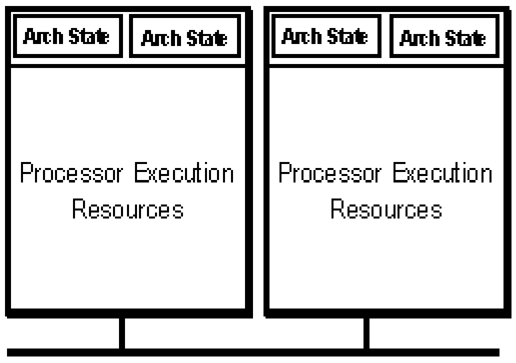

Figure 22 Processors with hyper-threading technology [5]

In order to reduce cost and hardware complexity, the two logical processors are not fully independent. Processors enabled with Hyper-Threading Technology share more resources (for example execution resources) on the same physical processor than multiple-core processors. For example, multiple-core processors have separate execution units and first-level memory caches.

Processor’s resources are divided into 3 categories [6, 14]:

Replicated

In order to keep both logical processors independent some resources must be replicated, for example the architectural register file that keeps track of current computations. Also the register renaming logic that keeps mapping from architectural registers to microarchitecture internal registers must be replicated.

Partitioned

Partitioned resources are shared in a way that each logical processor can use fixed part of the total resource available on the physical processor. For example the write buffers are partitioned so that each logical processor has access to only half of the write buffers on the physical processor.

Shared

Shared resources are the reason why hyper-threading meets its goals. More shared resources means their better utilization and growing performance without increasing the cost of the technology. Many of the internal resources (internal register file, cache memories, execution units) don’t care what logical processor’s data are processing. The most critical shared resource is the cache. By the fact that it is shared, the program can take advantage of shared data between the two processors, but it also means twice more cache trashing in the case when no data is shared which can significantly decrease performance.

When the processor is operating in non-hyper-threaded mode, all resources are available to the one logical processor that is in execution no matter how they are divided.

Replicated	Architectural registers
	Register renaming logic
	Translation lookaside buffers
Partitioned	Reorder buffer
	Load/Store buffers
	Various internal queues
Shared	Caches
	Microarchitectural registers
	Execution units

Table 2 Pentium 4 processor resources division

Resources in modern processors are generally underutilized, caused mainly by instruction dependences that limit the number of available operations. The main effect of hyper-threading is that it allows sharing of resources between logical processor whose instruction streams are independent and thus increase parallelism. But the fact that resources are shared can also cause problems and performance decrease in the case when logical processors are trashing the resource.

3.6 Performance counters

Intel Pentium 4 processor introduced several new hardware performance monitoring features [3, 4]:

- **Performance counters**. Pentium 4 supports 18 performance counters to collect program execution metrics.
- **Micro-operation tagging mechanisms**. These tagging mechanisms allow tagging micro-operation and/or instructions that have encountered certain performance events. Event detectors at the end of the pipeline can detect these tagged micro-operations and instructions as they retire, allowing obtaining non-speculative counts for these performance events.

- **Data address profile support**. allows to obtain profiles of data memory accesses that cause performance problems. (e.g., cache misses). With data address profiles we can attempt to improve the layout of data in memory to improve performance.

Intel Pentium 4 processor allows collecting several metrics that gives a view on program execution. These metrics are described in detail in [4].

Table 3 Performance metrics

General	Operation not specific to any sub-system of the microarchitecture (Clockticks, Retired instructions, Retired micro-operations, CPI…)
Branching	Branching activities (Branches retired, Mispredicted branches, Number of executed returns, calls, conditionals…)
Trace Cache and Front End	Front end activities and trace cache operation modes (ITLB misses, Number of cycles of trace cache in deliver/build mode, TC misses, speculative uops…)
Memory	Memory operation related to the cache hierarchy (cache misses, DTLB misses, 64kB aliasing conflicts, Loads/Stores retired, Split loads/stores)
Bus	Activities related to Front-Side Bus (Bus accesses from the processor, PSB data activity)
Characterization	Operation specific to the processor core (x87 assists, MMX and SSE metrics)

The most important set of metrics are memory and bus metrics, because the contentions for cache and FSB affects the performance in the biggest extent.

4 Software environment

This chapter describes software considerations for writing and running code on processors with hyper-threading enabled. This includes programming guides, operating system and execution environment choice and methodology to collect performance data.

4.1 Programming guides

There are already several guides that describe programming techniques for the Pentium 4 processor. This chapter is a short extract from the IA-32 Intel Architecture Optimization Manual [4]. It summarizes techniques suitable for execution in hyper-threaded environment. IA-32 Intel Architecture Optimization Manual states the following goals for well performing code:

- Good branch prediction
- Avoiding memory access stalls
- Good floating-point performance
- Instruction scheduling
- Thread synchronization

4.1.1 Branch prediction

Branch prediction can have significant impact on performance. In order to reduce branch misprediction penalty it is necessary to understand branch prediction algorithm implemented in Pentium 4. The basic branch optimizations are the following:

- Keep code and data separated
- Eliminate branches (using new instructions)
- Arrange code to be consistent with static prediction rules
- Use inline functions

The compiler best performs some of these optimizations; generally the code performs well if the programmer doesn’t explicitly break these rules. But the programmer can help the compiler to decide what function to inline.

4.1.2 Avoiding memory access stalls

An easy way to avoid many memory access stalls is to follow the following rules:

- Properly align data
- Place code and data on separate pages
- Exploit data locality
- Use data prefetching
- Use data forwarding
- Avoid access to the same data using different sizes
- Avoid data aliases

Again, the compiler takes care for most of these cases. The two last rules are among the most important. When trying to load (store) data from address that overlaps with the address used for previous store (load) with different operand size a significant penalty can occur. The IA-32 Intel Architecture Optimization Manual describes these cases in detail. The data aliasing cases were described in chapter 3.4.3 and it is strongly recommended to avoid them. The 64kB alias case results in flushing the aliasing data from L1 cache and this can cause significant penalty.

4.1.3 Good floating-point performance

There are hints to achieve good performance in floating point code.

- Use new instructions (SSE/SSE2)
- Use proper data type
- Vectorization

It is the programmer's responsibility to use the proper data types to avoid overflows, underflows and using denormalized values. Operations on out-of-range value cause significant overhead. On the other hand, calculations using single precision data are much faster than full precision. Vectorization is a program transformation that allows the hardware to perform the same operation on multiple data elements at the same time. Vectorization can be done by compiler or the programmer using compiler directives of vector math libraries.

4.1.4 Instruction scheduling

Instruction scheduling should be done so to maximize processor utilization and minimize resource conflicts for all generation of the processor family. This is not easy to get, but the rules for generation good code schedule are targeted primarily to compiler writers.

4.1.5 Thread synchronization

The programmer should prefer to use multithreaded libraries developed by hardware vendors, that are optimized for best performance. When developing a multithreaded code it is necessary to keep the synchronization overhead minimal. The common source of this overhead is busy waiting and false sharing. To reduce the cost of busy waiting Pentium 4 offers new instruction PAUSE that ensures better performance in hyper-threaded mode of execution. False sharing occurs when two threads access different variables that are placed in the same cache line (or two aliasing cache lines). To reduce the cost of false sharing the programmer must guarantee that the data doesn’t reside in the same cache line or both threads should create a private copy. This is preferable, because private data copy also increases data locality.

4.2 Execution environment

This chapter presents the execution environment used in experiments (Figure 23).

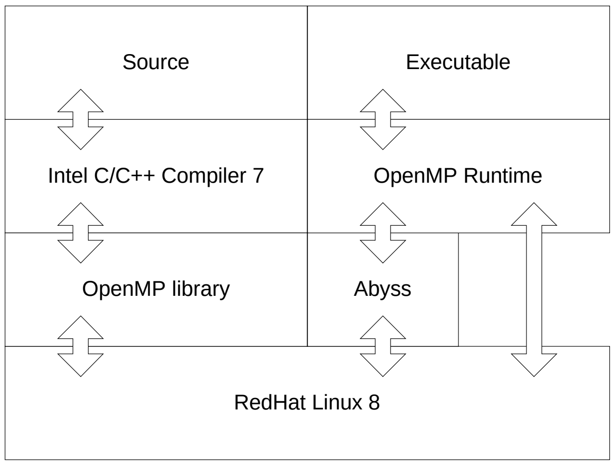

Figure 23 Execution environment

4.2.1 GNU/Linux

Linux is freely distributable UNIX based operating system kernel, originally written in 1991 for the IBM PC architecture. At this time many other ports of Linux exists on many platforms (m68000, MIPS, Sun Sparc, DEC Alpha/AXP and many other architectures). Linux is not a full operation system, it doesn't include applications for administration, graphics desktop, compilers, user application, but many of them are available from the GNU project.

The first Linux kernel with hyper-threading support was 2.4.18, but this kernel has serious limitations in the implementation of thread scheduler. Since 2.5.23 version, the Linux thread schedule has been improved and nowadays Linux supports hyper-threading without problems. The experiments were run on single processor Pentium 4 machine with enabled hyper-threading. The machine used Linux 2.4.18 SMP kernel with abyss 2.0 patch.

Hyper-threading on pre-2.4.18 kernels

Hyper-threading technology is not officially supported on these kernels. Because BIOS presents hyper-thread enabled processor as two logical processors, the pre-2.4.18 kernel can recognize two processors, but the thread scheduler in such a kernel is unaware of logical/physical processor distinction and can perform less optimal scheduling. A 2.4.18 or newer kernel is strongly recommended for HT configurations.

Hyper-Threading on 2.4.18 kernel

The performance under the 2.4.x kernel still shows some problems. This is because the scheduler still cannot make intelligent choices regarding logical/physical processors in many situations. Under some conditions, the 2.4 kernel will still schedule two active threads on the same physical CPU, causing performance degradation. This condition is often random, causing data from multithreaded benchmarks to vary considerably. Full hyper-thread scheduler support was not incorporated into the kernel until 2.5.32.

Hyper-Threading in the 2.5.x kernel

The 2.5.x kernel added a number of features to its thread scheduler that should extend the performance improvements of HT even further. The 2.5.32 kernel incorporated significant improvements that ensure good load balancing among logical and physical processors. The kernel ensures that all physical processors are occupied by active threads are before allowing two other threads sharing the same processor.

4.2.2 Intel C/C++ Compiler

Intel C/C++ compiler 7.1 is an advanced C/C++ compiler compatible with that offers aggressive optimizations and produce high quality code. The compiler offers:

- Support for the newest features of Pentium III, Pentium 4 and Itanium processors. It can utilize vectorization using floating point, SSE and SSE2 instruction sets.
- Manual selection of optimization in source code, these include memory aligning of data, loop unrolling and vectorization. All these optimization are also performed automatically, possibly with multiple code versions for different processor families.

- Supports parallel processing by automatic parallelization of OpenMP 2.0 standard.
- Compatibility with Microsoft Visual C++ and gcc compilers. It can be fully integrated to Visual C++ build environment.

The testing suites were compiled with Intel C/C++ compiler 7.1 with these settings:

Optimization level

Standard	-O2
Maximum	-O3
Aggressive	-O3 -xW -ipo

Where

-O2	Standard optimization set, should produce the optimal code in most cases.
-O3	The same as –O2, with loop transformations and data prefetching. The manual states that this may not improve performance.
-xW	Generate code only for Pentium 4 new instructions
-ipo	Multi file optimization

All the test suites are compiled with no presumption on aliasing (the compiler must discover all aliases) and full math precision.

4.2.3 OpenMP

OpenMP is a library that used to manage multithreaded execution. In the implementation of the test suites, the source code defines only the way how to distribute and parallelize the problem. All other parameters (namely number of threads and scheduling policy) can be read from environment variables. This project uses OpenMP 2.0 implementation as present in Intel C/C++ Compiler 7.1. For reference of OpenMP see chapter 2.3.1 or [28].

4.2.4 Brink & abyss

Brink and abyss software is a set of Pentium 4 performance monitoring tools available for Linux for 2.4.14, 2.4.17 and 2.4.18 kernels. Abyss and brink are tools that work together to provide easy access to all Pentium 4 native events. In the current version (2.0) Abyss offers access to global (process insensitive) counters only, thus some noise is added on the measured data. In Pentium 4's hyper-threading mode the performance counters are not fully independent and a care must be taken how the measuring tools handle this. One solution is to limit the measuring to only one thread, but this affects the final results more than global counters.

Abyss itself is composed of kernel patch (for 2.4.14, 2.4.17 and 2.4.18 kernels), kernel driver (abyss_dev.o) that enables access to performance counters from user programs and front-end program that easies access to the driver. The user should use only the front-end (abyss) in combination with brink.

Brink is a perl script and provides user-friendly interface to abyss. Brink uses two files for its task: a textual experiment configuration file and a global configuration file that describes hardware events available in the hardware. Brink & abyss is not the only package suited for the task. It was selected because it is fully preconfigured for Pentium 4 and offers enough good precision. The precision of time measurements is about 0.1 seconds. The testing suites runs about 5 to 10 seconds then this is a precision of 1 or 2% and it’s enough for evaluating the algorithms. The decision on parallelization method should be made on using more solid base than 1% performance difference.

To evaluate program performance the following metrics were collected:

Table 4 Metrics gathered in experiments

Metric	Description / Purpose	Event Name	Event mask value
Instructions Retired	Non-bogus IA-32 instructions executed to completion. May count more than once for some instructions with complex micro operations flow and were interrupted before retirement. The count may vary depending on the microarchitectural states when counting begins. This metric can reveal the overhead of parallel processing compared to sequential. High difference in this metric between sequential and parallel algorithm shows high amount of processing in the parallel algorithm that is not related to the problem being solved.	instr_retired	nbogusntag | nbogustag
µops Retired	Non-bogus micro-operations executed to completion. This metric has similar purpose as Instructions retired metric, but at the level of micro-operations rather than instructions.	uops_retired	nbogus
Branches Retired	All branch instructions executed to completion. This metric is used to calculate branch misprediction ratio.	branch_retired	MMTM | MMNM | MMTP | MMNP
Mispredicted Branches Retired	Mispredicted branch instructions executed to completion. This stat is often used in a per-instruction ratio.	mispred_branch_retired	NBOGUS

Metric	Description / Purpose	Event Name	Event mask value
	This metric counts branches that actually cause branch misprediction. This is an important metric that is related to code structure. High misprediction ratio should lead to code restructuring.		
1st-Level Cache Load Misses Retired	The number of retired µops that experienced 1st-Level cache load misses. This stat is often used in a per-instruction ratio. Similarly as branch metrics show code structure, memory metrics show memory access structure. High count of this metric shows poor cache utilization. In this case it is necessary to change memory access pattern.	Replay_event; With 1stL_cache_load_miss_retired tag	NBOGUS
64K Aliasing Conflicts	The number of 64K aliasing conflicts. A memory reference causing 64K aliasing conflict can be counted more than once in this stat. The performance penalty resulted from 64K-aliasing conflict can vary from being unnoticeable to considerable. Some implementations of the Pentium 4 processor family can incur significant penalties for loads that alias to preceding stores. 64kB aliases can be very expensive operations. It is necessary to keep them minimal. They can easily and unexpectedly appear on certain problem or block sizes and disappear when block size slightly changes.	Memory_cancel	64K_CONF
2nd-Level Cache Misses	The number of 2nd-level cache misses. Beware of granularity	BSQ_cache_reference	RD_2ndL_MISS |

Metric	Description / Purpose	Event Name	Event mask value
	differences. L2 cache miss serves the same purpose as L1 cache miss metric.		WR_2ndL_MISS
Loads Retired	The number of retired load operations that were tagged at the front end. The number and distribution of load and store accesses can be useful in comparing sequential and different sorts of parallel algorithms.	Front_end_event; set the following front end tag: Memory_loads.	NBOGUS
Stores Retired	The number of retired stored operations that were tagged at the front end. This stat is often used in a per-instruction ratio. The number and distribution of load and store accesses can be useful in comparing sequential and different sorts of parallel algorithms.	Front_end_event; set the following front end tag: Memory_stores.	NBOGUS
FSB Data Ready	The number of front-side bus clocks that the bus is transmitting data driven by this processor (includes full reads/writes and partial reads/writes and implicit writebacks). FSB is a shared resource among all devices connected there. High FSB utilization from one device can block all others.	FSB_data_activity	1. DRDY_OWN, DRDY_DRV 2. Enable edge filtering in the CCCR.

5 Matrix operations

Matrix operations are among the most important operations in linear algebra. This section begins with vector-vector multiply, and then follows vector matrix and matrix-vector and finally matrix-matrix multiplication. All problems them are approached from several parallelization methods.

5.1 Vector-vector multiplication

Vector-vector multiplication (dot product) is the most basic operation. It is used as a building block in any other matrix multiplications. Vector-vector multiplication is defined by

Equation 2 Vector-vector multiplication

$$z = x \cdot y = \sum_i x(i) \times y(i)$$

The common sequential algorithm is presented here:

```
procedure DOT_PRODUCT(x, y, z)
begin
     z := 0
     for i := 0 to n-1
          z := z + x[i] * y[i];
end
```

5.1.1 Parallelization

The iterations of the loop in dot product computation have interdependence caused by access to shared variable *z*. The variable *z* accumulates the results of partial multiplications and thus is called accumulation variable. This is kind of dependence can be optimized by compiler without a need for guarding access to the variable on every iteration.

There are many ways how to parallelize this algorithm. To achieve optimal performance we have to adhere to processor’s memory architecture. This can be done by choice of

1. Partitioning
2. Aligning to memory architecture

Partitioning

The partitioning method defines how is data distributed among processors. There are three basic methods of partitioning

- Cyclic partitioning
- Block partitioning
- Combined block-cyclic partitioning

Cyclic partitioning distributes data so that processor 0 gets every even element in data array and processor 1 gets every odd element. Cyclic partitioning (Figure 24) processes elements in small entities (individual data elements) and can provide good load balancing at the expense of higher overhead and higher resource conflicts (for example both processors can access the same cache block and increase the number of cache misses).

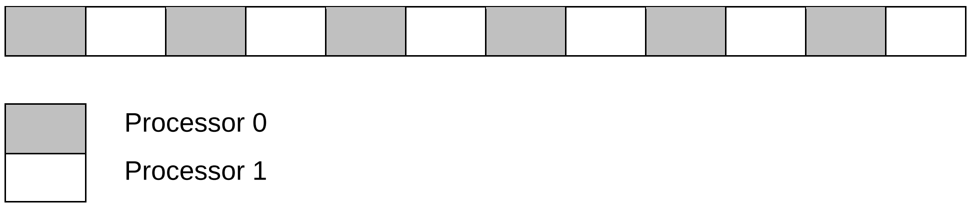

Figure 24 Cyclic partitioning of a vector

In block partitioning (Figure 25) the data vector is split into two halves, so that the first half is assigned to the first processor and the second half to the second processor. Its main advantage is low overhead, but doesn't offer any kind of load balancing. The other problem could be if the block size doesn't fit into the cache, then again cache performance can be poor.

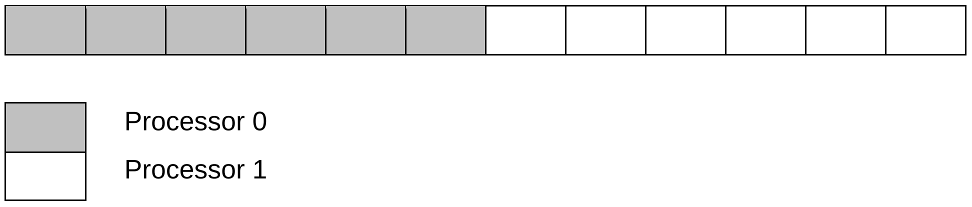

Figure 25 Block partitioning of a vector

Block-cyclic partitioning (Figure 26) combines both block partitioning and cyclic partitioning. It processes the data vector in cyclic manner, but instead of individual elements a larger blocks are processed. With proper block size the load balancing facility is kept and cache utilization is closer to the optimum.

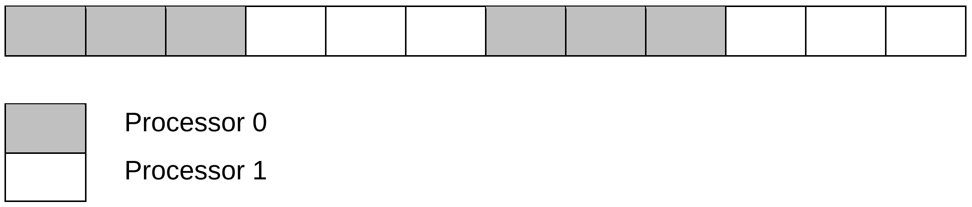

Figure 26 Block-cyclic partitioning of a vector

To achieve optimal cache performance, especially if the block is processed several times, the block size should fit in to the cache. The access time for data in level 1 cache is several times lower than access time to level 2 cache. The main memory access time is even several tens times higher and requires access to the front side bus. Clearly keeping high cache utilization can improve running time a lot.

Aligning to memory architecture

In every processor, the memory access has always several specifics. For example many processor architectures require the data to be size aligned (e.g. ints aligned to 4 byte boundary, doubles to 8 byte boundary). Pentium 4 doesn’t enforce such a restriction, but still access to aligned data is faster than access to unaligned data.

Next, due to the processor and memory architecture, the row-wise matrix processing is several times faster than column-wise. When data needs to be accessed column wise, it’s advantageous to consider transpose the data first, then process the operation and transpose back. Another example of memory architecture trick is aligning to the cache architecture. As stated before, Pentium 4’s level 1 cache has 8kB capacity and is organized as 4-way set-associative. This says that every access with 1kB difference refers to the same place in the cache. Avoiding 1kB difference accesses (together with other aliasing cases described in 3.4.3) can decrease cache miss ratio. Also, as the cache is shared between the two logical processors in hyper-threaded machine, consider assigning to the both processor blocks with starting address where

Equation 3 Non aliasing memory access

$$(start\_P0 - start\_P1) \bmod block\_size \geq cache\_line\_size$$

For Pentium 4:

Equation 4 Non aliasing memory access on Pentium 4

$$(start\_P0 - start\_P1) \bmod 1024 \geq 64$$

This guarantees that the starting address doesn't alias to the same cache line.

5.1.2 Results

These problem configurations have been tested. Each test was performed with and without hyper-threading.

Data types:	float, double
Data size:	$n = 2^x$, x = 8 to 26
Compiler optimizations:	standard, maximum, aggressive
Thread scheduling:	static, dynamic
Number of threads	1, 2, 3, 4
Hyper-threading:	enabled, disabled
Partitioning:	cyclic, block, block-cyclic with block size from 32 to 4096

For float data type (unlike for double data type) the compiler didn't generate code using the SSE2 instruction set even with aggressive optimizations. This affects performance negatively. An algorithm with block partitioning and block size B = 2^x is subject to memory aliasing, but the implementation changes slightly the starting address of block belonging to processor 1, so that this effect doesn't appear.

Data type float

Even with aggressive optimizations the compiler didn't generate code using the SSE2 instruction set. No matter how long the input vector is the sequential algorithm with aggressive performs best. The processor can execute vector operations so quickly that the overhead for parallel execution cannot be hidden. The fastest parallel solution is about 50 to 150 times slower than the fastest sequential one. Table 5 and figure 27 compares hyper-threaded, software threaded and sequential approach. The graph clearly shows that no parallelization is possible. The startup time of thread is too high.

	Hyper-threads		Software-Threads		Sequential	
	Execution Time	Number of Threads	Execution Time	Number of Threads	Execution Time	Number of Threads
16	4,52991	2	4,43895	2	0,098359	1
17	4,26039	2	4,37101	2	0,088117	1
18	5,1663	2	5,41141	2	0,088122	1
19	5,0551	2	5,44168	2	0,075609	1
20	4,9477	2	5,44522	2	0,075452	1
21	4,93112	2	5,45671	2	0,026365	1
22	4,90514	2	5,45326	4	0,089693	1
23	4,89879	2	5,46062	4	0,089905	1
24	4,93235	2	5,45991	2	0,075326	1
25	5,00223	2	5,44977	2	0,090904	1

Table 5 Vector-vector multiplication: Comparison of threading approaches (float data type)

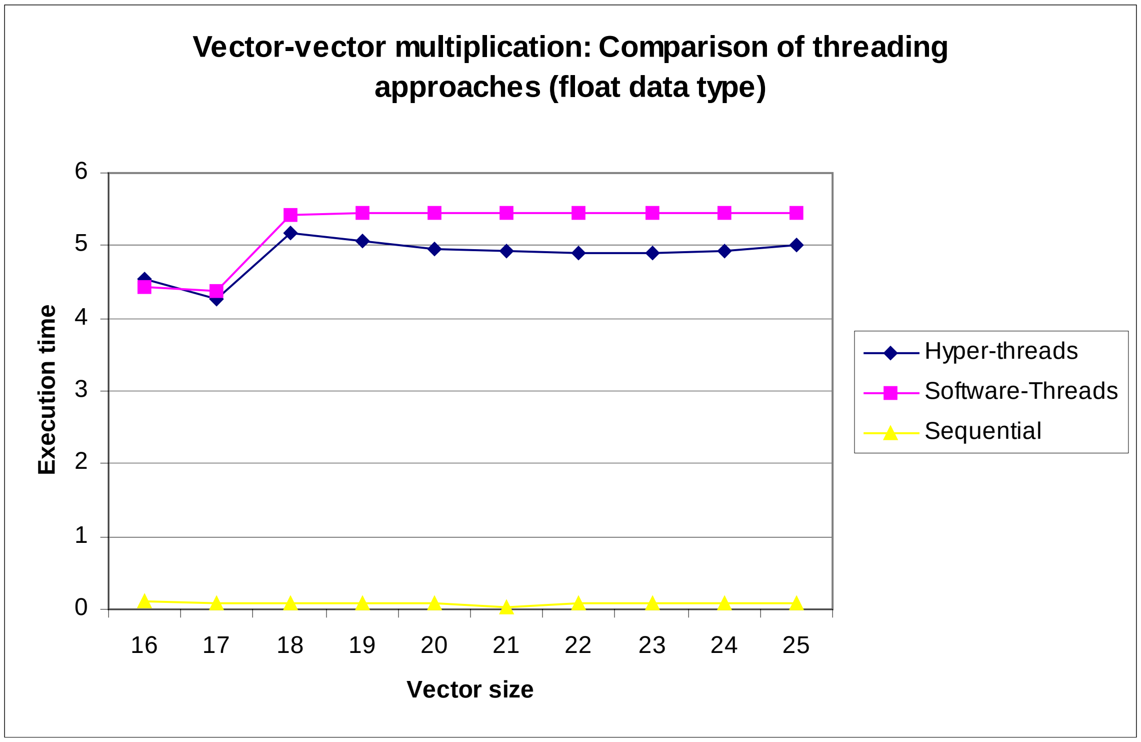

Figure 27 Vector-vector multiplication: Comparison of threading approaches (float data type)

Data type double

The code was generated using new SSE2 instruction set. The compiler correctly vectorized the loop and didn't use loop unrolling (it doesn't bring any performance advantage). Not surprisingly the code is faster than for float data type – this is the advantage of SSE2 instruction set. As in previous case for vectors with float data type, the overhead for parallel communication is significant. The sequential code using aggressive set of optimizations is faster than any other. Table 6 and figure 28 compares hyper-threaded, software threaded and sequential approach. The graph again clearly shows that no parallelization is possible. Surprisingly the solution with software threads is faster than hyper-threading. The hyper-threaded execution has about 10 times more L1 load cache misses, mainly due to the SSE2 instruction set that has lower overhead compared to non-SSE2 code and thus relatively higher memory demands. This, combined with truly concurrent execution in hyper-threaded code (while with software threads both threads are running in round robin scheme and don't really share the processor resources) results in slower execution.

	Hyper-threads		Software-Threads		Sequential	
	Execution Time	Number of Threads	Execution Time	Number of Threads	Execution Time	Number of Threads
16	3,76861	4	3,76861	4	0,100039	1
17	3,53222	4	3,53222	4	0,031013	1
18	3,56239	4	3,56239	4	0,094826	1
19	3,47113	4	3,47113	4	0,041801	1
20	3,55963	4	3,55963	4	0,041742	1
21	3,57945	2	3,57945	2	0,041814	1
22	3,6145	2	3,6145	2	0,096387	1
23	3,63199	2	3,63199	2	0,09808	1
24	3,67309	4	3,67309	4	0,041822	1
25	3,74494	4	3,74494	4	0,102824	1

Table 6 Vector-vector multiplication: Comparison of threading approaches (double data type)

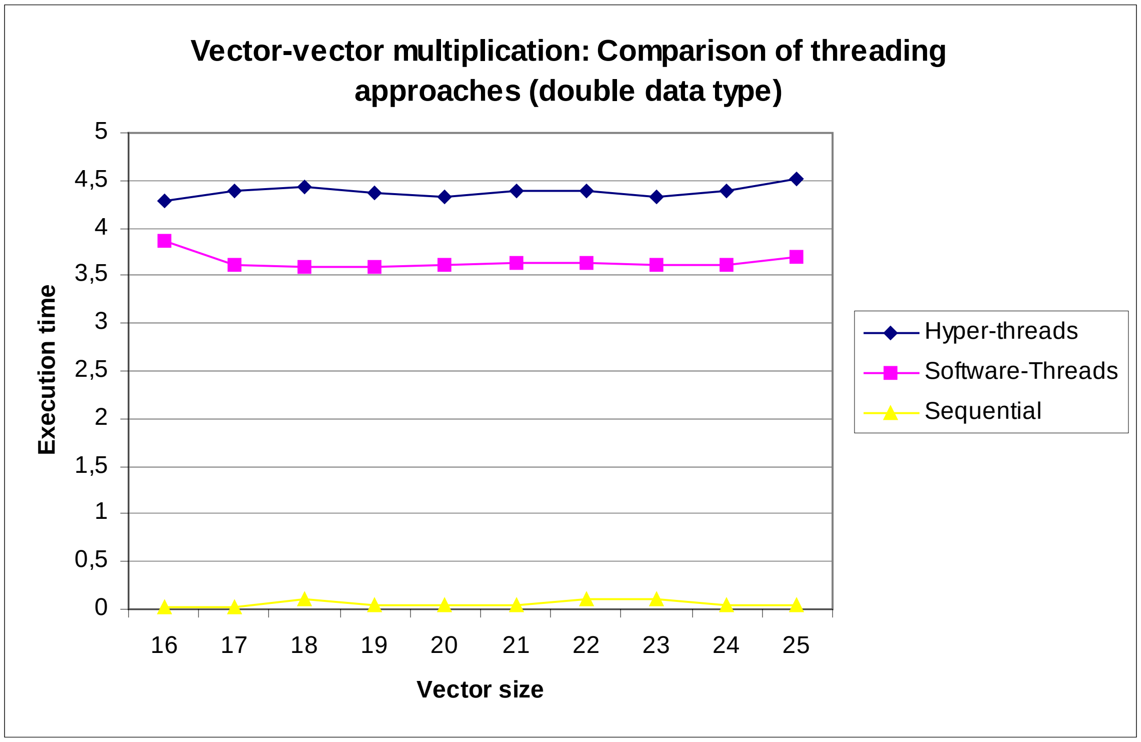

Figure 28 Vector-vector multiplication: Comparison of threading approaches (double data type)

5.1.3 Discussion

With double data type, the compiler generated optimal code. The code for dot product in float data type can be further speed up by manual coding using Intel's SSE and SSE2 intrinsic function, but the results stay the same. The experiment can be summarized as follows:

- The fastest way to scalar multiply two vectors is to use sequential algorithm. Today's processors and compilers are highly optimized to process regular data patterns and can generate efficient vectorized code.
- The only suitable optimization is aggressive, this allows code vectorization and allows using of new Pentium 4 instructions. The effect of optimizations varies depending on partitioning method
- The effect of data shifting in parallel execution was not measured. The processor automatically delays execution in one thread to avoid unnecessary data cache misses and aliasing.
- Algorithm implemented with SSE2 instruction set has significant advantage over non-SSE2 implementation, but its memory bandwidth demands are higher and truly-concurrent execution of two threads results in poor memory subsystem performance.

5.2 Matrix-vector multiplication

Matrix-vector multiplication is an operation that takes $n \times n$ matrix M and $n \times 1$ vector x to get as a result $1 \times n$ vector y, $M \times x = y$. A serial matrix-vector algorithm is presented here:

```
procedure MAT_VEC(A, x, y)
begin
    for i := 0 to n-1
    begin
        y[i] := 0;
        for j := 0 to n-1
            y[i] := y[i] + A[i, j] x x[j];
    endfor
end
```

Assume that even if the shape of vectors *x* and *y* is different, both vectors are stored sequentially in memory and the matrix *M* is stored in memory as two-dimensional array. Each element in matrix *M* is accessed only once, elements in the vector *x* are accessed multiple times, so it's advantageous to keep the vector in level 1 cache. There are several options to parallelize the algorithm.

5.2.1 Parallelization

At first define the partitioning method for matrix M (it implicitly defines partitioning for vector x). The matrix can be partitioned by rows, columns or both.

Row wise partitioning

Row wise partitioning is very simple. Each processor is assigned with certain amount of rows of the matrix *M* and has access to the whole vector *x* (Figure 29). This vector is for reading only, so its sharing shouldn't cause any performance penalty. Each processor computes one element of vector *y* and performs the following operation for each row *i*:

Equation 5 Row-wise matrix multiplication

$$y[i] = \sum_{j=0}^{n-1} M[i,j] \times x[j]$$

There could be two problems with row partitioning, first if the full vector *x* doesn't fit into level 1 cache, then increased and unnecessary amount of cache misses is generated. The second problem is that both processors starts at the computation at address that causes aliases. To solve this problem it is desirable to shift the beginning of the computation of processor 1 by small value so that the condition described above doesn't hold anymore.

$$y[i] = \sum_{j=0}^{n-1} M[i,j] \times x[j]$$

Equation 6 Row-wise matrix-vector multiplication (processor 0)

$$y[i] = \sum_{j=offset}^{n-1} M[i,j] \times x[j] + \sum_{j=o}^{offset} M[i,j] \times x[j]$$

Equation 7 Row-wise matrix-vector multiplication (processor 1)

This pattern should avoid the situation where both processors read/write the same cache line.

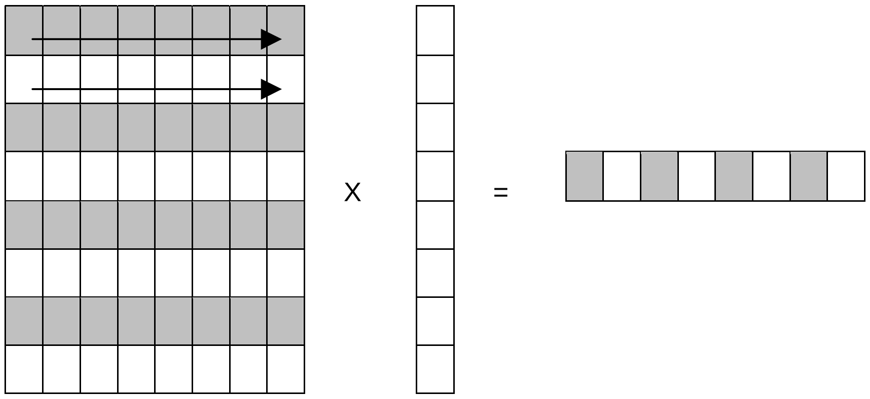

Figure 29   Row-wise matrix partitioning

Rows can be distributed among processors in block, cyclic and block-cyclic (with different block sizes) manner. Figure 30 show the execution time for different row distribution methods and different problem sizes. It also shows that the performance doesn't depend very much on the choice of the distribution method. Each test suite performs the same amount of basic operations (in this case each suite is run as many times needed to perform 1048576 multiplications).

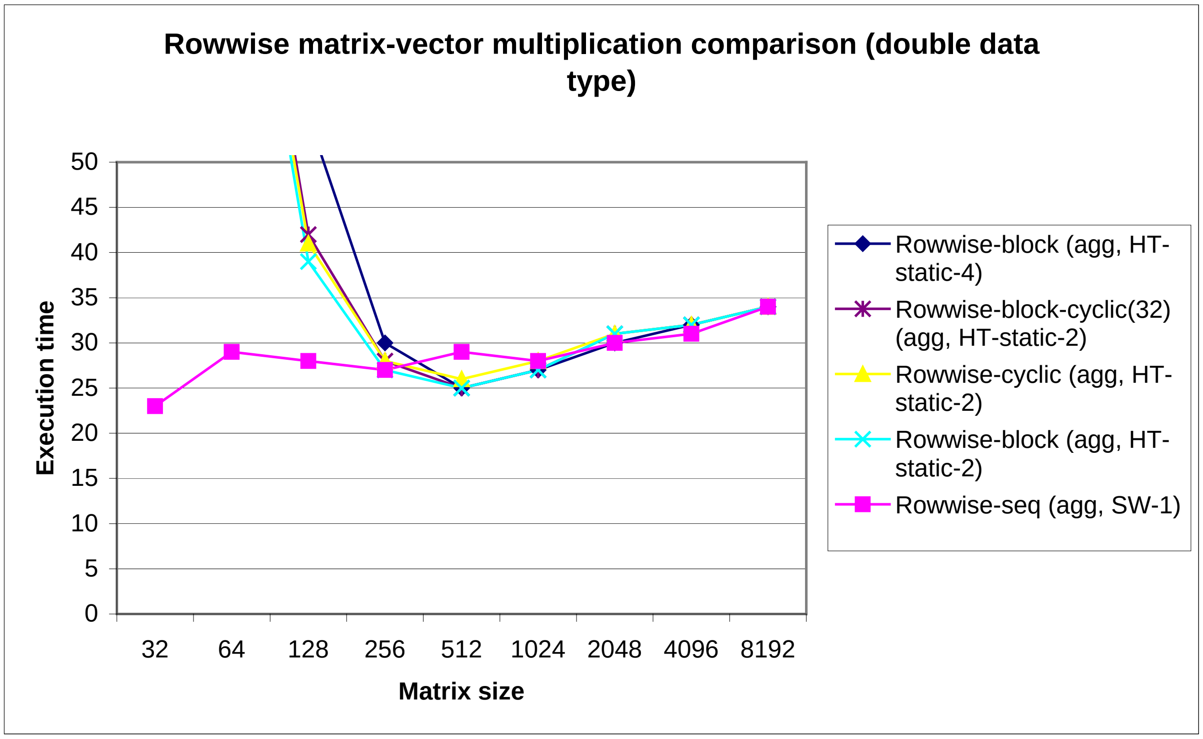

Figure 30 Row-wise matrix-vector multiplication comparison (double data type)

Column wise partitioning

The data matrix is divided by columns and the columns are assigned to single processors (Figure 31). Thus the matrix is accessed by columns, which is not optimal and the performance is expected to be worse than in row wise partitioning. Also note that all elements in a column alias to the same cache line.

Each processor computes for each column *j*

```
**for** $i$ := 0 to $n$-1
    $y[i] = y[i] + M[i, j] \times x[j]$
```

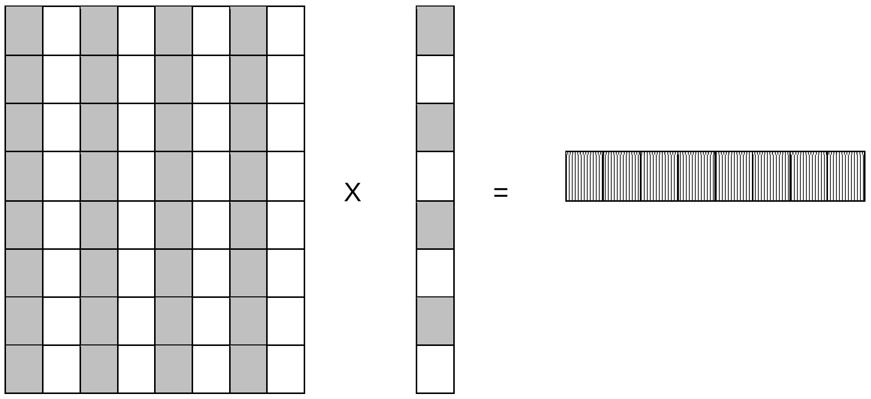

Figure 31 Column-wise cyclic partitioning

Columns are distributed among processors in block-cyclic manner. The width of columns block determines the performance. When the block size is too small the runtime is very high, but with block size of 512 of 1024 elements the runtime gets better especially for larger data sizes. Each test suite performs the same amount of basic operations (in this case each suite is run as many times needed to perform 1048576 multiplications). Figure 32 shows these results.

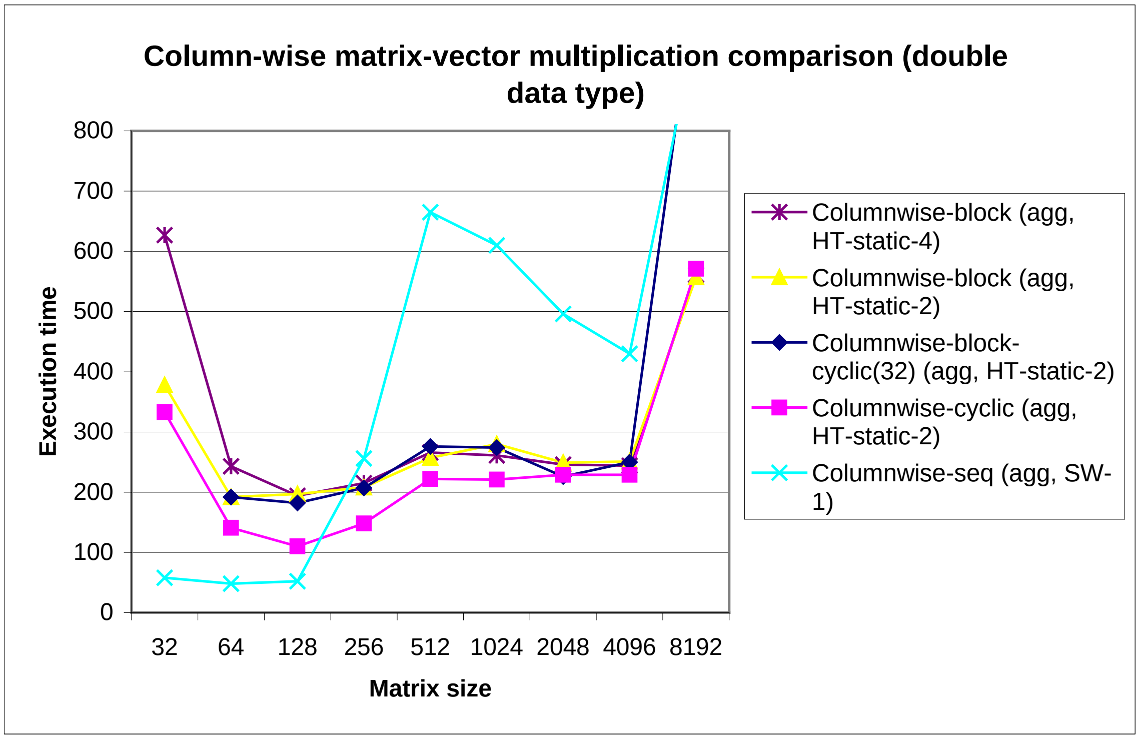

Figure 32 Column-wise matrix-vector multiplication comparison (double data type)

Compared to row-wise partitioning the execution time is significantly higher. This doesn't come as a surprise because many processor architectures (including Pentium 4) are optimized for row-wise memory access. What is noticeable is the execution time growth of sequential algorithm on 256 and more elements matrices. The next figures (figures 33 to 36) present selected memory metrics comparing sequential algorithm with parallel ones, showing corresponding dramatic growth of the FSB data activity metric on the affected sizes. This behavior is present in all variants of column-partitioned sequential algorithm and its origin is not clear.

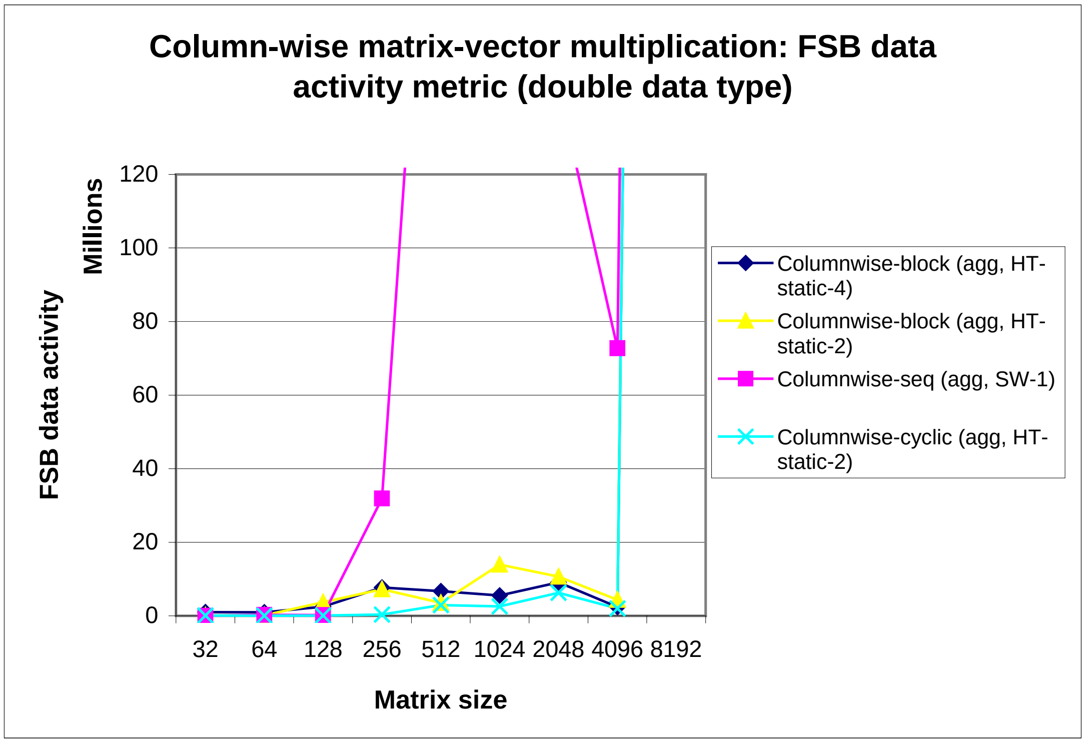

Figure 33 Column-wise matrix-vector multiplication: FSB data activity metric (double data type)

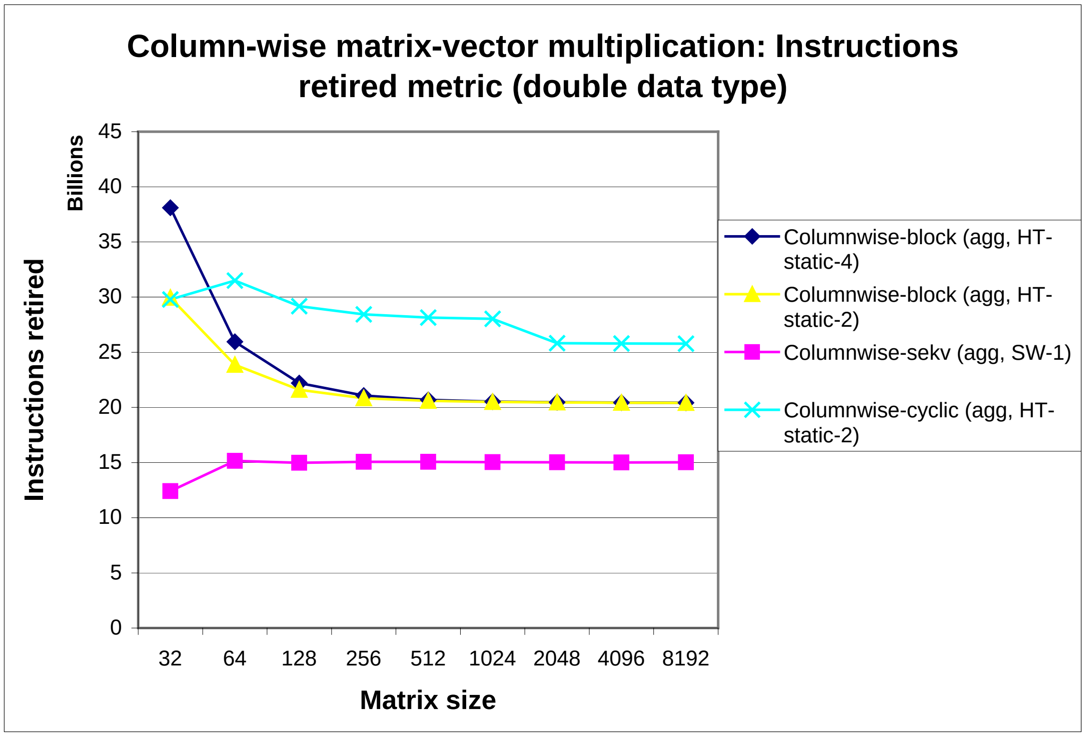

Figure 34 Column-wise matrix-vector multiplication: Instructions retired metric (double data type)

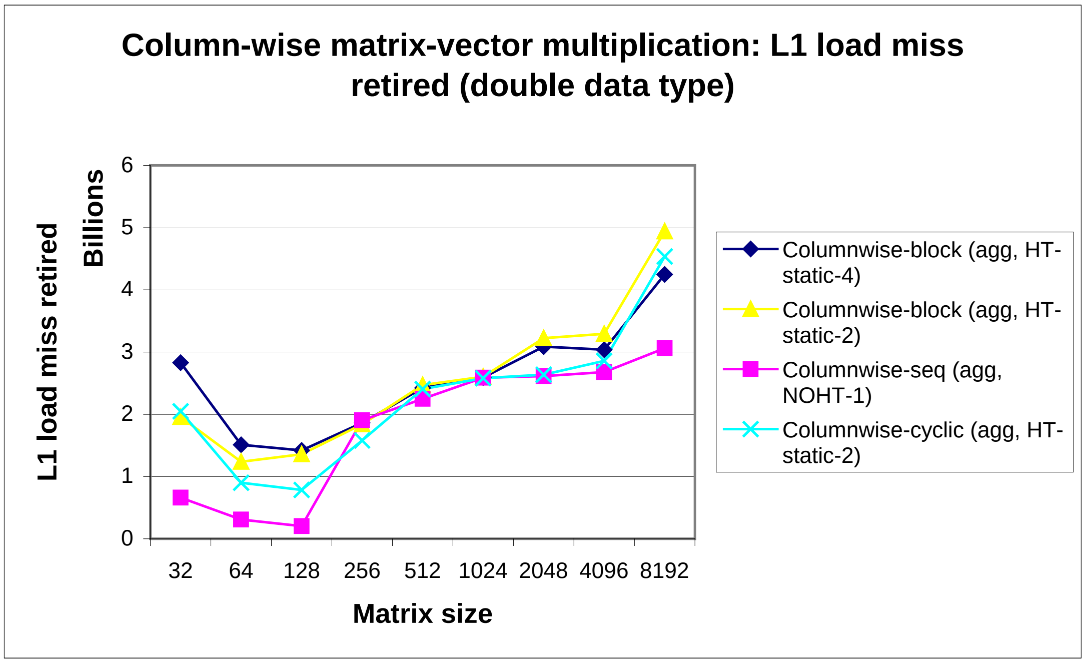

Figure 35 Column-wise matrix-vector multiplication: L1 load miss retired metric (double data type)

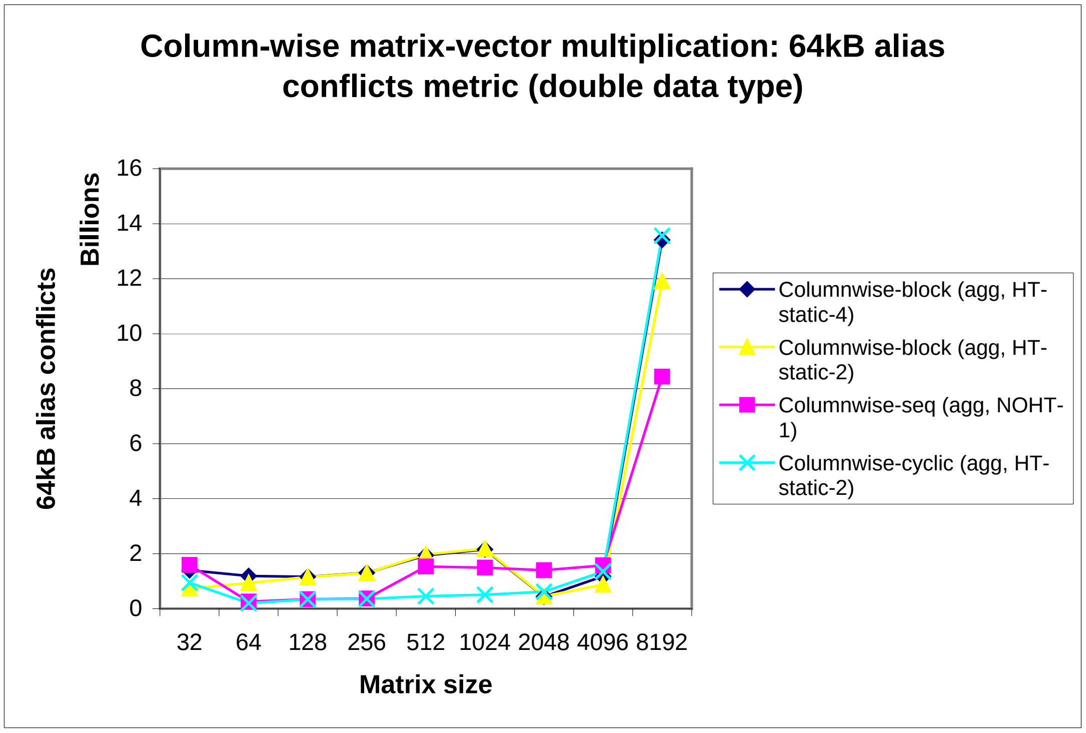

Figure 36 Column-wise matrix-vector multiplication: 64kB alias conflicts metric (double data type)

Row-column partitioning

Row column partitioning (Figure 37) can solve the problems of both row partitioning and column partitioning. The matrix is partitioned in both rows and columns and the data elements are accessed in row-major order. The block size is chosen so that it fits into level 1cache and reuse the data in vector *x* stored in cache.

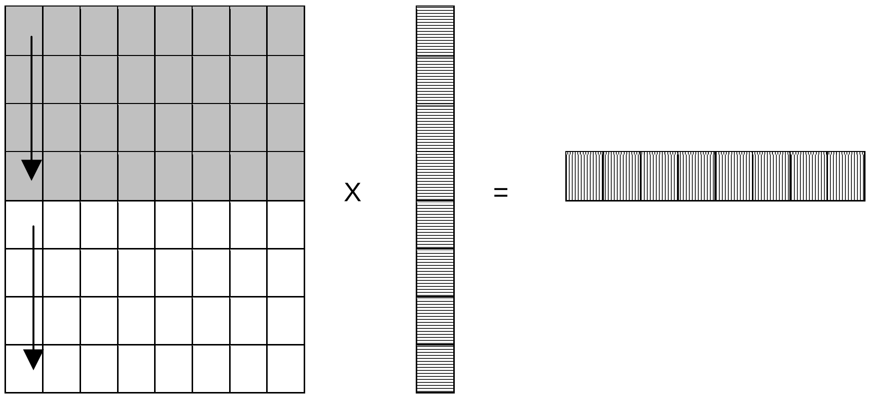

Figure 37 Row-column-wise partitioning

This approach is a merge of row wise and column wise partitioning. The matrix is divided by columns and the block-cyclic distributed by rows. This can increase cache efficiency. Figure 38 presents the results of this partitioning. Each test suite performs the same amount of basic operations (in this case each suite is run as many times needed to perform 1048576 multiplications).

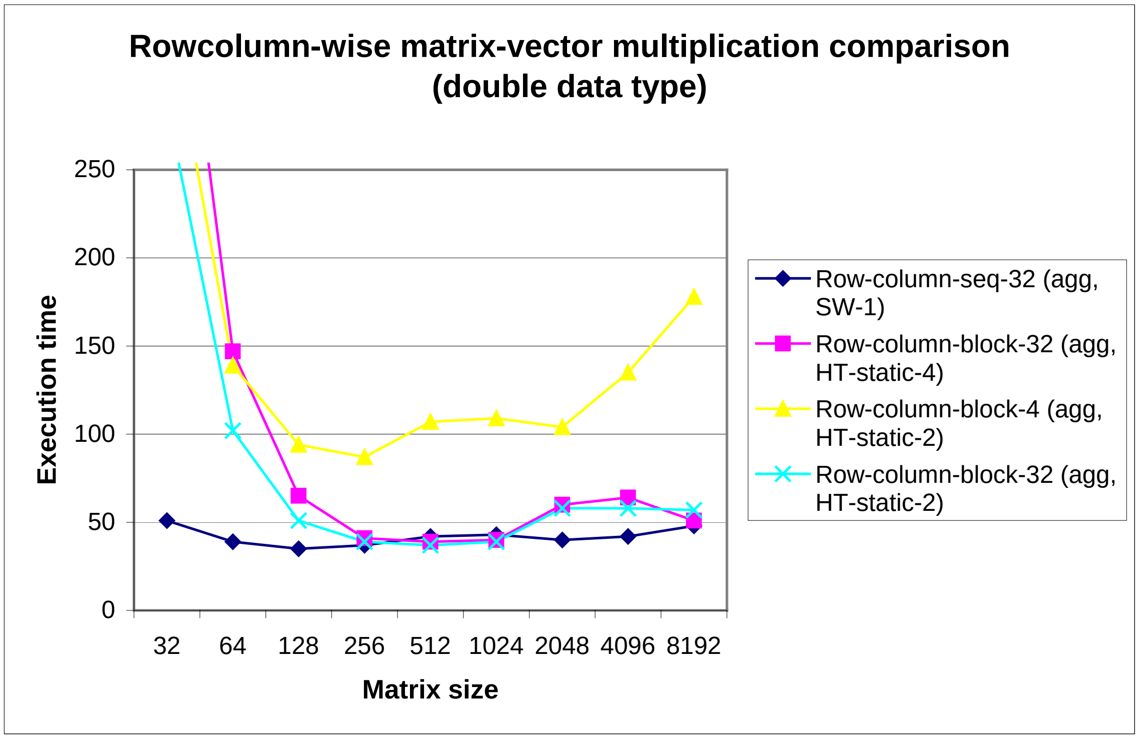

Figure 38 Row-column-wise matrix-vector multiplication comparison (double data type)

5.2.2 Results

These problem configurations have been tested.

Data types:	float, double
Matrix size:	$n = 2^x$, x = 5 to 13
Compiler optimizations:	standard, maximum, aggressive
Thread scheduling:	static, dynamic
Number of threads	1, 2, 3, 4
Hyper-threading:	enabled, disabled
Partitioning:	described above

Data type float

The raw data obtained from time measurement are listed in Table 7. If more than one method scores the same running time, only one of them is listed.

Matrix size	Parallelism	Parameters	optimization	Threading	Number of threads
32	rowwise	sequential	aggressive	software	1
64	rowwise	sequential	aggressive	software	2
128	rowwise	sequential	aggressive	ht-dynamic	1
256	rowwise	block-cyclic (4)	aggressive	ht-static	2
512	rowwise	block	aggressive	ht-dynamic	2
1024	rowwise	block	aggressive	ht-static	4
2048	rowwise	block	aggressive	ht-static	2
4096	rowwise	block	aggressive	ht-dynamic	2
8192	rowwise	sequential	aggressive	software	1

Table 7 Summary of matrix-vector multiply algorithms (float data type)

Figure 39 presents the data in graphical form. Parallel algorithms have high startup time; with block size less than 256 elements the sequential algorithm is in every case faster. Many algorithms have best performance at 512 and 1024 elements, with performance decrease later. The block-cyclic and block algorithms can keep more from its performance.

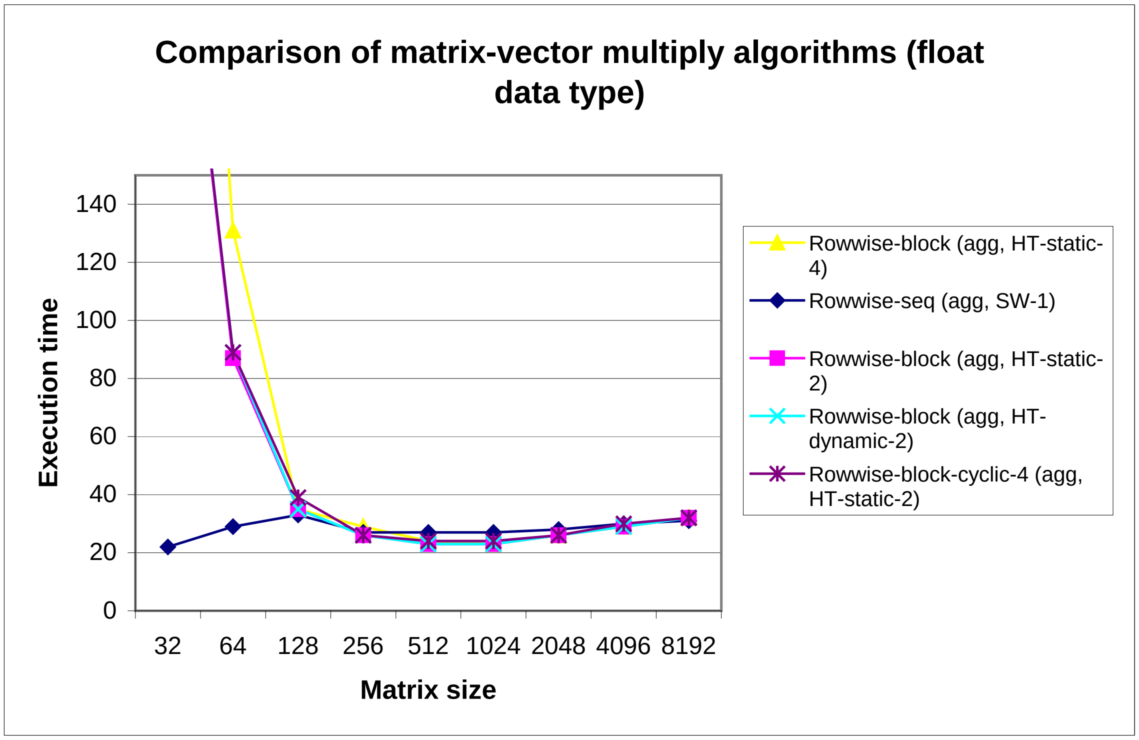

Figure 39 Comparison of matrix-vector multiply algorithms (float data type)

Table 8 and figure 40 compare the algorithms from different viewpoints. Now only the threading system is taken into account, an algorithm can be either sequential, implemented using hyper-threading or software threads. The graph shows that the performance of software threads and sequential algorithm is almost the same, hyper-threaded solution is significantly faster in problem sizes between 256 and 4096 and slower otherwise.

	Hyper-threads		Software-Threads		Sequential	
	Execution Time	Number of Threads	Execution Time	Number of Threads	Execution Time	Number of Threads
32	2,42188	2	2,32554	4	2,32545	1
64	3,16894	4	3,06789	2	3,06806	1
128	3,04923	4	2,94435	2	2,94429	1
256	2,78309	2	2,85528	2	2,85516	1
512	2,45854	2	2,85324	2	2,85181	1
1024	2,40942	4	2,85563	4	2,85534	1

2048	2,72718	2	2,92169	4	2,92667	1
4096	3,1334	2	3,16501	2	3,16411	1
8192	3,3741	2	3,28197	4	3,27045	1

Table 8 Matrix-vector multiplication: Comparison of threading approaches (float data type)

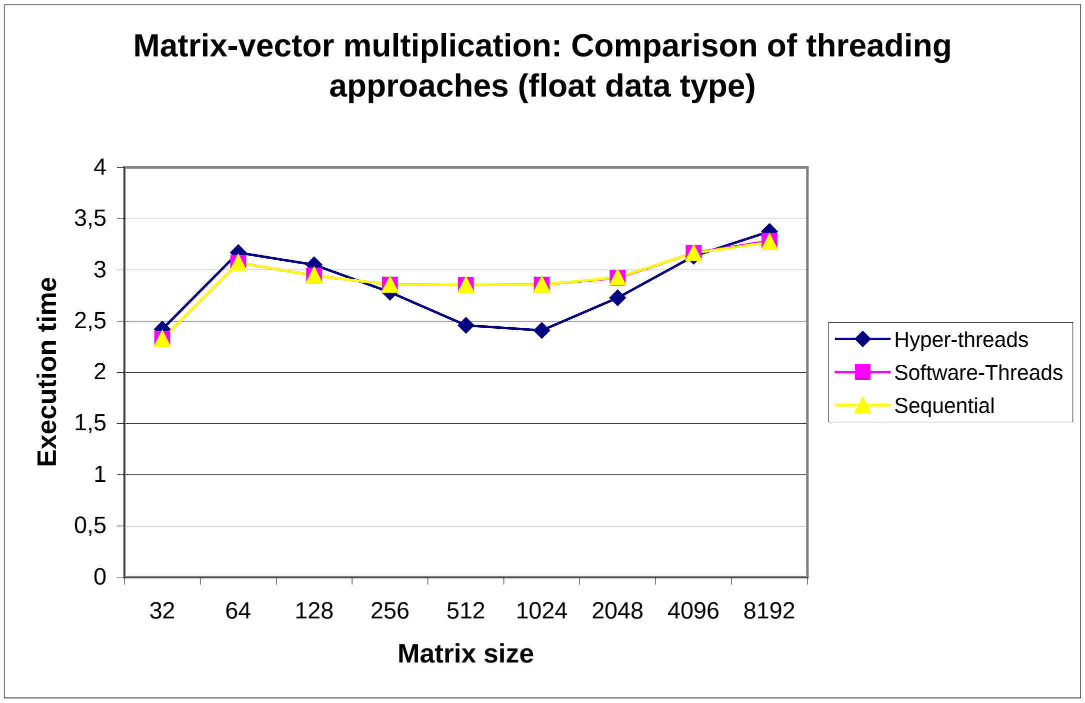

Figure 40 Matrix-vector multiplication: Comparison of threading approaches (float data type)

Data type double

The raw data obtained from time measurement are listed in Table 9. If more than one method scores the same running time, only one of them is listed. The code compiled with less aggressive optimizations performs better on longer vectors.

Matrix size	Parallelism	Parameters	optimization	Threading	Number of threads
32	rowwise	sequential	aggressive	software	2
64	rowwise	sequential	aggressive	software	1
128	rowwise	block	aggressive	ht-static	2
256	rowwise	block	aggressive	ht-dynamic	2
512	rowwise	block	aggressive	ht-dynamic	2

1024	rowwise	block	aggressive	ht-static	2
2048	rowwise	block	aggressive	ht-static	4
4096	rowwise	sequential	aggressive	software	1
8192	rowwise	block-cyclic (32)	aggressive	software	2

Table 9 Summary of matrix-vector multiply algorithms (double data type)

Figure 41 presents the data in graphical form. As previously with float data type the parallel algorithms have high startup time; with block size less than 256 elements the sequential algorithm is in every case faster.

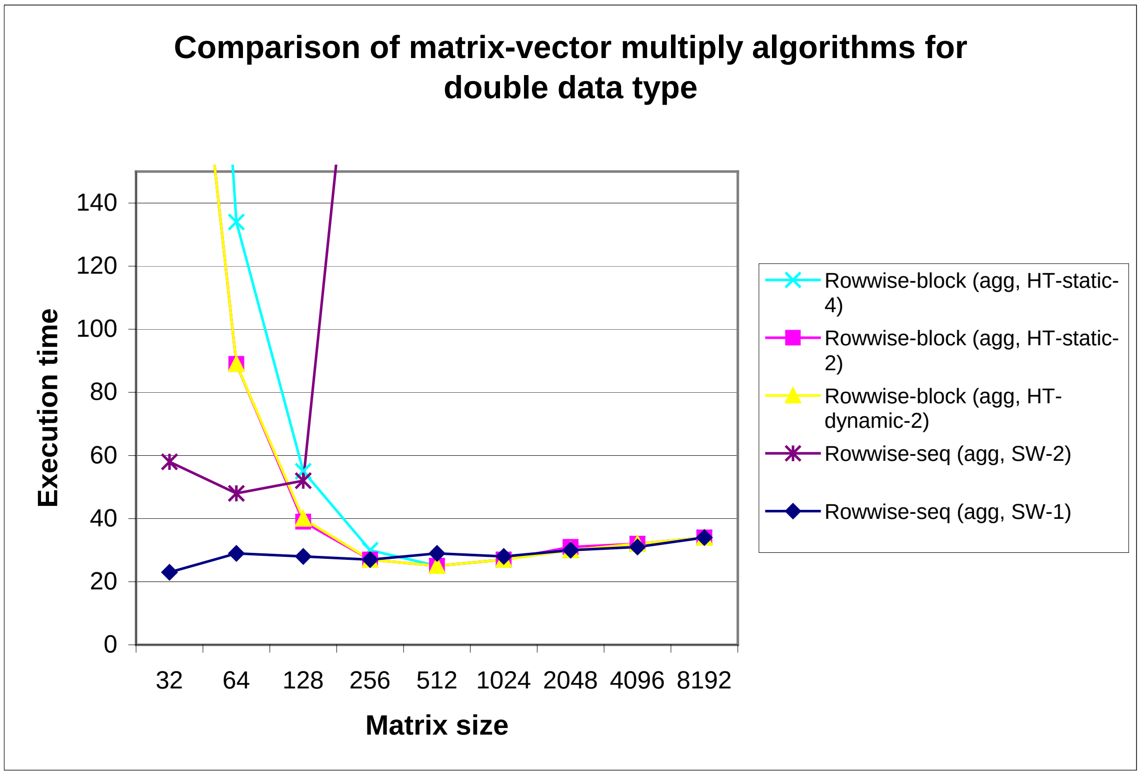

Figure 41 Comparison of matrix-vector multiply algorithms (double data type)

As for float data type, table 10 and figure 42 compare the algorithms from different viewpoints. Only the threading system is taken into account. The graph shows that the performance of software threads and sequential algorithm is almost the same, hyper-threaded solution is significantly faster in problem sizes between 256 and 1024 and slower otherwise.

	Hyper-threads		Software-Threads		Sequential	
	Execution Time	Number of Threads	Execution Time	Number of Threads	Execution Time	Number of Threads
32	2,4585	4	2,3646	2	2,36585	1
64	3,20367	2	3,09348	2	3,09269	1
128	3,05327	4	2,95608	4	2,95628	1
256	2,87456	2	2,88714	2	2,88163	1
512	2,58937	2	3,0159	2	3,02011	1
1024	2,83835	2	2,95986	4	2,95839	1
2048	3,23663	4	3,25197	4	3,252	1
4096	3,3845	2	3,36704	2	3,35353	1
8192	3,59049	2	3,54278	2	3,54594	1

Table 10 Matrix-vector multiplication: Comparison of threading approaches (double data type)

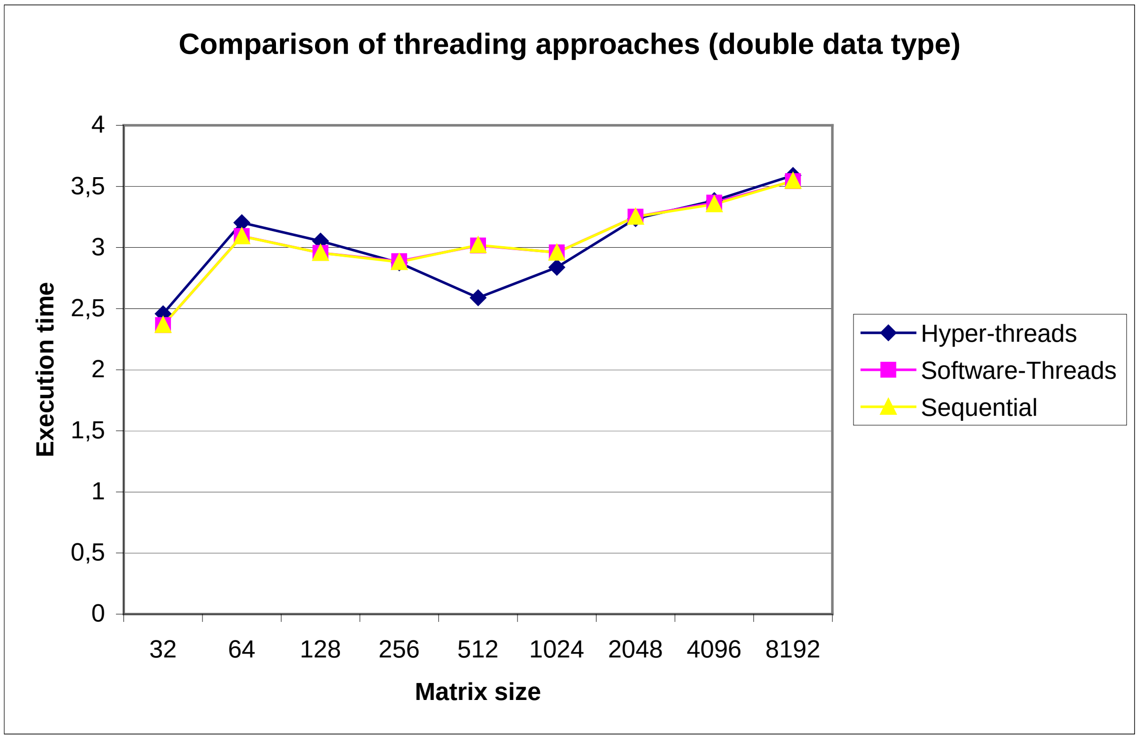

Figure 42 Matrix-vector multiplication: Comparison of threading approaches (double data type)

5.2.3 Discussion

Three different partitioning methods are compared; the row-wise partitioning is a clear winner. This shows the importance of memory access optimizations. The current processors have very good raw performance; the problematic factor is usually the memory architecture. Comparing parallel and sequential approaches, there is a point where parallel algorithms can hide their higher startup cost and become faster, but there is another point where the performance of memory subsystem drops and all approaches become roughly equal.

5.3 Matrix-matrix multiplication

Matrix-matrix multiplication is an operation that takes two $n \times n$ matrices A and B and produces $n \times n$ matrix C so that each element

Equation 8 Matrix-matrix multiplication

$$C[i, j] = \sum_{k=0}^{n-1} A[i,k] \times B[k, j]$$

A sequential matrix-matrix multiply algorithm is presented here:

```
procedure MAT_MULT(A, B, C)
begin
     for i := 0 to n-1
          for j := 0 to n-1
          begin
               C[i, j] := 0
               for k := 0 to n-1
                    C[i, j] := C[i, j] + A[i, k] + B[k, j]
          end
end
```

When the matrix size exceeds level 1 cache size, the performance can degrade significantly. A better block matrix multiplication can be achieved by recursively applying basic algorithm.

Equation 9 Matrix-matrix block multiplication [30]

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} & A_{1,2} \times B_{1,2} + A_{1,2} \times B_{2,2} \\ A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} & A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2} \end{bmatrix}$$

There are two possible further optimizations. First, the B matrix is accessed by columns. This is problematic on many architectures, an option is to transpose the matrix before and after the computation. Second, using small blocks should decrease the amount of L1 cache misses, in order to achieve this it is necessary to copy the block data from the original matrix to temporary matrix that exactly fits block size. If this step is omitted, then every access to following row causes memory alias and cache performance can be again poor.

The algorithm for multiplication of sub matrices is implemented as follows:

```
procedure MAT_MULT_BLOCK(A, posA, B, posB, C, posC)
begin
    subA = get_subblock(A, posA);
    subB = get_subblock(B, posB);
    subC = subA * subB;
    write_subblock(C, posC, subC);
end
```

5.3.1 Parallelization

Matrix operations generally can be easily parallelized. Many operations are independent and can be performed in parallel without the need for synchronization.

Loop parallelization

The basic algorithm can be easily parallelized. The iterations of the most inner loop (with induction variable *k*) share access to *C*[*i*, *j*] variable and in case of parallelizing this loop the access to *C*[*i*, *j*] must be guarded. The loops with induction variables *i* and *j* are independent and can be parallelized as they are. The implementation parallelizes the most outer loop. Figure 43 summarizes dependency of execution time on matrix size and block size.

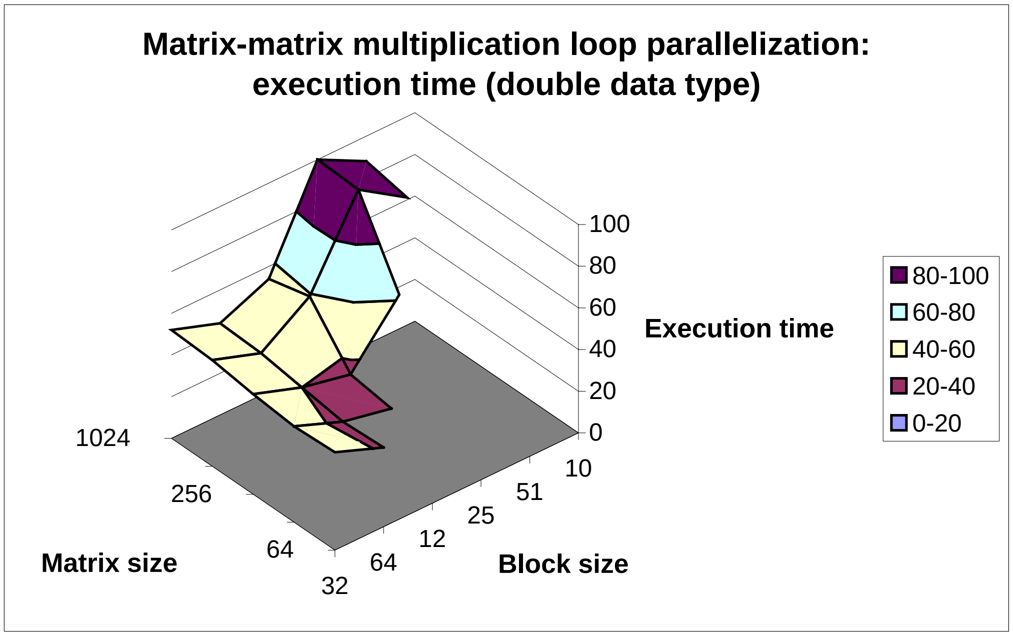

Figure 43 Matrix-matrix multiplication loop parallelization: execution time (double data type)

The graph shows that although small block size bring less cache misses (when multiplying 1024x1024 matrices the algorithm with block size of 32 elements makes about 5 times less cache misses than with 1024 elements) it also increases the number of instruction to execute (about 1.5 times in the example given).

Now let’s see the problem from another view. Figure 44 shows the correspondence between memory metrics and execution time when multiplying 1024x1024 matrices from double data type with varying block size. These metrics are used:

- Number of L1 cache misses. A cache miss can incur some penalty, but can also be hidden by executing other independent instructions. Also with hyper-threading enabled, the other logical processor can utilize the physical processor’s unutilized resources.
- Number of 64kB aliases. Again, it’s difficult to state the cost of this alias case. According to Intel Architecture optimization manual it can vary from being unnoticeable to significant. Anyway, it’s recommended to keep the value of this metrics minimal.
- FSB data activity. The number of clocks that the FSB is transmitting data. High bus utilization causes contentions and penalties.

- Number of instruction executed. This metric shows how effective the algorithm is in terms of utilizing execution resources.

For all metrics, lower numbers are better, but as they are dependent on each other, lower number on one metric doesn't guarantee better execution time. The right axis is execution time, the left is the metric count. The graph shows that all the metrics decrease when using smaller blocks, with the exception of instructions retired metric. The growth of instructions retired metric is responsible for growing execution time.

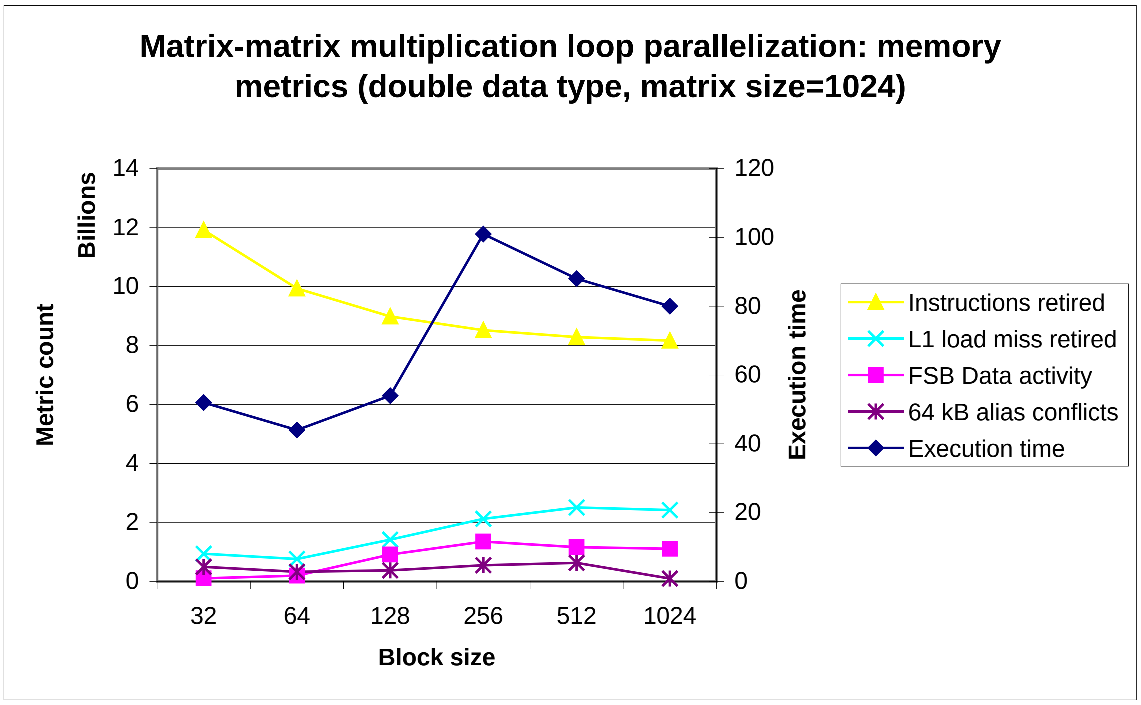

Figure 44 Matrix-matrix multiplication loop parallelization: memory metrics (double data type, matrix size=1024)

Dot-product

Parallelization of dot product calculation in the inner loop of matrix multiplication algorithm is not profitable. The granularity of the problem is not enough big to bring any speedup.

Parallel block computation

When using blocked matrix multiplication algorithm it is possible to allocate each sub block computation to different processor, as they are independent. This provides higher level of granularity. Figure 45 shows the relationship among matrix size, block size and execution time.

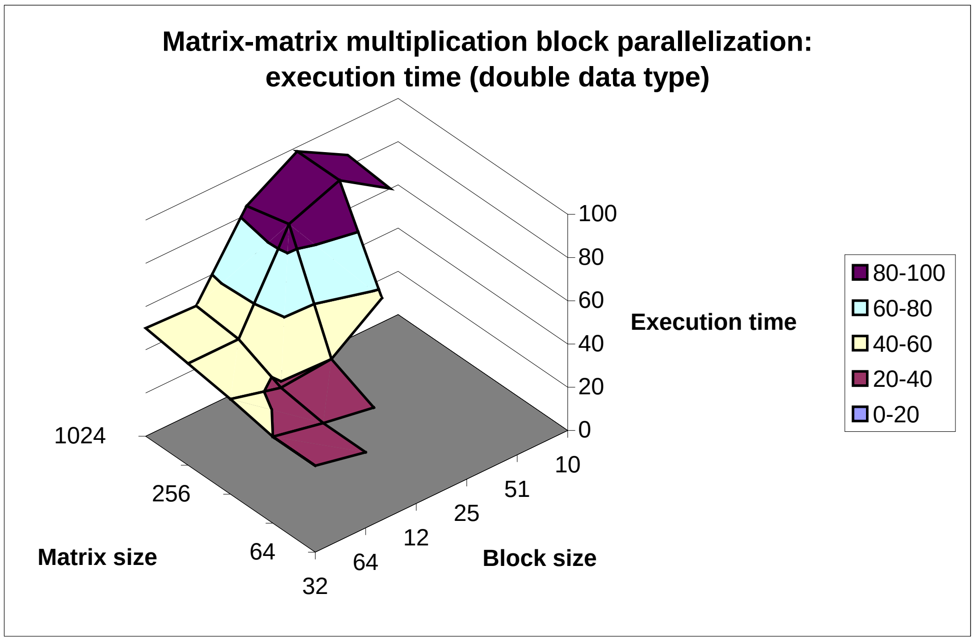

Figure 45 Matrix-matrix multiplication block parallelization: execution time (double data type)

And the metrics figure follows (Figure 46); compared to loop parallelization method shows higher number of cache misses and twice more instruction executed. This instruction overhead is caused by OpenMP and doesn’t significantly increase execution time. For all metrics, lower numbers are better, but as they are dependent on each other, lower number on one metric doesn’t guarantee better execution time. The right axis is execution time, the left is the metric count.

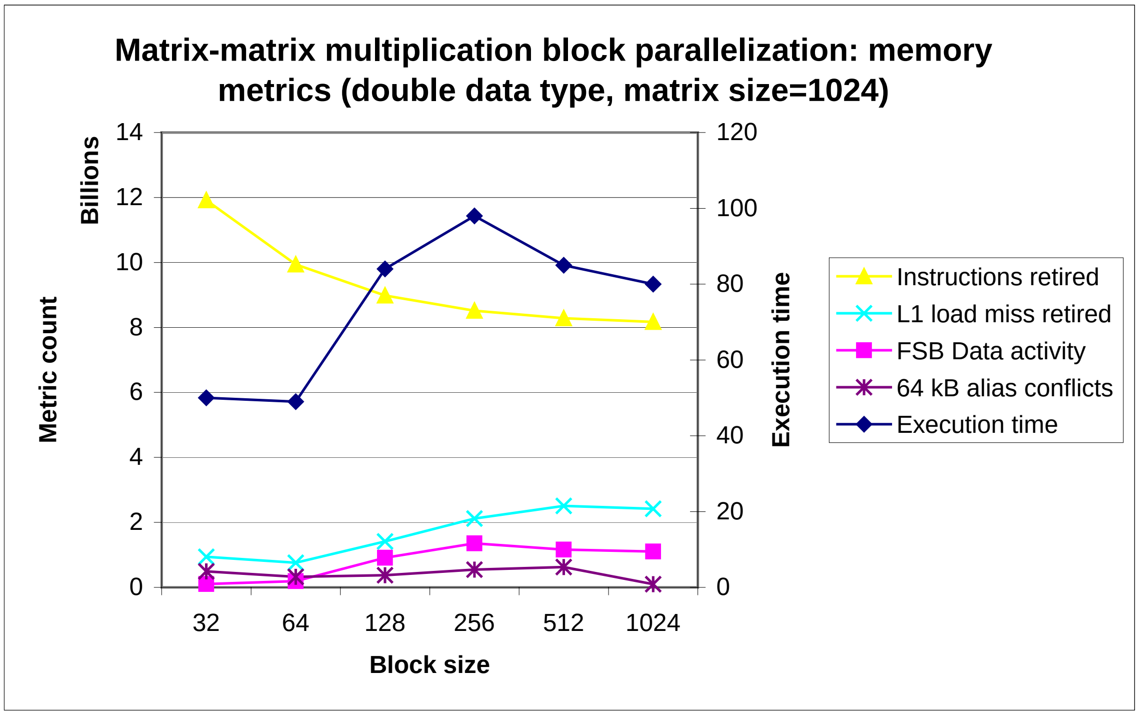

Figure 46 Matrix-matrix multiplication block parallelization: memory metrics (double data type, matrix size=1024)

5.3.2 Results

Several problem configurations have been tested. Several configurations achieve similar results, the table summarizes solutions that should be used in production code.

Data types:	float, double
Matrix size:	$n = 2^x$, x = 6 to 10
Compiler optimizations:	standard, maximum, aggressive
Thread scheduling:	static, dynamic
Number of threads	1, 2, 3, 4
Hyper-threading:	enabled, disabled
Partitioning:	described above

Data type float

Running several experiments with different settings give the following results.

Matrix size	Parallelism	Parameters	optimization	Threads	Number of threads
32	rowwise	Block-transpose-nocopy	aggressive	software	1
64	colwise	sequential- transpose -nocopy	aggressive	HT-static	2
128	rowwise	cyclic- transpose -nocopy	aggressive	HT-dynamic	4
256	rowwise	cyclic- transpose -nocopy	aggressive	HT-static	4
512	rowwise	Block- transpose -nocopy	aggressive	HT-dynamic	2
1024	rowwise	Block- transpose -nocopy	aggressive	HT-static	4
2048	rowwise	Block- transpose -nocopy	aggressive	HT-static	2
4096	rowwise	block- transpose -nocopy	aggressive	HT-dynamic	2
8192	rowwise	sequential- transpose -nocopy	aggressive	software	1

Table 11 Summary of matrix-matrix multiply algorithms (float data type)

Figure 47 shows a comparison of selected algorithms:

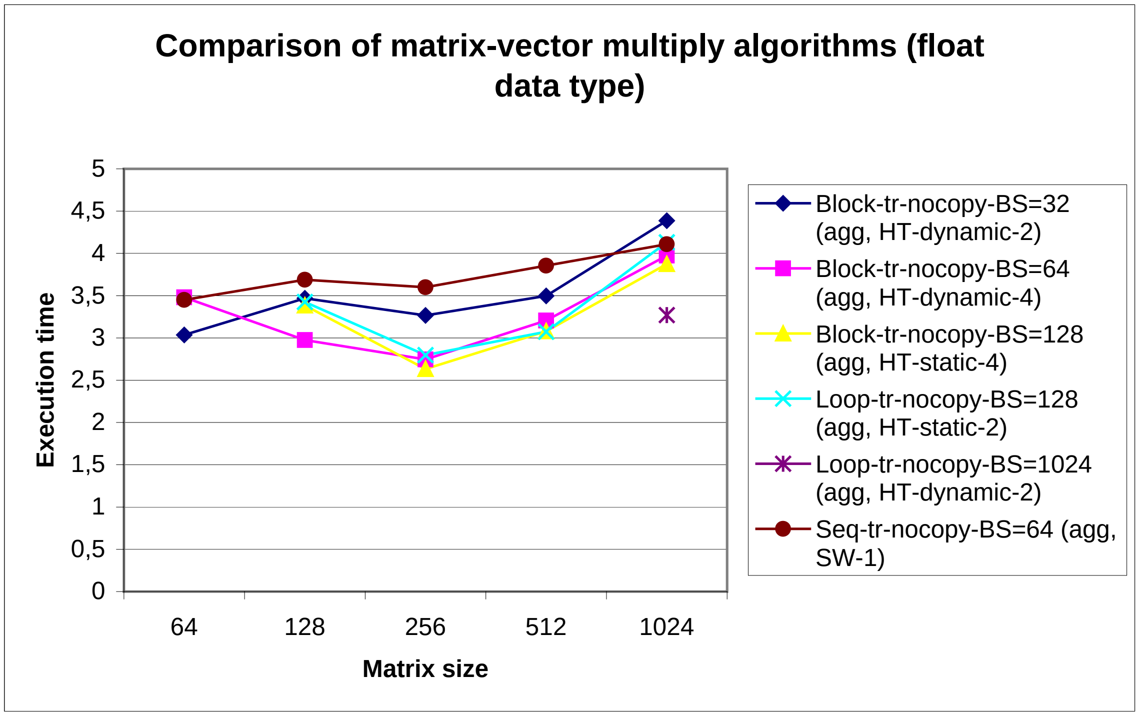

Figure 47 Comparison of matrix-vector multiply algorithms (float data type)

Matrix multiplication is expensive operation that can be easily parallelized and thus parallel algorithms perform better than sequential. Surprisingly algorithms with more threads than installed processors are among the fastest in every case. All the fastest access both matrices in rows (i.e. performs transposing) and don't use temporary are to store matrix sub blocks.

Table 12 and figure 48 compare the fastest versions of hyper-threaded approach, software-thread approach and sequential algorithm, disregarding data partitioning, distribution, optimization level and all other parameters. The data shows that the overhead of software threads in this problem is insignificant. The performance of hyper-threads is about 10% higher and is worth considering in implementation.

	Hyper-threads		Software-Threads		Sequential	
	Execution Time	Number of Threads	Execution Time	Number of Threads	Execution Time	Number of Threads
64	3,59148	2	3,61955	4	3,63435	1
128	2,83039	2	3,20609	4	3,20609	1
256	3,06519	2	3,49492	4	3,48334	1
512	4,28397	2	4,12484	2	4,11664	1
1024	4,22352	4	4,23518	4	4,24878	1

Table 12 Matrix-matrix multiplication: Comparison of threading approaches (float data type)

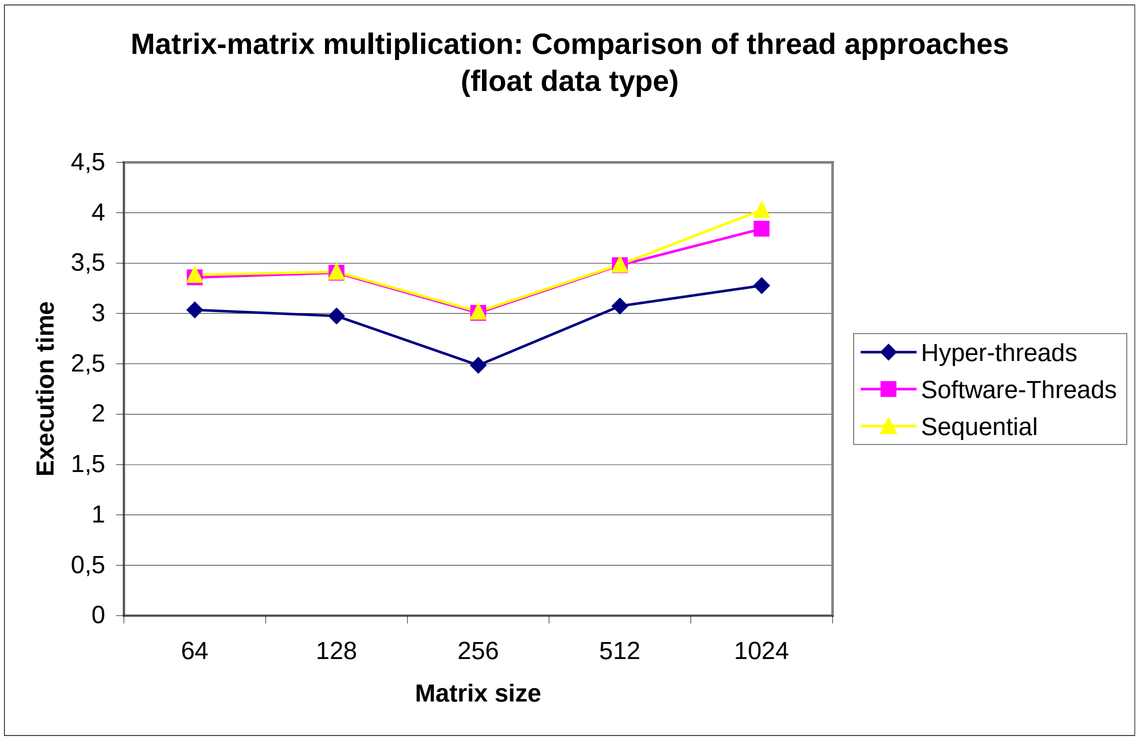

Figure 48 Matrix-matrix multiplication: Comparison of threading approaches (float data type)

Data type double

Running several experiments with different settings give the following results (Table 13).

Matrix size	Parallelism	Parameters	optimization	Threading	Number of threads
32	rowwise	cyclic-transpose-nocopy	aggressive	software	2
64	rowwise	sequential- transpose -nocopy	aggressive	HT-dynamic	2
128	colwise	block- transpose -nocopy	aggressive	HT-dynamic	1
256	colwise	block- transpose -nocopy	aggressive	HT-dynamic	1
512	rowwise	block-cyclic(32)- transpose -nocopy	aggressive	HT-static	2
1024	rowwise	block- transpose -nocopy	aggressive	HT-static	2
2048	rowwise	block- transpose -nocopy	aggressive	HT-static	4
4096	rowwise	sequential- transpose -nocopy	aggressive	software	1
8192	rowwise	block-cyclic-32- transpose -nocopy	aggressive	software	2

Table 13 Summary of matrix-matrix multiply algorithms (double data type)

Comparison of selected algorithms (Figure 49):

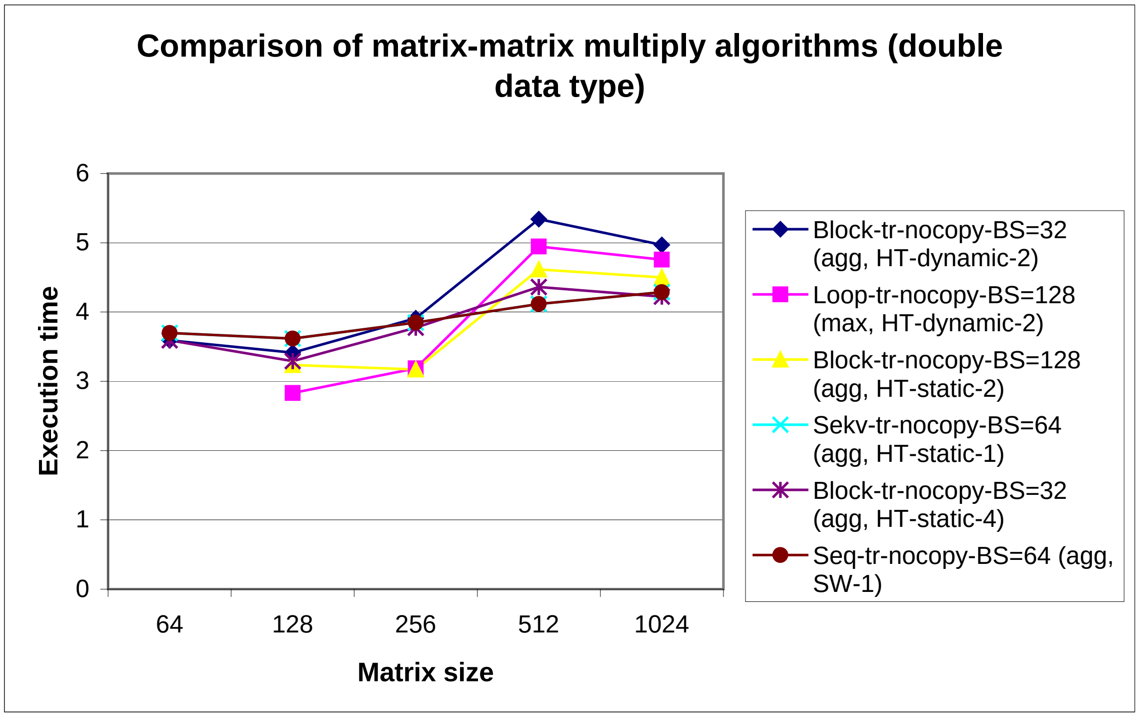

Figure 49 Comparison of matrix-matrix multiply algorithms (double data type)

Generally parallel algorithms perform better than sequential, the only exception are matrices with 64x64 and 512x512 elements, where sequential algorithm is slightly better. All top algorithms transpose the second matrix to allow row-wise access to both matrices. Variants that copies matrix sub blocks to temporary area to limit the number of memory aliases suffer the high cost of copying and don't show among the fastest algorithms.

Table 14 and figure 50 again compare the fastest versions of hyper-threaded approach, software-thread approach and sequential algorithm without regards to data partitioning, distribution, optimization level and all other parameters. The overhead of software threads in this problem is insignificant and the performance of hyper-threads varies from 10 to -5%.

	Hyper-threads		Software-Threads		Sequential	
	Execution Time	Number of Threads	Execution Time	Number of Threads	Execution Time	Number of Threads
64	3,03611	2	3,35727	4	3,38308	1
128	2,97586	4	3,40354	4	3,41391	1
256	2,48637	4	3,00477	4	3,01863	1
512	3,07372	2	3,47833	2	3,48457	1
1024	3,27765	2	3,84094	2	4,02724	1

Table 14 Matrix-matrix multiplication: Comparison of threading approaches (double data type)

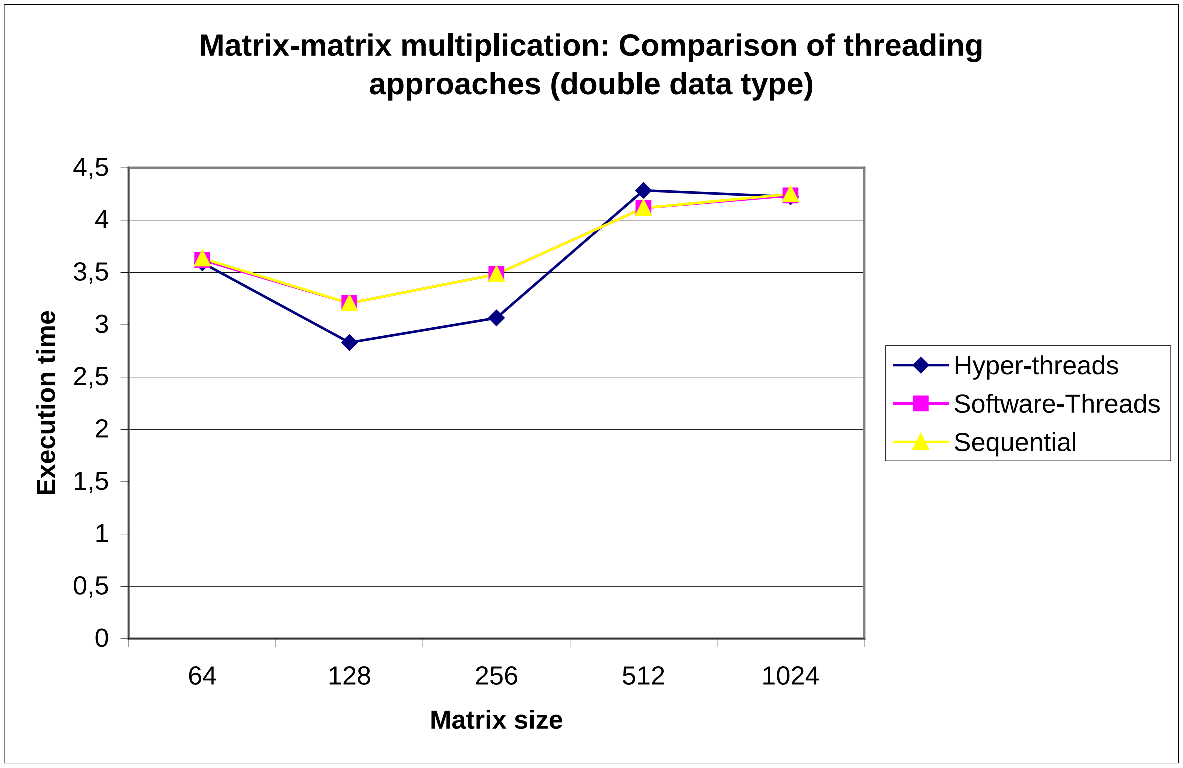

Figure 50 Matrix-matrix multiplication: Comparison of threading approaches (double data type)

5.3.3 Discussion

Matrix multiplication is an operation where parallel processing can show its advantages. Data access is regular and repeated to allow good cache utilization. Where the sequential versions are fastest their lead is very tight and within measurement error. Processing with enabled hyper-threading brings up to 10% advantage on some matrix sizes over sequential algorithm.

6 Sorting

Sorting is one of the most basic operations in computer science. There are many algorithms, suitable for different environment. This is a basic division of sorting algorithms:

Time and space complexity

Specifies amount of time and space units needed to perform the algorithm. It is expressed as a function of *n* – the number of input elements. Asymptotical complexity is a function that describes the behavior of complexity function for $n \to \infty$.

Stability

Stability is a property that says whether algorithm preserve the order of elements with the same value of their key.

Memory store.

Methods with internal sorting expect data in operating memory with random access to sorted values. External methods are used to sort data stored on disk devices.

Processor type

Parallel processor allows implementation of parallel sorting algorithm. Individual steps of the algorithm can overlap without waiting for terminating the previous step.

This section presents two different algorithms implemented in both sequential and parallel way. They are quick sort and radix sort.

6.1 Quicksort

Quick sort [32, 33] is a fast algorithm based on divide-and-conquer strategy. Although the algorithm is simple its implementation is not without problems.

Sequential recursive algorithm has 4 steps:

1. If there are one or zero elements in the array to be sorted, return immediately.
2. Pick an element in the array to serve as a pivot point.

3. Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
4. Recursively repeat the algorithm for both halves of the original array.

The fundamental problem is selection of pivot point, the worst-case results in $O(n^2)$ complexity, while the optimal running time is $O(n\log(n))$. The algorithm is not stable and in principle sequential.

6.1.1 Parallelization

Because the algorithm is based on divide and conquer strategy the performance of parallel execution is limited by the time of initial partitioning. Only the step 4 that executes recursively on both halves of the original array can be done in parallel. There are two basic ways how to effectively parallelize quick sort algorithm.

Sort & merge

The first method distributes the input array among available processors and let them perform sequential version of the algorithm and then merge the results. For the purpose of this test suite the input data array is divided in two halves and each of the two logical processors on Pentium 4 sorts one of them. The most costly operation is final merging, similarly to the split step in the basic algorithm it is performed sequentially.

Algorithm:

```
1. In parallel
     a. Sequentially sort input array from 1 to N/2-1
     b. Sequentially sort input array from N/2 to N
2. Merge results together
```

The other ways to parallelize quick sort takes the advantage of the fact that step 4 of the sequential algorithm can be executed in parallel. This can be done in static or dynamic manner.

Static

Static distribution algorithm performs similar task to the previous algorithm, only the last merge phase is replaced by preceding divide phase.

Algorithm:

1. Split the array into two parts according to chosen pivot point.
2. In parallel
    a. Sequentially sort input array from 1 to N/2-1
    b. Sequentially sort input array from N/2 to N

Dynamic

The dynamic distribution algorithm is implemented using a work pool pattern. This should achieve better load balancing than static distribution and avoid the need for later merging. The disadvantage is the overhead connected with work pool managements.

Dynamic quick sort algorithm steps:

1. Initialize work pool with the array to be sorted
2. Create worker threads
3. Wait for completion

Worker process steps:

1. If the work is done then exit
2. Pick a bag from work pool
3. If there are elements than a certain depth limit in the array to be sorted, sort sequentially and return.
4. Split the array into two parts according to chosen pivot point.
5. Add both halves into work pool
6. Go to step 1

6.1.2 Results

Several problem configurations have been tested.

Data types:	integer
Data size:	$n = 2^x$, x = 16 to 26
Compiler optimizations:	standard, maximal, aggressive
Thread scheduling:	static, dynamic
Number of threads	1, 2, 3, 4

Hyper-threading:	enabled, disabled
Partitioning:	defined above

Table 15 and figure 51 compare selected algorithm. The dynamic partitioning and static partitioning have similar execution time, but the static partitioning is the fastest (with one exception).

Vector size	Parallelism	Depth limit	Optimization	Threading	Number of threads
16	static		aggressive	HT-static	2
17	merge		maximum	HT-static	4
18	static		standard	HT-dynamic	2
19	static		standard	HT-dynamic	4
20	static		maximum	HT-static	2
21	static		standard	HT-static	4
22	static		standard	HT-dynamic	2
23	static		standard	HT-static	2
24	static		standard	HT-static	2
25	static		aggressive	HT-dynamic	2
26	static		aggressive	HT-static	2

Table 15 Summary of quick sort algorithms

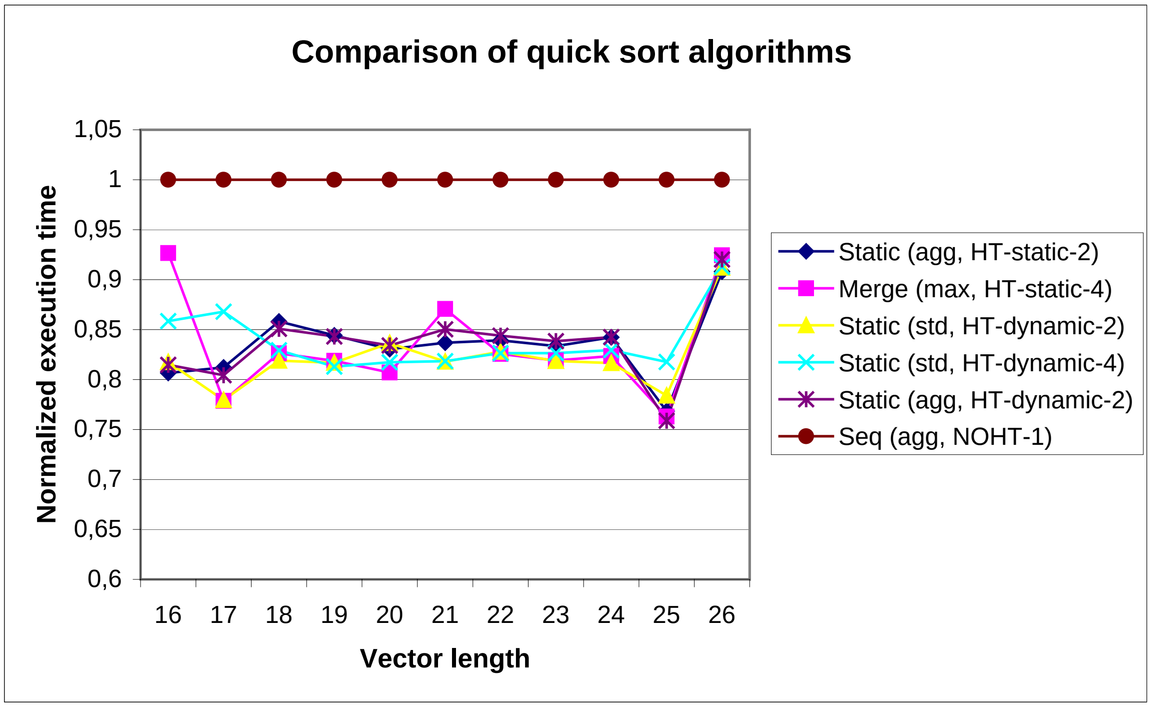

Figure 51 Comparison of quick sort algorithms

Parallel algorithm perform significantly better than sequential. The performance of static partitioned algorithm is higher than others (with the exception of vector length 2 to the power of 17), the reason is that the cost of merging (and also work pool management) is higher than static partitioning. Surprising fact can be the performance of algorithms with standard optimizations. The next few figures (Figures 52 to 56) compares metrics among several algorithms. These metrics show that the distribution step in static partitioning uses executes instructions and is more memory friendly than merging. This is because the memory access pattern in merging causes more conflicts in the architecture.

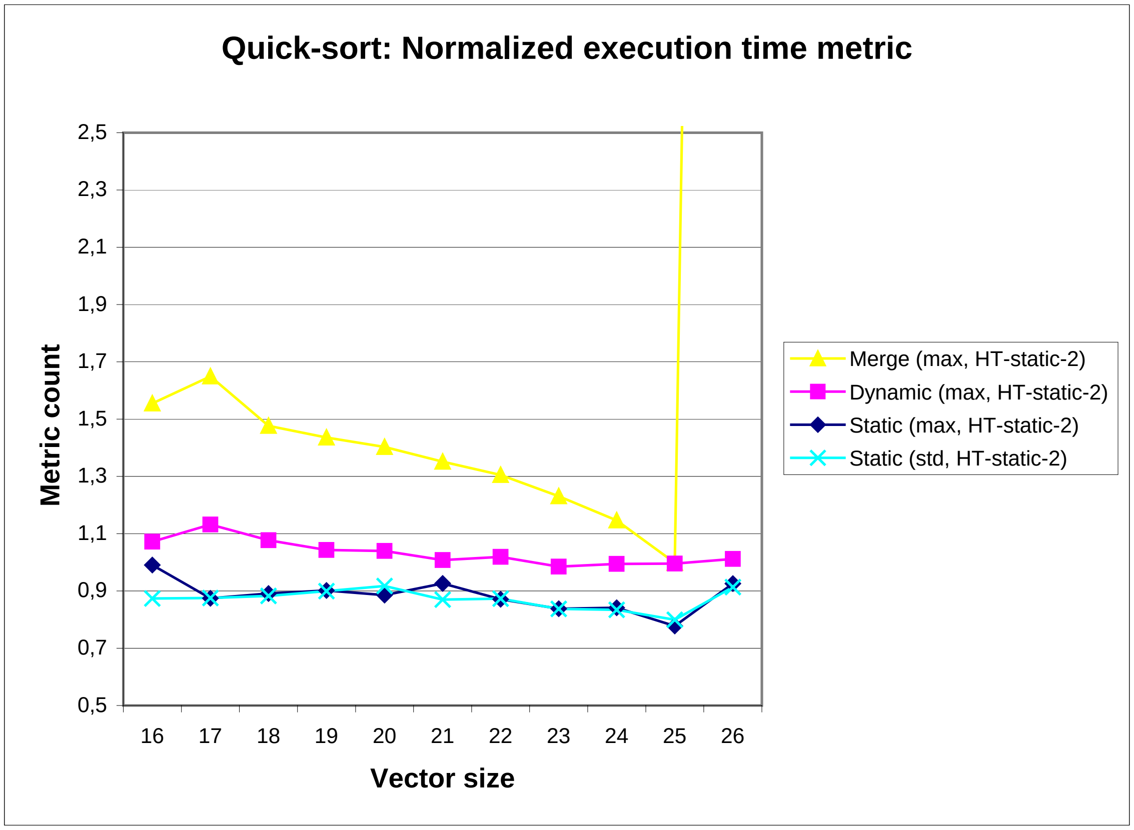

Figure 52 Quick-sort: Normalized execution time metric

These figures (Figure 53-56) compare metrics various parallelization and optimization methods of quick sort run with hyper-threading enabled, static thread scheduling and two execution threads. These metrics show the cost of these parallelization approaches.

The FSB data activity metric shows utilization of front side bus. High utilization is a result of bad cache performance or data conflicts among logical processors. Instructions retired metric shows the number of instructions retired during program execution. This can be considered as a native property of given parallelization and can't be easily influenced by modifying the method. L1 cache load miss retired and 64kB aliases count metrics show how effective the method is with respect to memory architecture.

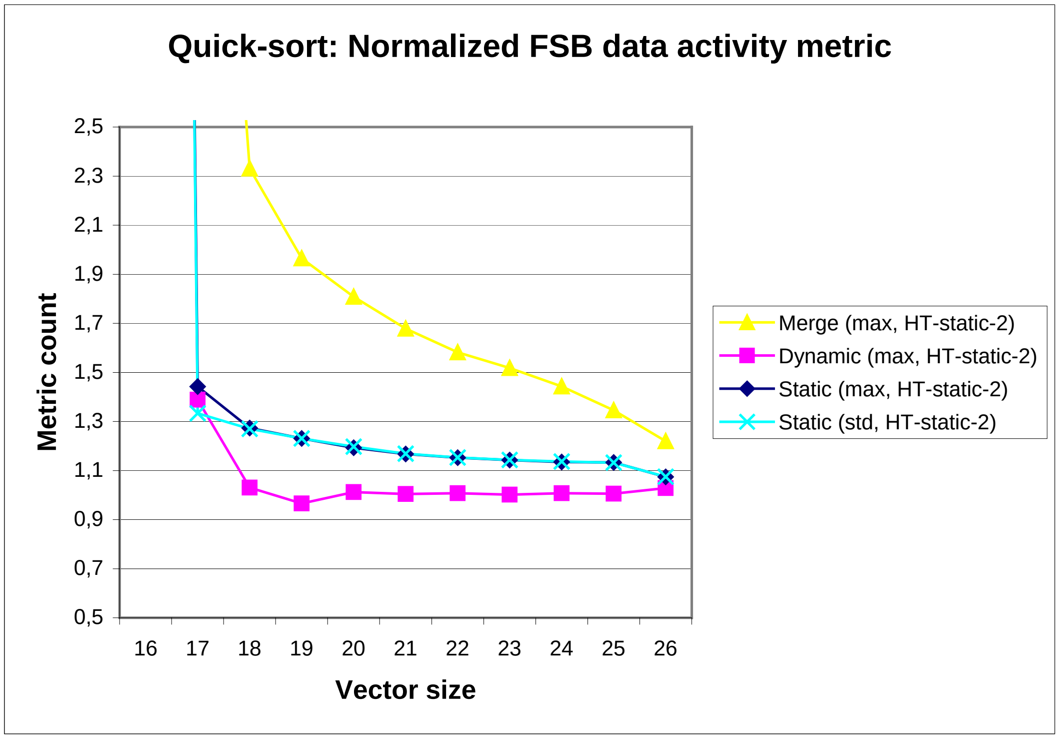

Figure 53 Quick-sort: Normalized FSB data activity metric

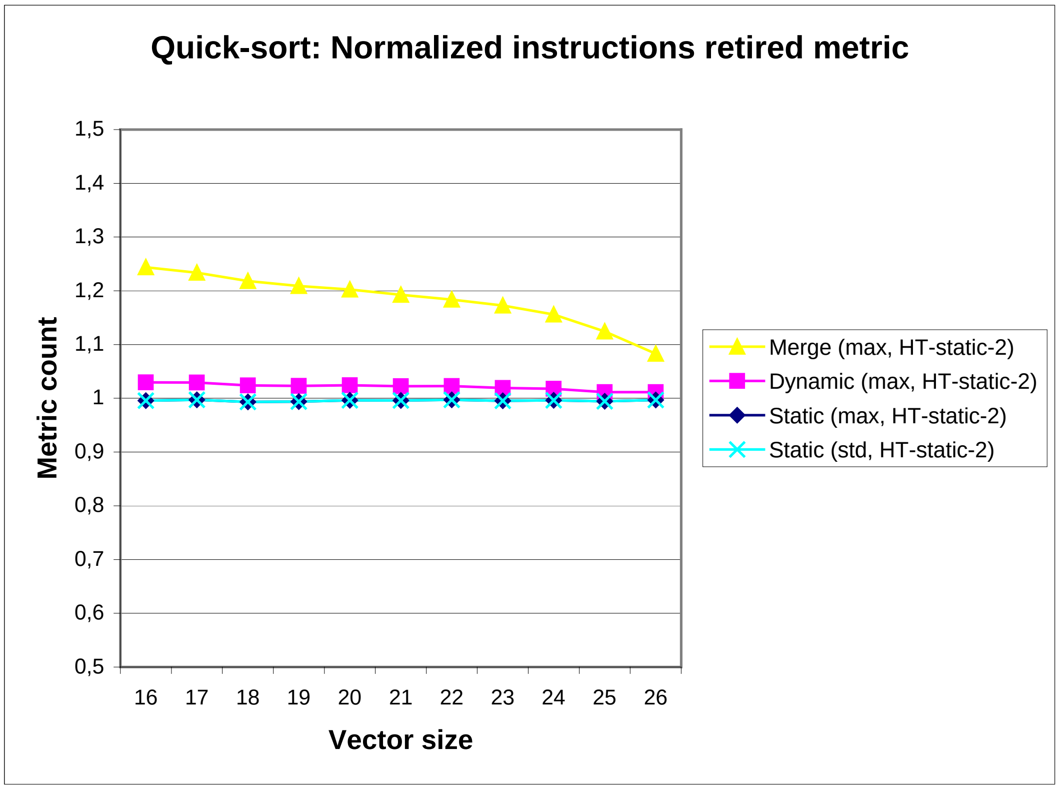

Figure 54 Quick-sort: Normalized instructions retired metric

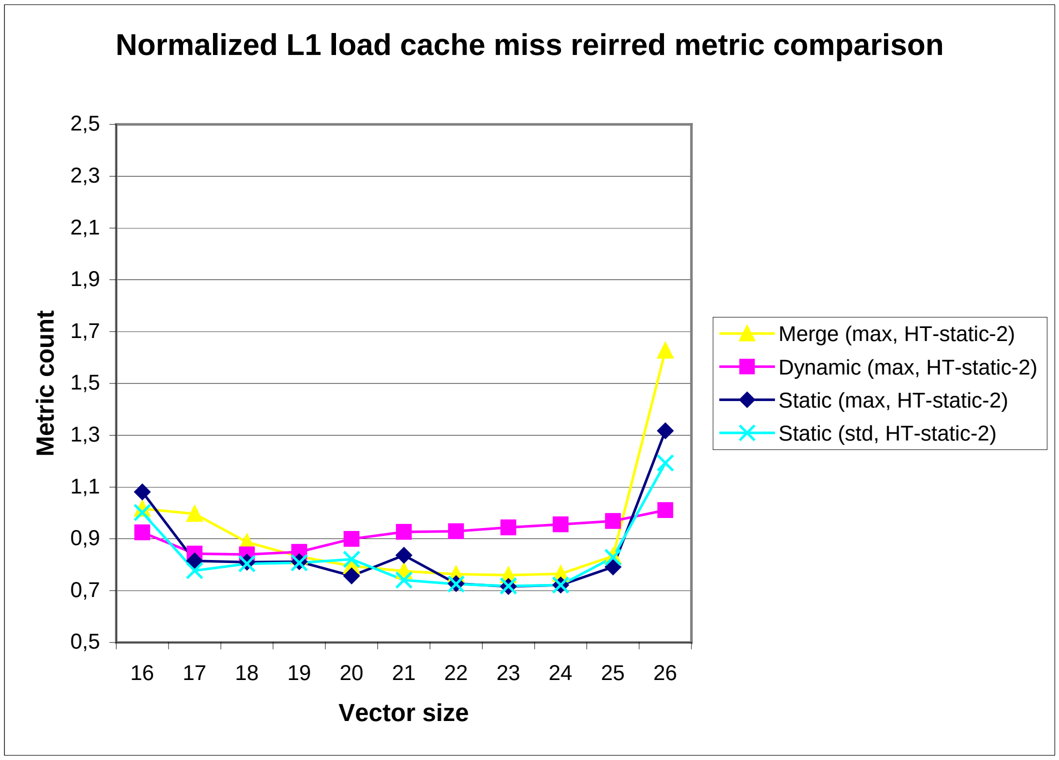

Figure 55 Quick-sort: Normalized L1 load cache miss metric

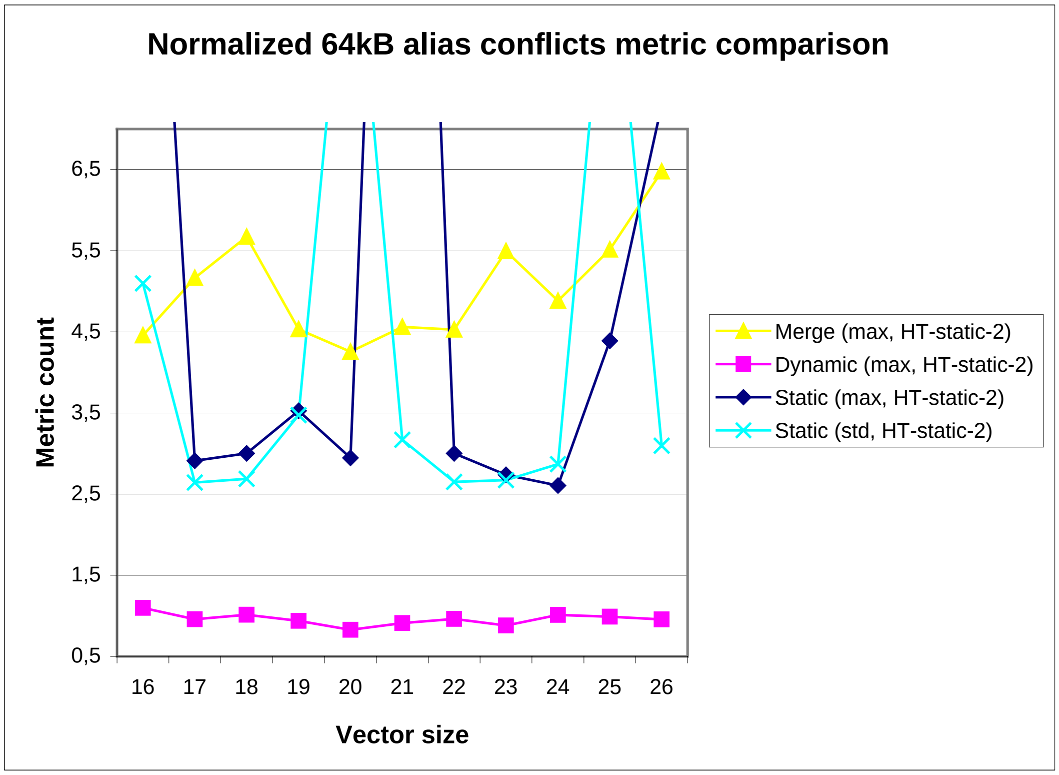

Figure 56 Quick-sort: Normalized 64kB alias conflicts metric

	Hyper-threads		Software-Threads		Sequential	
	Execution Time	Number of Threads	Execution Time	Number of Threads	Execution Time	Number of Threads
16	14,4235	2	17,5249	2	16,1632	1
17	14,4137	2	17,8111	2	16,8166	1
18	15,7515	2	18,6855	2	17,9861	1
19	15,3357	4	17,1073	4	17,1278	1
20	14,0676	2	16,0275	2	15,8801	1
21	12,7915	4	14,8742	2	14,6389	1
22	11,5534	2	13,429	2	13,2618	1
23	10,1935	2	11,7498	2	11,8025	1
24	9,13365	2	10,7003	2	10,7804	1
25	8,3982	2	10,7414	2	10,7692	1
26	11,8974	2	13,1048	2	13,0836	1

Table 16 Quick sort: Comparison of threading approaches

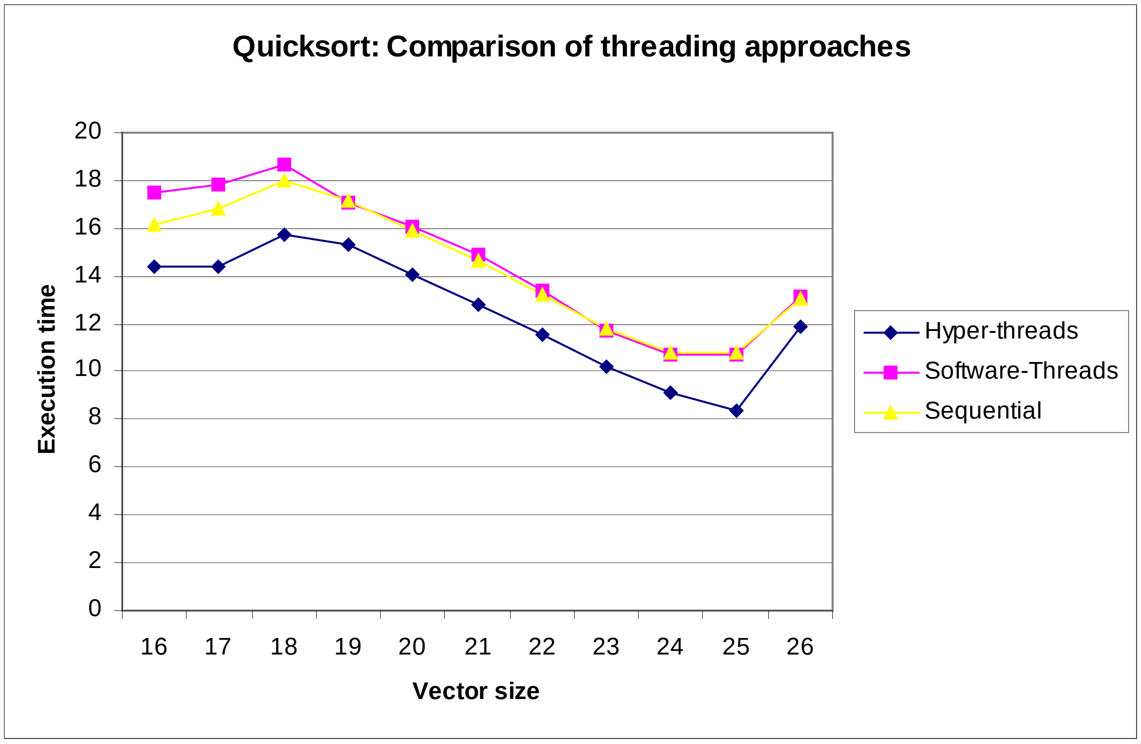

Figure 57 Quick sort: Comparison of threading approaches

6.1.3 Discussion

The static partitioning quick sort algorithm performs best at almost all problem sizes, it provides good balancing without the overhead of dynamic partitioning and can utilize both logical processors. The merge method provides both processor with the same amount of work, but the cost of merging is generally too high. Quick sort algorithm very well benefits from hyper-threading.

6.2 Radix sort

Radix sort is fast stable algorithm for sorting items identified by keys. It is based on sorting punched cards. The principle is to distribute keys in the first pass to 10 bins based on the value of the least significant digit. Then all the bins are concatenated together and in the next pass all numbers are again distributed according to the digit of higher order. These steps are repeated for all digits. The key point is that during distribution the order of keys sorted of lower order digits must be kept. The basic algorithm follows.

```
For $i$ = least_significant_digit to most_significant_digit
        Distribute all numbers in source array to bins according to i-th digit
        Concatenate all bins into the array
End
```

6.2.1 Parallelization

Radix sort is similarly to quick sort in principle sequential method. Nevertheless there are again three basic parallelization options.

Sort & merge

As with every sorting method, the first possibility is to distribute the work among all available processors, let them perform sequential version of the algorithm and then merge the results. For the purpose of this test suite the input data array is divided in two halves and each of the two logical processors on Pentium 4 sorts one of them.

Algorithm:

```
1. In parallel
        a. Sequentially sort input array from 1 to N/2-1
```

    b. Sequentially sort input array from N/2 to N
2. Merge results together

Static

Like quick sort, the radix sort algorithm is principally sequential, but it can be parallelized when the order of processed digits is reversed. The process goes from the most significant bit to the least significant bit and puts numbers with one in the examined bit to the right side of the array and numbers with zero in the examined bit to the right side of the array.

In the static partitioning solution the number array is at first sequentially distributed according to the most significant bit into two parts and then each processor takes one of them and performs sequential radix sort in its own part.

1. Distribute numbers in the array according to the most significant bit
2. In parallel
    a. Sort the part with the value of most significant bit equal to zero
    b. Sort the part with the value of most significant bit equal to one

Dynamic

The dynamic distribution algorithm is again implemented as a work pool. Using work pool should achieve better load balancing at the expense of higher overhead. The difficult part is to design the limit where to stop dividing the task into subtasks for further parallelization.

Algorithm sort:

1. Initialize work pool with the array to be sorted
2. Create worker threads
3. Wait for completion

Algorithm worker:

1. If the work is done then exit
2. Pick a workbag from work pool
3. If there are elements than a certain depth limit in the array to be sorted, sort sequentially and return.
4. Distribute numbers in the array according to the i-th bit

5. Add both parts into work pool and assign as an examination bit i-1
6. Go to step 1

6.2.2 Results

Several problem configurations have been tested.

Data types:	integer
Data size:	$n = 2^x$, x = 16 to 26
Compiler optimizations:	standard, maximum, aggressive
Thread scheduling:	static, dynamic
Number of threads	1, 2, 3, 4
Hyper-threading:	enabled, disabled
Partitioning:	defined by algorithm

Figure 58 and table 17 compare the execution time of selected algorithms. The static partitioning and dynamic partitioning algorithms have almost the same execution time.

Vector size	Parallelism	Depth limit	Optimization	Threading	Number of threads
16	static		aggressive	HT-static	2
17	static		aggressive	HT-dynamic	2
18	static		aggressive	HT-static	2
19	static		aggressive	HT-static	2
20	static		aggressive	HT-dynamic	4
21	static		aggressive	HT-static	4
22	static		aggressive	HT-static	2
23	static		aggressive	HT-dynamic	4
24	static		aggressive	HT-dynamic	4
25	static		aggressive	HT-dynamic	4
26	dynamic	24	aggressive	HT-static	2

Table 17 Summary of radix sort algorithms

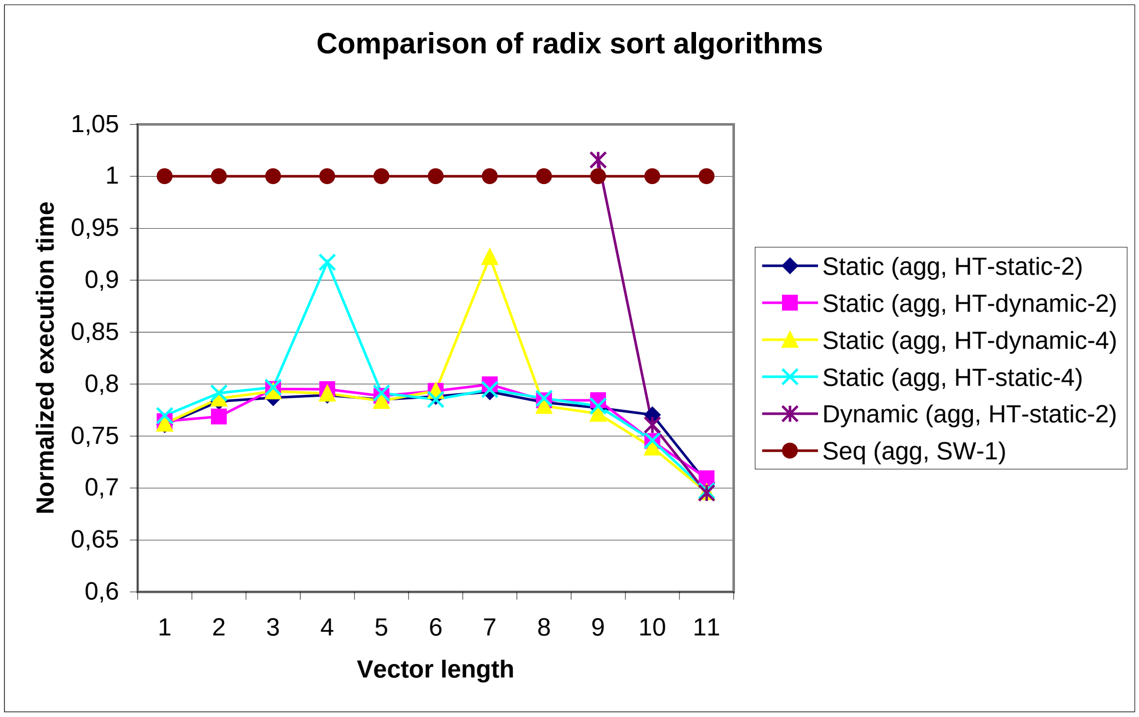

Figure 58 Comparison of radix sort algorithms

Parallel algorithms are significantly faster than sequential and static partitioned algorithm in various variants performs the best except the case with the longest vector. But even in that case static partitioned algorithm is vary fast and compared to the complexity of sort-merge or dynamic sort it should be the optimal choice.

	Hyper-threads		Software-Threads		Sequential	
	Execution Time	Number of Threads	Execution Time	Number of Threads	Execution Time	Number of Threads
16	40,9269	2	54,8683	2	53,1825	1
17	38,7532	2	52,1201	2	50,2572	1
18	36,8797	2	47,8271	2	46,8638	1
19	33,8188	2	43,8016	2	42,8437	1
20	30,4008	4	39,8155	2	38,7267	1
21	26,7971	4	35,5106	2	34,1272	1
22	23,3459	2	30,435	2	29,451	1

23	19,4078	4	25,579	2	24,9137	1
24	15,8586	4	20,6334	2	20,553	1
25	12,7924	4	17,183	2	17,2687	1
26	11,3088	2	16,1883	2	16,2656	1

Table 18 Radix sort: Comparison of threading approaches

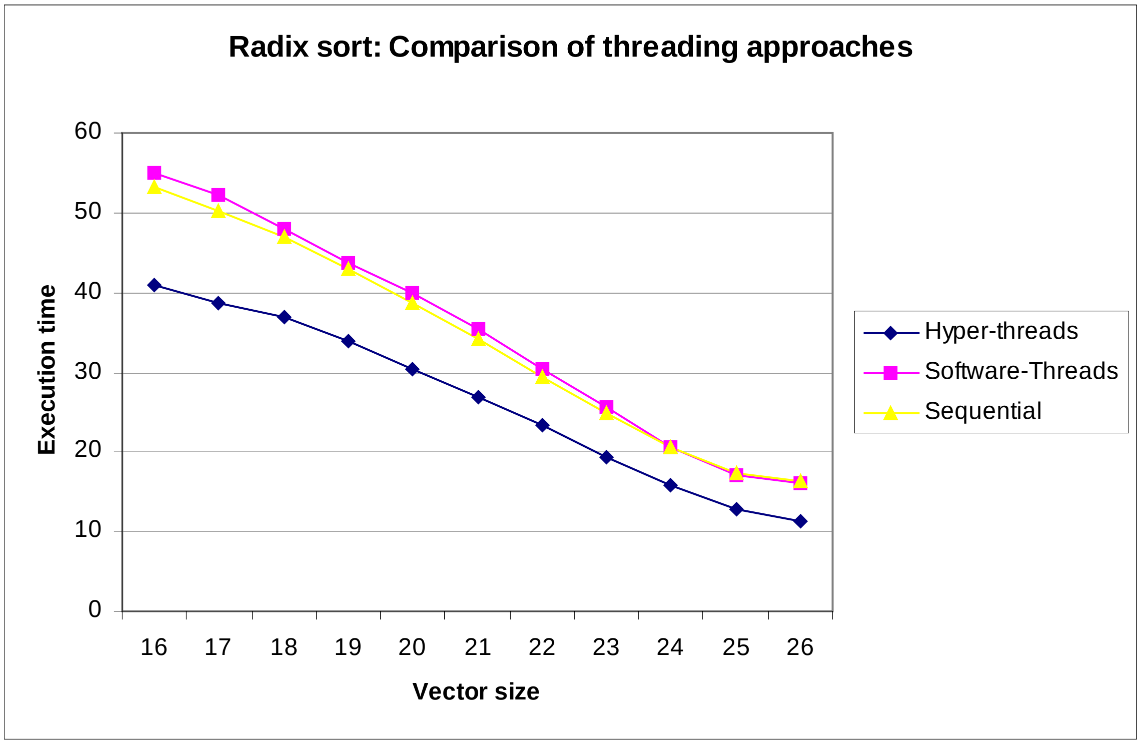

Figure 59 Radix sort: Comparison of threading approaches

6.2.3 Discussion

The static partitioning radix sort algorithm performs good at all problem sizes, it provides good balancing without the overhead of dynamic partitioning and can utilize both logical processors. The merge method provides both processor with the same amount of work, but the cost of merging is higher. As with quicksort, the radix sort also benefits from hyper-threading, the speedup varies between 20 to 40%.

Conclusions

Multithreading is a technology that has always been used only for large scale problems. This was mainly due to their high overhead. The operating system has to schedule and dispatch threads and these are costly operations, therefore multithreading was suitable only for overlapping I/O operations or for systems with multiple processors. The concept of multithreaded processor brings part of the thread management responsibilities to the hardware and enable efficient multithreaded processing on single processor systems and smaller scale problems. This allows to revisit the parallelism pyramid from chapter 2. Now hardware threads lie between instruction level parallelism and software threads (figure 60). All parallelization levels are complementary and still serve it’s purpose:

Software threads

Overlapping I/O, parallel processing on multiple physical processors (without threading support).

Hardware threads

Parallel processing on multiple logical processors inside one physical processor.

Instruction level parallelism

Parallel processing inside one thread.

SIMD processing

Parallel processing in one instruction.

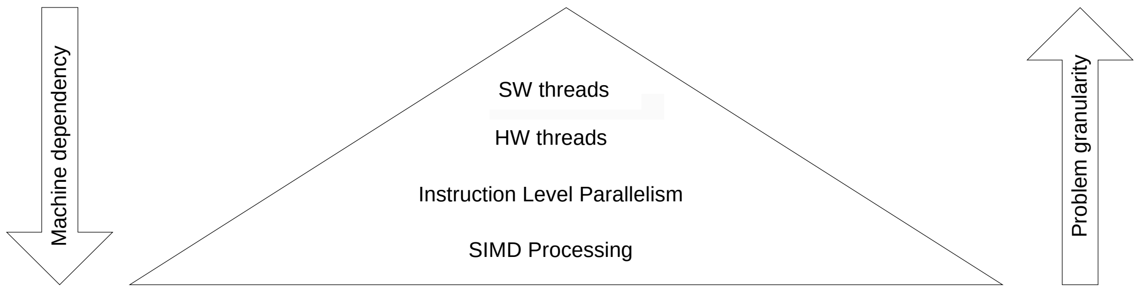

Figure 60   Parallelism pyramid revisited

Hyper-threading is a promising technology that creates the look of two logical processors inside one physical package. This allows two threads to execute concurrently and increase resource usage. Intel reports that hyper-threading technology costs only 5% of processor silicon and brings performance increase up to 20% [11]. This is much better performance to silicon ratio than in any other technique to improve performance. But in computer technology nothing is black and white and the support for hyper-threading has another costs on the software side. The two most important are:

Costs of parallel implementation

A parallel algorithm has a lot more to do than a sequential one. Not only that the sequential algorithm generally contains some sequential part and thus be parallelized as a whole, but also the parallel algorithm must perform initial distribution of data among participating threads, synchronize access to shared data and finally collect all partial results together. These tasks are performed sequentially and add to the overall execution time.

Cost of resource contention

The main advantage (and also a disadvantage) of hyper-threading is resource sharing. Well-done resource sharing allows substantial increase of processor utilization at almost no cost, but when there happens to be resource contention then the performance can be worse than in single processor mode. One of the most important resources is memory architecture in general (cache architecture in particular); poor memory access pattern brings performance down.

Hyper-threading is a technology that brings speedup to many problems, but not all. The speedup depends mainly on problem size and memory access pattern. Extra care must be taken where the memory access is regular (i.e. matrix operations) which can cause significant penalties in parallel algorithms.

References

[1] Intel Corp.: IA-32 Intel Architecture Software developer's Manual Volume 1: Basic Architecture, 2004

[2] Intel Corp.: IA-32 Intel Architecture Software developer's Manual Volume 2: Instruction Set Reference, 2004

[3] Intel Corp.: IA-32 Intel Architecture Software developer's Manual Volume 3: System Programming Guide, 2004

[4] Intel Corp.: IA-32 Intel Architecture Optimization Manual, 2004

[5] Intel Corp.: Intel Technology Journal, Volume 06, Issue 01, 2002

[6] Intel Corp.: Hyper-threading Technology, http://www.intel.com/technology/hyperthread/ (May 2004)

[7] Intel Corp.: Optimizing Applications with Intel(r) C++ and Fortran Compilers for Windows and Linux, 2002

[8] Intel Corp.: Intel C/C++ 7.0 Compiler help

[9] Pabst T.: Intel's New Pentium 4 Processor. Tom's hardware guide, 2000, http://www6.tomshardware.com/cpu/20001120/index.html (May 2004)

[10] Koufaty D., Marr D.: Hyper-threading technology in the NetBurst Architecture. Intel Corp., 2003

[11] Hilton G., Sager D., Upton M., Boggs D., Carmean D., Kyker A., Rousel P.: The Microarchitecture of the Pentium 4 Processor. Intel Corp., 2001

[12] Mehys A.: Hyper-threading and OpenMP. PowerSolutions, 2002

[13] Tian X., Bik A., Girkar M., Grey P.: Exploiting Thread-Level and Instruction Level Parallelism for Hyper-threading technology. Intel Corp. 2003

[14] Stokes J.: Introduction to Multithreading, Superthreading and Hyperthreading, Arstenica, http://www.arstechnica.com/paedia/h/hyperthreading/hyperthreading-1.html (March 2004)

[15] Reddy V., Sule A., Anantaraman A.: Hyper-threading on the Pentium 4. ECE 792E Advanced Microarchitecture, 2002

[16] Leng T., Ali R., Hsieh J., Mashayekhi V., Rooholamini R.: An Empirical Study of Hyper-Threading in High Performance Computing Clusters. Dell Computer Corp.,

[17] Vianney D.: Hyper-threading speeds Linux. IBM Linux Technology Center, January 2003, http://www-106.ibm.com/developerworks/linux/library/l-htl/?ca=dgr-lnxw01 (March 2004)

[18] Dvorak V., Drabek V.: Architektury Procesoru. Vutium, 1999

[19] Hennesy J., Patterson D.: Computer Architecture: A quantitative approach 3rd Edition. Elsevier Science & Technology, 2002

[20] Hennesy J., Patterson D.: Web Extension I: Survey of RISC Architectures (a part of Computer Organization and Design: The Hardware/Software Interface, 2nd Edition), http://www.websdeveloped.com/billy/COMPUTERORGANIZATION/Web%20Extensions/survey.htm (March 2004)

[21] Sudharsanan S.: MAJC-5200: A High Performance Processor for Multimedia Computing, IEEE Micro 2002

[22] Hannibal: Sun's MAJC and Intels's IA-64, Arstenica, http://www.arstechnica.com/cpu/4q99/majc/majc-1.html (March 2004)

[23] Sun Microsystems: MAJC Microprocessor Architecture for Java Computing, https://www.sun.com/processors/MAJC/ (October 2003)

[24] Alverson R., Callahan D., Cumming D., Koblenz B., Porterfield A., Smith B.: The TERA Computer System, Terra Computer Company

[25] NPACI: TERA Multithreaded Architecture MTA, http://www.navo.hpc.mil/pet/Video/Courses/SDSC/PDF/sdsc_hw4.pdf (May 2004)

[26] Bemowski P.: Hyper-threading Linux, http://www.linuxworld.com/story/33885.htm, (March 2004)

[27] Sprunt B.: Brink and Abyss: Pentium 4 Performance Counter Tools For Linux, http://www.eg.bucknell.edu/~bsprunt/emon/brink_abyss/brink_abyss.shtm (March 2003)

[28] OpenMP Forum: OpenMP C and C++ Application Program Interface, http://www.openmp.org/specs/mp-documents/cspec20.pdf (March 2004)

[29] Linear algebra and its applications (3rd edition), David Lay, Addison Wesley, 2002

[30] Wolfram Research: Block Matrix, http://mathworld.wolfram.com/BlockMatrix.html (March 2004)

[31] Kumar V., Grama A., Gupta A., Karypis G.: Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms 2nd Edition, Addison Wesley 2003

[32] Lang H.: Sequential and parallel sorting algorithms, http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm (March 2004)

[33] Honzik J., Hruska T., Macel M.: Vybrane Kapitoly z Programovacich Technik, VUT 1991, https://www.fit.vutbr.cz/study/courses/ADS/private/Texty/Skripta/ch7.pdf (March 2004)

[34] Pipenbrick N.: Radix sort Tutorial,
http://www.cubic.org/~submissive/sourcerer/radix.htm (March 2004)

[35] Eigemann R., Kuhn B., Mattson T., Menon R.: OpenMP tutorial,
http://anusf.anu.edu.au/~dbs900/OpenMP/OMP_tutorial/Module1/ (March 2004)