Performance Study of a Whole Genome Comparison Tool on a Hyper-Threading Multiprocessor*

Juan del Cuvillo[1], Xinmin Tian[2], Guang R. Gao[1], and Milind Girkar[2]

[1] Department of Electrical and Computer Engineering, University of Delaware, Newark, DE 19716, USA
[2] Intel Compiler Laboratory, SSG/EPG, Intel Corporation, 3600 Juliette Lane, Santa Clara, CA 95052, USA

Abstract. We developed a multithreaded parallel implementation of a sequence alignment algorithm that is able to align whole genomes with reliable output and reasonable cost. This paper presents a performance evaluation of the whole genome comparison tool called ATGC — Another Tool for Genome Comparison, on a Hyper-Threading multiprocessor. We use our application to determine the system scalability for this particular type of sequence comparison algorithm and the improvement due to Hyper-Threading technology. The experimental results show that despite of placing a great demand on the memory system, the multithreaded code generated by Intel compiler yields to a 3.3 absolute speedup on a quad-processor machine, with parallelization guided by OpenMP pragmas. Additionally, a relatively high 1^{st} level cache miss rate of 7-8% and a lack of memory bandwidth prevent logical processors with hyper-threading technology enabled from achieving further improvement.

1 Introduction

Multithreading with architecture and microarchitecture support is becoming increasingly commonplace: examples include the Intel Pentinum 4 Hyper-Threading Technology and the IBM Power 4. While using this multithreaded hardware to improve the throughput of multiple workloads is straightforward, using it to improve the performance of a single workload requires parallelization. The ideal solution would be to transform serial programs into parallel programs automatically, but unfortunately this is notoriously difficult. However, the OpenMP programming model has emerged as the de facto standard of expressing parallelism since it substantially simplifies the complex task of writing multithreaded programs on shared memory systems. The Intel C++/Fortran compiler supports OpenMP directive- and pragma-guided parallelization, which significantly increases the domain of applications amenable to effective parallelization. This

* This work was partially supported by NSF and DOE: NSF through the NGS program, grant 0103723; DOE grant DE-FC02-01ER25503.

paper focuses on the parallel implementation and performance study of a whole genome comparison tool using OpenMP on the Intel architecture.

Over the last decades, as the amount of biological sequence data available in databases worldwide grows at an exponential rate, researchers continue the never-ending quest for faster sequence comparison algorithms. This procedure is the core element of bioinformatic key applications such as database search and multiple sequence comparison, which can provide hints to predict the structure, function and evolutionary history of a new sequence. However, it has not been until recently that whole genomes have been completely sequenced, opening the door to the challenging task of whole genome comparison. Such an approach to comparative genomics has a great biological significance. Nonetheless, it can not be accomplished unless computer programs for pair-wise sequence comparison deal efficiently with both execution time and memory requirements for this large-scale comparison. With these two constraints in mind, a new wave of algorithms have been proposed in the last few years [1]. Some have showed to achieve good execution time at the expense of accuracy and they require a large amount of memory [3]. Others, based on heuristics such as "seed-extension" and hashing techniques offer a better trade off between execution time and memory consumption, and are the most commonly used nowadays [7]. More recently, the so called Normalized Local Alignment algorithm has been presented [1]. This iterative method uses the Smith-Waterman algorithm, which provides the accuracy other methods might not have, and a varying scoring system to determine local alignments with a maximum degree of similarity. It also solves the shadow and mosaic effects but increases the algorithm complexity. However, we have showed that by means of parallelization, the execution time for the Smith-Waterman algorithm decreases almost linearly, and hence the original algorithm itself as well as others based on the dynamic programming technique such as the NLA, become affordable for small and medium size genomes [2, 5, 6].

We developed an OpenMP implementation of the affine gap penalties version of the Smith-Waterman algorithm [10]. We also plugged our implementation into the NLA algorithm framework, which consists of an additional few lines of code, to verify the affordability of this method. The parallelization is achieved by means of OpenMP pragmas, more specifically, an *omp parallel pragma* construct that defines a parallel region, which is a region of the program that is to be executed by multiple threads in parallel. The algorithm's recurrence equation results in a relatively regular application, with data dependencies that easily fit into a consumer-producer model. Communication among threads under such a model is performed by shared rotating buffers. At any given time, a buffer is owned by a single thread. The contents of a buffer are consumed at the beginning of the computation stage, and once a new result is produced by the current owner, the buffer is released and its ownership automatically granted to the thread beneath it. As a consequence, neither locks nor critical sections are needed to prevent data-race conditions, a limiting factor in many parallel applications. Our OpenMP implementation runs on a hyper-threading multiprocessor system in

[1] A comprehensive study of related work can be found in the author's Master thesis.

which shared memory enables the mentioned features, i.e. easy programmability and minimum interthread communication overhead.

The rest of the paper is organized as follows. Section 2 describes our multithreaded parallel application. The results from a performance study based on this implementation are presented in section 3, and our conclusions in section 4.

2 Parallel Computation of a Sequence Alignment

A parallel version of the sequence comparison algorithm using dynamic programming must handle the data dependences presented by this method, yet it should perform as many operations as possible independently. The authors have showed that the similarity matrix calculation can be efficiently parallelized using fine-grain multithreading [2, 5, 6]. The implementation described in this paper is based upon that but it exploits parallelism by using OpenMP pragmas.

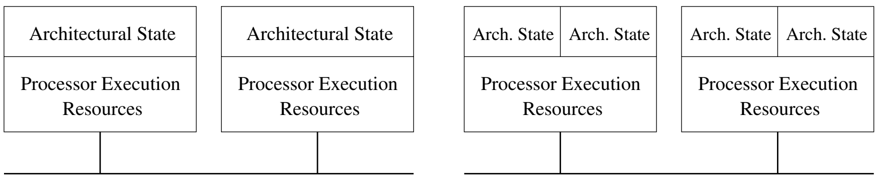
2.1 Hyper-Threading Technology

Intel's hyper-threading technology [4] is the first architecture implementation of the Simultaneous Multi-Threading (SMT) proposed by Tullsen [9]. By making a single physical processor appears as two logical processors, the operating system and user application can schedule processes or threads to be run in parallel instead of sequentially. Such an implementation of hyper-threading technology represents less than a 5% increase in die size and maximum power requirements. Nevertheless, it allows instructions from two independent streams to persist and execute simultaneously, yielding more efficient use of processor resources, improving performance.

As an example, Figure 1(a) shows a traditional multiprocessor system with two physical processors that are not hyper-threading technology-capable. Figure 1(b) shows a similar multiprocessor system with two physical processors and hyper-threading support. With two copies of the architectural state on each physical processor, the second system appears to have four logical processors.

2.2 An OpenMP Parallel Implementation

Our multithreaded implementation divides the scoring matrix into strips and each of these, in turn, into rectangular blocks. Generally speaking, it assigns


Architectural State
Processor Execution Resources
Architectural State
Processor Execution Resources
Arch. State
Arch. State
Processor Execution Resources
Arch. State
Arch. State
Processor Execution Resources

(a) Hyper-threading non-capable processors (b) Hyper-threading-enabled processors

Fig. 1. Traditional and HT-capable multiprocessor systems

the computation of each strip to a thread, having to 2 independent threads per physical processor, i.e. one thread per logical processor.

A thread iterates over blocks in a strip by means of a for loop. At each iteration, the computation of a block is followed by a communication phase in which the thread sends the scores and other information from the last row of its block to the thread beneath. In this way, alignments that cross processors' boundaries can be detected. Furthermore, since memory is shared by all processors this task can be accomplished efficiently without copying data.

Given the sizes of the input sequences, the scoring matrix can not be kept in memory throughout the computation as it is usually done when comparing short sequences. Instead, a pool of buffers is allocated at the beginning of the program execution and these buffers are used repeatedly by all threads to perform the computation. Each buffer holds only the scores of a single row of a block, requiring each thread two of these to compute a block.

Before the computation starts, a single buffer is assigned to each thread but the first, which gets all that is left in the pool. The first thread is the only with two buffers available and, based on data dependencies, the only that can start. When it finishes computing the first block, it makes the buffer containing the scores for the block's last row available to the thread below. To achieve this, it updates a variable that accounts for the number of buffers assigned to the second thread. Actually, this counter is updated in such a way it tells the thread that owns it which iteration should be performed with the data it contains. In other words, a thread just waits [2] until the producer signals it by updating this counter that says which iteration the consumer should be starting. Additionally, since this counter is only written by one thread, the producer, and read by the thread below, the consumer, no locking mechanism needs to be implemented.

At the end of the communication phase, the first thread starts the next iteration taking another buffer from the pool and reusing the one left from the previous iteration. Meanwhile the second thread, which owns two buffers now, starts computing its first block. When threads get a buffer to work with, it means that the data dependencies are satisfied. They then can start computing the corresponding block. Buffers released by the last thread are assigned back to the first thread so they can be reused.

Since each block does not represent exactly the same amount of work, as execution proceeds some threads might be idle waiting for the thread above to release a buffer. Having a pool of buffers allows a thread to work ahead, assuming a buffer is available, when an iteration is completed earlier. As an example of this behavior, the first thread can initially work ahead since buffers are available from the pool. It does not have to wait for a buffer to be used by all threads and become available again before starting the second iteration. That would have serialized the computation and made our implementation quite inefficient.

A snapshot of the computation of the similarity matrix using our multi-threaded implementation on a quad-processor system with hyper-threading support is illustrated in Figure 2. A thread is assigned to each horizontal strip and

[2] To avoid wasting cycles on this spin wait, the PAUSE instruction is used.

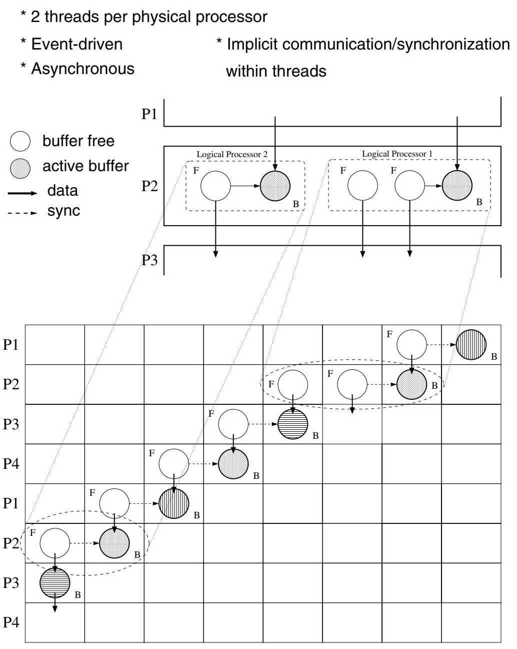


Fig. 2. Computation of the similarity matrix on a SMP system

the actual computation is done on the buffers labeled B(usy). Data or buffers assigned to the thread beneath are labeled F(ree). The figure shows the computation of the main anti-diagonal of the matrix. The arrows indicate data or buffer availability signals. For example, processor 2 sends data (downward arrows) to processor 3 and receives data from processor 1 [3]. Within a thread, that is, between blocks on the same strip, a synchronization signal is implicit (horizontal arrows) as results from one iteration are available to the next without explicit communication.

The number of possible alignments grows exponentially with the length of the sequences compared. Therefore, we can not simply report all the alignments. Instead we are interested in selecting only alignments with high scores. On each node, as each strip of the scoring matrix is calculated, scores above a given threshold are compared with the previous highest scores stored in an table. The number of entries in the table corresponds to the maximum number of alignments that a node can report. Among other information, the table stores the cells' position where an alignment starts and ends. This feature allows us to produce a plot of the alignments found. A point worth noticing is that high score alignments are selected as the similarity matrix is calculated, row by row, since the whole matrix is not stored in memory.

[3] Actually, threads 1 and 2 have worked faster than thread 3, which has now an additional F(ree) buffer ready to start working on without delay.

3 Performance Analysis

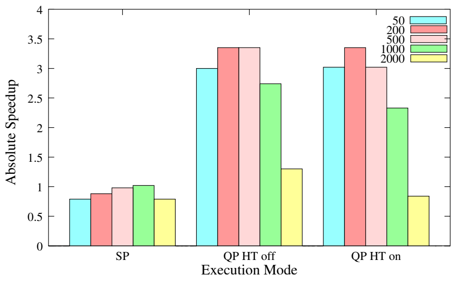
3.1 The System Configuration

The experiments described in this paper were carried out on a 4-way Intel Xeon processor at 1.4GHz with Hyper-Threading support, 2GB memory, a 32KB L1 cache (16KB instruction cache and 16KB two-way write-back data cache), a 256KB L2 cache, and a 512KB L3 cache. All programs were compiled with the Intel C++/Fortran OpenMP compiler version 7.0 Beta [8].

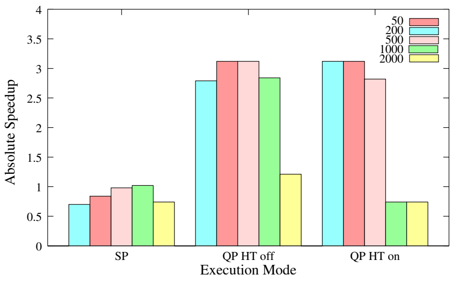
3.2 Results

Our OpenMP parallel implementation was used to align human and mice mitochondrial genomes, human and *Drosophila* mitochondrial genomes, and human herpesvirus 1 and human herpesvirus 2 genomes. We run each comparison several times to consider the effects of the number of threads, block size, and having hyper-threading enabled or disabled in the program execution time.

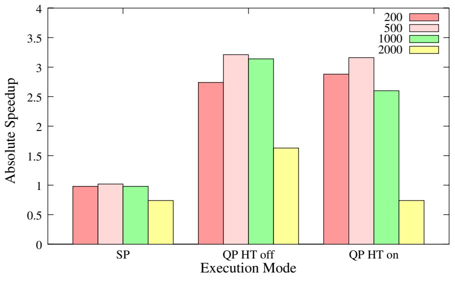
Figure 3 reports the absolute speedup achieved for the genome comparisons mentioned above. Each comparison runs under three execution modes: SP, single processor with a single thread, QP HT off, 4 threads running on 4 processors with hyper-threading disabled, QP HT on, hyper-threading support enabled and 8 threads running on 8 logical processors. For each execution mode several block



(a) Human vs. mice



(b) Human vs. *Drosophila*



(c) HHV1 vs. HHV2

Fig. 3. Absolute speedup for mitochondrial genome comparisons

Table 1. Workload characteristics for the human vs. mouse and human vs. Drosophila genome comparisons with block width 1,000

	SP	HT off	HT on	SP	HT off	HT on
Execution Time (seconds)	30	9	9	28	9	9
Instructions retired (millions)	24,547	42,606	44,819	24,429	42,967	42,347
μops retired (millions)	23,175	52,376	54,418	20,551	49,942	53,134
Trace cache delivery rate	74.40%	102.45%	87.96%	68.08%	98.11%	100.00%
Load accesses (millions)	9,990	17,035	17,873	9,816	16,658	16,662
L1 cache load misses (millions)	712	1,365	1,236	758	1,349	1,222
L2 cache load misses (millions)	32	290	27	30	338	29
L1 cache miss rate	7.13%	8.01%	7.62%	7.72%	8.09%	7.33%
L2 cache miss rate	0.32%	1.70%	0.15%	0.30%	2.03%	1.74%
L3 cache miss rate	0.02%	0.02%	0.04%	0.02%	0.02%	0.05%
Branches retired (millions)	3,111	5,255	5,528	3,318	5,823	5,473
Mispredicted branches (millions)	112	220	217	138	335	231
Branch misprediction rate	3.60%	4.19%	3.90%	4.16%	5.75%	4.22%

sizes are tested as well. The first observation we can make from both three figures is that a 3.3 absolute speedup is achieved for the three test cases. Second, performance does not improve when hyper-threading support is enabled. Runs with 8 threads on 8 logical processors report basically the same execution time as runs with 4 threads on 4 physical processors without hyper-threading support.

Table 1 summarizes the workload characteristics for the human and mouse and human and *Drosophila* genome comparisons, respectively. The first point worth noticing is the similarity between both tables. The *Drosophila* mitochondrial genome is longer than mice mitochondrial genome. Therefore, the amount of computation required to fill a larger similarity matrix is larger as well. However, the amount of work required by the second comparison to keep track of a fewer number of alignments seems to compensate the total amount of work required by each workload. Another issue is the relatively high 1^{st} level cache miss rate, always between 7% and 8%. Although we try to exploit as much locality as possible by selecting an appropriate block width, this parameter can not be set to an arbitrarily small value. When we do so performance decreases because of the small computation-communication ratio. Execution resources might be shared efficiently between logical processors running independent threads. However, if the memory bus can not keep up with the application demands (remember the high cache miss rate) instructions will not be able to proceed normally through the pipeline since store and load buffers are waiting for memory accesses initiated by previous instructions to complete. For this application, many of these instructions represent a memory operation. However, memory operations have to be queued because the memory bandwidth is limited (the same as what four threads see in the QP HT off execution mode). Once the load and store buffers are full, instructions can not be issued faster, regardless of hyper-threading, since resources are unavailable.

4 Conclusions

We have developed a multithreaded parallel implementation of a dynamic programming based algorithm for whole genome comparison that runs on an SMP system and meets the requirements for small and medium size genomes.

The experimental results show that despite being a memory bound application, which places a great demand on the memory system, the multithreaded code generated by Intel compiler yields to a 3.3 absolute speedup on a quad-processor machine, with parallelization guided by OpenMP pragmas. Additionally, a relatively high 1^{st} level cache miss rate of 7-8% and a lack of memory bandwidth prevent logical processors with hyper-threading technology enabled from achieving further improvement.

As future work we intend to investigate why the hyper-threading does not bring additional performance gain. In particular, we will focus on the overhead caused by the loop with the PAUSE instruction, which accounts for 5% of the execution time, and four load-store operations that account for more than 90% of the L2 cache misses. Should we tune our application such that hyper-threading effectively reduces these two factors, we can expect a significant performance improvement.

References

1. A. N. Arslan et al. A new approach to sequence comparison: Normalized sequence alignment. *Bioinformatics*, 17(4):327–337, 2001.
2. J. del Cuvillo. Whole genome comparison using a multithreaded parallel implementation. Master's thesis, U. of Delaware, Newark, Del., Jul. 2001.
3. A. L. Delcher et al. Alignment of whole genomes. *Nucleic Acids Res.*, 27(11):2369–2376, 1999.
4. D. T. Mar et al. Hyper-threading technology architecture and microarchitecture. *Intel Tech. J.*, 6(1):4–15, Feb. 2002.
5. W. S. Martins et al. Whole genome alignment using a multithreaded parallel implementation. In *Proc. of the 13th Symp. on Computer Architecture and High Performance Computing*, Pirenópolis, Brazil, Sep.10–12, 2001.
6. W. S. Martins et al. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In *Proc. of the Pacific Symp. on Biocomputing*, pages 311–322, Mauna Lani, Haw., Jan. 3–7, 2001.
7. S. Schwartz et al. PipMaker — A web server for aligning two genomic DNA sequences. *Genome Res.*, 10(4):577–586, April 2000.
8. X. Tian et al. Intel OpenMP C++/Fortran compiler for hyper-threading technology: Implementation and performance. *Intel Tech. J.*, 6(1):36–46, Feb. 2002.
9. D. M. Tullsen et al. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of the 22nd Ann. Intl. Symp. on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, Jun. 1995.
10. M. S. Waterman. *Introduction to Computational Biology: Maps, Sequences, and Genomes.* Chapman and Hall, 1995.