Parallel Processing of Multiple Pattern Matching Algorithms for Biological Sequences: Methods and Performance Results

Charalampos S. Kouzinopoulos[1], Panagiotis D. Michailidis[2] and
Konstantinos G. Margaritis[1]
[1]*University of Macedonia*
[2]*University of Western Macedonia*
Greece

1. Introduction

Multiple pattern matching is the computationally intensive kernel of many applications including information retrieval and intrusion detection systems, web and spam filters and virus scanners. The use of multiple pattern matching is very important in genomics where the algorithms are frequently used to locate nucleotide or amino acid sequence patterns in biological sequence databases. For example, when proteomics data is used for genome annotation in a process called proteogenomic mapping (Jaffe et al., 2004), a set of peptide identifications obtained using mass spectrometry is matched against a target genome translated in all six reading frames.

Given a sequence database (or text) $T = t_1 t_2 ... t_n$ of length n and a finite set of r patterns $P = p^1, p^2, ..., p^r$, where each p^i is a string $p^i = p^i_1 p^i_2 ... p^i_m$ of length m over a finite character set Σ, the multiple pattern matching problem can be defined as the way to locate all the occurrences of any of the patterns in the sequence database.

The naive solution to this problem is to perform r separate searches with one of the sequential algorithms (Navarro & Raffinot, 2002). While frequently used in the past, this technique is not efficient when a large pattern set is involved. The aim of all multiple pattern matching algorithms is to locate the occurrences of all patterns with a single pass of the sequence database. These algorithms are based of single-pattern matching algorithms, with some of their functions generalized to process multiple patterns simultaneously during the preprocessing phase, generally with the use of trie structures or hashing.

Multiple pattern matching is widely used in computational biology for a variety of pattern matching tasks. Brundo and Morgenstern used a simplified version of the Aho-Corasick algorithm to identify anchor points in their CHAOS algorithm for fast alignment of large genomic sequences (Brudno & Morgenstern, 2002; Brudno et al., 2004). Hyyro et al. demonstrated that Aho-Corasick outperforms other algorithms for locating unique oligonucleotides in the yeast genome (Hyyro et al., 2005). The SITEBLAST algorithm (Michael et al., 2005) employs the Aho-Corasick algorithm to retrieve all motif anchors for a local alignment procedure for genomic sequences that makes use of prior knowledge. Buhler et al use Aho-Corasick to design simultaneous seeds for DNA similarity search (Buhler et al., 2005). The AhoPro software package adapts the Aho-Corasick algorithm to compute the probability of simultaneous motif occurrences (Boeva et al., 2007).

As biological databases are growing almost exponentially in time and the current computational biology problems demand faster and more powerful searches, the performance of the most widely used sequential multiple pattern matching algorithms is not fast enough when used on conventional sequential computer systems. The recent advances in parallel computing are mature enough and can provide powerful computing means convenient to improve the performance of multiple pattern matching algorithms when used on large biological databases.

The goals of this chapter are (a) To discuss intelligent methods for speeding up the search phase of the presented algorithms on large biological sequence databases on both concurrent and shared memory/distributed memory parallel systems. We also present a hybrid parallelization technique that combines message passing between multicore nodes and memory sharing inside each node. This technique could potentially have a better performance than the traditional distributed and shared memory parallelization techniques, (b) to detail the experimental results of the parallel implementation of some well known multiple pattern matching algorithms for biological databases and (c) to identify a suitable and preferably fast parallel multiple pattern matching algorithm for several problem parameters such as the size and the alphabet of the sequence database, the amount of distributed processors and the number of cores per processor. To the best of our knowledge, no attempt has been made yet to implement the multiple pattern matching algorithms on multicore and multiprocessor computing systems using OpenMP and MPI respectively.

The rest of this chapter is organized as follows: the next section presents a short survey of multiple pattern matching algorithms and details the efficient and widely used Commentz-Walter, Wu-Manber and the Salmela-Tarhio-Kytöjoki algorithms. The third section discusses general aspects of parallel computing. The fourth section details methods for the parallel implementation of multiple pattern matching algorithms on clusters and multicore systems. Section five presents a hybrid parallel implementation of multiple pattern matching algorithms that use the MPI/OpenMP APIs. Section six discusses the experimental results of the proposed parallel implementations. Finally, the last section presents the conclusions of this chapter.

2. Multiple pattern matching algorithms

Based on the way the patterns are stored and the search is performed, the multiple pattern matching algorithms can generally be classified in to one of the four following approaches.

- **Prefix algorithms**: With prefix searching the patterns are stored in a trie, a data structure where the root node represents the empty string and every node represents a prefix of one of the patterns. At a given position i of the input string the algorithms traverse the trie looking for the longest possible suffix of $t_1...t_i$ that is a prefix of one of the patterns (Navarro & Raffinot, 2002). One of the well known prefix multiple pattern matching algorithms is Aho-Corasick (Aho & Corasick, 1975), an efficient algorithm based on the Knuth-Morris-Pratt algorithm, that preprocesses the pattern list in time linear in $|P|$ and searches the input string in time linear in n in the worst case. Multiple Shift-And, a bit-parallel algorithm generalization of the Shift-And algorithm for multiple pattern matching was introduced in (Navarro & Raffinot, 2002) but is only useful for a small size of $|P|$ since the pattern set must fit in a few computer words.

- **Suffix algorithms**: The suffix algorithms store the patterns backwards in a suffix automaton, a rooted directed tree that represents the suffixes of all patterns. At each position of the input string, the algorithms search for a suffix of any of the patterns from right to left to skip some of the characters. Commentz-Walter (Commentz-Walter, 1979) is an extension of the Boyer-Moore algorithm to multiple pattern matching that uses a suffix trie. A simpler variant of Commentz-Walter is Set Horspool (Navarro & Raffinot, 2002), an extension of the Horspool algorithm (Horspool, 1980) that can locate all the occurrences of multiple patterns in $O(n \times m)$ time in the worst case. Suffix searching is generally considered to be more efficient than prefix searching.
- **Factor algorithms**: The factor searching algorithms build a factor oracle, a trie with additional transitions that can recognize any substring (or factor) of the patterns. Dawg-Match (Crochemore et al., 1999) and MultiBDM (Crochemore & Rytter, 1994) were the first two factor algorithms, algorithms complicated and with a poor performance in practice (Navarro & Raffinot, 2002). The Set Backward Oracle Matching and the Set Backward Dawg Matching algorithms (Navarro & Raffinot, 2002) are natural extensions of the Backward Oracle Matching (Allauzen et al., 1999) and the Backward Dawg Matching (Crochemore et al., 1994) algorithms respectively for multiple pattern matching.
- **Hashing algorithms**: The algorithms following this approach use hashing to reduce their memory footprint usually in conjunction with other techniques. Wu-Manber (Wu & Manber, 1994) is one such algorithm that is based on Horspool. It reads the input string in blocks to effectively increase the size of the alphabet and then applies a hashing technique to reduce the necessary memory space. Zhou et al. (Zhou et al., 2007) proposed an algorithm called MDH, a variant of Wu-Manber for large-scale pattern sets. Kim and Kim introduced in (Kim & Kim, 1999) a multiple pattern matching algorithm that also takes the hashing approach. The Salmela-Tarhio-Kytöjoki (Salmela et al., 2006) variants of the Horspool, Shift-Or (Baeza-Yates & Gonnet, 1992) and BNDM (Navarro & Raffinot, 1998) algorithms can locate *candidate* matches by excluding positions of the input string that do not match to any of the patterns. They combine hashing and a technique called q-grams to increase the alphabet size, similar to the method used by Wu-Manber.

2.1 Commentz-Walter

Commentz-Walter combines the filtering functions of the single pattern matching Boyer-Moore algorithm and a suffix automaton to search for the occurrence of multiple patterns in an input string. The trie used by Commentz-Walter is similar to that of Aho-Corasick but is created from the reversed patterns. The original paper presented two versions of the algorithm, B and $B1$. The B algorithm creates during the preprocessing phase a trie structure from the reversed patterns of the patterns set where each node corresponds to a single character, constructs the two shift functions of the Boyer-Moore algorithm extended to multiple patterns and specifies the exit nodes that indicate that a complete match is found in $\mathcal{O}(|P|)$ time. The trie is then positioned with its starting node aligned with the m^{th} character of the input string and is compared backwards to the text making state transitions as necessary, until the end of the input string is reached in $\mathcal{O}(n \times m)$ worst case time. When a mismatch is encountered, the trie is shifted to the right using the shift functions based on the knowledge of the suffix of the text that matches to a suffix of one of the patterns. The $B1$ algorithm is a modification of the B algorithm that has a linear search time in the worst case since it stores in memory the input characters that were already scanned. Due to the complicated preprocessing and the higher memory overhead, the original paper discourages its usage.

2.2 Wu-Manber

Wu-Manber is a generalization of the Horspool algorithm, a simple variant of the Boyer-Moore algorithm that uses only the bad-character shift, for multiple pattern matching. To achieve a better performance when $|P|$ is increased, the algorithm essentially enlarges the alphabet size by considering the text as blocks of size B instead of single characters. As recommended in (Wu & Manber, 1994), a good value for B is $log_{\Sigma}2|P|$ although usually B could be equal to 2 for a small pattern set size or to 3 otherwise. In the preprocessing phase, three tables are built, the SHIFT table, the HASH table and the PREFIX table. SHIFT is similar to the bad-character shift table of the Horspool algorithm and is used to determine the number of characters that can be safely skipped based on the previous B characters on each text position. Since the maximum amount of characters that can be skipped based on the value of the SHIFT table is equal to $m - B + 1$, the Wu-Manber algorithm is not very efficient when short patterns are used. The PREFIX table stores a hashed value of the B-characters prefix of each pattern while the HASH table contains a list of all patterns with the same prefix. When the value of SHIFT is greater than 0, the search window is shifted and a new substring B of the input string is considered. When no shift is possible at a given position of the input string, a candidate match is found and the hashed value of the previous B characters of the input string is then compared with the hashed values stored at the PREFIX table to determine if an exact match exists. As the experiments of this chapter involve large pattern set sizes, Wu-Manber was implemented with a block size of $B = 3$.

2.3 Salmela-Tarhio-Kytöjoki

Salmela-Tarhio-Kytöjoki presented three algorithms called HG, SOG and BG, that extend the single pattern Horspool, Shift-Or (Baeza-Yates & Gonnet, 1992) and BNDM algorithms respectively for multiple pattern matching. The algorithms are character class filters; they essentially construct a generalized pattern with a length of m characters in $\mathcal{O}(|P|)$ time for BG and SOG and $\mathcal{O}(|P| \times m)$ time for the HG algorithm that simultaneously matches all the patterns. As $|P|$ increases, the efficiency of the filters is expected to be reduced since a candidate match would occur in almost every position (Salmela, 2009). To solve this problem, the algorithms are using a similar technique to the Wu-Manber algorithm. They treat the input string and the patterns in groups of q characters, effectively enlarging the alphabet size to Σ^q characters. That way, the performance of the algorithms is improved but with an additional memory space cost since an alphabet of 28 characters will grow to 2^{16} characters with 2-grams or 2^{24} with 3-grams. When 3-grams are used, a hashing technique can be applied to reduce the required memory space to 2^{21} bytes. When a candidate match is found, it is verified using a combination of the Karp-Rabin (Karp & Rabin, 1987) algorithm and binary search in $\mathcal{O}(n(logr + m))$ worst case complexity. To improve the efficiency of the verification, a two-level hashing technique is used as detailed in (Muth & Manber, 1996). The combined filtering and verification time is $\mathcal{O}(n)$ for SOG and $\mathcal{O}(nlog_{\Sigma}(|P|)/m)$ for the BG and HG algorithms on average. For the experiments of this chapter the HG, SOG and BG algorithms were implemented using hashed 3-grams.

For this chapter, the Commentz-Walter, Wu-Manber and the Salmela-Tarhio-Kytöjoki multiple pattern algorithms were used. The Commentz-Walter algorithm is substantially faster in practice than the Aho-Corasick algorithm, particularly when long patterns are involved (Wu and Manber, 2004). Wu-Manber is considered to be a practical, simple and efficient algorithm for multiple pattern matching (Navarro & Raffinot, 2002). Finally, Salmela-Tarhio-Kytöjoki is a recently introduced family of algorithms that has a reportedly good performance on specific types of data (Kouzinopoulos & Margaritis, 2010; 2011). For further details and pseudocode of the above algorithms, the reader is referred to (Kouzinopoulos & Margaritis, 2010; 2011) and the original references.

3. Parallel computing

Very often computational applications need more computing power than a sequential computer can provide. One way of overcoming this limitation is to improve the operating speed of processors and other components so that they can offer the power required by computationally intensive applications. Even though this is currently possible to certain extent, future improvements are constrained by the speed of light, thermodynamic laws, and the high financial costs for processor fabrication. A viable and cost-effective alternative solution is to coordinate the efforts of multiple interconnected processors and share computational tasks among them.

Parallel computing can be classified into two basic techniques based on the way the communication between the processing nodes occurs: distributed memory and shared memory. In distributed memory parallel systems (most commonly clusters of computers) the processing elements are loosely-coupled; each has its own local memory and the communication between the elements takes place through an interconnected network, usually with the use of message passing. Shared memory parallel systems (most commonly multi-processors and multi-core processors) on the other hand are tightly-coupled; they have a shared access to a common memory area that is also used for the communication between the processing elements.

3.1 Multi-core system

A multi-core processor is a type of parallel system, which consists of a single component with two or more independent actual processors (called "cores"). In other words, it is a single integrated circuit chip or die that includes more than one processing unit. Each core may independently implement optimizations such as superscalar execution (a CPU architecture that allows more than one instruction to be executed in one clock cycle), pipelining (a standard feature in RISC processors and Graphics Processor Units that is much like an assembly line: the processor works on different steps of the instruction at the same time), and multithreading (a specialized form of multitasking enabling concurrent execution of pieces of the same program). Using multiple processors on a single piece of silicon enables increased parallelism, saves space on a printed circuit board which enables smaller footprint boards and related cost savings, reduces distance between processors which enables faster intercommunication with less signal degradation than if signals had to travel off-chip between processors and reduces the dependence on growth of processor speeds and the related increasing gap between processor and memory speeds. Hyper-Threading is a technique used in Intel processors that makes a single physical processor appear to the operating system as two logical processors by sharing the physical execution resources and duplicating the architecture state for the two logical processors.

Threads are a popular paradigm for concurrent programming on uniprocessor as well as on multiprocessor machines. On multiprocessor systems, threads are primarily used to simultaneously utilize all the available processors while in uniprocessor systems, threads are used to utilize the system resources effectively. This is achieved by exploiting the asynchronous behaviour of an application for overlapping computation and communication. Multithreaded applications offer quicker response to user input and run faster. Unlike forked process, thread creation is cheaper and easier to manage. Threads communicate using shared variables as they are created within their parent process address space. Threads are potentially portable, as there exists an IEEE standard for POSIX threads interface, popularly called pthreads (Nichols et al., 1996) that is available on PCs, workstations, SMPs and clusters. Threads have been extensively used in developing both application and system software.

The most widely used API for shared memory parallel processing is OpenMP, a set of directives, runtime library routines and environmental variables that is supported on a wide range of multicore systems, shared memory processors, clusters and compilers (Leow et al., 2006). The approach of OpenMP is to start with a normal sequential programming language but create the parallel specifications by the judicious use of embedded compiler directives. The API consists of a set of specifications for parallelizing programs on shared memory parallel computer systems without the explicit need for threads management.

3.2 Cluster system

A cluster is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource.

A computer node can be a single or multiprocessor system (PCs, workstations, SMPs and multi-core processors) with memory, I/O facilities, and an operating system. A cluster generally refers to two or more computer nodes connected together. The nodes can exist in a single cabinet or be physically separated and connected via a LAN. An interconnected (LAN-based) cluster of computers can appear as a single system to users and applications. Such a system can provide a cost-effective way to gain features and benefits (fast and reliable services) that have historically been found only on more expensive proprietary shared memory systems.

Message passing libraries allow efficient parallel programs to be written for distributed memory systems. These libraries provide routines to initiate and configure the messaging environment as well as sending and receiving packets of data. Currently, the two most popular high-level message-passing systems for scientific and engineering application are the PVM (Parallel Virtual Machine) (Geist et al., 1994) from Oak Ridge National Laboratory, and MPI (Message Passing Interface) defined by MPI Forum (Snir et al., 1996).

MPI is the most popular message passing library that is used to develop portable message passing programs using either C or Fortran. The MPI standard defines both the syntax as well as the semantics of a core set of library functions that are very useful in writing message passing programs. MPI was developed by a group of researchers from academia and industry and has enjoyed wide support by almost all the hardware vendors. Vendor implementations of MPI are available on almost all parallel systems. The MPI library contains over 125 functions but the number of key concepts is much smaller. These functions provide support for starting and terminating the MPI library, getting information about the parallel computing environment, point-to-point and collective communications.

3.3 Parallel programming models

Parallel applications can be classified into some well defined programming models. This section presents a brief overview of two popular programming models, the data parallel and master-worker models. For further details on parallel programming models, the reader is referred to (Buyya, 1999; Grama et al., 2003).

3.3.1 The data-parallel model

Data-parallel (Grama et al., 2003) is a programming model where the tasks are statically or semi-statically mapped onto processes and each task performs similar operations on different data. This type of parallelism that is a result of identical operations being applied concurrently on different data items is called data parallelism. The work may be done in phases and the data operated upon in different phases may be different. Typically, data-parallel computation phases are interspersed with interactions to synchronize the tasks or to get fresh data to the tasks. Since all tasks perform similar computations, the decomposition of the problem into tasks is usually based on data partitioning because a uniform partitioning of data followed by a static mapping is sufficient to guarantee load balance.

Data-parallel algorithms can be implemented in both shared-memory and message-passing paradigms. However, the partitioned address-space in a message-passing paradigm may allow better control of placement, and thus may offer a better handle on locality. On the other hand, shared-memory can ease the programming effort, especially if the distribution of data is different in different phases of the algorithm.

3.3.2 The master-worker model

The master-worker model (Buyya, 1999) consists of two entities: the master and multiple workers. The master is responsible for decomposing the problem into small tasks (and distributes these tasks among a farm of worker processes), as well as for gathering the partial results in order to produce the final result of the computation. The worker processes execute in a very simple cycle: get a message with the task, process the task, and send the result to the master. Usually, the communication takes place only between the master and the workers while only rarely do the workers communicate with each other.

Master-worker may either use static load-balancing or dynamic load-balancing. In the first case, the distribution of tasks is all performed at the beginning of the computation, which allows the master to participate in the computation after each worker has been allocated a fraction of the work. The allocation of tasks can be done once or in a cyclic way. The other way is to use a dynamically load-balanced master/worker paradigm, which can be more suitable when the number of tasks exceeds the number of available processors, or when the number of tasks is unknown at the start of the application, or when the execution times are not predictable, or when we are dealing with unbalanced problems.

The master-worker model can be generalized to the hierarchical or multi-level master - worker model in which the top-level master feeds large chunks of tasks to second-level masters, who further subdivide the tasks among their own workers and may perform part of the work themselves. This model is generally equally suitable to shared memory or message- passing paradigms since the interaction is naturally two-way; i.e., the master knows that it needs to give out work and workers know that they need to get work from the master.

While using the master-worker model, care should be taken to ensure that the master does not become a bottleneck, which may happen if the tasks are too small (or the workers are relatively fast). The granularity of tasks should be chosen such that the cost of doing work dominates the cost of transferring work and the cost of synchronization. Asynchronous interaction may help overlap interaction and the computation associated with work generation by the master. It may also reduce waiting times if the nature of requests from workers is non-deterministic.

4. Parallel implementations for distributed and shared memory systems

To implement the Commentz-Walter, Wu-Manber and the Salmela-Tarhio-Kytöjoki algorithms on a cluster and multi–core environment we followed the process of two phases: dividing a computation into smaller parts and assigning them to different processes for parallel execution. A major source of overhead in parallel systems is the time the processes stay idle due to uneven distribution of load. To decrease the execution time, the available data set must be decomposed and mapped to the available processes in such a way that this overhead is minimized.

There are two available mapping techniques, the static mapping technique and the dynamic mapping technique. Static mapping is commonly used in a homogeneous environment where all processes have the same characteristics while the dynamic mapping is best suited in heterogeneous set-ups. A simple and efficient way to divide the data set into smaller parts, especially when no interaction occurs between neighbour array elements, is by using a line partitioning where the data is divided on a per line basis.

Let p be the number of available processes and w the number of parts that a text is decomposed to. In the case of static mapping, each process receives a part consisting of $\lceil \frac{n}{p} \rceil + m - 1$ text characters prior to the execution of the algorithms. When a dynamic mapping is used instead, the data set is decomposed into more parts than the available processes $(w > p)$ and each process receives $sb + m - 1$ characters where sb is the chosen block size during the execution of the algorithms. There is an overlap of $m - 1$ characters on each part to ensure that each process has all the data needed, resulting in $w(m - 1)$ additional characters to be processed for the dynamic mapping and $p(m - 1)$ for the static.

For the experiments of this chapter the Master-Worker model was used, as it was concluded in (Cringean et al., 1988) that is the most appropriate model for pattern matching on either message passing or shared memory systems. For the data distribution between the master and the workers, both a dynamic and a static distribution of text pointers was considered as detailed in (Michailidis & Margaritis, 2003). The biological databases and the patterns resided locally on each node. The pattern set was preprocessed first by each worker and the master then distributed a pointer offset to each worker to indicate the area of the text that was assigned for scanning during the search phase.

As opposed to distributed memory parallelization, shared memory parallelization does not actually involve a distribution of data since all threads have access to a common memory area. OpenMP provides the programmer with a set of scheduling clauses to control the way the iterations of a parallel loop are assigned to threads, the static, dynamic and guided clauses. With the static schedule clause, the assignment of iterations is defined before the computation while with both the dynamic and guided clause, the assignment is performed dynamically at computation time (Ayguade et al., 2003).

When a block size is not specified, OpenMP divides the data set into p blocks of equal size for the static clause, where p is the number of processes, while for the dynamic and guided clause the default block size is 1 iteration per thread, which provides the best level of workload distribution but at the same the biggest overhead due to synchronization when scheduling work (Bailey, 2006).

5. Hybrid parallel implementation

In this section we propose a hybrid parallelization approach that combines the advantages of both shared and distributed memory parallelization on a cluster system consisting of multiple interconnected multi-core computers using a hierarchical model. At the first level, parallelism is implemented on the multi-core computers using MPI where each node is responsible for one MPI process. In the next level, the MPI processes spread parallelism to the local processors with the use of OpenMP directives; each OpenMP thread is assigned to a different processor core. More specifically, a static or dynamic distribution of text pointers is used for the MPI processes to parallelize the computation and distribute the corresponding data. Within each MPI process, OpenMP is used to further parallelize the multiple pattern matching algorithms by using a combined parallel work-sharing construct for each computation, namely a parallel *for* directive with either the static, dynamic or guided scheduling clauses. Figure 1 presents a pseudocode of the proposed hybrid technique.

6. Experimental methodology

To compare the performance of the parallel implementations of the multiple pattern matching algorithms, the practical running time was used as a measure. Practical running time is the total time in seconds an algorithm needs to find all occurrences of a pattern in an input string including any preprocessing time and was measured using the MPI_Wtime function of the Message Passing Interface since it has a better resolution than the standard $clock()$ function.

The data set used consisted of the genome of Escherichia coli from the Large Canterbury Corpus, the SWISS-PROT Amino Acid sequence database and the FASTA Amino Acid (FAA) and FASTA Nucleidic Acid (FNA) sequences of the A-thaliana genome:

- The genome of Escherichia coli from the Large Canterbury Corpus with a size of n = 4.638.690 characters and the FASTA Nucleidic Acid (FNA) of the A-thaliana genome with a size of n = 118.100.062 characters. The alphabet $\Sigma = \{a, c, g, t\}$ of both genomes consisted of the four nucleotides a, c, g and t used to encode DNA.
- The FASTA Amino Acid (FAA) of the A-thaliana genome with a size of n = 11.273.437 characters and the SWISS-PROT Amino Acid sequence database with a size of n = 182.116.687 characters. The alphabet $\Sigma = \{a, c, d, e, f, g, h, i, k, l, m, n, p, q, r, s, t, v, w, y\}$ used by the databases had a size of 20 different characters.

The pattern set used consisted of 100.000 patterns where each pattern had a length of $m = 8$ characters.

The experiments were executed on a homogeneous computer cluster consisting of 10 nodes with an Intel Core i3 CPU with Hyper-Threading that had a 2.93GHz clock rate and 4 Gb of memory, a shared 4MB L3 cache and two microprocessors cores, each with 64 KB L1 cache

```
Main procedure
main()
{
   1. Initialize MPI and OpenMP routines;
   2. If (process==master) then call master(); else call worker();
   3. Exit message passing operations;
}

Master sub-procedure
master()
{
   1. Broadcast the name of the pattern set and text to workers; (MPI_Bcast)
   2. Broadcast the offset of the text, the blocksize and the number of
      threads to workers; (MPI_Bcast)
   3. Receive the results (i.e. matches) from all workers; (MPI_Reduce)
   4. Print the total results;
}

Worker sub-procedure
worker()
{
   1. Receive the name of the pattern set and text; (MPI_Bcast)
   2. Preprocess the pattern set;
   3. Receive the offset of the text, the blocksize and the
      number of threads; (MPI_Bcast)
   4. Open the pattern set and text files from the local disk and store the
      local subtext (from text + offset to text + offset + blocksize) in memory;
   5. Call the chosen multiple pattern matching algorithm passing
      a pointer to the subtext in memory;
   6. Divide the subtext among the available threads (#pragma omp parallel for);
   7. Determine the number of matches from each thread (reduction(+: matches));
   8. Send the results (i.e. matches) to master;
}
```

Fig. 1. Pseudocode of the hybrid implementation

and 256 KB L2 cache. The nodes were connected using Realtek Gigabit Ethernet controllers. Additional experiments were executed on a Core 2 Quad CPU with 2.40GHz clock rate and 8 Gb of memory, 4 × 32 KB L1 instruction cache, 4 × 32 KB L1 data cache and 2 × 4 MB L2 cache. The Ubuntu Linux operating system was used on all systems and during the experiments only the typical background processes ran. To decrease random variation, the time results were averages of 100 runs. All algorithms were implemented using the ANSI C programming language and were compiled using the GCC 4.4.3 compiler with the “-O2” and “-funroll-loops” optimization flags.

7. Experimental results

This section discusses the performance speedup achieved by the Commentz-Walter, Wu-Manber and the Salmela-Tarhio-Kytöjoki family of multiple pattern matching algorithms when executed in parallel using OpenMP, MPI and a hybrid OpenMP/MPI parallel technique. The total execution time T_{tot} of a multiple pattern matching algorithm on a cluster of distributed nodes generally equals to the summation of the average processing time T_p on

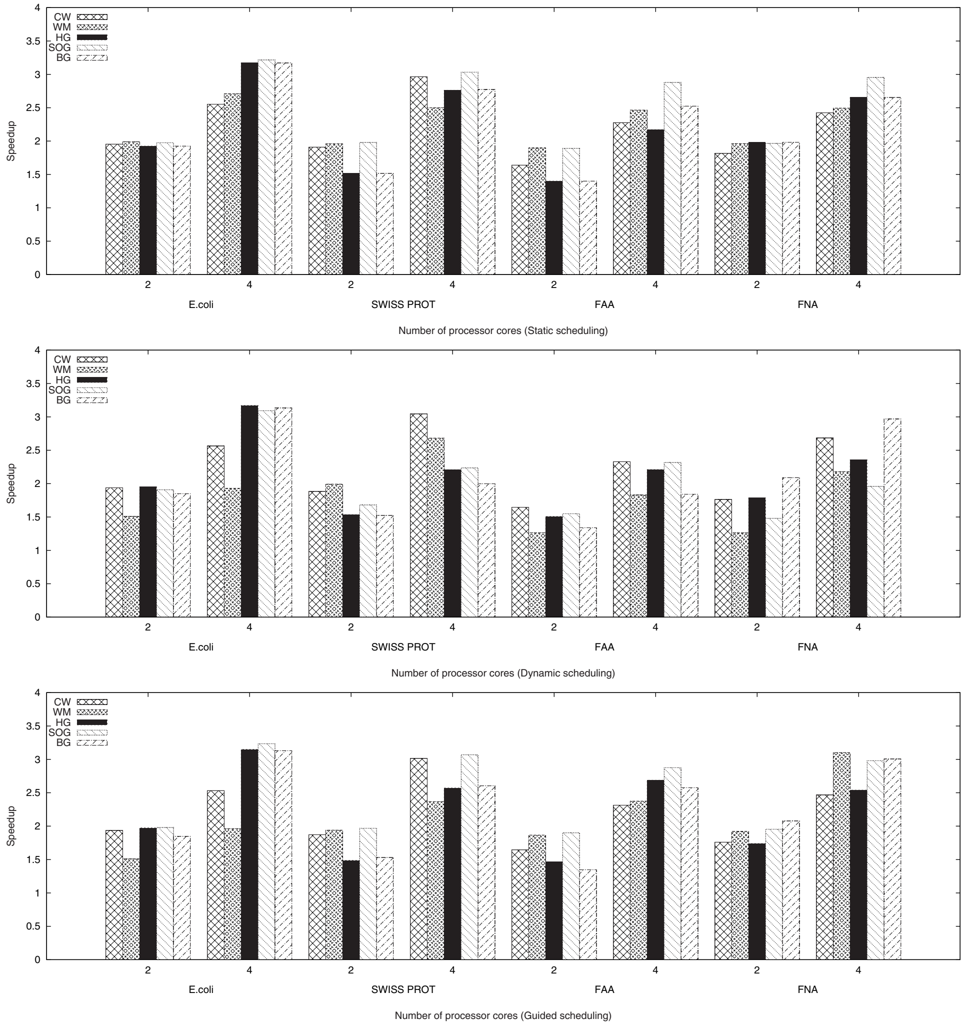

Fig. 2. Speedup of all algorithms for different number of cores and all three scheduling clauses on a Hyper-Threading CPU

a single node plus the total communication time T_c to send the text pointers from the master to the workers and receive back the results.

$$T_{tot} = T_p + T_c \tag{1}$$

As opposed to distributed memory parallelization, shared memory parallelization does not actually involve a distribution of data since all threads have access to a common memory area and thus the total execution time T_{tot} equals to the average processing time T_p per core.

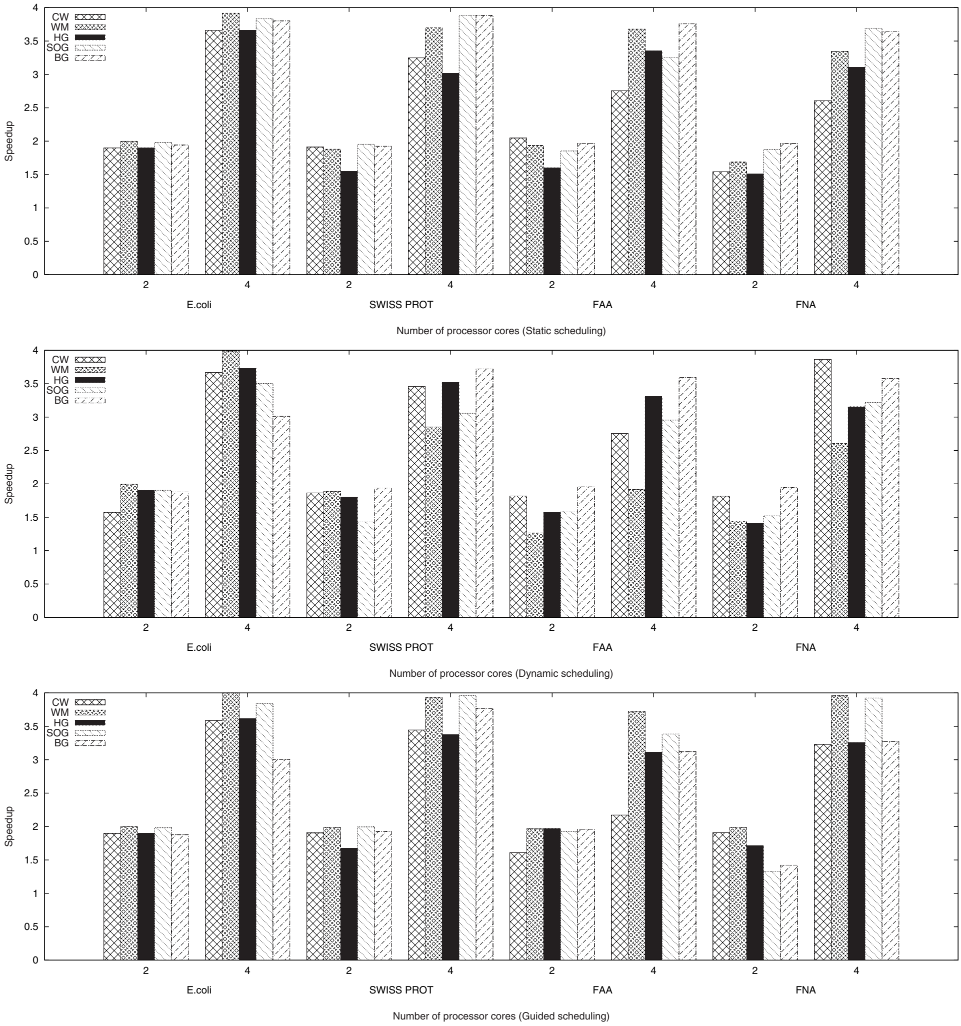

Fig. 3. Speedup of all algorithms for different number of cores and all three scheduling clauses on a Quad Core CPU

Speedup or parallelization rate S_p refers to the running time increase of a parallel algorithm over a corresponding sequential when executed on a cluster of p processing elements.

$$S_p = \frac{T_{seq}}{T_p} \tag{2}$$

An analytical performance prediction model for string matching algorithms when executed in parallel on a cluster of distributed nodes can be found in (Michailidis & Margaritis, 2002).

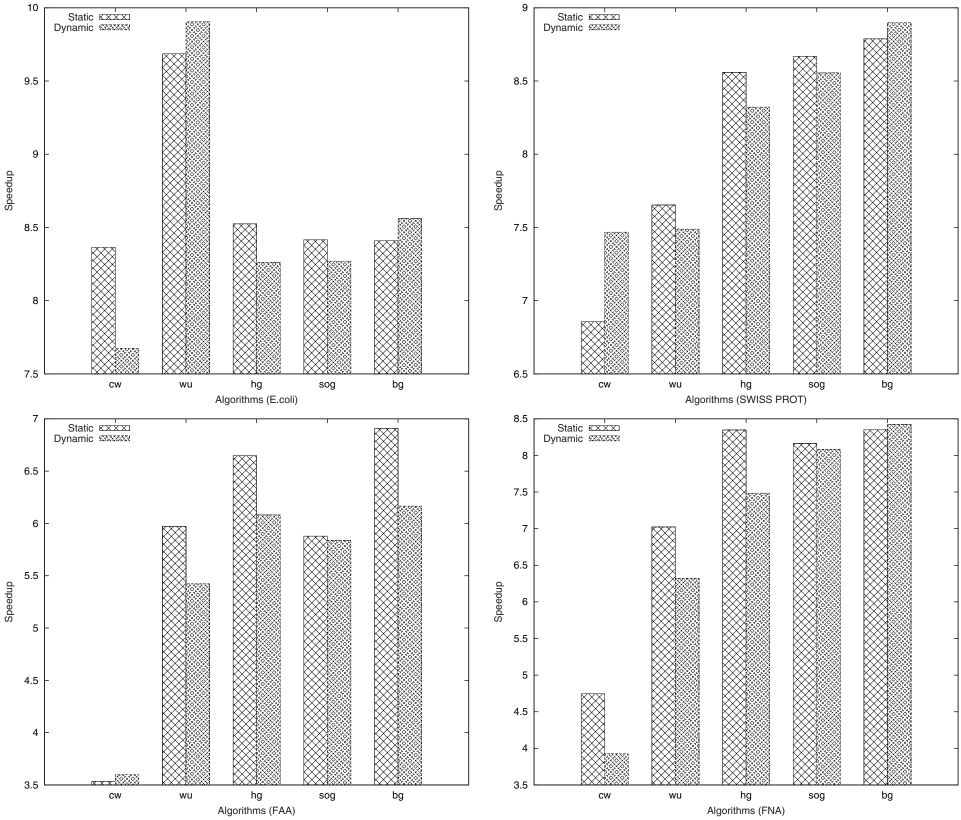

Fig. 4. Comparison of the speedup of the algorithms for a static and dynamic pointer distribution

7.1 Shared memory parallelization

Figure 2 presents the speedup achieved with the use of the OpenMP API when parallel executing the multiple pattern matching algorithms on a single Intel Core I3 processor with 2 and 4 threads as opposed to their execution on a single core. The biological data set used included the genome of Escherichia coli, the SWISS-PROT Amino Acid sequence database and the FASTA Amino Acid (FAA) and FASTA Nucleidic Acid (FNA) sequences of the A-thaliana genome, a pattern set size of 100.000 patterns and a pattern length of $m = 8$. Since the Core I3 processor consists of only two physical and two logical cores with the use of the Hyper-Threading technology, Figure 3 depicts for comparison purposes the speedup of the algorithms on the same data set when executed on an Intel Core 2 Quad processor, that has four physical cores, with 2 and 4 threads.

As illustrated in Figures 2 and 3, the parallel speedup achieved by all algorithms on both processors and for all biological sequence databases using the static clause of OpenMP was similar when two threads were used. In that case, a parallelization rate close to 2 was achieved. With the use of 4 threads though, the Core 2 Quad processor had a significant advantage in terms of performance with a speedup between 3.5 and 4 times the running time of the

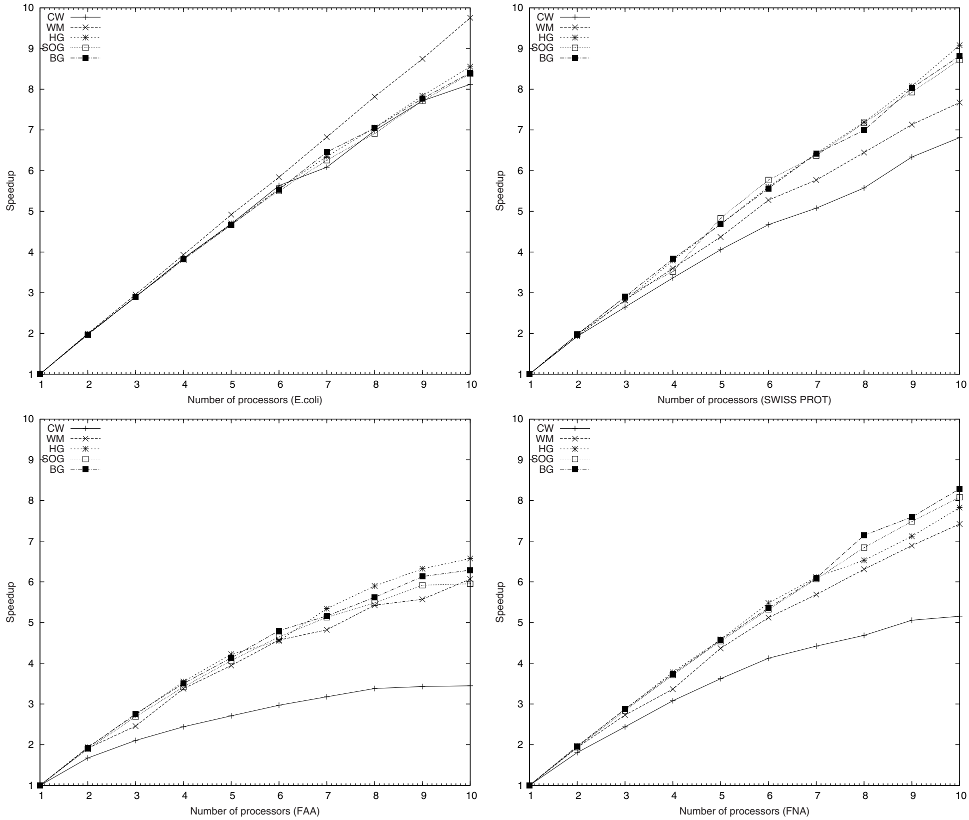

Fig. 5. Speedup of all algorithms with MPI for different number of processors using a static pointer distribution

sequential algorithms on all biological databases as opposed to a speedup between 2.5 and 3.5 of the Core I3 processor. More specifically, when 4 threads were used on the Core I3 processor, the speedup achieved by the Commentz-Walter algorithm was close to 3 for the SWISS-PROT sequence database and between 2.5 and 3 for the E.coli, FAA and FNA databases. Wu-Manber had a parallelization rate of between 2.5 and 3 on all biological databases. Finally, the speedup of the HG, SOG and BG algorithms was close to 3 for the E.coli database and between 2.5 and 3 for the SWISS-PROT, FAA and FNA databases. On the Core 2 Quad processor, the speedup achieved by all algorithms was uniform; their parallelization rate for all biological databases was close to 2 when executed on two processor cores and between 3.5 and 4 when executed on all four physical cores of the CPU.

Since the percentage of hits and misses was generally balanced across the data set and the use of dynamic scheduling usually incurs high overheads and tends to degrade data locality (Ayguade et al., 2003), it was expected that the static scheduling clause of OpenMP would be best suited for the experiments of this chapter. A similar conclusion was drawn in (Kouzinopoulos & Margaritis, 2009) where the static scheduling clause with the default chunk size had a better performance than the dynamic and guided scheduling clauses for two

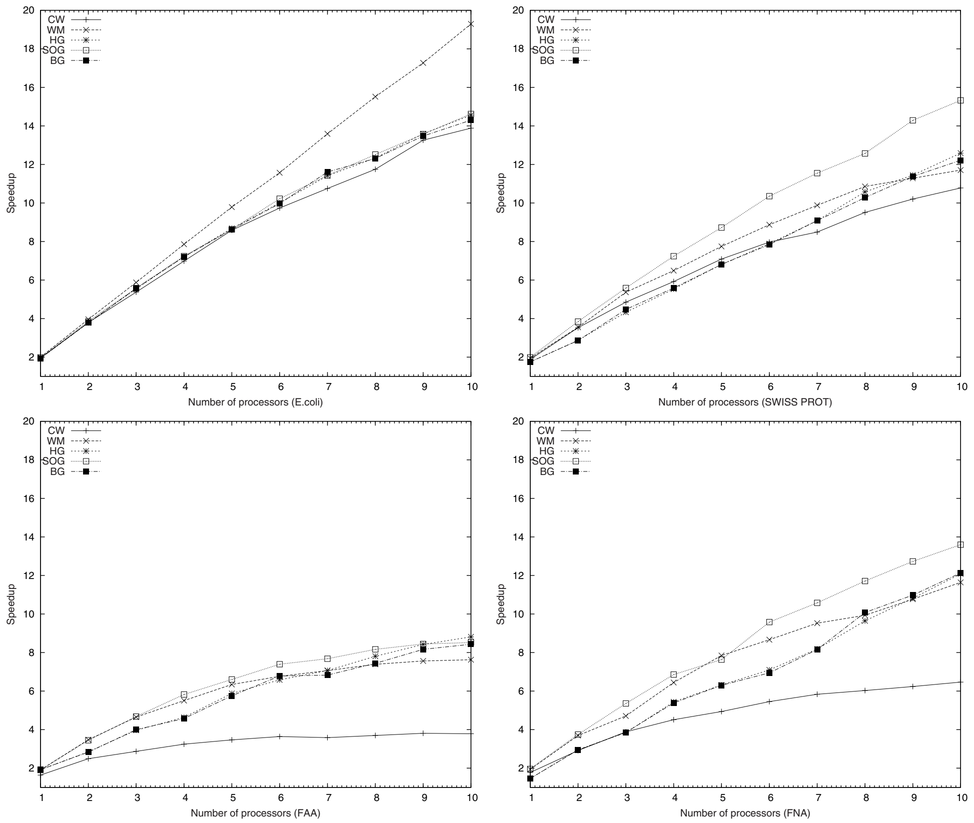

Fig. 6. Speedup of all algorithms with hybrid OpenMP/MPI for different number of processors with two cores

dimensional pattern matching algorithms. The experimental results confirm this expectation since a higher parallelization rate was achieved in most cases when the static scheduling clause was used instead of the dynamic and guided clauses. The performance speedup of the Commentz-Walter algorithm was roughly similar on all biological databases, independent of the scheduling clause used. The speedup of the Wu-Manber algorithm was significantly decreased when a dynamic scheduling clause with 4 threads on the Core I3 processor was used. Finally, the parallelization rate of the Salmela-Tarhio-Kytöjoki algorithms was slightly reduced when the dynamic and guided scheduling clauses were used instead of the static clause.

The performance increase of a parallel task that is executed on a processor with four physical cores over a processor with two physical cores and Hyper-Threading was expected, it is interesting though that the parallel execution of a multiple pattern algorithm using 4 threads on a CPU with two cores and Hyper-Threading exhibits a performance increase of 1.25 to 1.5. This performance boost is achieved by the increase in parallelization that helps hiding a number of hardware and software issues including memory latency, branch misprediction and data dependencies on the instruction stream (Marr et al., 2002) that often leave processor

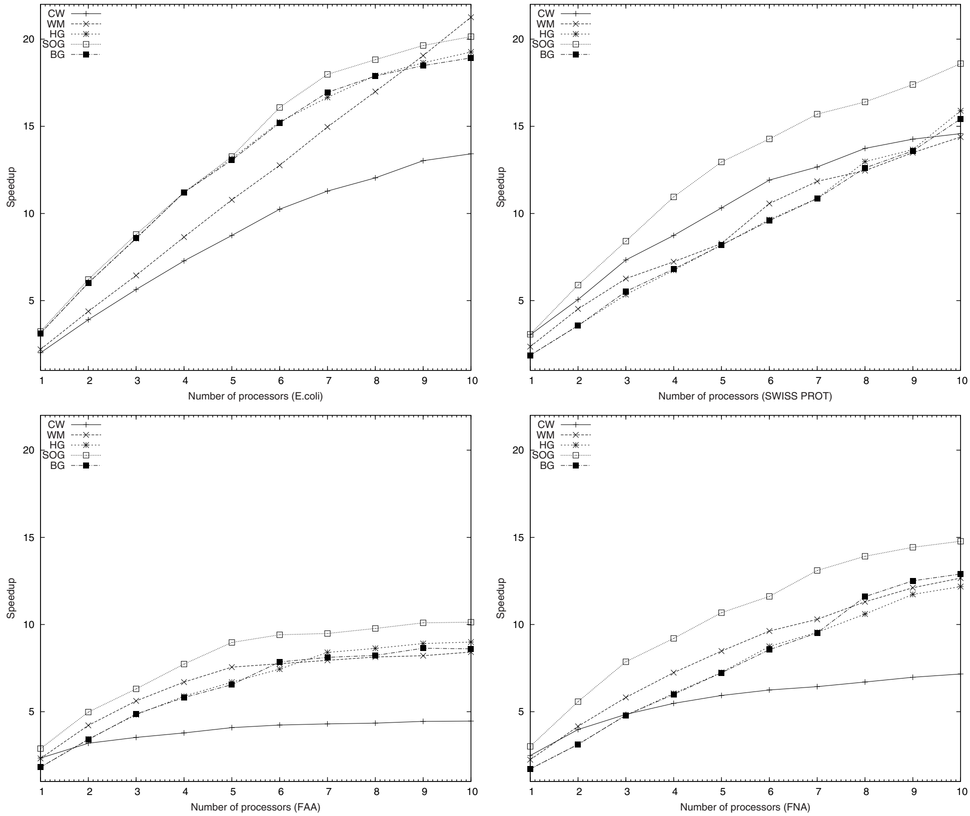

Fig. 7. Speedup of all algorithms with hybrid OpenMP/MPI for different number of processors with four cores

resources unused. Similar performance gains were reported in (Tian et al., 2002) with an average performance speedup of 1.2 to 1.4 on image processing and genetics experiments.

7.2 Distributed memory parallelization

Figure 4 presents a performance comparison in terms of parallel speedup of the Commentz-Walter, Wu-Manber and the Salmela-Tarhio-Kytöjoki multiple pattern matching algorithms using the MPI library for a static and dynamic distribution of pointers from the master to the worker nodes on a homogeneous cluster of 10 nodes. For the static distribution of pointers, each node received a block consisting of $\lceil \frac{n}{p} \rceil + m - 1$ bytes as already discussed. For the dynamic distribution, measurements showed that a block size of $\lceil \frac{n}{1000} \rceil + m - 1$ bytes provided the nodes with enough blocks to equate any load unbalances caused by the distribution of pattern match locations in the biological databases while at the same time keeping low the communication cost. Due to the homogeneity of the cluster nodes and the generally balanced data set used it was expected that the algorithms should have a better performance in terms of the parallelization rate achieved when the static distribution of pointers was used over the dynamic as confirmed by the experimental results.

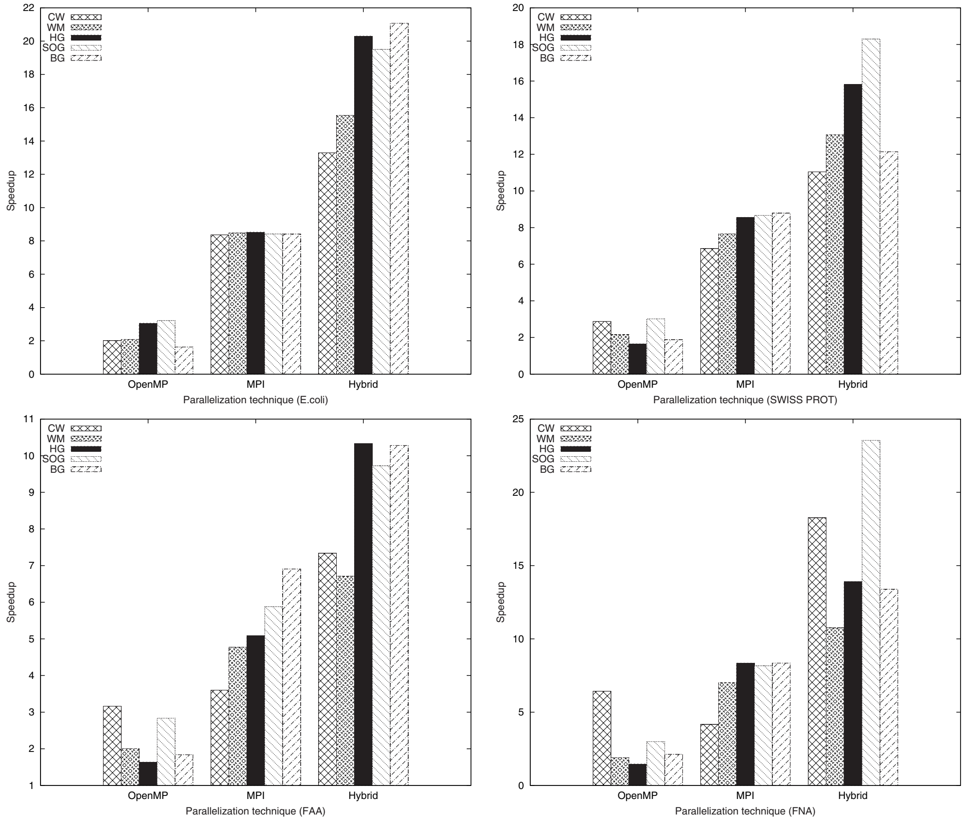

Fig. 8. Speedup achieved with OpenMP, MPI and a hybrid OpenMP/MPI system

More specifically, the Commentz-Walter algorithm had a better performance on the E.coli and the FNA sequence databases with the static distribution of pointers while HG and SOG were consistently faster for all types of biological databases when a static pointer distribution was used. The parallel implementation of the Wu-Manber algorithm also was faster when a static distribution of pointers was chosen instead of the dynamic. Interestingly, the Commentz-Walter algorithm had a better performance on the SWISS-PROT database when a dynamic distribution of pointers was used. The advantage of the dynamic distribution of pointers over the static for Commentz-Walter can be explained by the fact that the sequential implementation of the algorithm is outperformed by the Wu-Manber and the Salmela-Tarhio-Kytöjoki family of multiple pattern matching algorithms when used on data sets with a big alphabet size, including the SWISS-PROT and the FAA databases as detailed in (Kouzinopoulos & Margaritis, 2011). This fact can be also confirmed by the very low parallelization rate that is achieved by the Commentz-Walter algorithm as presented in Figures 4 and 5.

Figure 5 depicts the speedup of the multiple pattern matching algorithms as a factor of the number of worker nodes utilized in the cluster. The static pointer distribution was chosen as it was concluded that it was best suited to the specific biological databases and the cluster topology used. As discussed in (Michailidis & Margaritis, 2003), there is an inverse relation between the parallel execution time and the number of workstations on a distributed memory system, since the total communication time is much lower than the processing time on each node. It is clear from Figure 5 that by distributing a computation task over two worker nodes resulted in approximately doubling its performance. On each subsequent workstation introduced, the performance of the algorithms increased but in most cases with a decreasing rate since the communication cost between the master and the worker nodes also increased. This parallelization rate generally depended on the algorithm and varied with the type of the sequence database used.

For the E.coli sequence database, the speedup of all multiple pattern matching algorithms increased roughly linear in the number of distributed nodes in the cluster. The Wu-Manber algorithm improved its performance by 9.7 times when parallel executed on a cluster with 10 nodes while the Commentz-Walter, HG, SOG and BG algorithms improved their performance by 8.4 times. The parallelization rate of the algorithms was similar for the SWISS-PROT and the FNA sequence databases. For both genomes, the speedup of the algorithms increased with a linear rate; the maximum speedup achieved for SWISS-PROT and FNA was 9 and 8 respectively for the Salmela-Tarhio-Kytöjoki family of algorithms, 7.6 and 7.4 for the Wu-Manber algorithm and 6.8 and 5.1 for the Commentz-Walter algorithm. It is interesting that for the FAA sequence database, the parallelization of all algorithms increased with a logarithmic rate in the number of worker nodes with the HG, SOG and BG achieving a maximum speedup of 6.5 times, Wu-Manber a speedup of 6 times and Commentz-Walter reaching a speedup of 3.4 times. Based on the experimental findings it can be concluded in general that the hashing multiple pattern matching algorithms have a better parallelization rate than the trie-based Commentz-Walter algorithm on all biological databases.

7.3 Hybrid parallelization

Figures 6 and 7 illustrate the performance increase of the Commentz-Walter, Wu-Manber and the Salmela-Tarhio-Kytöjoki family of multiple pattern matching algorithms using the proposed hybrid OpenMP/MPI technique on a homogeneous cluster of 10 nodes with a Core I3 processor on each node for 2 and 4 threads. To distribute the data across the worker nodes of the cluster and subsequently across the cores of each processor, the static distribution of pointers was chosen for MPI and the static scheduling clause was used for OpenMP as were the best suited options for the specific algorithms, biological databases and cluster topology used. As can be seen in both Figures, the parallelization rate achieved by the algorithms was significant better when using the hybrid OpenMP/MPI technique instead of either shared memory or distributed memory parallelization. Additionally it can be seen that the type of sequence database that is used can greatly affect the performance of the parallel implementation of the algorithms.

When the two physical cores of the Core I3 processor where used and for the E.coli sequence database, the parallelization rate of the algorithms increased linear in the number of cluster nodes. The Wu-Manber algorithm was up to 19.2 times faster than its sequential implementation while the Commentz-Walter, HG, SOG and BG algorithms had on average a 14.5 times better performance. As with the distributed memory parallelization, the speedup of the multiple pattern matching algorithms was similar for the SWISS-PROT and the FNA sequence databases; the speedup of the SOG algorithm was 15.3 and 13.5 respectively, of the HG, BG and Wu-Manber algorithms was 12 on average while the maximum parallelization rate of the Commentz-Walter algorithm was 10.7 and 6.4. Finally for the FAA genome, the Wu-Manber and the Salmela-Tarhio-Kytöjoki family of multiple pattern matching algorithms had a similar speedup of 8.4 on average while Commentz-Walter had a parallelization rate of 3.7. When all four processing cores were utilized per processor (two physical and two logical) with the use of the Hyper-Threading technique, the performance of all algorithms increased by an additional 1.2 to 1.3 times on all biological sequence databases as can be seen on Figure 7.

Figure 8 presents a comparison of the speedup achieved by the Commentz-Walter, Wu-Manber and the Salmela-Tarhio-Kytöjoki family of multiple pattern matching algorithms using the three presented parallelization techniques; shared memory with 4 threads per node, distributed memory with 10 homogeneous cluster nodes and a hybrid technique that combines the advantages of both shared and distributed memory parallelization. As can be seen by Figure 8, the proposed hybrid implementation of the algorithms was roughly 5 to 8 times faster than the shared memory implementation using two processor cores and 2 to 3 times faster than the distributed memory parallelization for all sequence databases.

8. Conclusions

This chapter presented implementations and experimental results of the Commentz-Walter, Wu-Manber and the Salmela-Tarhio-Kytöjoki family of multiple pattern matching algorithms when executed in parallel. The algorithms were used to locate all the appearances of any pattern from a finite pattern set on four biological databases; the genome of Escherichia coli from the Large Canterbury Corpus, the SWISS-PROT Amino Acid sequence database and the FASTA Amino Acid (FAA) and FASTA Nucleidic Acid (FNA) sequences of the A-thaliana genome. The pattern set used consisted of 100.000 patterns where each pattern had a length of $m = 8$ characters.

To compare the speedup achieved on multicore processors, the parallel algorithms were implemented using shared memory parallelization on processors with two and four physical cores using 2 and 4 threads and with either the static, dynamic or guided scheduling clause for the data distribution between each processor core. For the performance evaluation of the parallel implementations on a homogeneous cluster of 10 worker nodes, the algorithms were implemented using distributed memory parallelization with a static or dynamic pointer distribution. Finally a hybrid OpenMP/MPI technique was proposed that combined the advantages of both shared and distributed parallelization.

It was discussed that the parallel execution of multiple pattern matching algorithms on a homogeneous cluster with multicore processors using the specific types of biological databases is more efficient when the static scheduling clause and a static pointer allocation are used for the OpenMP and MPI APIs respectively. Moreover it was concluded in general that the hashing multiple pattern matching algorithms have a better parallelization rate than the trie-based Commentz-Walter algorithm on all biological databases. Finally, it was shown that the proposed hybrid implementation of the algorithms was roughly 5 to 8 times faster than the shared memory implementation using two processor cores and 2 to 3 times faster than the distributed memory parallelization for all sequence databases.

The work presented in this chapter could be extended with experiments that use additional parameters like patterns of varying length and larger pattern sets. Since biological databases and sets of multiple patterns are usually inherently parallel in nature, future research could focus on the performance evaluation of the presented algorithms when parallel processed on modern parallel architectures such as Graphics Processor Units.

9. References

Aho, A. & Corasick, M. (1975). Efficient string matching: an aid to bibliographic search, *Communications of the ACM* 18(6): 333–340.

Allauzen, C., Crochemore, M. & Raffinot, M. (1999). Factor oracle: A new structure for pattern matching, 1725: 758–758.

Ayguade, E., Blainey, B., Duran, A., Labarta, J., Martinez, F., Martorell, X. & Silvera, R. (2003). Is the schedule clause really necessary in openmp?, *International workshop on OpenMP applications and tools*, Vol. 2716, pp. 147–159.

Baeza-Yates, R. & Gonnet, G. (1992). A new approach to text searching, *Communications of the ACM* 35(10): 74–82.

Bailey, A. (2006). Openmp: Getting fancy with implicit parallelism, Website. URL: *http://developer.amd.com/documentation/articles/Pages/ 1121200682.aspx*.

Boeva, V., Clement, J., Regnier, M., Roytberg, M. & V.J., M. (2007). Exact p-value calculation for heterotypic clusters of regulatory motifs and its application in computational annotation of cis-regulatory modules, *Algorithms for Molecular Biololoy* 2(1): 13.

Brudno, M. & Morgenstern, B. (2002). Fast and sensitive alignment of large genomic sequences, *IEEE Computer Society Bioinformatics Conference*, Vol. 1, pp. 138–147.

Brudno, M., Steinkamp, R. & Morgenstern, B. (2004). The CHAOS/DIALIGN WWW server for multiple alignment of genomic sequences, *Nucleic Acids Research* 32: 41–44.

Buhler, J., Keich, U. & Y., S. (2005). Designing seeds for similarity search in genomic dna, *Journal of Computer and System Sciences* 70(3): 342–363.

Buyya, R. (1999). *High Performance Cluster Computing: Programming and Applications*, Vol. 2, Prentice Hall.

Commentz-Walter, B. (1979). A string matching algorithm fast on the average, *Proceedings of the 6th Colloquium, on Automata, Languages and Programming* pp. 118–132.

Cringean, J. K., Manson, G. A., Wilett, P. & G.A., W. (1988). Efficiency of text scanning in bibliographic databases using microprocessor-based, multiprocessor networks, *Journal of Information Science* 14(6): 335–345.

Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W. & Rytter, W. (1994). Speeding up two string-matching algorithms, *Algorithmica* 12(4): 247–267.

Crochemore, M., Czumaj, A., Gasieniec, L., Lecroq, T., Plandowski, W. & Rytter, W. (1999). Fast practical multi-pattern matching, *Information Processing Letters* 71(3-4): 107 – 113.

Crochemore, M. & Rytter, W. (1994). *Text algorithms*, Oxford University Press, Inc., New York, NY, USA.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. & V., S. (1994). *PVM: Parallel Virtual Machine, A Users Guide and Tutorial for Networked Parallel Computing*, The MIT Press.

Grama, A., Karypis, G., Kumar, V. & Gupta, A. (2003). *Introduction to Parallel Computing*, Addison Wesley.

Horspool, R. (1980). Practical fast searching in strings, *Software: Practice and Experience* 10(6): 501–506.

Hyyro, H., Juhola, M. & Vihinen, M. (2005). On exact string matching of unique oligonucleotides, *Computers in Biology and Medicine* 35(2): 173–181.

Jaffe, J., Berg, H. & G.M., C. (2004). Proteogenomic mapping as a complementary method to perform genome annotation, *Proteomics* 4(1): 59–77.

Karp, R. & Rabin, M. (1987). Efficient randomized pattern-matching algorithms, *IBM Journal of Research and Development* 31(2): 249–260.

Kim, S. & Kim, Y. (1999). A fast multiple string-pattern matching algorithm, *Proceedings of the 17th AoM/IAoM Inernational Conference on Computer Science* pp. 1–6.

Kouzinopoulos, C. & Margaritis, K. (2009). Parallel Implementation of Exact Two Dimensional Pattern Matching Algorithms using MPI and OpenMP, *Hellenic European Research on Computer Mathematics and its Applications*.

Kouzinopoulos, C. & Margaritis, K. (2010). Experimental Results on Algorithms for Multiple Keyword Matching, *IADIS International Conference on Informatics*.

Kouzinopoulos, C. S. & Margaritis, K. G. (2011). Algorithms for multiple keyword matching: Survey and experimental results. Technical report.

Leow, Y., Ng, C. & W.F., W. (2006). Generating hardware from OpenMP programs, *Proceedings of IEEE International Conference on Field Programmable Technology*, pp. 73–80.

Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A. & Upton, M. (2002). Hyper-Threading Technology Architecture and Microarchitecture, *Intel Technology Journal* 6(1): 4–15.

Michael, M., Dieterich, C. & Vingron, M. (2005). Siteblast–rapid and sensitive local alignment of genomic sequences employing motif anchors, *Bioinformatics* 21(9): 2093–2094.

Michailidis, P. & Margaritis, K. (2002). Parallel implementations for string matching problem on a cluster of distributed workstations, *Neural, Parallel and Scientific Computations* 10(3): 312.

Michailidis, P. & Margaritis, K. (2003). Performance evaluation of load balancing strategies for approximate string matching application on an MPI cluster of heterogeneous workstations, *Future Generation Computer Systems* 19(7): 1075–1104.

Muth, R. & Manber, U. (1996). Approximate multiple string search, *Combinatorial Pattern Matching*, Springer, pp. 75–86.

Navarro, G. & Raffinot, M. (1998). A bit-parallel approach to suffix automata: Fast extended string matching, *Lecture Notes in Computer Science* 1448: 14–33.

Navarro, G. & Raffinot, M. (2002). *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*, Cambridge University Press.

Nichols, B., Buttlar, D. & Farrell, J. (1996). *Pthreads Programming*, OReilly.

Salmela, L. (2009). *Improved Algorithms for String Searching Problems*, PhD thesis, Helsinki University of Technology.

Salmela, L., Tarhio, J. & Kytöjoki, J. (2006). Multipattern string matching with q -grams, *Journal of Experimental Algorithmics* 11: 1–19.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D. & J., D. (1996). *MPI: The complete reference*, The MIT Press.

Tian, X., Bik, A., Girkar, M., Grey, P., Saito, H. & Su, E. (2002). Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance, *Intel Technology Journal* 6(1): 36–46.

Wu, S. & Manber, U. (1994). A fast algorithm for multi-pattern searching, pp. 1–11. Technical report TR-94-17.

Zhou, Z., Xue, Y., Liu, J., Zhang, W. & Li, J. (2007). Mdh: A high speed multi-phase dynamic hash string matching algorithm for large-scale pattern set, 4861: 201–215.

Systems and Computational Biology - Bioinformatics and Computational Modeling
Edited by Prof. Ning-Sun Yang

ISBN 978-953-307-875-5
Hard cover, 334 pages
Publisher InTech
Published online 12, September, 2011
Published in print edition September, 2011

Whereas some â€œmicroarrayâ€ or â€œbioinformaticsâ€ scientists among us may have been criticized as doing â€œcataloging researchâ€, the majority of us believe that we are sincerely exploring new scientific and technological systems to benefit human health, human food and animal feed production, and environmental protections. Indeed, we are humbled by the complexity, extent and beauty of cross-talks in various biological systems; on the other hand, we are becoming more educated and are able to start addressing honestly and skillfully the various important issues concerning translational medicine, global agriculture, and the environment. The two volumes of this book present a series of high-quality research or review articles in a timely fashion to this emerging research field of our scientific community.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Charalampos S. Kouzinopoulos, Panagiotis D. Michailidis and Konstantinos G. Margaritis (2011). Parallel Processing of Multiple Pattern Matching Algorithms for Biological Sequences: Methods and Performance Results, Systems and Computational Biology - Bioinformatics and Computational Modeling, Prof. Ning-Sun Yang (Ed.), ISBN: 978-953-307-875-5, InTech, Available from: http://www.intechopen.com/books/systems-and-computational-biology-bioinformatics-and-computational-modeling/parallel-processing-of-multiple-pattern-matching-algorithms-for-biological-sequences-methods-and-per

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821