CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

MPI-CMS: a hybrid parallel approach to geometrical motif search in proteins

Marco Ferretti[1], Mirto Musci[1*] and Luigi Santangelo[1]

[1] *University of Pavia, Department of Computer Engineering, Via Ferrata 5, 27100 Pavia, Italy*

SUMMARY

This paper describes MPI-CMS, a parallel implementation of the Cross Motif Search (CMS) algorithm. It is an extension of a series of previous papers [1, 2, 3] and specifically improves on the results obtained in [4].
CMS is a bioinformatics algorithm, based on the SSC algorithm [5, 6], whose goal is to search for geometrical motifs in proteins. Indeed, the retrieval and identification of motifs is an important and open problem in bioinformatics [7].
CMS is able to compare any two proteins in an arbitrary set, and identify the precise location of each structural similarity between the pair. For the purpose of a complete characterization of protein similarities, it would be important to run CMS on the largest possible dataset, up in theory to the entire Protein Data Bank (PDB) [8]. Unfortunately, due to its precision, CMS is inherently slow; thus it was originally implemented using a shared memory parallel paradigm. A complete implementation of CMS using OpenMP (MP-CMS) was discussed in [1].
In [4] we proved that the MP-CMS, while it can indeed improve the performance of CMS, is extremely inefficient and cannot scale adequately neither on large datasets nor on large machines. To solve the problem, we designed a new parallel implementation of CMS based on a hybrid shared memory and message passing paradigm, and implemented using both OpenMP and OpenMPI. This implementation is called MPI-CMS or Hybrid-CMS. In this extended paper we will discuss the design and implementation of MPI-CMS in far more details as before.
This paper reconsiders MPI-CMS with the target to port it on a supercomputing machine (likely the Fermi IBM Blue Gene/Q at the Cineca facility in Bologna, Italy) to address the whole database of currently known secondary structures of all proteins. The focus is on the dependence of performance in the hybrid approach on the workload unbalance; the effect of this unbalance is traced to the idle time of some OpenMP threads generated for comparing couple of proteins that have modest, if not minimal similarities in their secondary structure. Using a simple statistical analysis of the workload we discuss several strategies through which we can improve the design of MPI-CMS. We conclude the paper proposing a revised implementation of MPI-CMS which takes into account the size of the protein pairs to fine-tune the parallelization strategy and hopefully increase the overall performance of the hybrid algorithm. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: geometrical motif search, hybrid parallelism, workload characterization

*Correspondence to: mirto.musci01@ateneopv.it

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls *[Version: 2010/05/13 v3.00]*

1. INTRODUCTION

The retrieval and identification of protein motifs is an important and open problem in bioinformatics [7]. Put in very simple terms, motifs are recurring patterns of structural elements of a protein. Motifs are defined at several levels of the protein structure: they can be simple strings of amino-acids or complex 3D structures. For the purpose of this paper, we are mostly interested in motifs of Secondary Structure Elements (SSE).

Cross Motif Search (CMS) is a bioinformatics algorithm based on Secondary Structures Co-Occurrences [5, 6], and on the General Hough Transform [9], which is able to perform an exhaustive search for every common secondary structure motif in a given pair of proteins. CMS has been thoroughly described in [1, 3, 2]. The natural extension of CMS is Complete CMS (CCMS), where the same process is applied to a set of proteins, instead of just to a protein pair.

The key idea underlying CMS is the innovative focus that it puts on the geometrical description of the structural motifs, which could be simply viewed as line segments, rather than on the topological/biological description employed by competing algorithms such as ProSMoS [10, 11], PROMOTIF [12] or MASS [13]. With respect to other geometrical algorithms, such as SSM [14], CMS is way more precise, as SSM only gives a similarity score for the geometrical structures of pair of proteins. From a higher point-of view, the main goal of CCMS is to look for previously unknown common geometrical structures in so-called “unfamiliar” proteins.

The main shortcoming of CMS with respect to the state of the art is performance, as precise geometrical approaches are inherently slower. Thus, efficient parallel implementations become extremely important. At first approximation, the computational complexity of a Complete CMS run is quadratic in the number of proteins in the data-set, multiplied by the execution time of a single run. However, due to the internal heuristics of CMS, the complexity of each run is very hard to determine, as it depends in a non-trivial way on the similarity score between each pair of proteins, on the tolerance used in the geometrical description, and in the number of SSEs of both proteins.

In previous works [1, 2, 3] we described an OpenMP implementation of the CMS algorithm, and evaluated its performance. The key idea of the OpenMP parallelization is to parallelize the inner kernel of a single CMS run, so that the search for the geometrical motifs in a single pair can be distributed on multiple threads. We measured the execution times of a Complete CMS between the 200 proteins in the dataset, making the number of cores vary from a minimum of 2, to the available maximum of 32 (Fig. 1.1). The figure clearly shows that the OpenMP implementation of the CMS algorithm is able to scale up to only 8 cores. Moreover, the efficiency of the parallel implementation is extremely poor, as it is intrinsically limited by the structure of the algorithm [1].

Due to this inefficiency, as the number of proteins in the set increases, the execution time of CMS quickly becomes unmanageable and clearly the OpenMP implementation cannot keep up just by increasing the number of cores. The situation is particularly critic if one wants to analyze a huge data-set, or even the entirety of the Protein Data Bank (PDB) [8]. Thus, if one wants to improve the performance of CCMS, the OpenMP approach has to be abandoned, or at least complemented in some way. The most natural path is to try to implement a hybrid message-passing and shared-memory version of the CMS algorithm.

In [4] we showed how this approach, called MPI-CMS, or Hybrid-CMS, can be designed and fine-tuned for optimal load balancing. This extended version presents a more detailed and in-depth

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

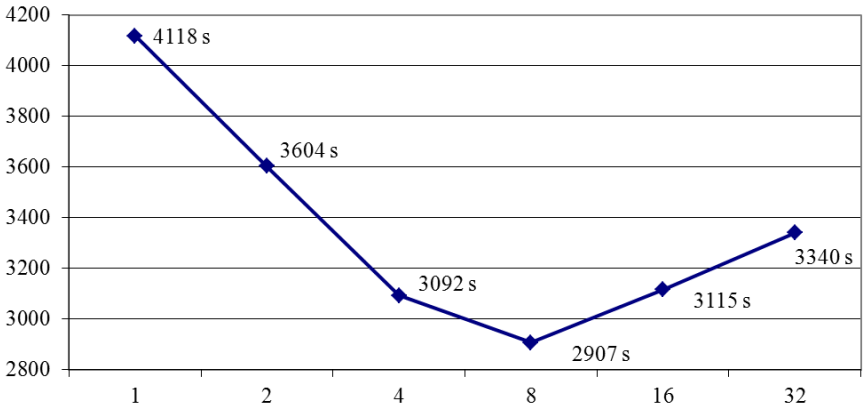

Figure 1.1. The performance of a pure OpenMP implementation of CMS. On the x-axis the number of cores, on the y-axis the execution time in seconds. *Source*: [4]

analysis of the design and the performance of the MPI-CMS algorithm. In particular it provides a few previously undisclosed implementation details. Most important, this extended version presents a simple characterization of the workload of the algorithm using statistical analysis and the proposal of a revision of MPI-CMS which takes into account the peculiarities of the domain to improve the performance.

The paper is organized as follows: Sec. 2 and 3 respectively describes the design and implementation of MPI-CMS. Then, Sec. 4 presents some experimental results which elaborates from the ones presented in [4] and provides a new point-of-view on the performance of the MPI-CMS algorithm. Sec. 5 describes a revised implementation of the algorithm, which is based on a new design which exploit simple workload characterization to improve the performance. Sec. 6 concludes the paper and discusses future research work.

2. DESIGN

MPI-CMS is a hybrid shared-memory and message-passing parallel algorithm. MPI-CMS decomposes the whole CCMS problem into x MPI processes,each consisting of a single MP-CMS with y threads. In what follows, the hybrid decomposition will be referred to with the $x : y$ notation.

The computations related to a single run of CMS are still done with a shared memory paradigm, while the x multiple MP-CMS instances run on different subsets of computational units; the distribution of the workload (proteins to be processed) among them uses a message passing approach.

In designing the approach, the strategy to subdivide the computational units between MPI processes and OpenMP threads has been discussed [4]. Fig. 1.1 provides some hints, since it shows the intrinsic limit to the number of cores that can be profitably used in a single OpenMP run; it is clear that the optimal decomposition may be different on different systems. In Sec. 4 we will show how the execution times of Hybrid-CMS vary with different decompositions on a reference machine.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Logically, MPI-CMS is composed of two main parts: the *master*, a process dedicated to prepare, manage and distribute the computational load; and a series of *slaves* which simply receive data from the master (the protein pairs), and then perform the actual processing.

The main concern with the design of MPI-CMS has been to obtain the best possible load-balancing. The reason is due to the nature of the load itself. The exhaustive search for similarities for two protein pairs can require very different amounts of time; in the worst case the difference is between a few milliseconds and a few minutes (4-5 orders of magnitude!). Worse: there is very little that we can do to characterize the load a priori. Sec. 5.1, discusses the peculiarity of the computational load in much more details and how its features can be exploited to improved the load-balancing performance.

According to the Casavant and Kuhl taxonomy [15], it can be stated that the load-balancer of MPI-CMS is a *dynamic*, *synchronized*, *demand-driven*, *centralized* and *one-time assignment* algorithm. Moreover, it is clear that MPI-CMS is a *high granularity* algorithm. In fact, as it will become clear later, long periods of computation are separated by a very small amount of communication between the master and the slaves. In other words, if (in the best possible scenario) the average execution time for a pair of proteins is in the order of 10 milliseconds, the short messages required to manage said computation are sent and received in the span of a few microseconds. Load-balancing is *centralized*, as it is entirely performed by the master process, identified by the MPI ID zero.

Fig. 2.1 shows the communication diagram of MPI-CMS and is helpful to understand the basic behavior of the load-balancer.

At the startup, the master process retrieves (generally from the filesystem) the list of all XML files containing the description of the proteins of interest in the current CCMS run. In order for both the master and the slaves to access the files, the entire dataset is maintained in a shared partition, which must be accessible, at least in read-only mode, by all running processes.

At this point, the master generates all possible pairs, without repetition, of proteins in the set, and creates and maintains an ordered list of each of these pairs. Each element in the queue is a string containing just the filename of each protein of interest. The current pair of proteins to processes, respectively `protein_source` and `protein_search`, are determined just before they are sent to the slaves. It is therefore up to the master to keep track of the protein pairs that have already been sent for processing. In order to minimize the amount of data kept in the RAM and also to improve the master multithreading performance (Sec. 3.3), the master keeps only a single integer index, called `next_element`, through which it can derive the identifiers i and j of the two proteins in the queue that will form the pair to be sent to the slave for processing.

At run-time, the master distributes one or more protein pairs to each slave process, updating the list accordingly. The number of pairs of proteins sent is the smallest unit of computation (the *chunksize*) and can be specified in a configuration file or with an environment variable.

At the start of the computation, each slave performs its initializations and sends a ready message to the master, with which it communicates that it is ready to process the next block of protein pairs (the *chunk*). In reply to a ready message, the master determines the identifiers of all proteins in the pairs that must be sent to the slave, and then creates a block of protein pairs whose size is defined by the distribution policy. Finally, the master send this block of protein pairs to the slave applicant. This model is referred to as *demand-driven*.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

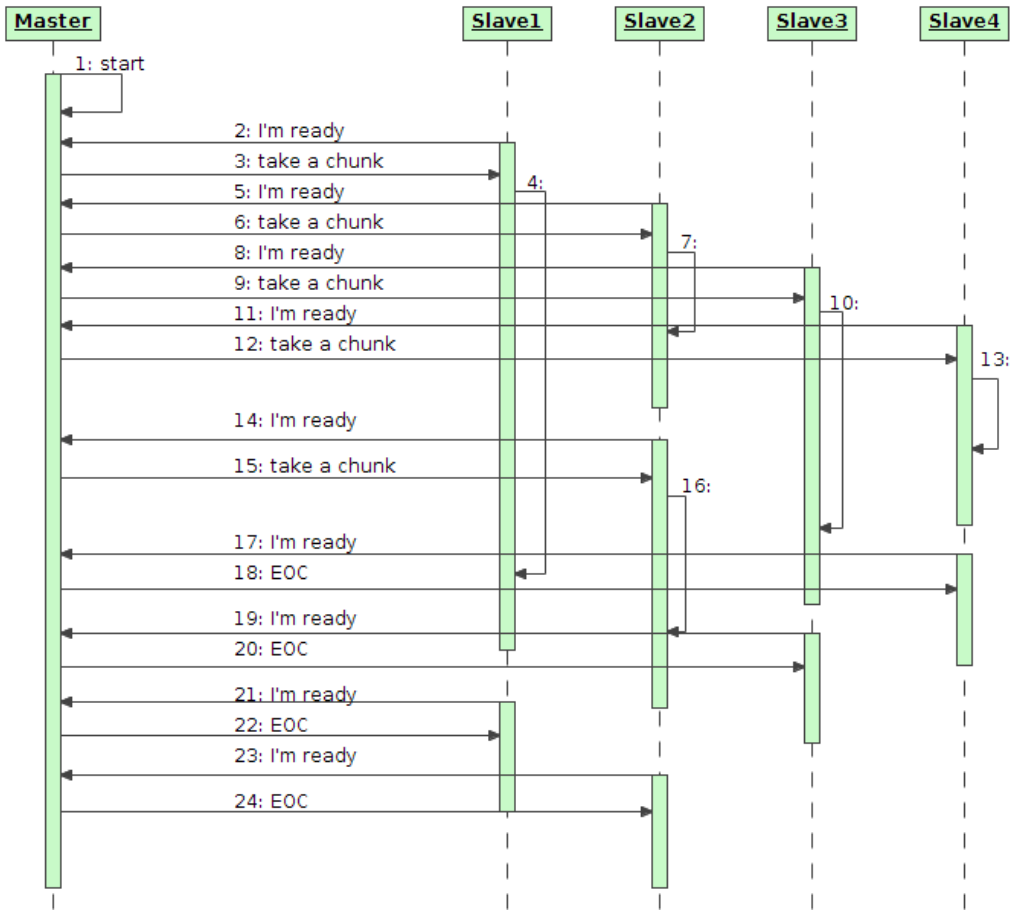

Figure 2.1. UML interaction diagram between the master and the slave processes of MPI-CMS

When a message containing a chunk is received, the slave performs a CMS run on the pairs of proteins listed in the message. At the end of the computation, the slave sends a new request to the master, thus calling for a new computational cycle.

During the entire process, master and slave keep their own log file in order to enable the analysis of the performance and activities of each process through a custom log analyzer tool. In order to avoid concurrent accesses to the log file, each process has its own log file that is uniquely identified with the rank of the process.

When the master process has distributed every protein pair in the list, it broadcasts every slave an EOC (end of computation) message. The slaves conclude their current computation chunk, receive the message, write their log file (for both bookkeeping and performance analysis) and return, without needing an acknowledgment by the master, to avoid potential stalls in case one of the slaves (or the master) crash. At the same time, the master also concludes its execution. This distribution process is clearly dynamic, as it is possible for different slaves to receive and process a different number of chunks, and the distribution is non-deterministic; i.e. it can be different for every run of MPI-CMS.

The process through which the work is distributed among the slaves is clearly *dynamic*: different slaves receive and process a different number of chunks, according to their dynamic performance. Slaves with greater computational capacity, if any, or slaves which are receiving “simple” protein

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

pairs (and therefore are completing CMS runs in a few tenths of a second rather than in several second or in a few minutes), will probably get a number of chunks higher than other slaves. The distribution of the chunk of protein is also *non-deterministic*: different runs of MPI-CMS on the same data-set and on the same machine, may cause a different data distribution.

The communication in MPI-CMS is *point-to-point only*, as only two parties are involved in any communication. In fact, in the current version of the algorithm, the only possible communication is between the master and a slave, as there is no need for slaves to communicate with each other. Moreover, the slave-master communication is *synchronous*, as the master cannot send a different computation chunk, until one (and only one) slave has accepted the current chunk. With the exception of the distribution messages, MPI-CMS does not require any other synchronization point. This is another hint that points to a very low communication overhead.

With the current design, a slave who is assigned a chunk of proteins cannot transfer part (or the whole) of it to another slave process. This means the MPI-CMS is a *one-time-assignment* algorithm. Variable assignment could be useful to increase the balance of a message passing application. However, it was not implemented, as the advantage in the CMS case would be negligible if one keeps the chunksize relatively low. The matter on hand is further discussed in Sec. 4.

2.1. Chunksize

An important parameter in the design of MPI-CMS is the *chunksize*, that is the number of proteins pairs in the block sent to a slave in the context of a single send operation. The usage of a rather large chunksize reduces the total number of send and receive operations, thus reducing the overall communication overhead. On the other hand, since the load-balancing policy does not allows the slaves to redistribute part of their workload, there is a very real risk that, when all the other slaves have completed their execution and all possible pairs of proteins have been distributed, there will be a slave still processing its last data chunk. The bigger the chunks, the longer it will take for the last slave to process the last of them. This results in a increase of the overall execution time and a consequent reduction in efficiency. The optimal value for the chunksize is machine dependent and can only be determined experimentally.

3. IMPLEMENTATION

The MPI-CMS algorithm has been implemented in C++. Fig. 3.1 shows all the logical modules of the application and their mutual relationship. MPI-CMS interacts with several components of the MP implementation of CMS and entrusts them with the actual processing of a single protein pair and the outputting of an XML file containing the result of the computation.

The two main components are referred to as *Master* and *Slave*. The Master implements the load balancing approach discussed in the previous section. It consists of two sub-components, that are generally executed on the same computing node: the first, called *Primary Master*, is responsible to the distribution of protein blocks to the slaves; the second, called *Secondary Master*, instead performs some useful work in order to maximize the performance of the algorithm (Sec. 3.3).

The *Slave* module is responsible of the processing of all proteins pairs received from the *PrimaryMaster*. The communication between *PrimaryMaster* and *SecondaryMaster* takes place via

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

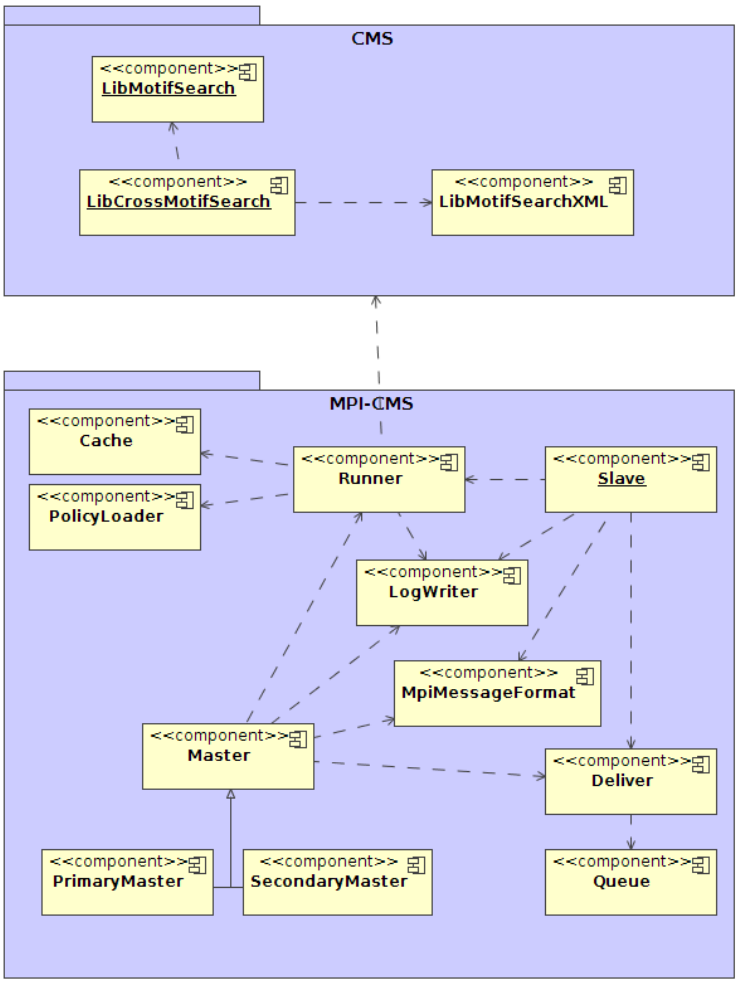

Figure 3.1. Component diagram of all of the software modules of the MPI-CMS implementation.

shared memory (OpenMP), while the communication between Primary Master and Slave occurs through messages (MPI). The structure of the MPI messages exchanged between the parties is defined inside *MpiMessageFormat* (Sec. 3.1). Both Secondary Master and Slave use the *Runner* component as an interface to CMS. In order to optimize memory accesses, the *Runner* makes use of a local cache (Sec. 3.2) for the temporary storage of proteins. The cache is implemented by the *ProteinCache* module.

In what follows, some of the optimizations introduced in the code will be discussed. In particular, the next few sections describe the format of the messages used inside MPI-CMS, the introduction of a protein cache to reduce the memory accesses and the multithread nature of the *Master* component.

3.1. Message Structure

In order to minimize the overhead introduced by the message passing communication model, instead of using predefined types made available from MPI, a new custom data type has been defined. The new data type is the interface through which MPI transfers a block of protein pairs having the size specified by the distribution policy. In this way, a single send message may transfer up to b pairs

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

of proteins. However, packing the entire protein structure into the message would certainly increase the complexity of the algorithm and the transmission time of the message, due the excessive size of the package. The choice, therefore, was to insert in the datatype only a string corresponding to the filename of the proteins to process. Upon receiving the message, the slave is responsible to retrieve all complete proteins description from the file system. Obviously, this design implies the presence of a shared disk partition accessible by all processes (master and slave) containing the entire dataset of proteins of interest.

3.2. Protein cache

The total number of CMS runs in a CCMS run is quadratic with respect to the size of the dataset [1]. That is, if n is the number of proteins present in the dataset, the number of comparisons between all possible pairs of proteins is equal to

$$\binom{n}{2} = \frac{n \cdot (n-1)}{2}$$

.

Therefore, if p is the number of slave processes running in a parallel architecture, the average number of pairs of proteins that each slave will process is equal to

$$\frac{n \cdot (n-1)}{2 \cdot p}$$

.

Since p is much smaller than n, is extremely likely that each process will be asked to process the same protein many times, although in different pairs. This means that each slave process might need to load the same file several times ($n-1$ times in the worst case). Retrieving a file from the file system, although small in size, requires several tens of milliseconds, compared to a few hundred nanoseconds required to access the RAM (a difference of 5 orders of magnitude). The latency may be even greater when the partition containing the file is remote and shared through NFS. Obviously the protein cache needs to be entirely maintained in RAM. From an implementation point-of-view, the cache has been created with a C++ Map data structure, which guarantees a logarithmic complexity on the number of elements (up to n) for the insertion operation, while the access to an element and its deletion are guaranteed to complete in constant time. The access to the data structure is via a key of type string, which represents the filename of the protein. On the other hand, the elements of the map are complex data structures, called `CacheElement`. Each `CacheElement` is a struct of two variables: `usingTime`, which is the time instant in which the process has first inserted or last retrieved the element, and `Protein`, a the convenient protein representation created by the underlying CMS library.

When a slave receives a chunk containing the protein pairs to process, it verifies for each one of them if any of its two proteins is already present in the cache. In the case of a *cache hit*, the slave recovers the protein (that is the `Protein` structure) directly from the cache and updates `usingTime` with the instant of time at which the access has occurred. In the case of a *cache miss*, the file is retrieved from the shared file system, a new `CacheElement` is created and stored in

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

the cache so that the `Protein` structure is stored in the local memory for future references. In this case, `usingTime` is set to the moment where the new element has been inserted.

It is important to emphasize that the cache keeps track of individual proteins and not protein pairs. The maximum cache size is thus at most equal to n, i.e. the number of proteins present in the dataset. To prevent a large dataset of protein to be fully loaded into the cache and saturating the RAM memory as a result, our memory management subsystem uses the environment variable `MAX_CACHE_SIZE` to specify the maximum size of the cache. If the value of `MAX_CACHE_SIZE` is less than or equal to the number of proteins present in any local cache, an exception is raised during the attempt to insert a new element due to saturation of the resource.

According to this description, it is clear that the protein cache adopts a *Last Recently Used* (LRU) policy to locate the item to be deleted from the cache and to allow the insertion of the new element. In fact, thanks to the timestamp stored in the attribute `usingTime`, it is possible to identify the least recently used element in order to replace it with a new one. The cache can be enabled or disabled by the *PolicyLoader* component.

As it was shown in [4], the introduction of the cache improves the performance of the algorithm, even in the case where the shared partition is local to all processes, and not physically distributed.

3.3. Multithread master

As previously explained, the main activity of the master process is to sequentially prepare and send chunks of protein pairs to the slaves. Due to the coarse granularity of CCMS and the need for synchronized send and receive operation, the master process spends most of its time in an idle state, waiting for the slaves requests. In order to maximize the performance of the algorithm, it is essential that the computational unit running the master process contributes to the processing. To enact this behavior, the master process has been implemented as an OpenMP thread.

At the startup, the master process creates two threads. The primary thread answers the requests from the slaves and acts as the algorithm load-balancer, while the secondary thread can perform some processing when the primary thread is idle. The secondary thread therefore can be considered as one of the slaves, but unlike the latter, the communication with the load-balancer takes place via shared memory.

The master uses a smart iterator, called `next_element`, through which it can determine the next pair of protein to send for processing. Both the primary and the secondary thread must be able to access this iterator and, at the same time, must be able to update its value. In order to avoid race conditions and ensure mutual exclusion, we used the OpenMP critical directive. Using a critical section, it is possible to prevent multiple parallel threads to simultaneously access the variables subject to race conditions. Unfortunately, critical sections can be extremely expensive in terms of overhead. To keep the critical section as small as possible, each thread stores in a local variable (`local_next_element`) the current value of `next_element`, then safely increases the global variable at an opportune time.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

4. PERFORMANCE ANALYSIS

In this section we present an analysis of the performance of Hybrid-CMS, and extend what has been stated in [4] with new evidence. The outcome of this critical assessment is the proposal for a more advanced algorithm, that will be sketched in Sec. *5*.

The results discussed in this section refer to the same testing machine used for collecting the measurements reported in [4], namely a custom cluster of four servers interconnected with a Gigabit Ethernet backbone. Each server in the testing configuration has four AMD Opteron 6272 boards with 16 cores each, for a total of 64 cores and 252 GB of RAM per machine and a grand total of 256 cores and almost 1 TB of RAM in the cluster. Each core runs at 2.1 MHz, and has 16 MB of L3 cache.

Moreover, all the results refer to the same testing dataset of proteins. The dataset has been built taking a single protein for each superfamily in the PDB, thus it is composed of exactly 1549 elements. Thanks to this choice, any pair of proteins in the set exhibits a very low similarity score. This is an important property; in fact the behavior, and in particular the performance, of the CMS algorithm changes dramatically if some proteins in the set exhibits too much repetition in their secondary structures [1]. Most importantly, the goal of CMS is to uncover previously unseen similarities in unfamiliar proteins (see Sec. 1). Thus, the choice of the dataset is sound and representative of larger-scale problems.

In the previous work paper, we assumed, on the basis of the results obtained by the pure OpenMP implementation of CMS, that the optimal setup for the shared memory component of Hybrid-CMS was using four OpenMP threads to concurrently evaluate the similarities of a single pair of proteins. This assumption was validated on a smaller testing machine, a single Intel Xeon server with 32 cores. Moreover we showed that the MPI component of the parallel implementation scaled almost linearly when increasing the number of MPI processes if the number of MP threads is fixed to four. Fig. 4.1, drawn from the original paper, validates these statements.

An analytic estimate of the quality of the hybrid implementation is difficult to do. However, we can provide two extremely simple but significant figures: the speedup and the efficiency of the MPI component of the parallel implementation:

$$Speedup_{MPI} = \frac{T_m}{T_p} = \frac{T\,(1:4)}{T\,(64:4)} = \frac{264,917s}{4,259s} \approx 62.202 \tag{4.1}$$

$$Efficiency_{MPI} = \frac{T_m}{p \cdot T\,(p)} = \frac{T\,(1:4)}{64 \cdot T\,(64:4)} = \frac{264,917s}{64 \cdot 4,259s} \approx 0.972 \tag{4.2}$$

where T_m is the execution time to process the entire dataset using only one MPI process, while T_p is the time employed to elaborate the same dataset using the entirety of the testing machine. In particular, as in this scenario we were only considering the case of 4 OpenMP threads for each MPI process and the machine had 256 cores, T_p is the execution time of the run with 64 processes. Note that in Eq. 4.2, the value of p indicates only the number of MPI processes, not the total number of cores of the testing machine.

Moreover, we can provide a more in-depth estimation of the performance of MPI-CMS. To do that we can start with a graphical estimation of the load balancing performance of the algorithm thanks to Fig. 4.2. The figure shows the individual execution times of each process in a given test run.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

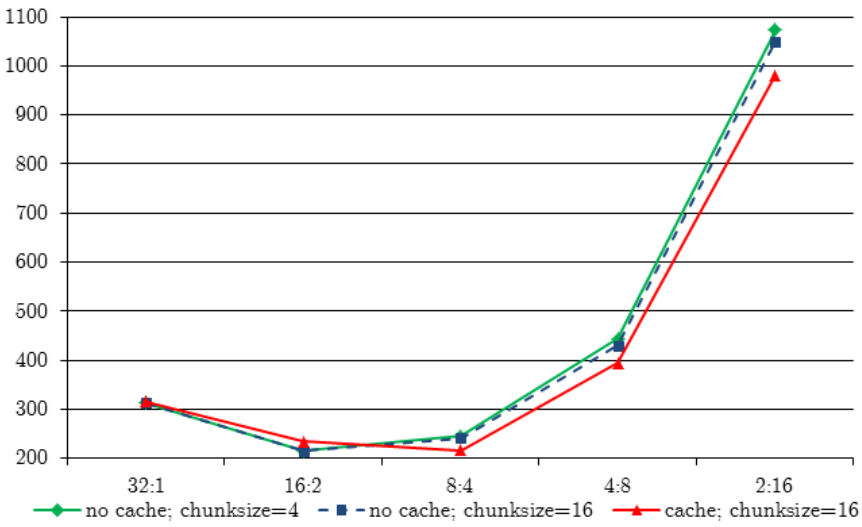

Figure 4.1. The performance of Hybrid-CMS in different configurations. On the y-axis the execution times in seconds. The figure also shows the effects of the introduction of the protein cache (Sec. 3.2) and of different chunk size (Sec. 2.1). *Source*: [4].

Note that a few processes conclude their execution somewhat later than the other ones. Due to the non-deterministic nature of the algorithm the exact behavior of the processes can vary significantly, thus Fig. 4.2 is just a representation of a random run. The behavior, however, is similar across different runs, even if the actual distribution can appear to be very different. The reason of this (relatively small) unbalancing is due to two factors: the first is the size of the computational chunk, as explained in Sec. 2.1, and the second and most important one is the different computational times of each protein pair in any dataset. For instance, as we will see later (Sec. 5.1), the fastest protein pair can be evaluated in a few microseconds, while the slowest one can take up to nine minutes. Clearly, if the heaviest protein pairs are distributed at the end of the computation, the unbalance can be worse than what results from this figure. Note, however, that this kind of behavior is highly unlikely.

We can analytically estimate the quality of the global load balancing (GLB) [15] of any parallel algorithm and of MPI-CMS in particular, as follows:

$$GLB = \frac{\text{avg}\,(T_p)}{\max\,(T_p)} = \frac{4210.145}{4258.649} \approx 0.989G \tag{4.3}$$

According to the definition, if the average execution time of all processes, i.e. avg (T_p) time in Eq. 4.3) is nearby the execution time of the slowest one, i.e.max (T_p) again in Eq. 4.3, a load-balancer can be considered near optimal, as in the case of MPI-CMS. The ideal case, of course, is when $GLB = 1$.

We could try to even improve the Global Load Balancing of the MPI component of Hybrid-CMS. To do that, we could start by reducing the chunksize to the lowest possible value, i.e. one. To estimate the effect of such of a reduction we can compute the average and standard deviation of the individual execution times across various runs of MPI-CMS. For instance we have that:

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

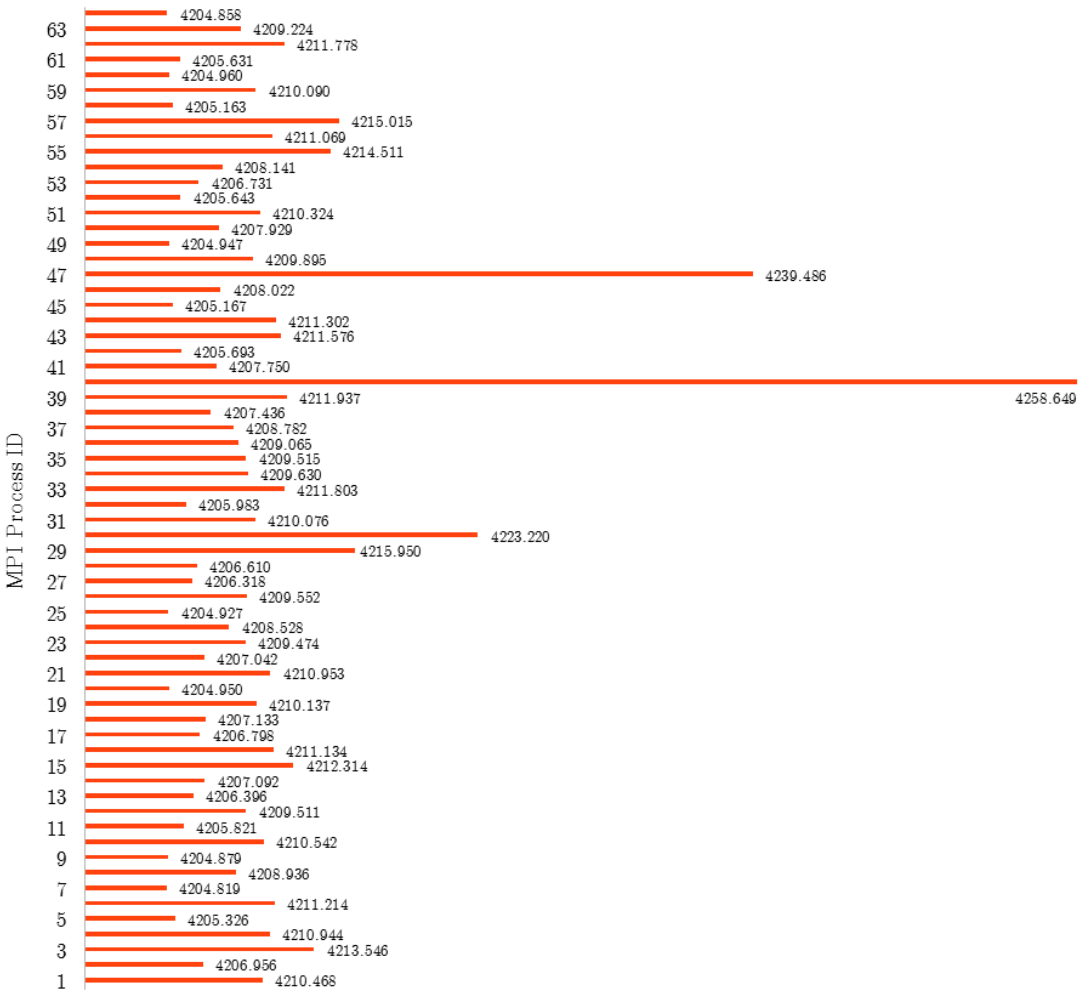

Figure 4.2. Individual execution times, in seconds, of each of the 64 MPI-CMS processes on a random run. To outline the differences between the processes, the figure shows a zoom on the terminal part of the complete graph (the baseline value is 4200 seconds).

$$\sigma_{chunk=4} \approx 7.962s$$
$$\sigma_{chunk=1} \approx 5.983s$$

Unfortunately, as the average execution time across different run is more than four thousand seconds, the coefficient of variation c_v is negligible in both cases. For instance, for the second scenario where the chunksize is set equal to one we obtain:

$$c_v = \frac{\sigma_{chunk=1}}{\text{avg}\,(T_P)} \cdot 100\% \approx 0.142\%$$

Thus, we decided to not pursue this strategy any longer. Finally we can also conclude that a strategy based on the redistribution of chunks between different process would be equally ineffective, as the current global load balancing is already near optimal, as noted before.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

5. A NEW PROPOSAL

5.1. Workload characterization

After the initial design and development of MPI-CMS, described in the previous sections of this paper, we continued to work on the project. The goal, as outlined in the conclusions of the original paper was to port Hybrid-CMS to a real supercomputing machine. In particular, for our case, the target is Fermi, the IBM Blue Gene/Q machine available at the CINECA facility in Bologna, Italy. The motivation behind the porting of MPI-CMS is to extend the analysis carried out up to the present moment to a far bigger dataset of proteins. The reader should remember that the tests presented so far were restricted to a database of 1549 proteins, while the entirety of the Protein Data Bank stores about 100,000 proteins. Even if, for domain-specific reasons which are mostly outside the scope of this paper, the analysis on the entirety of the PDB is not of interest from the point-of-view of a geometrical motif extraction algorithm, it would be nevertheless extremely interesting to extend the analysis to a dataset containing at least one member for each protein *family* instead of of a member for each protein *superfamily*, which would improve the amount of computations by at least two orders of magnitude [16].

In preparation for the aforementioned porting we decided to analyze the performance of MPI-CMS in even more details than before (Sec. 4) so to ascertain and remove any remaining inefficiency.

Eq. 4.1 and Eq. 4.2 show, respectively, the speedup and efficiency of MPI-CMS. However, these value can be misleading, as they only refer to the message passing component of the implementation. If we take into account the shared memory component into the final formulas, the figures change drastically:

$$Speedup_{Hybrid} = \frac{T(s)}{T(p)} = \frac{T(1:1)}{T(64:4)} = \frac{452,870s}{4,259s} \approx 106.332 \tag{5.1}$$

$$Efficiency_{Hybrid} = \frac{T(s)}{p \cdot T(p)} = \frac{T(1:1)}{256 \cdot T(64:4)} = \frac{452,870s}{256 \cdot 4,259s} \approx 0.419 \tag{5.2}$$

This time, $T_s$is the actual serial execution time of the algorithm, and so we have to use 256 as the value for p (these figures, of course, refers to the testing machine described in Sec. 4).

Eq. 5.1 and Eq. 5.2 show that the overall efficiency of Hybrid-CMS is below what is generally considered to be acceptable, even if the absolute value for the speedup is quite remarkable. Comparing Eq. 4.2 and Eq. 5.2, it is clear the the low value for the efficiency is entirely due to the poor scalability of the OpenMP implementation of CMS, as already noted.

To mitigate such inefficiency we have explored several strategies; however we cannot improve the performance of MP-CMS. Put in a very simple way, CMS uses an aggressive strategy to reduce the space in which to search possible similarities of each protein pair. However, the same strategy also causes an unpredictable unbalancing of the internal execution loops of the algorithm which makes it extremely hard to parallelize in a balanced way. An exhaustive discussion on this point can be found in [1].

We have hypothesize that in some scenarios it could be better to perform some CMS run in serial, that is to allow some MPI process to only run on a single OpenMP thread. To demonstrate the

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

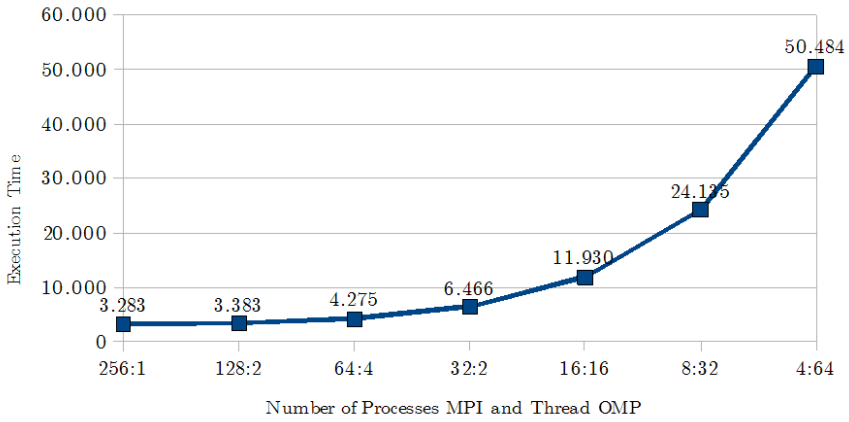

Figure 5.1. Execution times in seconds of MPI-CMS on the AMD Opteron Cluster (y-axis). On the x-axis different configurations of the implementation according to the notation presented in Sec. 2. According to the figure, it appears that a pure MPI configuration performs better than any hybrid ones.

validity of this hypothesis we tried an extreme approach, that is to process the entire dataset using only one OpenMP thread per MPI process.

Fig. 5.1 shows the execution times of MPI-CMS in different balancing configuration between OpenMP and OpenMPI, according to the notation in Sec. 2. Note the every configuration exploits the same number of cores (256, due to the given testing machine) but we obtain faster and faster execution times and thus better and better efficiency as the ratio between OpenMP and MPI gets skewed in favor of the second one.

At the end, it is possible to note that a pure MPI implementation of Hybrid-CMS performs better than any hybrid solution. This appears to be in contrast to what we have concluded in the original paper. In fact, this behavior was not present with the smaller testing machine and the smaller dataset employed in the previous iteration of this work.

We can speculate that this behavior is due to the fact that most or at least many OpenMP threads remain idle during most of the run. We confirmed this hypothesis using both the TAU profiler application and the basic Unix `top` tool. A further proof of this behavior can be found in Table I, where we eventually overloaded the testing machine with a number of thread higher than the number of available cores. As in the previous case, the testing dataset is the same described in Sec. 4. Due to the fact the the overall execution time improves even in the two overload scenario presented in the Table, we can conclude that the OS scheduler has been preempting idle threads with ready threads. Again, the reason behind this behavior has to be in the inherent unbalancing of the parallelization scheme of the OpenMP implementation of CMS.

Luckily the shared memory unbalancing, and thus the high idle time of the OpenMP threads, is not uniform across the entire protein dataset. In [1], we showed that the protein pairs processed by CMS in the longest time, are extremely well-balanced across multiple threads. However, the same paper also shows that is extremely difficult to predict the execution time of a given protein pair. Four factors were listed in the paper as concurrent causes to the length of processing of a pair of proteins: 1) geometrical tolerances; 2) maximum size of the motif searched for by CMS; 3) similarity score

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

MPI Processes	Execution time (sec.)
4:4	66597.758
8:4	30954.752
16:4	15050.973
32:4	7489.563
64:4	4259.023
128:4	3321.659
256:4	3497.297

Table I. Execution time of MPI-CMS in different configurations. The final two rows represent a configuration when the testing machine is overloaded with a number of threads higher than the number of available cores. This table is an extension of the table presented in appendix of the original paper [4].

of the proteins in the pair; 4) sizes of the proteins in the pair. Factors 1) and 2) are constant in a given run, so they are not useful for the present analysis. Factor 3) requires an in-depth knowledge of the application domain and is difficult to evaluate without introducing much more processing into the CMS algorithm. Thus, we can work only with factor 4), that is the size of the protein pair. Put in other words, we need to take into account some workload characterization to improved the load-balancing of MPI-CMS.

To validate the idea we can compute the statistical correlation between the size of protein pairs and the execution time of their CMS computation, which is as follows:

$$\rho = 0.653 \tag{5.3}$$

where the pair size can be simply defined as the product between the number of secondary structures in the first protein and the number of secondary structures in the other protein in the pair.

Eq 5.3 shows a good correlation between execution time and pair size. Due to the pure textural representation of proteins employed by CMS, the product is almost perfectly correlated to the product of the size in byte of the XML representation of the two proteins. Thus, we can state that we can reliably predict the execution time of a protein pair simply by looking at the size of the input data.

To further refine the analysis, we can plot the histogram in Fig. 5.2, which shows the distribution of protein pairs according to their measured execution time.

From the figure it is clear that a very large number of protein pairs (more than 500,000, that is almost 50% of total) actually takes only a few hundredths of a second or even less to complete, while only very few pairs take more than a couple of seconds.

Starting from these consideration we can design a new implementation of MPI-CMS, which exploits the distribution of the protein pairs according to their execution time and the correlation between the latter and the pair size. This revised implementation is described in the following section.

5.2. New design

In this section we present a first draft of a new parallel implementation of MPI-CMS which exploits the consideration in Sec. 5.1.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

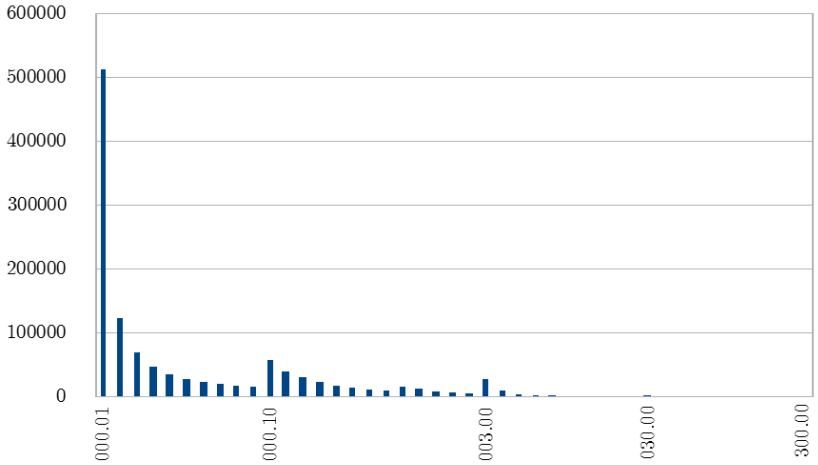

Figure 5.2. Histogram of the distribution of protein pairs according to their measured execution time. On the x-axis the per-pair execution time (in seconds), subdivided in a number of bins. The bins are not uniform, as the graphs would be otherwise unreadable. On the y-axis the number of protein pairs belonging to each bin.

The base idea behind the new implementation is to classify each pair of proteins depending on an estimate of its processing time, in order to allow the slaves to adopt different processing strategies depending on the classification of each protein pair.

The classification of each pair is performed by the master process before the chunk is sent to the slave. The master estimates the complexity of processing on the basis of the pair size, since, as we have observed in the previous section, this value has a more than sufficient correlation with the execution time.

In the initialization phase, the master process creates two different lists of proteins pairs, called respectively *light* (L) and *heavy* (H). The list L includes all pairs of proteins labeled as light, that is those pair which size is less than a given threshold value. The choice of a good threshold value is not trivial, and is discussed in more detail in Sec. 5.2.1. With a good probability, these proteins will take a rather short processing time, according to what was said in Sec. 5.1. The search for similarities in light protein pairs would be extremely unbalanced and thus inefficient if performed using MP-CMS. Thus, to improve performance, protein pairs in this class need to be executed in a purely serial fashion in a single MPI process. On the other hand, list H includes all proteins pairs labeled as heavy, that is those pairs whose size is greater than or equal to the threshold value. Likely, these protein pairs will take a rather long time to be processed and therefore the performance will be better if multiple OpenMP thread will concur to its processing.

When a slave request a new chunk from the master, the latter send to the former the first available one from one of the two lists. The MPI message will contain an additional information with respect to the format described in Sec. 2, that is the class (light or heavy) of the chunk. It is up to the distribution policy to define which chunk to send to a particular slave if more than one is available in different lists.

The size of the two different types of chunk cannot be clearly the same from a performance optimization point-of-view. The chunk containing light proteins has to be larger in size, for instance it could contain 32 or 64 pairs of proteins, while the chunk containing heavy proteins has to be

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

smaller, e.g. it could contain one or two pairs of proteins. With reference to the previous design, we move from a fixed chunksize, to a dynamic one.

Upon receipt of the chunk, the slave process adopts a different processing strategy according to the detected chunk class. When a slave receives a light chunk, it creates n OpenMP threads, where n is the number of available computational units, then distributes the m protein pairs contained in the chunk between the different threads. The distribution of the workload can take place using dynamic OpenMP scheduling to ensure that faster threads will likely receive a larger number of pairs. Only after the join of all OpenMP threads the slave process requests the next chunk from the master.

On the other hand, when a slave receives a heavy chunk, it creates n OpenMP threads that cooperates to the processing of the m pairs in the chunk. While in the previous scenario we could have up to n small concurrent pair processing at the same time, in this scenario we have m sequential operations, which are concurrently executed by n threads.

Thanks to the variable chunksize, the characterization of the workload and the implementation of different parallelization strategies, the revised hybrid implementation can optimize the use of the computing resources, and improve its parallel performance well-above the level of a pure MPI implementation.

At the time of writing, the revised implementation is still work in progress and it will be released shortly.

5.2.1. Threshold value From the above discussion, it is clear that an adequate choice of the threshold value, that is the cutoff value which determines if a protein is going to be classified as *light* or *heavy* by the algorithm, is crucial to maximize performance. Assigning a low threshold value would in fact keep in the implementation some amount of thread idleness, as described in Sec. 5.1. In this way we would not reach the full optimization potential of the hybrid implementation. Conversely, using a high threshold value would lead to reduce the parallelism of the algorithms, as some protein pair which would benefit from shared memory execution would be serially executed.

We have attempted, through statistical analysis of the testing data, to identify an ideal theoretical value for the threshold. However, we was not able to do that as this ideal value is too dependent on the dataset. Therefore, the only way is to use a reasonable value for the threshold that can be only determined experimentally case-by-case.

6. CONCLUSIONS

In this paper we have presented both the design and implementation of Hybrid-CMS, a shared memory and message passing hybrid version of the Cross Motif Search (CMS) algorithm, which is able to efficiently perform exhaustive searches for geometrical motifs in large datasets of proteins. We have shown, using a in-depth performance analysis, that even if the current version of Hybrid-CMS vastly outperforms the previous OpenMP implementation, it has a few remaining inefficiencies. Finally we have shown how the current design can be further improved using a simple statistical analysis of the workload. The revised version of MPI-CMS is still work in progress: however we have described its design and conducted some preliminary experiments.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

The final goal and main motivation of this paper, as stated in Sec. 5, is to port MPI-CMS on a massive supercomputing machine. We have already been granted an access to Fermi, the IBM Blue Gene/Q machine of the Cineca facility in Bologna, Italy. At the moment, the port is almost completed and it will be discussed in much details in a new paper.

References

1. Ferretti M, Musci M. Geometrical Motifs Search in Proteins: A Parallel Approach. *Parallel Computing* ; (in press, accepted 30 September 2014).
2. *New Trends in Image Analysis and Processing — ICIAP 2013*, *Lecture Notes in Computer Science*, vol. 8158, Springer: Heidelberg, 2013.
3. Cantoni V, Ferretti M, Musci M, Nugrahaningsih N. Structural Motifs Identification and Retrieval: A Geometrical Approach. *Pattern Recognition in Computational Molecular Biology: Techniques and Approaches, 1st edition*, Elloumi M, Iliopoulos CS, Wang JTL, Zomaya AY (eds.). John Wiley and Co.: New York, USA, 2014.
4. Ferretti M, Musci M, Santangelo L. A Hybrid Openmp and Openmpi Approach to Geometrical Motif Search in Proteins. *Proceedings of the IEEE International Conference on Cluster Computing (IEEE Cluster 2014)*, IEEE Computer Society, 2014; 298–304.
5. Cantoni V, Ferone A, Özbudak O, Petrosino A. Structural analysis of protein secondary structure by GHT. *21st International Conference on Pattern Recognition, ICPR'12, Nov. 11–15, Tsukuba, Japan*, IEEE Computer Society Press, 2012; 1767–1770.
6. Cantoni V, Ferone A, Özbudak O, Petrosino A. Motif Retrieval by Exhaustive Matching and Couple Co-occurrences. *9th International Meeting on Computational Intelligence Methods for Bioinformatics and Biostatistics, CIBB'12, July 12–14, Texas*, 2012; 1767–1770.
7. Richardson JS. The anatomy and taxonomy of protein structure. *Advances in Protein Chemistry* 1981; **34**.
8. Research Collaboratory for Structural Bioinformatics. Protein Data Bank. `http://www.rcsb.org/pdb` 2013. [Online; accessed 5-December-2013].
9. Ballard DH. Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition* 1981; **13**(2):111–122.
10. Shi S, Zhong Y, Majumdar I, Krishna SS, Grishin NV. Searching for three-dimensional secondary structural patterns in proteins with ProSMoS. *Bioinformatics* 2007; **23**(11):1331–1338.
11. Shi S, Chitturi B, Grishin NV. ProSMoS server: a pattern-based search using interaction matrix representation of protein structures. *Nucleic acids research* July 2009; **37**(Web Server issue).
12. Hutchinson G, Thornton JM. PROMOTIF — a program to identify and analyze structural motifs in proteins. *Protein Science* 1996; **5**:212–220.
13. Dror O, Benyamini H, Nussinov R, Wolfson H. MASS: multiple structural alignment by secondary structures. *Bioinformatics* Jul 2003; **19**(1):i95–i104.
14. Krissinel E, Henrick K. Secondary-structure matching (SSM), a new tool for fast protein structure alignment in three dimensions. *Acta Crystallographica Section D: Biological Crystallography* 2004; **60**:2256–2268.
15. Casavant L, Kuhl JG. A Taxonomy of Scheduling in General Purpose Distributed Computing Systems 1988.
16. Hubbard TJ, Ailey B, Brenner SE, Murzin AG, Chothia C. SCOP: a structural classification of proteins database. *Nucleic Acids Research* 1999; **27**(1):254–256.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe