Scalability Analysis of Progressive Alignment on a Multicore

Sebastian Isaza*, Friman Sanchez†, Georgi Gaydadjiev*, Alex Ramirez†‡ and Mateo Valero†‡
*Computer Engineering Laboratory, Delft University of Technology
{s.isazaramirez,g.n.gaydadjiev}@tudelft.nl
†Computer Architecture Department, Technical University of Catalonia
fsanchez@ac.upc.edu
‡Barcelona Supercomputing Center
{alex.ramirez,mateo.valero}@bsc.es

***Abstract*—Sequence alignment is a fundamental instrument in Bioinformatics. In recent years, numerous proposals have been addressing the problem of accelerating this class of applications. This, due to the rapid growth of sequence databases in combination with the high computational demands imposed by the algorithms. In this paper we focus on the analysis of the progressive alignment in ClustalW, a widely used program for performing multiple sequence alignment. We have parallelized ClustalW for the Cell processor architecture and have carefully analyzed the scalability of its different phases with both the number of cores used and the input size. Experimental results show that computing profile scores scales well up to 16 SPE cores. With the increase of the input size, profiles initialization in the PPE core becomes the predominant bottleneck.**

I. Introduction

Bioinformatics is the discipline that applies computational techniques to solve problems in biology [1]. Due to the huge amount of data involved, computers are used to assist researchers. Many algorithms and software tools have been then proposed in the last years to assist scientists while performing many different analysis on large amounts of biological data. Multiple Sequence Alignment (MSA) is one of the essential tasks in bioinformatics. When having a group of DNA sequences an MSA can reveal sections that are common to most of the sequences in the group. This information has many uses, for example, it can indicate evolutionary relationships between species or, in the case of protein analysis, it may determine a protein's function based on other known functionalities in the reference group.

ClustalW [2] is one of the most commonly used programs to perform MSA. Unlike pairwise sequence alignment tools like BLAST [3] and FASTA [4], ClustalW aligns a set of sequences all together to produce an MSA. Given the intractability of computing an optimal MSA, ClustalW uses heuristics. The program is divided in three main phases, namely: Pairwise Alignment (PW), Guide Tree (GT) and Progressive Alignment (PA).

Improvements in sequencing technologies have led to a very rapid growth of biological databases. As these databases grow, analyses are more ambitious and the need for computational power increases. On the other hand, the use of expensive multiprocessor machines to meet the performance demands has two main limitations: only few can afford it and the energy needed is extremely high. In the search for efficient solutions previous studies did propose accelerating ClustalW using FPGAs, GPUs and multicore processors.

In this paper we use the Cell processor [5], [6] to perform our study. Figure 1 shows the Cell architecture composed by eight Synergistic Processing Elements (SPEs), a PowerPC Processing Element (PPE) and the Element Interconnect Bus (EIB). SPEs are dual-issue in-order SIMD cores with 256KB Local Stores (LS) and 128 registers, 128-bit wide. The PPE is a 2-way Simultaneous Multithreading (SMT) dual-issue in-order PowerPC processor. The EIB is a circular ring comprising four 16B-wide unidirectional channels that connects the SPEs, the PPE, two memory controllers and two I/O controllers. The operating system runs on the PPE and software can spawn threads in the SPEs. Data has to be explicitly copied to the SPEs LSs using Direct Memory Access (DMA) commands. The Memory Flow Controller (MFC) in each SPE takes care of these DMA transfers and it does it in parallel to the SPEs' execution. This allows for hiding DMA transfers latency. Although these features enable programmers to write highly efficient code, it also significantly increases programming difficulty.

In this paper we analyze the scalability of ClustalW running on the Cell processor. Previous work has shown that the PW phase is highly scalable so we focus on the PA part. We have generated a scalable parallelization of the *prfscore* function within PA and have merged it with the parallel forward (FWD) and backward (BWD) loops, as explained later in Section III. The contributions of this work are:

- measuring and analyzing how the different parts of PA affect the performance when increasing number of cores and input data size;
- producing a scalable parallelization of *prfscore* in PA.

This analysis provides crucial knowledge that is needed if we are to build multicore architectures that can be efficiently used for bioinformatics applications. The rest of the paper is organized as follows. Section II describes other works with similar aim and how ours is different. Section III describes ClustalW phases and the way it is parallelized. In section IV

978-0-7695-3967-6/10 $26.00 © 2010 IEEE
DOI 10.1109/CISIS.2010.149

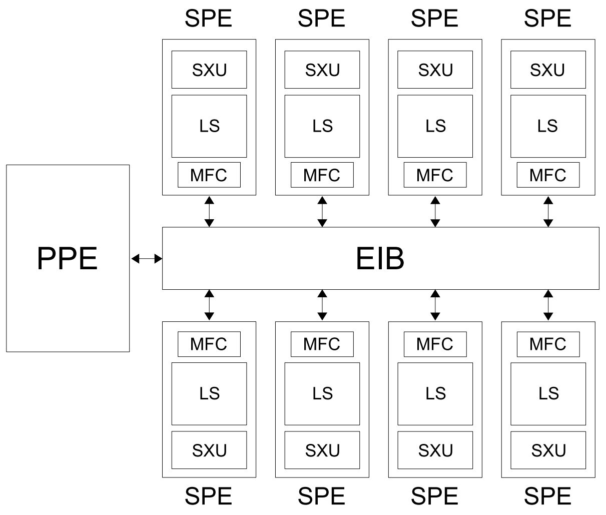

Fig. 1. Cell processor block diagram.

we describe the experimental setup. In section V we analyze the experimental results obtained. Section VI concludes the paper.

II. Related Work

There have been different approaches to accelerating ClustalW, from using multiprocessor machines and Graphic Processing Units (GPUs) to designing HW accelerators implemented on FPGAs. In [7], [8], distributed memory parallelizations (using *MPI*) of ClustalW are presented. They parallelized both PW and PA targetting computer clusters and report speedups without any analysis. In [7], performance scalability is fairly good up to 8 processors while in [8] it lasts up to 16 processors. The main reason for this difference is that the input sequences used in [8] are about 4 times longer on average. This results in the PW phase taking much longer time than the other phases. As a consequence and since PW is the most scalable phase, the overall performance scalability is better. In these cases, the main performance bottleneck is the long computation time needed for each PW alignment. On the other hand, in [9] ClustalW is parallelized for Cell. PW scales close to linearly as well but now the inside of PW is parallelized with SIMD execution which greatly reduces its execution time. As a consequence, PA becomes the predominant stage and due to its more limited parallelism, the overall application's scalability saturates much more quickly.

In [10] *pthreads* have been used to parallelize ClustalW using a shared memory model. Experiments reported focused more on varying the number of input sequences and their length. However, a machine supporting only 4 threads is used and the analysis done is minimal. SGI has also announced an *OpenMP* parallelization for shared memory machines but the source code is not available.

A number of recent works have ported ClustalW to Cell. In [11], [12], the PW phase is parallelized showing linear scalability. This is due to its inherent parallelism that requires minimal synchronization and has no data dependencies. On the other hand, PA's parallelism is more limited. Vandierendonck et al. [9] have produced a highly optimized version of PA specifically tuned for using 6 SPEs. Two master SPEs independently compute the forward (FWD) and backward (BWD) loops while each of them get help from two other SPEs (four SPE helpers in total) that take care of computing the *prfscore* function, needed in the inner FWD/BWD loops. Most of the code running on the SPEs is vectorized and data communication is double-buffered. Although the speedup achieved with respect to the sequential code running on the PPE is significant (5.8X), the parallelization is fixed, that is, the code is only able to run with 6 SPEs, not with more, not with less. This does not allow to study scalability behavior when the aim is to look forward into future architectures where more cores will be available. Different to what has been done in previous works, our aim in this paper is to measure and analyze how the contribution to performance changes for different PA parts when varying input size and number of cores. For this purpose, we have parallelized PA in a flexible way that can use an arbitrary number of SPEs. Although the forward and backward loops remain *2-way* parallel, the *prfscore* is now *n-way* parallel. Details, benefits and drawbacks of doing this are discussed in Section III.

Lastly, an analysis of *Smith-Waterman* [13] algorithm running on different multiprocessor machines was presented in [14]. Our work goes in the same direction and complements it by looking at MSA.

III. Application's Description

ClustalW v.1.83 is a software tool to perform MSA. It was developed by J.D. Thomson et al [2] and is available online through the European Bioinformatics Institute website [15]. Although ClustalW2 is already available [16], [17], it is a C++ re-write that incorporates the same algorithms, therefore not affecting our analysis.

A. Program's Structure

Recalling from section I, ClustalW is composed of three phases: PW, GT and PA. PW computes a similarity matrix by performing pairwise alignments of all possible pairs of sequences. GT uses a neighbor-joining algorithm [18] to build a tree that clusters together sequences that are more similar while putting others in different branches. As shown in previous articles [11], [9], [12], GT takes an almost negligible time in most cases, certainly in the ones we are interested in where sequences are in the order of thousand symbols. Lastly, PA progressively aligns sequences following the tree structure. It walks the tree by aligning the most related sequences first. After traversing the full tree, the MSA is produced along with a global score.

Figure 2a shows the basic PA operation. The main function in PA is *prfalign*. It first computes the two 2D arrays that contain the *profiles* and then call the *pdiff* function to align them. *pdiff* computes the alignment with a dynamic programming algorithm [19] that uses two independent loops: one that moves forward (FWD) and one that moves backwards (BWD). The green arrow between the two corresponding boxes in figure 2a emphasizes the fact that there are no data dependencies. A third small loop (not shown for simplicity) collects results and a recursive call to *pdiff* follows. Every recursive *pdiff* instance incrementally reduces the loops' scope until a final score is computed. This condition is checked for at the beginning of *pdiff* as shown in figure 2a. *ptracepath* takes care of updating the alignment that will subsequently be aligned to a sequence or to another alignment (by further calls to *prfalign* and *pdiff*), by walking the tree nodes.

B. Porting to Cell

The PW phase is parallelized by letting the SPEs compute individual pairwise alignments in parallel. Since PW is mostly compute-bound, performance does not get affected by the job distribution policy. More details can be found in [11], [9].

PA has a less straightforward parallelism, as shown in Figure 2b. The FWD and BWD loops are independent (see figure 2a) so we concurrently compute them using two SPEs, as in [9]. Figure 2b shows the basic interaction between the PPE and one SPE where only FWD is indicated. An equivalent pair of threads (one in the PPE and one in the SPE) computes BWD in the same way.

We have made the observation that the *prfscore* calls inside FWD/BWD can be fully computed in advance as they only depend on the input *profiles* that do not change inside *pdiff*. This has two benefits. On one hand, it allows for a realtively easy distribution of work among all available SPEs. On the other one, *prfscore* results can be stored in a 2D array that is to be used by all recursive instances of *pdiff*. This saves the unnecessary repeated *prfscore* calls in the original algorithm that was aimed at reducing the amount of memory needed. In consequence, all *prfscore* calls are precomputed by an arbitrary number of SPEs. A *prfscore matrix* is computed and then used as input for FWD and BWD. That is, in the inner body, FWD and BWD do not compute the scores anymore but rather get them from main memory. These DMA transfers happen once for every row and are double-buffered. Therefore, the *prfscore* stage will use N SPEs while in the FWD/BWD stage only two will be running.

Although not shown in the figure 2, a check is done on the *profiles*' sizes before running FWD/BWD. This is to avoid sending too tiny pieces of FWD/BWD work when inside recursive *pdiff* instances.

Figure 3 illustrates the computation of the *prfscore matrix*. The inputs are two 2D arrays called *profiles* (A and B in figure 3) where each one represent either a sequence or the result of a previous alignment. The *prfscore matrix* will have *M* rows and *N* columns where position *"i,j"* is computed as the dot-product between rows *i* and *j* of the two input arrays respectively. This work is then performed by the SPEs together. *Profile A* is split in sections that are assigned to the SPEs in a round-robin fashion. *Profile B* has to be divided in chunks due to the LS size limitation. While processing one chunk, the next one is being fetched to hide the transfer latency.

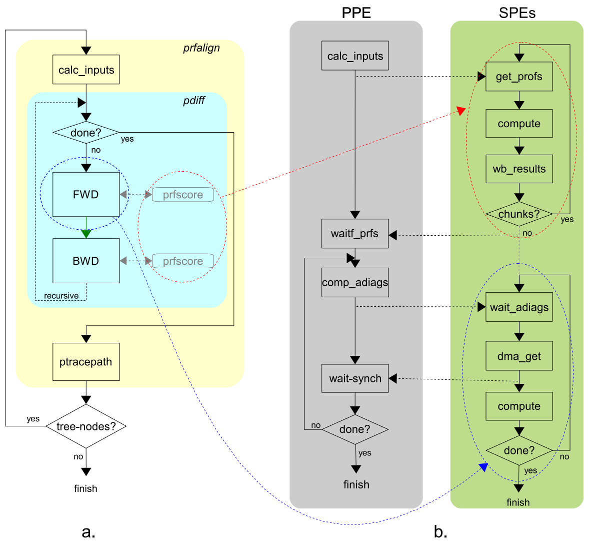

Fig. 2. PA workflow. a.) Baseline sequential version. b.) Parallel version showing the PPE and one of the SPEs.

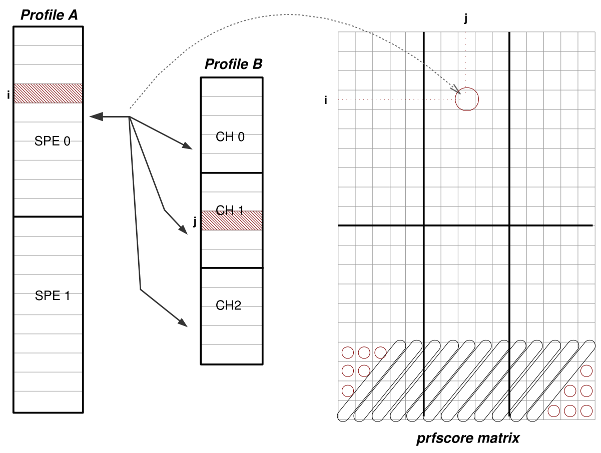

Fig. 3. Generation of the *prfscore matrix* by dot-products of all rows in the two input *profiles*.

Since the SPE FWD/BWD code is SIMD and because of data dependencies (like in [11]), the *prfscore matrix* needs to be reorganized in a way that elements of 4-wide antidiagonals form vectors, as shown in the lower part of figure 3. For that purpose, we split the matrix in two parts (one that concerns FWD and one for BWD) that are concurrently reorganized by two PPE threads. These also handle synchronization with their corresponding consumer SPE threads. Furthermore, the corner elements that do not belong to any 4-wide antidiagonal are fetched one-by-one from the SPEs. The two PPE threads benefit from the SMT capabilities.

TABLE I
PA PHASE NAMES USED.

Name	Meaning	PPE/SPE
calc-inputs	compute the two input *profiles*	PPE
wait-prfs	wait until the *prfscore matrix* is computed	PPE
comp-adiags	reorganize *prfscore matrix* in antidiagonals	PPE
wait-synch	synchronize reception of antidiagonals	PPE
others	*ptracepath* and other minor phases	PPE
wait-adiag	FWD/BWD waiting for an antidiagonal to be computed	SPE
total-dma	DMA waiting time for corner elements and antidiagonals	SPE
compute	compute the *prfscore matrix* and the FWD/BWD loops	SPE
get-profs	DMA waiting time for input *profiles*	SPE
wb-results	DMA waiting time for matrix data	SPE

IV. Experimental Setup

We have ported ClustalW v.1.83 to Cell using libspe2 and performed the experiments in an IBM QS21 Blade featuring two Cell processors running at 3.2GHz and 4GB of RAM. The code has been compiled with GCC4.1.1 and -O3 flag. The source code has been manually instrumented to measure the time that various processing or waiting phases consume. Four different input sets have been used with 66, 100, 200 and 313 sequences of 1000 symbols average length.

V. Results and Analysis

This section presents a detailed profiling analysis of the different PA phases. PW analysis can be found in [11], [9] and GT is not considered as it took less than 1% of the total execution time in all cases analyzed.

Several optimizations like double-buffering, the use of SMT threads and the precomputation of *prfscore* have been implemented. However, since this paper's focus is not on software optimizations, the impact of those is not measured nor presented here. Results shown used all the above optimizations.

Table I indicates the meaning of the different phase names used in the graphs and the text.

A. *prfscore behavior - SPE side*

The *prfscore* parallelization requires only minimal synchronization and the PPE just waits until SPEs are done. Therefore, here we analyze the behavior from the SPE perspective only. Figure 4 shows the time share of the *prfscore* processing in the SPEs. Looking at the DMA transfers, there are two contrasting results. While fetching the input data (*get-profs*) from main memory appears negligible, the time for writing results back (*wb-results*) is considerable and gets worse with more SPEs. Although both types of DMA transfers are double-buffered, output data is much larger than input data (see figure 3). Moreover, as more SPEs come into play, more data is being transferred simultaneously, thus putting pressure in the communication infrastructure. On the other hand, even when

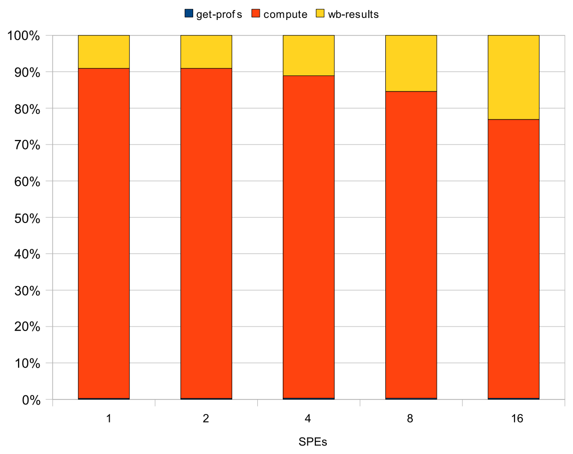

Fig. 4. Execution time distribution for *prfscore* phases in the SPEs.

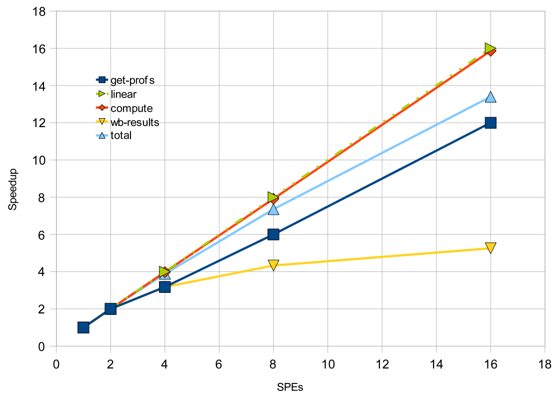

Fig. 5. Scalability of *prfscore* processing on the SPEs.

using 16 SPEs, most of the time is spent in performing useful computations (orange).

Figure 5 shows that up to 16 SPEs the overall scalability is relatively good (light blue line). However, *wb-results* phase will soon become the main bottleneck if adding more cores.

Results shown for the *prfscore* analysis are with 313 sequences as input. Very similar results were obtained with the other input sets. That is, the three *prfscore* phases grow at the same pace with respect to the number of sequences processed. Since we are interested in analyzing scalability when using an increasing number of cores, the baseline for the speedup is the execution time using one SPE (not PPE).

B. *FWD/BWD behavior - SPE side*

Unlike for the *prfscore* analysis, for FWD/BWD we use the number of sequences in the input set as the independent variable (X axis in figures 7 and 6). The FWD/BWD code in the SPEs is heavily affected by the *comp-adiags* phase in the PPE. As a consequence, only 40%-50% of the time is spent on computing. Around 50% of the time the SPEs are idle waiting for the PPE to reorganize the current row.

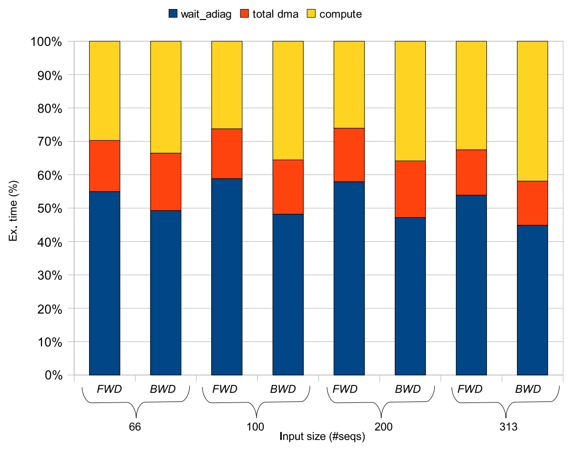

Fig. 6. Execution time distribution of PA phases in the SPEs.

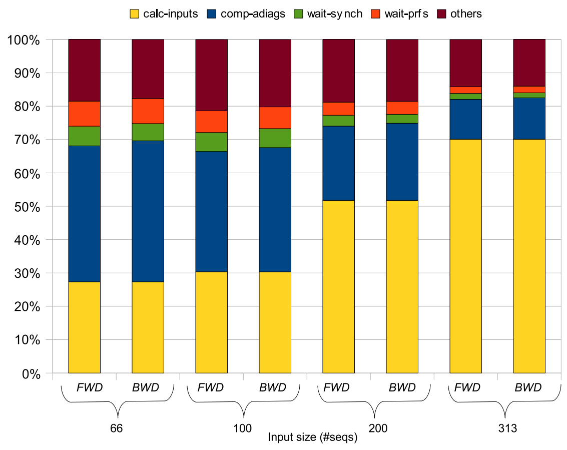

Fig. 8. Execution time distribution of PA phases in the PPE.

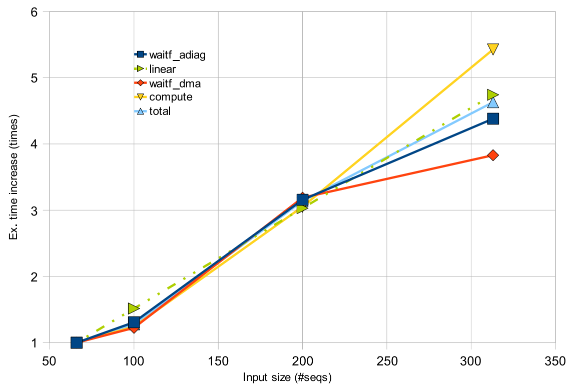

Fig. 7. Time growth of PA phases in the SPEs.

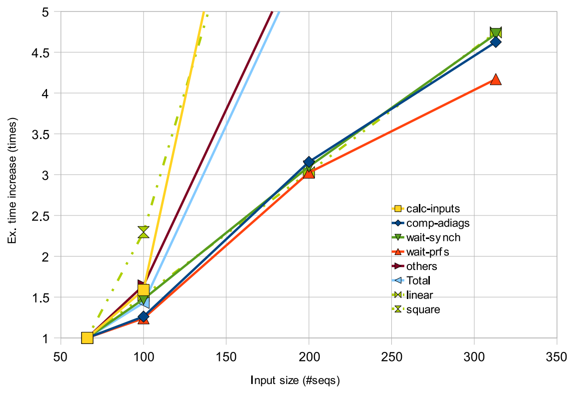

Fig. 9. Time growth of PA phases in the PPE.

Eventhough the corner elements represent only 0.3% of the *prfscore matrix*, its associated DMA latency is consuming nearly 15% of the FWD/BWD SPE time. One way to improve this is by flattening corners in vectors so that they can be accessed faster from the SPEs.

Lastly, figure 7 shows that increasing the number of sequences does not significantly deviate the growth lines from the linear behavior.

C. PA behavior - PPE side

Now we look at the PPE side behavior with respect to the SPE phases already discussed. In figure 8 the time share reveals interesting results. After efficiently parallelizing *prfscore*, the portion it takes now is very small (orange). While the SPEs are busy with FWD/BWD, the PPE spends the time reorganizing the *prfscore matrix* (blue and green). The most important observation here is that when using the largest input size, the preparation of inputs (yellow) becomes the most time consuming part. Besides that, new phases become significant.

In figure 9 we show how fast is the growth of the execution time for the PA phases when the input size increases. Two phases (yellow and brown) do not follow a linear growth. Moreover, *calc-inputs* time grows faster than n^2 (see the dashed lines representing the linear and squared functions). Computing the input *profiles* becomes then the highly predominant applications bottleneck when increasing the number sequences to be analyzed.

In figure 8, bars are grouped to show the behavior of the two SMT threads running on the PPE. As expected (because they reorganize half matrix each) the workload is well balanced among the two.

Although not shown here, one of the procedures that becomes predominant in the *others* category is the *ptracepath* function. *ptracepath* is called after *pdiff* finished aligning two *profiles* and it is in charge of reconstructing the path information so that the alignment can be stored.

D. Overall application's behavior

As a complement to the PA analysis presented so far, here we look at the overall ClustalW behavior (see figure 10). Although PW dominates for the single core case, it quickly decreases due to its linear scalability. PA runtime gets also reduced but in a much more limited way and GT remains negligible.

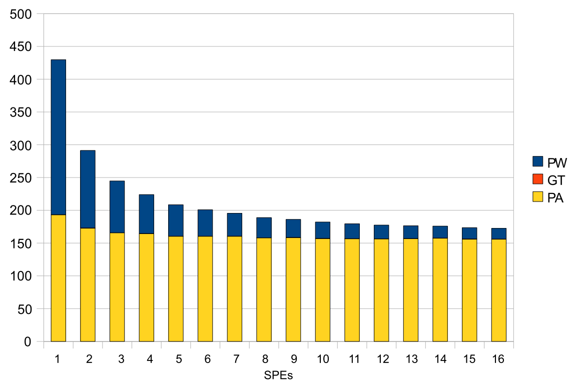

Fig. 10. Execution time of ClustalW phases with varying number of SPEs.

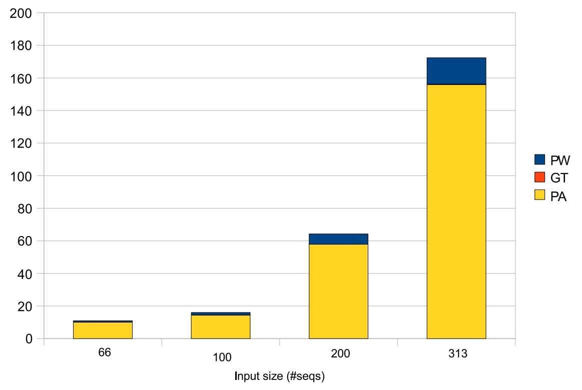

Fig. 11. Execution time of ClustalW phases with varying number of input sequences.

With respect to increasing the input size, figure 11 shows that both PW and PA grow at a similar rate, faster than linearly.

VI. Conclusions and Future Work

We have carefully profiled and analyzed the behavior of the PA phases in ClustalW. We have also parallelized the *prfscore* computations so that an arbitrary number of SPEs can be used. Results show that the parallel *prfscore* version scales close to linearly with both the number of SPEs and the number of input sequences. On the other hand, the FWD/BWD part gets affected by the limited task-level parallelism and the slow matrix reorganization process in the PPE. Cores with stride memory access capabilities would be able to tackle this bottleneck by reducing the number of load operations required.

Medium to fine grained parallelization for PA does not greatly benefit from adding more cores as only *prfscore* allows scaling. FWD/BWD remain fixed to use two cores and other sequential parts in the PPE (like *calc-inputs* and *ptracepath*) become the bottlenecks. These phases suffer from the poor PPE ability to extract ILP.

Due the its computational complexity, *calc-inputs* dominates the PA time share for input sets of 200 or more sequences.

An interesting future work is to implement a coarser grain parallelization of PA in Cell, that is, at the tree level. In that case, available parallelism would largely depend on the tree structure, that is, on the relatedness of the input sequences. Furthermore, depending on the way the parallelization is implemented, the LS size and/or the PPE speed to coordinate the SPEs will be the likely bottlenecks.

Acknowledgment

This work was sponsored by the European Commission in the context of the SARC Integrated Project #27648 (FP6), the HiPEAC Network of Excellence, the FEDER funds under contract TIN2007-60625 and by the Spanish Ministry of Science. The authors would like to thank the Barcelona Supercomputer Center for the access to the Cell blades and tools.

References

[1] J. Cohen, “Bioinformatics-an introduction for computer scientists,” *ACM Computing Surveys*, pp. 122–158, 2004.
[2] J. Thompson, D. Higgins, and T. Gibson, “Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice,” *Nucleic Acids Research*, vol. 22, pp. 4673–4680, 1994.
[3] “Blast web server at ncbi,” http://blast.ncbi.nlm.nih.gov.
[4] “Fasta web server at embl-ebi,” http://www.ebi.ac.uk/Tools/fasta/.
[5] J. Kahle, M. Day, H. Hofstee, C. Johns, and D. Shippy, “Introduction to the cell multiprocessor,” *IBM Systems Journal*, vol. 49, no. 4/5, pp. 589–604, 2005.
[6] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, “Synergistic processing in cell’s multicore architecture,” *IEEE Micro*, pp. 10–24, 2006.
[7] J. Cheetham, F. K. H. A. Dehne, S. Pitre, A. Rau-Chaplin, and P. J. Taillon, “Parallel clustal w for pc clusters,” in *ICCSA (2)*, 2003, pp. 300–309.
[8] K. bin Li, “Clustalw-mpi: Clustalw analysis using distributed and parallel computing,” *Bioinformatics*, vol. 19, pp. 1585–1586, 2003.
[9] H. Vandierendonck, S. Rul, M. Questier, and K. D. Bosschere, “Accelerating multiple sequence alignment with the cell be processor,” *The Computer Journal - Advance Access*, Sept. 11, 2009.
[10] K. Chaichoompu, S. Kittitornkun, and S. Tongsima, “Mt-clustalw: multithreading multiple sequence alignment,” in *IPDPS*, 2006.
[11] S. Isaza, F. Sanchez, G. N. Gaydadjiev, A. Ramirez, and M. Valero, “Preliminary analysis of the cell be processor limitations for sequence alignment applications,” in *Proceedings of the 8th International Workshop SAMOS 2008*, July 2008.
[12] V. Sachdeva, M. Kistler, E. Speight, and T.-H. K. Tzeng, “Exploring the viability of the cell broadband engine for bioinformatics applications,” *Parallel Computing*, vol. 34, no. 11, pp. 616–626, 2008.
[13] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.
[14] F. Sanchez, A. Ramirez, and M. Valero, “Quantitative analysis of sequence alignment applications on multiprocessor architectures,” in *Conf. Computing Frontiers*, 2009, pp. 61–70.
[15] “Ebi - clustalw web server,” http://www.ebi.ac.uk/Tools/clustalw/.
[16] J. Thompson, D. Higgins, and T. Gibson, “Clustal w and clustal x version 2.0,” *Bioinformatics*, vol. 23, pp. 2947–2948, 2007.
[17] “Ebi - clustalw2 web server,” http://www.ebi.ac.uk/Tools/clustalw2/.
[18] N. Saitou and M. Nei, “The neighbor-joining method: A new method for reconstructing phylogenetics trees,” *Molecular Biology and Evolution*, vol. 4, no. 4, pp. 406–425, 1987.
[19] E. W. Meyers and W. Miller, “Optimal alignments in linear space,” *Bioinformatics*, vol. 4, pp. 11–17, 1988.