Long DNA Sequence Comparison on Multicore Architectures

Friman Sánchez[1], Felipe Cabarcas[2,3], Alex Ramirez[1,2], and Mateo Valero[1,2]

[1] Technical University of Catalonia, Barcelona, Spain
[2] Barcelona Supercomputing Center, BSC, Spain
[3] Universidad de Antioquia, Colombia
{fsanchez}@ac.upc.es, {felipe.cabarcas,alex.ramirez,mateo.valero}@bsc.es

Abstract. Biological sequence comparison is one of the most important tasks in Bioinformatics. Due to the growth of biological databases, sequence comparison is becoming an important challenge for high performance computing, especially when very long sequences are compared. The Smith-Waterman (SW) algorithm is an exact method based on dynamic programming to quantify local similarity between sequences. The inherent large parallelism of the algorithm makes it ideal for architectures supporting multiple dimensions of parallelism (TLP, DLP and ILP). In this work, we show how long sequences comparison takes advantage of current and future multicore architectures. We analyze two different SW implementations on the CellBE and use simulation tools to study the performance scalability in a multicore architecture. We study the memory organization that delivers the maximum bandwidth with the minimum cost. Our results show that a heterogeneous architecture is an valid alternative to execute challenging bioinformatic workloads.

1 Introduction

Bioinformatics is an emerging technology that is attractring the attention of computer architects, due to the important challenges it presents from the performance point of view. Sequence comparison is one of the fundamental tasks of bioinformatics and the starting point of almost all analysis that imply more complex tasks. This is basically an inference algorithm oriented to identify similarities between sequences. The need for speeding up this process is consequence of the continuous growth of sequence length. Usually, biologists compare long DNA sequences of entire genomes (coding and non-coding regions) looking for matched regions which mean similar functionality or conserved regions in the evolution; or unmatched regions showing functional differences, foreign fragments, etc.

Dynamic programming based algorithms (DP) are recognized as optimal methods for sequence comparison. The Smith-Waterman algorithm [16] (SW) is a well-known exact method to find the best local alignment between sequences. However, because DP based algorithm's complexity is $O(nm)$ (being n and m the length of sequences), comparing very long sequences becomes a challenging scenario. In such a case, it is common to obtain many optimal solutions, which

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part II, LNCS 6272, pp. 247–259, 2010.

can be relevant from the biological point of view. The time and space requirements of SW algorithms limit its use. As alternative, heuristics solutions have been proposed. FASTA [13] and BLAST [4] are widely used heuristics which allow fast comparisons, but at the expense of sensitivity. For these reasons, the use of parallel architectures that are able to exploit the several levels of parallelism existing in this workload is mandatory to get high quality results in a reduced time. At the same time, computer architects have been moving towards the paradigm of multicore architectures, which rely on the existence of sufficient thread-level parallelism (TLP) to exploit the large number of cores. In this context, we consider the use of multicore architectures in bioinformatics to provide the computing performance required by this workload. In this paper we analyze how large-scale sequence comparisons can be performed efficiently using modern parallel multicore architectures. As a baseline we take the IBM CellBE architecture, which has proved to be an efficient alternative for highly parallel applications [11][3][14]. We study the performance scalability of this workload in terms of speedup when many processing units are used concurrently in a multicore environment. Additionally, we study the memory organization that the algorithm requires and how to overcome the memory space limitation of the architecture. Furthermore, we analyze two different synchronization strategies.

This paper is organized as follows: Section 2 discusses related work on parallel alternatives for sequence comparison. Section 3 describes the SW algorithm and the strategy of parallelism. Section 4 describes the baseline architecture and presents two parallel implementations of the SW algorithm on CellBE. Section 5 describes the experimental methodology. Section 6 discusses the results of our experiments. Finally, section 7 concludes the paper with a general outlook.

2 Related Work

Researchers have developed many parallel versions of the SW algorithm [7][8], each designed for a specific machine. These works are able to find a short set of optimal solutions when comparing two very long sequences. The problem of finding many optimal solutions grows exponentially with the sequences length, becoming this a more complex problem. Azzedine et al [6] SW implementation avoids the excessive memory requirements and obtains all the best local alignments between long sequences in a reduced time. In that work, the process is divided in two stages: First, the score matrix is computed and the maximum scores and their coordinates are stored. Second, with this information, part of the matrix is recomputed with the inverses sequences (smaller than the original sequences) and the best local alignments are retrieved. The important point is that the compute time of the first stage is much higher than the needed in phase two. Despite the efforts to reduce time and space, the common feature is that the score matrix computation is required, which is still the most time-consuming part. There are some works about the SW implementations on modern multicore architectures. Svetlin [12] describes an implementation on the Nvidia's Graphics Processing Units (GPU). Sachdeva et al [15] present results on the use of the

CellBE to compare few and short pairs of sequences that fit entirely in the Local Storage (LS) of each processor. Sánchez [10] compares SW implementation on several modern multicore architectures like SGI Altix, IBM Power6 and CellBE, which support multiple dimension of parallelism (ILP, DLP and TLP).

Furthermore, several FPGAs and custom VLSI hardware solutions have been designed for sequence comparison [1][5]. They are able to process millions of matrix cells per second. Among those alternatives, it is important to highlight the Kestrel processor [5], which is a single instruction multiple data (SIMD) parallel processor with 512 processing elements organized as a systolic array. The system originally focuses on efficient high-throughput DNA and protein sequence comparison. Designers argue that although this is a specific processor, it can be considered to be in the midpoint of dedicated hardware and general purpose hardware due to its programmability and reconfigurable architecture.

Multicore architectures can deliver high performance in a wide range of applications like games, multimedia, scientific algorithms, etc. However, achieving high performance with these systems is a complex task: as the number of cores per chip and/or the number of threads per core increases, new challenges emerge in terms of power, scalability, design complexity, memory organization, bandwidth, programalibility, etc. In this work we make the following contributions:

- We implement the SW on the CellBE. However, unlike previous works, we focus on long sequences comparison. We present two implementations that exploit TLP and DLP. In the first one, the memory is used as a centralized data storage because the SPE LS is small to hold sequences and temporal data. In the second one, each SPE stores parts of the matrix in its own LS and other SPEs synchronously read data via DMA operations. It requires to handle data dependencies in a multicore environment, synchronization mechanisms between cores, on-chip and off-chip traffic management, double buffering use for hiding data communication latency, SIMD programming for extracting fine-grain data parallelism, etc.
- As a major contribution, we use simulation techniques to explore the SW performance scalability along different number of cores working in parallel. We investigate the memory organization that deliveres the maximum bandwidth with the minimum hardware cost, and analyze the impact of including shared cache that can be accessed by all the cores. We also study the impact of memory latency and the synchronization overhead on the performance.

3 Algorithm Description and Parallelism

The SW algorithm determines the optimal local alignment between two sequences of length l_x and l_y by assigning scores to each character-to-character comparison: positive for exact matches/substitutions, negative for insertions/deletions. The process is done recursively and the data dependencies are shown in figure 1a. The matrix cell (i, j) computation depends on results $(i-1, j)$, $(i, j-1)$ and $(i-1, j-1)$. However, cell across the antidiagonals are independent. The final score is reached when all the symbols have been compared. After

computing the similarity matrix, to obtain the best local alignment, the process starts from the cell which has the highest score, following the arrows until the value zero is reached.

3.1 Available Parallelism

Because most of the time is spent computing the score matrix, this is the part usually parallelized. The commonly used strategy is the wavefront method in which computation advances parallel to the antidiagonals. As figure 1a shows, the maximum available parallelism is obtained when the main antidiagonal is reached. Before developing an specific implementation, it is necessary to understand the parameters that influence performance. Figure 1c and table 1 illustrate these parameters and their description. The computation is done by blocks of a determined size. We can identify three types of relevant parameters: first, those which depend on the input data set, (the sequence lengths l_x and l_y); second, those which depend on the algorithm implementation like b, k (the vertical and horizontal block lengths); and third, those which depend on the architecture (the number of workers p and the time $T_{block(b,k)}$ required to compute a block of size $b*k$). There are studies on the parallelism in this kind of problems [2][9], Instead of developing a new model, we just want to summarize this remarking that the total time to compute the comparison can be expressed as follows:

$$Total\_time_{parallel} = T_{seq\_part} + T_{comp(b,k)} + T_{transf(b,k)} + T_{sync(b,k)} \quad (1)$$

Where T_{seq_part} is the intrinsic sequential part of the execution; $T_{comp(b,k)}$ is the time to process the matrix in parallel with p processors and with a specific block size $b * k$; $T_{transf(b,k)}$ is the time spent transferring all blocks with size $b * k$ used in the computation; and $T_{sync(b,k)}$ is the synchronization overhead. Each synchronization is done after a block of size $b * k$ is computed. On one hand $T_{comp(b,k)}$ basically depends on p, b and k, as the number of processors increases, this time decreases, The limit is given by the processors speed and the main antidiagonal, that is, if l_y and l_x are different, the maximum parallelism continues for $|l_x - l_y|$ stages and then decreases again. Small values of b and k increases the number of parallel blocks, making the use of a larger number of processor effective. On the contrary, larger values of b and k reduce parallelism, therefore $T_{comp(b,k)}$ increases. On the other hand, $T_{sync(b,k)}$ also depends on b, k. Small values of them increase the number of synchronization events, which can degrade performance seriously. Finally, $T_{transf(b,k)}$ increases with large values of b and k but also with very small values of them. The latter situation happens because it increases the number of inefficient data transfers due to the size.

4 Parallel Implementations on a Multicore Architecture

Comparing long sequences presents many challenges to any parallel architecture. Many relevant issues like synchronization, data partition, bandwidth use, memory space and data organization should be studied carefully to efficiently use the available features of a machine to minimize equation 1.

Table 1. Parameters involved in the execution of the SW implementation on CellBE

Name	Description
b	Horizontal block size (in number of symbols (bytes))
k	Vertical block size (in number of symbols (bytes))
l_x	Length of sequence in the horizontal direction
l_y	Length of sequence in the vertical direction
p	Number of processors (workers), SPE is the case of CellBE
$T_{block(b,k)}$	Time required to process a block of size $b * k$

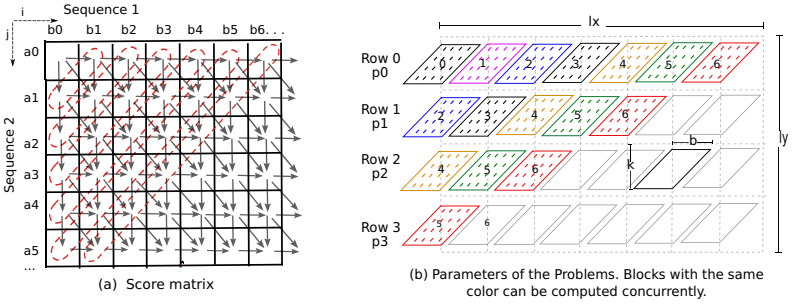

Fig. 1. (a) Data dependency (b) Different optimal regions (c) Computation distribution

To exploit TLP in the SW, a master thread takes sequences and preprocess them: makes profiling computation according to a substitution score matrix, prepares the worker execution contexts and receives results. Those issues correspond to the sequential part of the execution, the first term of equation 1. Workers compute the similarity matrix as figure 1b shows, that is, each worker computes different rows of the matrix. For example, if $p = 8$, p_0 computes row 0, row 8, row 16, etc; p_1 computes row 1, row 9, row 17, and so on. Since SIMD registers of the workers are 16-bytes long, it is possible to compute 8 symbols in parallel, (having 2 bytes per temporal scores), that is, $k = 8$ symbols. Each worker has to store temporal matrix values which will be used by the next worker, for example, in figure 1b, computing block 2 by p_0 generates temporal data used in the computation of block 1 by p_1. This feature leads to several possible implementations. In this work, we show two, both having advantages and disadvantages, and being affected differently by synchronization and communications.

4.1 Centralized Data Storage Approach

Due to the small size of the scratch pad memory (LS of 256KB in CellBE, shared between instructions and data), a buffer per worker is defined in memory to store data as figure 2a shows. Each worker reads from its own buffer and write to the next worker's buffer via DMA GET and PUT operations. Shared data correspond to the border of consecutive rows as shown in figure 1b. That implies that all workers are continuously reading/writing from/to memory, which is a possible problem from the BW point of view, but it is easy to program. The $T_{transf(b,k)}$

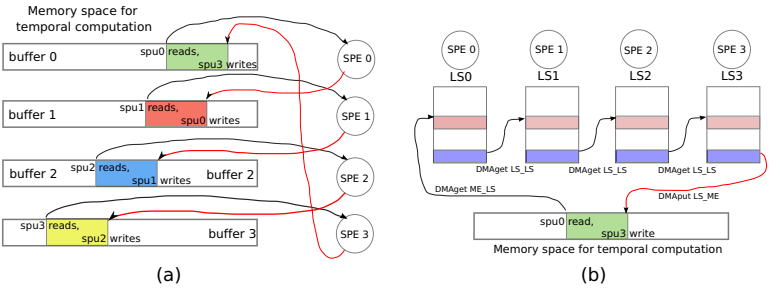

Fig. 2. (a) SPEs store data in memory. (b) SPEs store data in an internal buffer.

term of equation 1 is minimized using double buffering. It reduces the impact of DMA operation latency, overlapping computation with data transfer. Atomic operations are used to synchronize workers and guarantee data dependencies.

4.2 Distributed Data Storage Approach

Here, each worker defines a small local buffer in its own LS to store temporal results (figure 2b). When p_i computes a block, it signals p_{i+1} indicating that a block is ready. When p_{i+1} receives this signal, it starts a DMA_GET operation to bring data from the LS of p_i to its own LS. When data arrives, p_{i+1} handshakes p_i sending an `ack` signal. Once p_i receives this signal, it knows that the buffer is available for storing new data. The process continues until all blocks are computed. However, due to the limited size of LS, the first and the last workers in the chain read from and write to memory. This approach reduces data traffic generated in previous approach. However it is more complex to program.

There are three types of DMA: between workers to transfer data from LS to LS (on-chip traffic), from memory to LS and from LS to memory (off-chip traffic). Synchronization overhead is reduced taking into account that one SPE does not have to wait immediately for the `ack` signal from other SPE, it can start the computation of the next block and later wait for the `ack` signal. Synchronization is done by using the signal operations available in the CellBE.

5 Experimental Methodology

As a starting point, we execute both SW implementations on the CellBE with 1 to 16 SPUs. We evaluate the performance impact of the block size and the bandwidth requirements. Then, using simulation, we study the performance scalability and the impact of different memory organizations. We obtain traces from the execution to feed the architecture simulator (TaskSim) and carry out an analysis in more complex multicore scenarios. TaskSim is base on the idea that, in distributed memory architectures, the computation time on a processor does not depend on what is happening in the rest of the system, as it happens on the CellBE. Execution time depends on inter-thread synchronization, and

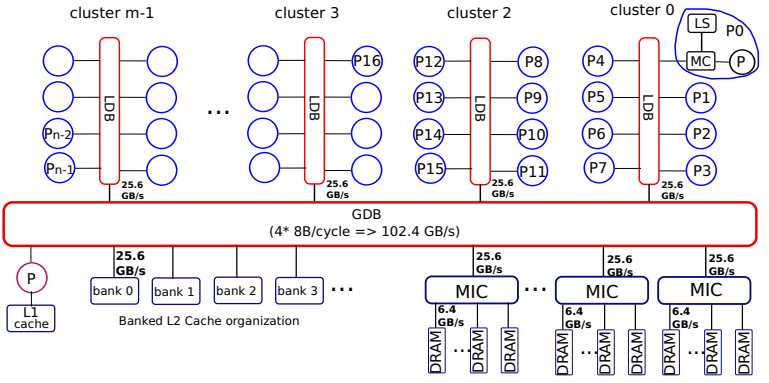

Fig. 3. Modeled system

Table 2. Evaluated configurations ranked by L2 bandwidth and Memory bandwidth

L2 Cache Organization	Number of Banks	1	2	4	8	16
	Bandwidth [GB/s]	25.6	51,2	102.4	204.8	409.6
Memory Organization	mics/dram per mics	1/1	1/2 or 2/1	1/4 or 4/1	2/4 or 4/2	4/4
	Bandwidth [GB/s]	6.4	12.8	25.6	51.2	102.4
Each combination of L2 BW and memory BW is a possible configuration, e.i, 2 L2 banks and 2 mics with 2 dram/mic deliver 51.2 GB/s to L2 and 25.6 GB/s BW to memory						

the memory system for DMA transfers. TaskSim models the memory system in cycle-accurate mode: the DMA controller, the interconnection buses, the Memory Interface Controller, the DRAM channels, and the DIMMs. TaskSim does not model the processors themselves, it relies on the computation time recorded in the trace to measure the delay between memory operations (DMAs) or interprocessor synchronizations (modeled as blocking semaphores).

As inputs, we use sequences with length 3.4M and 1.8M symbols, for real execution. However, to obtain traces with manageable size for simulation, we take shorter sequences ensuring available parallelism up to 112 workers, using block transfer of 16KB. The code running on PPU and SPU side are coded in C and were compiled with ppu-gcc and spu-gcc 4.1.1 respectively, with -O3 option. The executions run on a IBM BladeCenter QS20 system composed by 2 CellBE at 3.2 GHz. As a result, it is possible to have up to 16 SPEs running concurrently.

5.1 Modeled Systems

Figure 3 shows the general organization of the evaluated multicore architectures using simulation. It is comprised of several processing units integrated into clusters, the working frequency is 3.2 GHz. The main elements are:

- A control processor P: a PowerPC core with SIMD capabilities.
- Accelerators: 128 SIMD cores, connected into clusters of eight core each. Each core is connected with a private LS and a DMA controller.

- A Global Data Bus (GDB) connects the LDBs, L2 cache and memory controllers. It allows 4 8-bytes request per cycle, with 102.4 GB/s of bandwidth.
- Local Data Buses (LDB) connects each cluster to GDB. Each GDB to LDB connection is 8-bytes/cycle. (25,6 GB/s).
- A shared L2 cache distributed into 1 to 16 banks as table 2 describes. Each bank is a 8-way set associative with size ranging from 4KB to 4MB.
- On-chip memory interface controllers (MIC) connect GDB and memory, providing up to 25.6 GB/s each (4x 6.4 GB/s Multi-channel DDR-2 modules).

Our baseline blade CellBE-like machine consists of: 2 clusters with 8 workers each, without L2 cache, one GDB providing a peak BW of 102.4 GB/s for on-chip data transfer. One MIC providing a peak BW of 25.6 GB/s to memory. And four DRAM modules connected to the MIC with 6.4 GB/s each.

6 Experimental Results

6.1 Speedup in the Real Machine

Figures 4 and 5 show performance results for the centralized and distributed approaches on CellBE. The baseline is the execution with one worker. Figures show the performance impact of the block size (parameter b of table 1). When using small block sizes (128B or 256B), the application does not exploit the available parallelism due to two reasons: First, although more parallel blocks are available, the number of inefficient DMAs increases. Second, because each block transfer is synchronized, the amount of synchronization operations also increases and the introduced overhead degrades performance. With larger blocks like 16KB, the parallelism decreases, but the synchronization overhead decreases. Less aggressive impact is observed in the distributed SW because the synchronization mechanism is direct between workers and data transfers are done directly between LSs. With blocks of 16KB, performance of both approaches is similar (14X for centralized and 15X for distributed with 16 workers), that is, synchronization and data transfer in both approaches are hidden by computation.

6.2 Bandwidth Requirements

Data traffic is measured in both centralized and distributed cases. With these results, we perform some estimation of the on-chip and off-chip BW requirements, dividing the total traffic by the time to compute the matrix. Figure 6 depicts results of these measures. Up to 16 workers, the curves reflect the real execution on CellBE, for the rest points, we made a mathematical extrapolation that gives some idea of the BW required when using more workers. Figure shows that centralized SW doubles the BW requirements of distributed case when using equal number of workers. The reason is that in the centralized case, a worker sends data from a LS to memory first, and then, another worker bring this data from memory to its own LS, besides, all the traffic is off-chip. In the distributed case, data travels only once: from a LS to another LS and most of the traffic

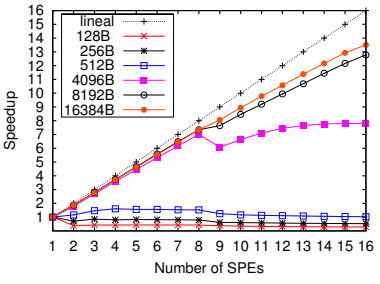

Fig. 4. Centralized SW implem.

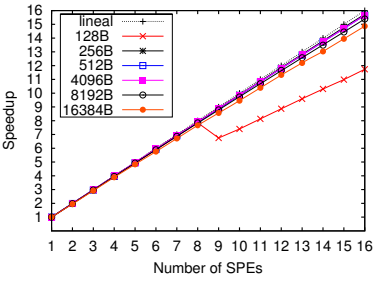

Fig. 5. Distributed SW implem.

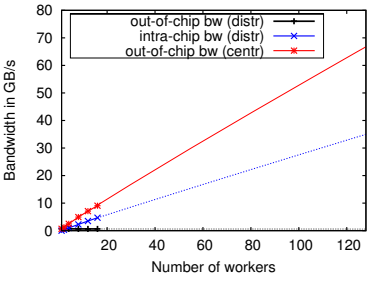

Fig. 6. Bandwidth requirements

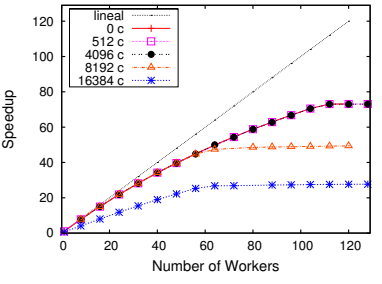

Fig. 7. Memory latency impact, cent.

is on-chip. For example, for 16 workers off-chip BW arrives to 9.1 GB/s in the first case, and on-chip BW is around 4.7 GB/s in the second one. Although the current CellBE architecture can deliver these BW for 16 cores, it is clear this demand is unsustainable when more than 16 workers are used, as figure shows.

6.3 Simulation Results

This section presents simulation results according to the configurations of section 3. We show results for both SW implementations using 16KB of block size. We study the memory latency with perfect memory, the real memory system impact, inclusion of L2 cache impact and the synchronization overhead impact.

Memory Latency Impact. We perform experiments using up to 128 cores, without L2 cache, with sufficient memory bandwidth, with different latencies in a perfect memory and without synchronization overhead. Figure 7 shows results for the centralized SW. The first observation is that even in the ideal case (0 cycles), the execution does not reach a linear performance. This is because of Amdahl's law: with 1 worker, the sequential part of the execution takes 0.41% of time, but with 128 workers it takes around 30.2% of time. Figure also shows this implementation hides the latency properly due to the double buffering use. The degratation starts with latencies near to 8K cycles, when more than 64 workers are used. Finally, the performance does not increase with more than 112 cores

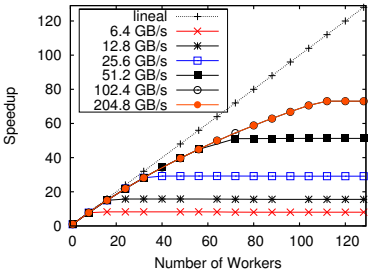

Fig. 8. Memory BW, centralized

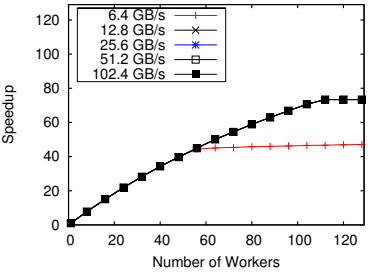

Fig. 9. Memory BW, distributed

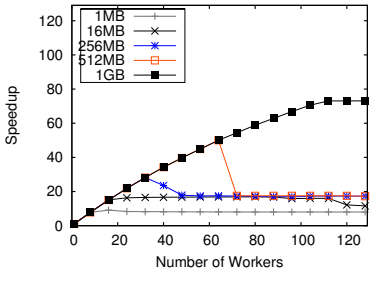

Fig. 10. Cache Sizes, 6.4 Gb/s BW, cent.

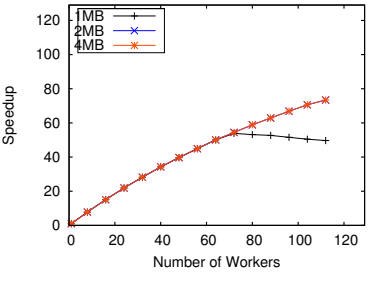

Fig. 11. Cache Sizes, 6.4 Gb/s BW, dist.

because the simulated trace only has parallelism for this amount of cores, as explained in section 5. Results for the distributed case exhibit similar behavior.

Real Memory System Without L2 Cache. Figures 8 and 9 show the performance results when using several combinations of MICs and DRAMS per MIC, without L2 caches. As shown, having 128 workers in the centralized case, the required BW to obtain the maximum performance is between 51.2 GB/s (2 MICS and 4 DRAM/MIC) and 102.4GB/s (4 MICS and 4 DRAMS/MIC). Comparing the results with the extrapolation of figure 6, we conclude that the required BW with 128 workers is near to 65 GB/s. However, having some configurations like them is unrealistic because of that physical connections do not scale in this way. For the distributed case, 12.8 GB/s (1 MIC and 2 DRAMS/MIC) is sufficient to obtaining the maximum performance. This is because the off-chip traffic in this case is very small (figure 6). Basically all the traffic is kept inside the chip.

Impact of the L2 Cache and Local Storage. There are several ways to include cache or local memory in the system. We evaluate two options: first, adding a bank-partitoned L2 cache connected to the GDB; second, adding small size of LS to each worker. These two models differ in the way data locallity and interprocessor communication is managed. Figure 10 shows results for the centralized case in which only one MIC and one DRAM module is used (6.4

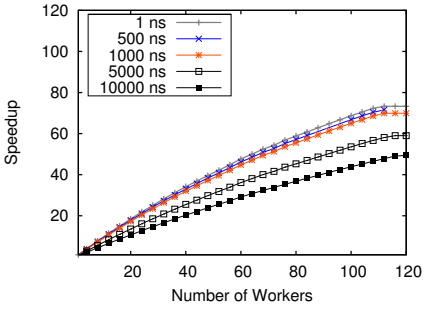

Fig. 12. Synchronization Overhead

GB/s of memory BW) and a maximum 204.5 GB/s of L2 BW is available (L2 is distributed in 8 banks). As shown, the cache requirement is very high due to that the matrix to compute is bigger than L2 and data reuse is very small: when a block is computed, data is used once for another worker, after that, it is replaced by a new block. Additionally, with many workers the conflict misses increase significantly with small L2 caches, therefore, the miss rate increases and performance degrades. Figure 11 show results for the distributed case, where each worker has a 256KB LS (as CellBE) and there is L2 cache distributed in 8 banks to access date that are not part of the matrix computation. Results show that a shared L2 cache of 2MB is enought to capture the on-chip traffic.

Synchronization Overhead. To obtain small synchronization overhead it is required to used a proper synchronization technique that match well in the target machine. So far, we have made experiments regardless of synchronization overhead. Now, this is included in the performance analysis. Each time a worker computes a block, it communicates to another worker that data is available, as explained in section 4.1. We include this overhead assuming that the time to perform a synchronization event (signal o wait) after a block is computed is a fraction of the requirede time to compute it, that is, $T_{sync_block(b,k)} = \alpha * T_{block(b,k)}$. We give this information to our simulator and measure the performance for different values of α. Figure 12 shows the experiment results for the centralized SW implementation. As observed, the system assimilate the impact of up to 1000 nanoseconds of latency in each synchronization event. The results of the distributed approach exhibit a similar behavior.

7 Conclusions

This paper describes the implementation of DP algorithms for long sequences comparisons on modern multicore architectures that exploit several levels of parallelism. We have studied different SW implementations that efficiently use the CellBE hardware and achieve speedups near to linear with respect to the number of workers. Furthermore, the major contribution of our work is the use

of simulation tools to study more complex multicore configurations. We have studied key aspects like memory latency impact, efficient memory organization capable of delivering maximum BW and synchronization overhead. We observed that it is possible to minimize the memory latency impact by using techniques like double buffering while large data blocks are computed. Besides, we have shown that due to the sequential part of the algorithm, the performance does not scale linerly with large number of workers. It becomes necessary to perform optimizations in the sequential part of the SW implementations.

We investigated the memory configuration that deliveres maximum BW to satisfy tens and even hundreds of cores on a single chip. As a result, we determined that for the SW algorithm it is more efficient to distribute small size of LS accross the workers instead of having a shared L2 on-chip data cache connected to the GDB. This is consequence of the streaming nature of the application, in which data reuse is low. However, the use of LS makes more challenging the programming, because the communication is always managed at the user-level.

Finally, we observed that our synchronization strategy minimizes the impact of this overhead because it prevents a worker to wait inmediatly for the response of a previous signal. In this way, the application can endure an overhead of up to thousand ns with a maximum performance degradation of around 3%.

Acknowledgements

This work was sponsored by the European Commission (ENCORE Project, contract 248647), the HiPEAC Network of Excellence and the Spanish Ministry of Science (contract TIN2007-60625). Program AlBan (Scholarship E05D058240CO).

References

1. Fast data finder (fdf) and genematcher (2000), `http://www.paracel.com`
2. Aji, A.M., Feng, W.c., Blagojevic, F., Nikolopoulos, D.S.: Cell-swat: modeling and scheduling wavefront computations on the cell broadband engine. In: CF 2008: Proceedings of the 5th conference on Computing frontiers, pp. 13–22. ACM, New York (2008)
3. Alam, S.R., Meredith, J.S., Vetter, J.S.: Balancing productivity and performance on the cell broadband engine. In: IEEE International Conference on Cluster Computing, pp. 149–158 (2007)
4. Altschul, S.F., Madden, T.L., Schffer, A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.: Gapped blast and psi-blast: a new generation of protein database serach programs. Nucleic acids research 25, 3389–3402 (1997)
5. Blas, A.D., Karplus, K., Keller, H., Kendrick, M., Mesa-Martinez, F.J., Hughey, R.: The ucsc kestrel parallel processor. IEEE Transactions on Parallel and Distributed systems (January 2005)
6. Boukerche, A., Magalhaes, A.C., Ayala, M., Santana, T.M.: Parallel strategies for local biological sequence alignment in a cluster of workstations. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS. IEEE Computer Society, Los Alamitos (2005)

7. Boukerche, A., Melo, A.C., Sandes, E.F., Ayala-Rincon, M.: An exact parallel algorithm to compare very long biological sequences in clusters of workstations. Cluster Computing 10(2), 187–202 (2007)
8. Chen, C., Schmidt, B.: Computing large-scale alignments on a multi-cluster. In: IEEE International Conference on Cluster Computing, vol. 38 (2003)
9. Edmiston, E.E., Core, N.G., Saltz, J.H., Smith, R.M.: Parallel processing of biological sequence comparison algorithms. Int. J. Parallel Program. 17(3) (1988)
10. Friman, S., Ramirez, A., Valero, M.: Quantitative analysis of sequence alignment applications on multiprocessor architectures. In: CF 2009: Proceedings of the 6th ACM conference on Computing frontiers, pp. 61–70. ACM, New York (2009)
11. Gedik, B., Bordawekar, R.R., Yu, P.S.: Cellsort: high performance sorting on the cell processor. In: VLDB 2007: Proceedings of the 33rd international conference on Very large data bases, pp. 1286–1297, VLDB Endowment (2007)
12. Manavski, S.A., Valle, G.: Cuda compatible gpu cards as efficient hardwarer accelerator for smith-waterman sequence alignment. BMC Bioinformatics 9 (2008)
13. Pearson, W.R.: Searching protein sequence libraries: comparison of the sensitivity and selectivity of the smith-waterman and FASTA algorithms. Genomics 11 (1991)
14. Petrini, F., Fossum, G., Fernández, J., Varbanescu, A.L., Kistler, M., Perrone, M.: Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In: IPDPS, pp. 1–10 (2007)
15. Sachdeva, V., Kistler, M., Speight, E., Tzeng, T.H.K.: Exploring the viability of the cell broadband engine for bioinformatics applications. In: Proceedings of the 6th Workshop on High Performance Computational Biology, pp. 1–8 (2007)
16. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of Molecular Biology 147, 195–197 (1981)