International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:2, 2008

Using the PGAS Programming Paradigm for Biological Sequence Alignment on a Chip Multi-Threading Architecture

M. Bakhouya, S. A. Bahra, and T. El-Ghazawi

***Abstract*—The Partitioned Global Address Space (PGAS) programming paradigm offers ease-of-use in expressing parallelism through a global shared address space while emphasizing performance by providing locality awareness through the partitioning of this address space. Therefore, the interest in PGAS programming languages is growing and many new languages have emerged and are becoming ubiquitously available on nearly all modern parallel architectures. Recently, new parallel machines with multiple cores are designed for targeting high performance applications. Most of the efforts have gone into benchmarking but there are a few examples of real high performance applications running on multicore machines. In this paper, we present and evaluate a parallelization technique for implementing a local DNA sequence alignment algorithm using a PGAS based language, UPC (Unified Parallel C) on a chip multithreading architecture, the UltraSPARC T1.

***Keywords*—Partitioned Global Address Space, Unified Parallel C, Multicore machines, Multi-threading Architecture, Sequence alignment.

I. Introduction

Sequence comparison and sequence database searching have played a crucial role in biological research advancements. The massive computational resources required for the search and the comparison of sequence databases represents today a big challenge for biologists [13]. To meet this requirement, many parallel computation methods have been developed on high performance computing systems. Most of these applications are implemented for distributed memory architectures using the message-passing paradigm [11].

Recently, new parallel machines and processor architectures have been introduced for throughput computing. One of these most successful introductions come in the form of chip multi-threading architectures. For example, the Sun Fire T2000 [8] contains a T1 processor with eight processing cores each with four hardware threads. However, most of the efforts have gone into benchmarking the real world performance of enterprise I/O-bound applications [8],[17], but there are a few examples of compute-centric application analysis for this architecture [15],[14]. Despite the short-comings of this architecture in computational through-put, problems with a fair share of both communication and computational requirements may exploit the high scalability the architecture provides. In addition, to the best of our knowledge, there are no such applications implemented using PGAS languages and optimized for these machines. Therefore, it is interesting to study a PGAS implementation of these applications for this novel architecture. We have selected the local sequence alignment application, since it represents one of high performance computings most demanding applications with a fair share of both communication and computational requirements.

Recall that the PGAS model supports a programming model in which programmers design and write high level descriptions of their algorithms. UPC is one of partitioned global address space programming language based on C and extended with global pointers and data distribution declarations for shared data [1] [18] [19]. A number of threads in UPC can work independently in a Simple Program Multiple Data (SPMD) model. Threads communicate through the shared memory and can access shared data while a private object may be accessed only by its own thread. The UPC memory model supports three different kinds of pointers: private pointers pointing to the shared address space, pointers living in shared space that also point to shared data, and private pointers pointing to data in the threads own private space [1]. UPC provides also synchronization mechanisms that are barriers, split-phase barriers and memory consistency control which may increase delays incurred by waiting jobs. According to these UPC features, the fine-grained UPC programming model is simple and easy to use. In other words, the advantage of PGAS programming model is that programmers need to only specify the data to be distributed across processors, and reference them through special global pointers during processing [5],[20]. However, UPC performance largely depends on the number of threads and how they are accessing the shared space [20].

In this paper, we present an implementation of a PGAS algorithm for local sequence alignment on chip multi-threading architectures, in particular the Sun T1. More precisely, we show how the proposed PGAS algorithm, a Smith-Waterman derivative, for sequence alignment can benefit from this type of multicore architecture.

The rest of the paper is organized as follows. In section 2, we present the basic concepts of Smith-Waterman algorithm. Section 3 presents a description of the PGAS-based algorithm for local sequence alignment. Experimental results are given in section 4. Conclusions and future work are presented in section 5.

II. Smith-Waterman Algorithm

Local sequence alignment is one of the most important problems in bioinformatics. Two known algorithms based

M. Bakhouya, S. A. Bahra, and T. El-Ghazawi are with the Department of Electrical and Computer Engineering, High Performance Computing Laboratory, The George Washington University, email: bakhouya@gmail.com, {Bahra,tarek}@gwu.edu

	- 0	G 1	A 2	T 3	C 4	G 5	G 6	A 7	A 8	T 9	A 10	G 11
- 0	0	0	0	0	0	0	0	0	0	0	0	0
G 1	0	**2**	1	0	0	2	2	1	0	0	0	2
A 2	0	1	**4**	**3**	2	1	1	4	3	2	2	1
C 3	0	0	3	3	**5**	4	3	3	3	2	1	1
G 4	0	2	2	2	4	**7**	6	5	4	3	2	3
G 5	0	2	1	1	3	6	**9**	8	7	6	5	4
A 6	0	1	4	3	2	5	8	**11**	10	9	8	7
T 7	0	0	3	6	5	4	7	10	**10**	12	11	10
T 8	0	0	2	5	5	4	6	9	9	**12**	11	10
A 9	0	0	2	4	4	4	5	8	11	11	**14**	13

Fig. 1. The score matrix of the sequences X and Y, where the highest score is 14.

on dynamic programming techniques have been proposed to determine the maximal alignment of two sequences: the global alignment introduced by Needleman and Wunsch [3] and the local alignment algorithm introduced by Smith and Waterman [4]. These algorithms are used to determine a score that represents the degree of similarity between two sequences. The global alignment algorithm computes a similarity score between two sequences as the sum of all individual elementary similarities. The local alignment algorithm however, finds the most similar subsequences of two sequences. This algorithm is based on dynamic programming techniques on two phases. The first phase computes the total score that indicates the degree of the similarity by building a score matrix. The second phase, called tracing back phase, identifies the corresponding alignment using the score matrix built in the first phase. Formally, the local alignment algorithm takes two input sequences $X(1, ..., m-1)$ and $Y(1, ..., n-1)$, which their elements are in an alphabet that contains s symbols. For example, proteins are sequences of 20 different letters (amino acids), and DNA sequences can be represented by sequences of 4 letters (i.e., s=4) [6]. To facilitate the representation, X and Y can be represented as one-dimensional table of size m and n respectively, where $X(0) = Y(0) = Nil$. The similarity matrix C is a two-dimensional table of size and each element $C(i, j)$ can be computed as follows [16],[2],[12].

$$C(i,j) = max \left\{ \begin{array}{l} C(i-1, j-1) + sbt(X(i), Y(j)) \\ C(i, j-1) + \sigma \\ C(i-1, j) + \sigma \\ 0 \end{array} \right.$$

Where $sbt(X(i), Y(j)$ is a substitution cost related to the similarity of the character $X(i)$ and $Y(j)$. The building phase starts with the $C(1, 1)$ and the continuously searches stops at $C(m-1, n-1)$. During this phase, the elements of C are calculated row by row, left to right on each row using the formula described above. To illustrate this phase, an example to compute the score matrix of the sequence $X = GACGGATTA$ and the sequence $Y = GATCGGAATAG$ is given in figure 1. In this example, we consider that $\sigma = -1$ and the substitution cost is equal +2 if the characters $X(i)$ and $Y(j)$ are identical and -1 otherwise. The first row and the first column are initialized to 0 before starting the computation.

This first process cannot tell us what the corresponding alignment actually is, but only gives the highest score, C(9,10)=14. The second phase starts from the cell that has the highest score and traces back to generate the alignment of X and Y. This process continues until a position in column 0 or row 0, or $C(i, j) = 0$ is reached. Two lists XL and YL of size m and n respectively are used to store the computed corresponding alignment. The corresponding alignment, in this example, is represented as the bold text in the figure 1, i.e., $XL = GACGGATTA$ and $YL = GATCGGAATA$.

III. PGAS Parallel Local Sequence Alignment

In this section, a parallel PGAS algorithm for computing the similarity between two sequences X and Y is presented. Two phases are considered in the algorithm: the score matrix building phase that computes the score matrix C and the back-tracing phase that finds the corresponding alignment. When the first phase finishes building the matrix and determines the cell $C(i, j)$ having the highest score, the second phase starts from this cell to determine an actual corresponding alignment by examining the elements of this matrix.

In the first phase, to compute the value $C(i, j)$, the value of the upper cell $C(i-1, j)$, the left $C(i, j-1)$ and the upper-left $C(i-1, j-1)$ should be checked. This strong dependency leads to a regular parallelism. The algorithm largely used to paralyze the building phase is known as the wave-front algorithm [10]. This algorithm uses the BSP/CGM (Bulk Synchronous Parallel/Coarse Grained Multicomputer) model. This algorithm has been proposed mainly for string editing problem with the main goal to minimize the communication complexity of parallel searches. An extension of this algorithm for biological sequence comparison is proposed in [11]. This extended algorithm, used mainly to build the score matrix, is based on a compromise between the workload of each processor and the number of communication rounds required using a parameterized scheduling scheme. Recall that the BSP/CGM computer model consists of a set of p processors with $\frac{n}{p}$ local memory per processor, where n is the space needed by the sequential algorithm [9]. These processors are connected through any interconnection network and communicate by sending messages in a point-to-point manner. BSP/CGM computational model can be also used to predict the performance of the algorithms when implemented on a distributed memory machine [7].

The algorithm proposed in this paper takes two inputs, the sequence Y of size $n-1$ and the sequence X of size $m-1$, and produces a score matrix C of size $(n \times m)$. This matrix is considered as an input to the second phase that produces the corresponding alignment of X and Y as an output. Unlike the algorithm presented in [11] that builds a score matrix of size $(m \times n)$, in the proposed algorithm we use the transpose of this matrix by using one of size $(n \times m)$ and we add another factor that allows the reduction of the load imbalance between threads. Two mechanisms are used for data and work distribution. In the data distribution, we consider that the sequence X is local to each thread, i.e., each thread has a copy of X. The sequence Y is distributed in round robin manner to all threads. More precisely, this distribution assigns $\frac{\alpha n}{T}$ elements of Y to each threads, where T is the number of threads or processors and α is an adjustment factor that allows

T_0^0	T_0^1	. . .	T_0^j	. . .	$T_0^{T/\beta-1}$
T_1^1	T_1^2		T_1^{j+1}		$T_0^{T/\beta}$
⋮	⋮	⋮	⋮ $m\beta/T$	⋮	⋮
T_i^i	T_i^{i+1}	. . $n\alpha/T$	T_i^{i+j}		$T_i^{i+T/\beta-1}$
⋮	⋮	⋮	⋮	⋮	⋮
$T_{T-1}^{T/\alpha-1}$	$T_{T-1}^{T/\alpha}$	. . .	$T_{T-1}^{T/\alpha+j-1}$	. . .	$T_{T-1}^{T/\alpha+T/\beta-2}$

Fig. 2. A general distribution scheme of the workload between threads.

the load balancing between threads as described in [10]. The matrix C is distributed, in round robin manner, by blocks of rows. More precisely, this distribution assigns $\frac{\alpha nm}{T}$ elements to each thread. We consider that each thread can be executed in one processor with $O(\frac{nm}{p})$ memory space, where p is the number of processors.

In the workload distribution scheme between threads, instead of using a block size of $\frac{\alpha nm}{T^2}$, used in wavefront algorithm, we have worked with a variable block size of $\frac{\alpha\beta nm}{T^2}$, where T is the number of threads executed on p processors ($T = p$), $0 < \alpha, \beta \leq 1$ (see figure 2). In this distribution scheme, we assume that ($\alpha\beta nm$ mod T^2) is equal to 0. We can see for example that in the first steep only the thread T_0 computes the sub-matrix, threads T_0 and T_1 in the second steep. This phase continues until all matrix elements will be computed.

To illustrate this algorithm, we use the same example described above that computes the score matrix of the sequence $Y = GATCGGAATAG$ and the sequence $X = GACGGATTA$. Let us consider, for example, that the number of threads is equal to 2 ($T = 2$), $\alpha = 2$ and $\beta = 1$. The figure 3 shows that only the thread starts computing its first block of size (3×5) elements. After that, T_0 and T_1 can compute in parallel next blocks. When the thread T_0 finishes processing its last block of the first band, it can starts computing the first block of the second band. This process continues until all elements of C will be computed, i.e., the last block that will be processed by the thread ($T = 1$). Note that the elements of this matrix are note necessarily similar to the elements of the matrix used usually in sequential and parallel wavefront algorithms, but the obtained result is similar (i.e., the highest score and the corresponding alignment are similar).

According to this data and workload distribution scheme, we can see that large values of α and β tend to increase the idle time of the threads except for the thread T_0 which starts immediately. It is notable that small values of α will incur huge communication requests in distributed systems. However, this is not relevant on multicore architectures which have negligible communication overhead when communicating over local memory. A major factor here in order to optimize application performance especially on shared cache multicore architectures is cache utilization. As such, α and β are analyzed in order to deduct the best values for fair cache utilization.

After building the score matrix, all threads can compute

		- 0	G 1	A 2	C 3	G 4	G 5	A 6	T 7	T 8	A 9
	- 0	0	0	0	0	0	0	0	0	0	0
T0	G 1	0	**2**	1	0	2	2	1	0	0	0
	A 2	0	1	**4**	3	2	1	4	3	2	2
	T 3	0	0	**3**	3	2	1	3	6	5	4
T1	C 4	0	0	2	**5**	4	3	2	5	5	4
	G 5	0	2	1	4	**7**	6	5	4	4	4
	G 6	0	2	1	3	6	**9**	8	7	6	5
T0	A 7	0	1	4	3	5	8	**11**	10	9	8
	A 8	0	0	3	3	4	7	10	**10**	9	11
	T 9	0	0	2	2	3	6	9	12	**12**	11
T1	A 10	0	0	2	1	2	5	8	11	11	**14**
	G 11	0	2	1	1	3	4	7	10	10	13

Fig. 3. The score matrix of X and Y, the highest score is c(10,9)=14.

locally in round robin manner a part of the matrix of size $\frac{\alpha nm}{T}$, but in sequential manner as follows. The thread T_{k-1} starts computing at the cell having the highest score and stops when the next row to process is not local to it or the column 0 is reached. When one of these conditions is reached, T_{k-1} activates the thread T_{k-2} by giving it the next column to be computed. The thread T_{k-2} can start computing from its last row, processed during the building phase, and the column given by T_{k-1}. This process is repeated until at least one of the following conditions is reached, a cell $C(i, j)$ having the value 0, the row 0, or the column 0. For example, figure 3 shows the corresponding path found (elements with bold font) between the two sequences X and Y. The corresponding alignment between X and Y are the subsequences $XL = GACGGATT$ and $YL = GATCGGAATA$.

IV. Experimental Results

The PGAS-based algorithm to perform the longest common subsequence is implemented using the Berkeley UPC running on the Sun Fire T2000 server. The figure 4 shows a general architecture used by multicore machines. A number of processing cores integrated on to a single chip. These cores have their own private L1 cache and share a common L2 cache. The bandwidth between the L2 cache and main memory is shared by all the processing cores [15]. The Sun Fire T2000 server, used in our experiments, has eight cores running at 1.2GHz, each with four hardware threads. Each processing core has its own private L1 cache and shares L2 cache with the other cores. The bandwidth between L2 and the main memory is shared by all processing cores [8].

In the experiments, two sequences X and Y are considered (with sizes 1536 and 2304 respectively). The elements of these sequences are generated randomly from the alphabet $\Phi = \{A, C, G, T\}$. Speedups achieved on the Sun Fire T2000 for computing the local alignment of X and Y, with 1, 2, 4, 8, 16, and 24 threads are shown in figure 5. We can see that the algorithm is sensitive to the variation of α. For example, the speedup increases up to 8 threads, plateaux between 8 and 16 threads, but decrease between 16 and 24 threads. This is because decreasing the value leads to a decrease of the synchronization time and an increase of the

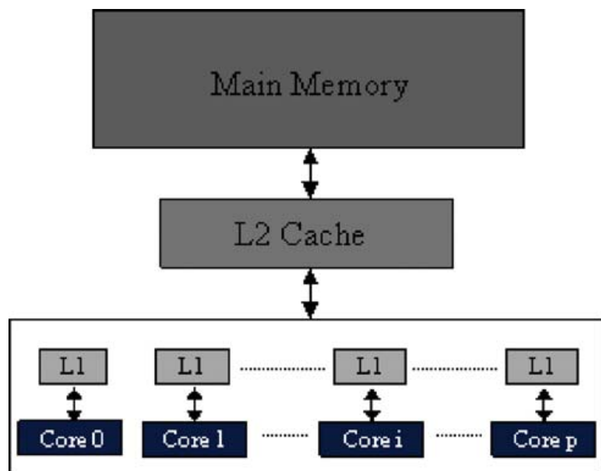

Fig. 4. A general architecture for multicore systems with p cores.

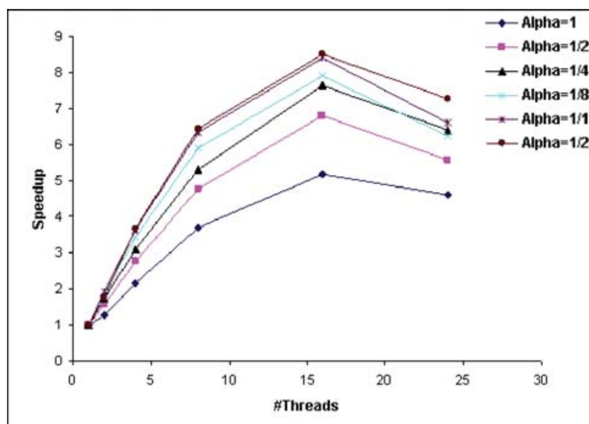

Fig. 5. The speedup with number of threads and varying the value of α.

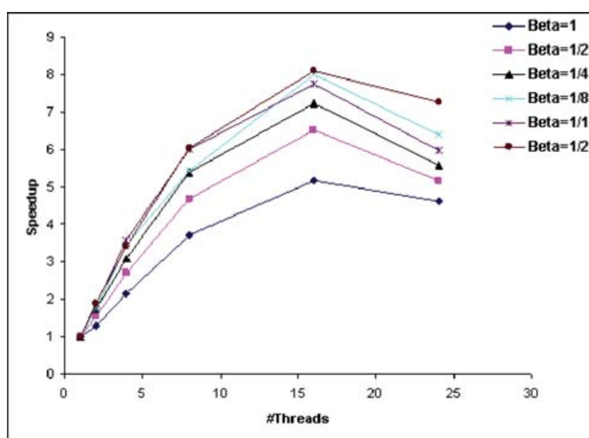

Fig. 6. The speedup with number of threads and varying the value of β.

communication time. Since the memory is shared, the increase of the communication time can be due to insufficient memory bandwidth consumed by the large memory requests. This large number of memory requests can be the result of cache misses, since L1 cache is shared between all threads in the chip and L2 is shared between four threads in one core.

It should be noted that, unlike traditional parallel systems, the Sun T2000 has extremely low communication overhead. As such, it was found that α and β values maximize performance when both workload and data distribution maximize cache utilization. In this specific case, as α is set to lower values the spacial locality of the application improves allowing for better L2 cache utilization. It is critical however that the α value reflects fair cache line utilization in this shared cache architecture. In addition, decreasing the value of α leads also to a decrease of the synchronization time. From the experimental results depicted in the figure 5, the best speedup for the alignment of these two sequences were obtained when the value of α is equal $\frac{1}{16}$ and $\frac{1}{24}$ and the number of threads is equal 16.

To study the impact of the parameter β on the performance, let us consider that the value of α is equal to 1 and varying the value of β. Speedups achieved with 1, 2, 4, 8, 16, and 24 threads are illustrated in figure 6. We can see that the algorithm is also sensitive to the variation of the value of β. Alike the sensitivity to the variation of α, the speedup increases nearly linearly up to 8 threads, plateaus between 8 and 16 threads, and decreases between 16 and 24 threads. We can see that the best speedup were obtained when the value of β is equal $\frac{1}{8}$ or $\frac{1}{24}$ using 16 threads, α here is considered equal 1. With a fixed α it can be seen that smaller values of β improve performance by reducing idle time.

According to these two figures, 5 and 6, the same behavior is shown when varying independently α and β. Since varying the value of α influences both cache utilization and thread idle time. The selection of the best values of α must be done by taking into consideration the possible values of β. To illustrate this point, let us now fix the number of threads to 16. Figure 7 presents the variation of the speedup versus α and β. The same behavior like in figures 5 and 6 is shown, but with a higher speedup given when (α, β)= $(\frac{1}{2}, \frac{1}{16})$. The objective is to select the couple (α, β) that results with good performance, i.e, minimize idle time and maximize cache utilization.

The optimal values of β in a shared cache multicore must reflect values that equally share cache line availability. It is found that α and β should generally be set to values in order to better utilize available cache lines (in other words, computation should work horizontally on the matrix). However, it is also important to note that decreasing α and β decreases idle time of threads (excluding thread T_0).

From figure 8 it can be concluded that spatial locality of both compute workload and data distribution must both optimize spacial locality. The values of α and β that yield the best performance all have a workload distribution that have cache line contention of 40% or less for data distribution. Most importantly these values provide access patterns which better utilize cache line availability (in the case of the T2000, a cache line is at a 128-byte boundary). The values of β and α which have shown poor performance all are due to extremely

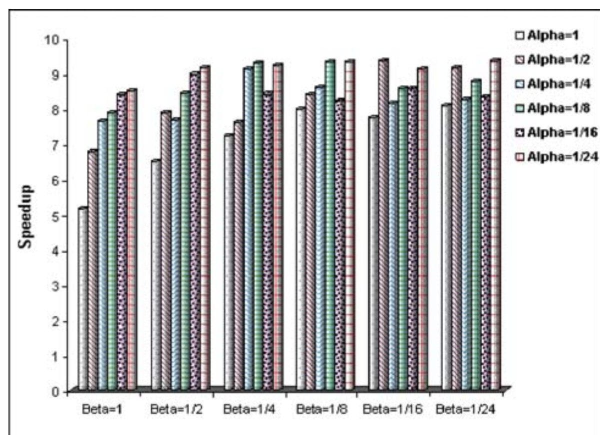

Fig. 7. Speedups for 16 threads by varying the values of α and β.

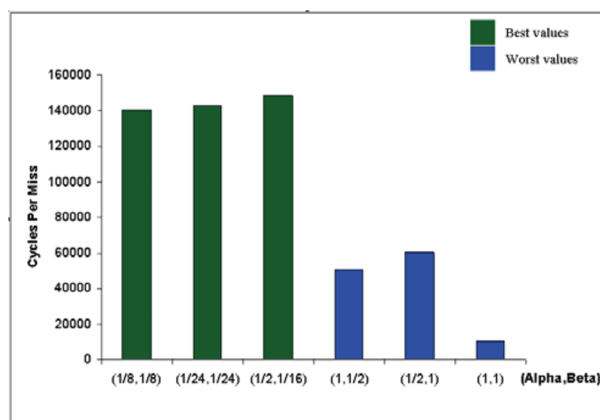

Fig. 8. Cycles Per L2 Cache Miss on Load for best and worst performing combination of α and β.

bad cache utilization with to over 50% cache contention [1]. In the values for (α, β)=($\frac{1}{2}$, $\frac{1}{16}$) it can be seen there is a slightly higher miss rate. However both data and workload distribution are maximized here leading to better performance for these values. It is worth noting that larger values of β and α also increase idle time detrimentally towards performance. Hence, to select the best over-all α and β, the user must minimize idle time and maximize cache utilization.

V. Conclusions and Future Work

In this paper, we evaluated a PGAS parallel algorithm for local sequence alignment on the T1 chip multi-threading architecture. The performance of the algorithm is based on two tuning parameters, α and β (which ensure the optimal data and workload distributions). The programmer should be able to empirically select the best values for these parameters to simultaneously minimize the thread idle time while maximizing cache utilization. The results obtained show that the values of these parameters have a great impact on the performance of the application. Future work involves further optimization of the algorithm to exploit the concurrency semantics of hardware threading in order to maximize compute throughput for this specific problem. Other optimization techniques, such as message aggregation and privatizing local shared accesses, should be added in order to increase the performance of the algorithm by hiding the latencies associated with remote shared accesses, since are not integrated yet in the version of UPC used.

[1]This data was collected from the T1 Instr_Cnt and L2_Dmiss_ld performance counters extracted from the Solaris cputrack tool

References

[1] El-Ghazawi T., Carlson W., Sterling T, Yelick K.: UPC: Distributed Shared Memory Programming. Book, John Wiley and Sons Inc., NewYork. ISBN: 0-471-22048-5, 2005.
[2] Yap T. K., FriederO., Martino R. L.: Parallel Computation in Biological Sequence Analysis. IEEE Transactions on Parallel and Distributed Systems, Vol. 9, N 3, pp. 283-294, 1998.
[3] Needleman, S. B., Wunsch C. D: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology, Vol. 47, pp. 443-453, 1970.
[4] Smith W., Waterman M.: Identification of Common Molecular Subsequences. Journal of Molecular Biology Vol. 147, pp. 195-197, 1981.
[5] Gaber J.: Complexity Measure Approach for Partitioned Shared Memory Model, Application to UPC. Research report RR-10-04. Universite de Technologie de Belfort-Montbeliard, 2004.
[6] Lu M., Lin L.: Parallel algorithms for the Longest Common Subsequence Problem. IEEE Transaction on Parallel and Distributed System, Vol.5, pp.835-848, 1994.
[7] Valiant L.G.: A bridging model for parallel computation. Communication of the ACM, Vol. 33, N 8, pp. 103-111, 1990.
[8] http://www.sun.com/servers/coolthreads/t1000/benchmarks.jsp
[9] Garcia T., Myoupo J-F., Seme D.: A Coarse-Grained Multi-computer algorithm for the longest common subsequence problem. 11-th Euromicro Conference on Parallel Distributed and Network based Processing, 2003.
[10] Alves C. E. R., Cceres E. N., Dehne F.: Parallel Dynamic Programming for Solving the String Editing Problem on a CGM/BSP. In proceeding of ACM SPAA'02, pp. 275-281, 2002.
[11] Alves C. E. R. Cceres E. N., Dehne F. , Song S. W.: A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison. International Conference on Computational Science and its Applications, Montreal, Canada, May 18-21, Lecture Notes in Computer Science, V. 2668, pp. 249-258, 2003.
[12] Nicholas P. P.: Searching Biological Sequence Databases Using Distributed Adaptive Computing. Master thesis, Master of Science in Computer Engineering, Virginia Polytechnic Institute and State University, 2003, available at http://scholar.lib.vt.edu/theses/
[13] Chen Y., Wan A., Liu W.: A fast Parallel Algorithm for Finding the Longest Common Sequence of Multiple Bio-sequences. Symposium of Computations in Bioinformatics and Bioscience (SCBB06). In conjunction with the International Multi-Symposiums on Computer and Computational Sciences 2006 (IMSCCS06), 2006.
[14] Bader D.A., Madduri K.,: A Graph-Theoretic Analysis of the Human Protein-Interaction Network Using Multicore Parallel Algorithms. Sixth IEEE International Workshop on High Performance Computational Biology (HiCOMB'07), 2007.
[15] Bader D. A., Kanade V., Madduri K.: SWARM, A Parallel Programming Framework for Multicore Processors. First Workshop on Multithreaded Architectures and Applications (MTAAP'07), 2007.
[16] Voss G., Schrder A., Mller-Wittig W. Schmidt B.: Biological Sequence Alignment on Graphics Processing Units. Available at http://www.ntu.edu.sg/home/asbschmidt/paper/BioGPU.pdf
[17] Kayi A., Yao Y., El-Ghazawi T., Newby G.: Experimental Evaluation ofEmerging Multi-core Architectures, Workshop on Performance Modeling, Evaluation, and Optimisation of Ubiquitous Computing and Networked Systems (PMEO07) IPDPS07 Proceedings, pp. 1-6, 2007.
[18] Chen W-Y., Bonachea D., Duell J., Husbands P., Iancu C., Yelick K.: A Perfor- mance Analysis of the Berkley UPC Compiler. In Annual International Conference on Supercomputing (ICS), 2003.
[19] Cantonnet F., El Ghazawi T., Lorenz P., Gaber J.: Fast Address Translation Tech- niques for Distributed Shared Memory Compilers. International Parallel and Dis- tributed Processing Symposium IPDPS06, 2006.
[20] Bakhouya M., Gaber J., El-Ghazawi T.: Towards a Complexity Model for Design and Analysis of PGAS-Based Algorithms, HPCC 2007 Proceedings, LNCS 4782 Springer, ISBN 978-3-540-75443-5, pp.672-682, 2007.