A Parallel BSP Algorithm for Irregular Dynamic Programming

Malcolm Yoke Hean Low[1], Weiguo Liu[1], and Bertil Schmidt[2]

[1] School of Computer Engineering, Nanyang Technological University, Singapore 639798
{yhlow, liuweiguo}@ntu.edu.sg
[2] University of New South Wales Asia, 1 Kay Siang Road, Singapore 248922
bertil.schmidt@unswasia.edu.sg

Abstract. Dynamic programming is a widely applied algorithm design technique in many areas such as computational biology and scientific computing. Typical applications using this technique are compute-intensive and suffer from long runtimes on sequential architectures. Therefore, several parallel algorithms for both fine-grained and coarse-grained architectures have been introduced. However, the commonly used data partitioning scheme can not be efficiently applied to irregular dynamic programming algorithms, i.e. dynamic programming algorithms with an uneven load density pattern. In this paper we present a tunable parallel Bulk Synchronous Parallel (BSP) algorithm for such kind of applications. This new algorithm can balance the workload among processors using a tunable block-cyclic data partitioning method and thus is capable of getting almost linear performance gains. We present a theoretical analysis and experimentally show that it leads to significant runtime savings for pairwise sequence alignment with general gap penalties using BSPonMPI on a PC cluster.

Keywords: BSP, Irregular Dynamic Programming, Partitioning, Load Balancing, Scientific Computing.

1 Introduction

Dynamic programming (DP) is a popular algorithm design technique for optimization problems. Problems such as string editing [1], genome sequence alignment [14, 22], RNA and protein structure prediction [6, 17, 24], context-free grammar recognition [7, 19], and optimal static search tree construction [9] have efficient sequential DP solutions. In order to reduce the high computing cost of DP problems, many efficient parallel algorithms on different parallel architectures have been introduced [1, 2]. On fine-grained architectures, the computation of each cell within an anti-diagonal is parallelized [20, 21]. However, this way is only efficient on architectures such as systolic arrays, which have an extremely fast inter-processor communication. On coarse-grained architectures like PC clusters it is more convenient to assign an equal number of adjacent columns to each processor as shown in Figure 1. In order to reduce communication time further, matrix cells can be grouped into blocks.

M. Xu et al. (Eds.): APPT 2007, LNCS 4847, pp. 151–160, 2007.

Processor P_i then computes all the cells within a block after receiving the required data from processor P_{i-1}. Figure 1 shows an example of the computation for 4 processors, 8 columns and a block size of 2×2, the numbers 1 to 7 represent consecutive phases in which the cells are computed. We call this method *blockbased*. It works efficiently for regular DP computations with an even workload across matrix cells, i.e. each matrix cell is computed from the same number of other matrix cells.

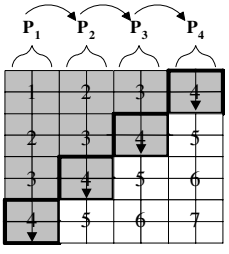

Fig. 1. Parallel computation for 4 processors, 8 columns and a 2×2 block size

In practice, there are many irregular DP applications where the workload of a cell varies across the matrix. Figure 2 shows an example of such an application. The workload to compute one matrix cell will increase along the shift direction of the computation. We call this the *load computation density*. Figure 2 shows the change of load computation density along the computation shift direction by using increasingly blacking shades. We can see that the load computation density at the bottom right-hand corner is much higher than that in the top left-hand corner. The column-based partitioning method in Figure 1 will therefore lead to a poor performance, since the workload on processor P_i is much higher than on the processor P_{i-1}.

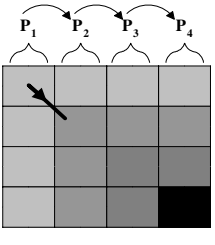

Fig. 2. Example of an irregular DP computation

In this paper, we propose a general parameterized parallel BSP algorithm to solve this problem. By introducing two performance-related parameters, we can get the trade-off between load balancing and communication time by tuning these two parameters and thus obtain the maximum possible performance. We demonstrate how this algorithm can lead to substantial performance gains for irregular DP applications.

The rest of the paper is organized as follows: Section 2 describes the characters and classification for irregular DP algorithms. The BSP model is briefly reviewed in Section 3. Section 4 presents the parallel BSP algorithm. Section 5 evaluates the performance on a PC clusters using BSPonMPI. Section 6 concludes this paper.

2 Irregular DP Algorithms

DP algorithms can be classified according to the matrix size and the dependency relationship of each matrix cell [10]: a DP algorithm for a problem of size n is called a *tD*/*eD* algorithm if its matrix size is $O(n^t)$ and each matrix cell depends on $O(n^e)$ other cells. The DP formulation of a problem always yields an obvious algorithm whose time complexity is determined by the matrix size and the dependency relationship. If a DP algorithm is a *tD*/*eD* problem, it takes time $O(n^{t+e})$ provided that the computation of each term takes constant time. Three examples are given in Algorithm 1 to 3.

Algorithm 1. (2*D*/0*D*): Given $D[i,0]$ and $D[0, j]$ for $1 \le i, j \le n$,
$D[i, j] = \min\{D[i\text{-}1, j] + x_i, D[i, j\text{-}1] + y_j, D[i\text{-}1, j\text{-}1] + z_{i,j}\}$ where x_i, y_j and $z_{i,j}$ are computed in constant time.

Algorithm 2. (2*D*/1*D*): Given $w(i, j)$ for $1 \le i<j \le n$; $D[i,i] = 0$ for $1 \le i \le n$

$$D[i,j] = w(i, j) + \min_{i<k\le j} \{D[i,k-1] + D[k, j]\} \quad \text{for } 1 \le i, j \le n$$

Algorithm 3. (2*D*/2*D*): Given $w(i, j)$ for $1 \le i<j \le 2n$; $D[i, 0]$ and $D[0, j]$ for $0 \le i, j \le n$,

$$D[i, j] = \min_{\substack{0\le i'<i \\ 0\le j'<j}} \{D[i',j'] + w(i'+j',i+j)\} \quad \text{for } 1 \le i, j \le n$$

Table 1. A classification for the popular DP algorithms in CB

Algorithm	Time complexity	Application Field	Reference
Smith-Waterman algorithm with linear and affine gap penalty	$O(n^2)$	Genome alignment	[14, 22]
Syntenic alignment		Generalized genome global alignment	
Smith-Waterman algorithm with general gap penalty	$O(n^3)$	Genome alignment	[8, 22]
Nussinov algorithm		RNA base pair maximization	
Viterbi Algorithm	$O(n^2)$~$O(n^4)$	Gene sequence alignment using HMMs, Multiple sequence alignment	[8]
Double DP algorithm	$O(n^4)$	Protein threading	[17]
Spliced Alignment	$O(n^3)$	Gene finding	[11]
Zuker Algorithm	$O(n^3)$~$O(n^4)$	RNA secondary structure prediction	[24]
CYK Algorithm		RNA secondary structure alignment	[8]

There are many DP algorithms in Computational Biology (*CB*). DP is used for assembling DNA sequence data from the fragments that are delivered by automated sequencing machines [3], and to determine the intron/exon structure of eukaryotic genes [12]. It is used to infer function of proteins by homology to other proteins with known function [18, 23] and it is used to predict the secondary structure of functional RNA genes or regulatory elements. In some areas of CB, DP problems arise in such

variety that a specific code generation system for implementing such algorithms has been developed [4]. However, the development of a successful parallel DP algorithm is a matter of experience, talent, and luck. The typical matrix recurrence relations that make up a parallel DP algorithm are intricate to construct, and difficult to implement reliably. No general problem independent guidance is available. Table 1 shows the classification of some popular DP algorithms in CB.

(a) Nussinov: Given a sequence A of length L with symbols $x_{1,\ldots}\, x_L$. Let $\delta(i, j) = 1$ if x_i and x_j are a complementary base pair, else $\delta(i, j) = 0$. We will recursively calculate scores $M(i, j)$ which are the maximal number of base pairs that can be formed for subsequence $x_i, \ldots, x_j$.

Initialization:
for $i = 2$ **to** L **do**
  $M(i,i-1) = 0$
for $i = 1$ **to** L **do**
  $M(i,i) = 0$

Recursion:
$M(i, j) = \max\{M(i+1, j), M(i, j-1),$
$M(i+1, j-1) + \delta(i, j),$
$\max_{i<k<j}[M(i, k)+M(k+1, j)]\}$

(b) SkylineMatrix: The skyline matrix problem can be formulated as follows: Given an $N \times N$ skyline matrix A and an N-vector b, we seek to find an N-vector x such that $Ax = b$. An efficient and widely used technique for solving $Ax = b$ in the general case is the LU-Decomposition. This method decomposes A into two matrices L and U. The algorithm used for sequential LU-Decomposition is "Doolittle's Method". Generally, the algorithm works as follows:

for $i = 1$ **to** N **do**
  for $j = 1$ **to** $i-1$ **do**
    $L_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} L_{ik} U_{kj}\right) / U_{jj}$
  for $j = 1$ **to** i **do**
    $U_{ji} = a_{ji} - \sum_{k=1}^{j-1} L_{jk} U_{ki}$

(c) SW with general gap penalty function: Consider two strings A and B of length l_1 and l_2, a substitution matrix s and a general gap penalty function $\gamma(g)$. To identify common subsequences, they compute the similarity matrix $M(i, j)$ of two sequences ending at position i and j.

$$M(i,j) = \max \begin{cases} M(i-1, j-1) + s(A_i, B_j), \\ M(k, j) + \gamma(i-k), \quad k = 0,\ldots,i-1, \\ M(i,k) + \gamma(j-k), \quad k = 0,\ldots,j-1. \end{cases}$$

Fig. 3. The recurrence formulas for three 2D/1D DP algorithms: (a) Nussinov algorithm, (b) Skyline matrix problem, (c) Smith-Waterman algorithm with general gap penalty function

In this paper we concentrate on the parallelization of DP algorithms of the type 2D/1D. This is an important DP algorithm with many applications. Figure 3 shows three well-known DP algorithms of type 2D/1D. Although these DP algorithms look different, they share similar characteristics. These 2*D*/1*D* DP algorithms are all irregular with load computation density changes along the computation shift direction.

Figure 4 shows the change of load computation density along the computation shift direction by using increasingly blacking shades. For these algorithms, the column-based partitioning method of Figure 1 leads to poor load balancing. Thus, a more efficient data partitioning scheme is needed. The problem of determining an appropriate data partitioning scheme is to maximize system performance by balancing the computational load among processors. Since the data partitioning scheme largely determines the performance and scalability of a parallel algorithm, a great deal of research has aimed at studying different data partitioning schemes. As a result the

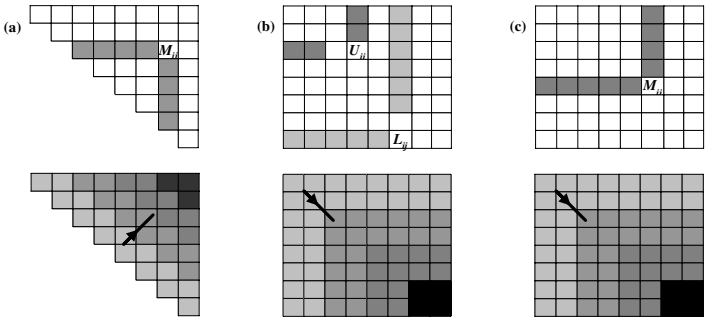

Fig. 4. Dependency relationship and distribution of load computation density along computation shift direction for (a) Nussinov, (b) Skyline matrix problem, (c) Smith-Waterman algorithm with general gap penalty function

block-cyclic partitioning has been suggested as a general-purpose basic scheme for parallel algorithms because of its scalability, load balancing and communication properties [15]. In this paper, we introduce a tunable block-cyclic based distribution of columns for irregular DP algorithms to balance the workload among processors.

3 The Bulk Synchronous Parallel (BSP) Model

The BSP model first proposed in [23] is designed to be a general purpose approach to parallel computing that allows the separation of concerns between computation, synchronization and communication costs. It has a simple cost model for predicting the performance of BSP algorithms on different parallel platforms. A BSP programming model consists of P processors linked by an inter-connecting network and each with its own pool of memory.

A BSP algorithm consists of a set of processors each executing a series of supersteps. Each superstep consists of three ordered phases: 1) a local computation phase, where each processor can perform computation using local data and issue communication requests; 2) a global communication phase, where data is exchanged between processors according to the requests made during the local computation phase; and 3) a barrier synchronization, which waits for all data transfers to complete and makes the transferred data available to the processors for use in the next superstep. The BSP cost model for a BSP algorithm S can be expressed as

$$\text{cost(S)} = \text{sum}\{\ w(i) + gh(i) + L\ \} \text{ for superstep } i = 1 \ \ldots\ n_s$$

where n_s is the total number of supersteps; $w(i)$ is the maximum computation cost by any processor in superstep i; and $h(i)$ is the maximum number of messages sent or received respectively by any processor in superstep i. The architecture dependent parameters g and L represent the communication and synchronization costs respectively. From the BSP cost model, we can see that the performance of a BSP algorithm relies on three factors: a) computation balance; b) communication balance; and c) n_s, the total number of supersteps.

While a BSP library consists of a small set of architectural independent programming interface that support the BSP programming model, the efficiency of a BSP algorithm depends on how the underlying BSP library implementation optimizes the architecture dependent parameters *g* and *L*. Existing BSP library implementation such as the Oxford BSP library [13] and the Paderborn University BSP (PUB) Library [5] are often optimized for a selection of parallel hardware platforms. To keep up with changes and development in these platforms, these libraries have to be constantly updated. The BSPonMPI library (http://bsponmpi.sourceforge.net) is an effort to create a BSP library that runs on any machine that has MPI installed. This ensures that any BSP program compiled using BSPonMPI will benefit from improvements and optimizations in the MPI library for a particular hardware platform.

4 Parallel BSP Algorithm

In this section, we describe a tunable parallel BSP algorithm for solving irregular DP problems. The algorithm proceeds in a series of wavefront diagonally across the matrix *M*. Figure 5 illustrate the concept of the algorithm for an 8×8 matrix with a column-wise block-cyclic partition. The parameter *division* is used to implement a block cyclic distribution of columns to processors. The parameter *rowwidth* is used to control the size of messages that P_i will send to other processors. In the figure, $P_{i,dj}^{\ k}$ denotes that the cell is updated by processor P_i at division *j* of wavefront *k*. Each wavefront corresponds to a superstep in the BSP computation. For example, in wavefront 4, processor P_1 and P_2 are active in both division 1 and 2.

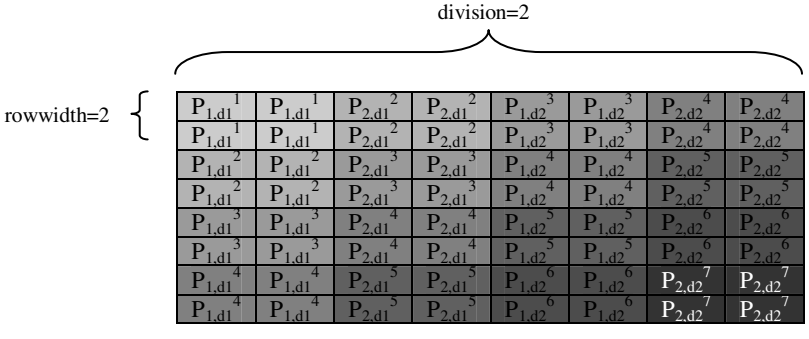

Fig. 5. The tunable block-cyclic partitioning method for irregular dynamic programming

Increasing the number of cyclic divisions and decreasing the size of messages may lead to better load balancing at the expense of increase in communication overhead. Thus, the choice of the parameter for *division* and *rowwidth* is a trade-off between load balancing and communication time. Figure 6 shows the BSP algorithm for irregular dynamic programming. In each superstep (or wavefront), each processor updates the block allocated to it in all its active divisions and sends the updated block to other processors. In this implementation, we use the BSP shared memory primitive `bsp_put()` to update the matrix block. A barrier synchronization is called at the end of each superstep. All processors will receive the updated matrix by the beginning

```
Input:   The number of processors Np, the number of division Nd, the
         row width R. (n×n is the size of matrix M, dt denotes the t-
         th division, wt denotes the t-th wave, C denotes the column
         width).
Output:  Depending on the requirements of the given applications,
         the output will be the optimal score M[1,n] or the whole
         matrix M.

Nwaves = p*Nd + n/R;
C = n/Nd;

bsp_begin(Np)
  pid = bsp_pid();
  // beginning of a superstep, do for each wavefront
  for wt = 1 to Nwaves
    for dt = 1 to Nd  // do for each division
      // if processor pid is active in this division
      if pid+(dt × Np) <= wt
        Sc = (pid + (dt-1)×Np) × C; // compute starting column
        Sr = (wt - pid -(dt × Np)) × R; // compute starting row
        for i = Sc to Sc + C
          for j = Sr to Sr + R
          compute(M[i,j]);
          endfor
        endfor
        send_block();//send updated block to other processors
      endif
    endfor
    // end of superstep
    bsp_sync();
  endfor
bsp_end()
```

Fig. 6. The BSP algorithm for irregular dynamic programming

of the next superstep. Note that for sake of simplicity, the algorithm presented assumes the dimension of the matrix n is exactly divisible by C and R. The actual algorithm implemented does not have this assumption.

5 Performance Evaluation

We carried out a set of experiments using the BSP algorithm described in section 4 to parallelize the Smith-Waterman algorithm with general gap penalty function. The hardware platform used is an 8-node Dual-Processor Linux cluster with a 1GBit/sec Myrinet switch used as inter-cluster connection. The BSP algorithm is compiled with Myrinet MPICH ver 1.2.6 and linked with the BSPonMPI ver 2.0 library.

Table 2 shows the speedup results using the BSP algorithm for irregular dynamic programming on different number of processors. With different number of processors, the best speedup (shown in bold) is obtained with different combination of N_d and R.

In the first implementation, each processor is allocated equal number of columns in each division. When the dimension of the matrix is not exactly divisible by the N_d and the number of processor, the remainder columns are allocated to the first processor in the first division. For example, in the case of N_p=16 and N_d=50, each processor will be allocated 3 columns in each division. In the first division, processor P_1 will be allocated the remaining 600 columns in addition to the 3 columns allocated to each processor! Since processor P_1 will be active in division 1 for n/R supersteps, this

allocation will result in computation and communication imbalance during the BSP computation. Table 3 shows the number of extra columns allocated to processor P_1 in division 1. Except for N_p=2, the best speedup numbers from Table 2 clearly matches the value of N_d that gives the smallest number of extra columns in division 1.

In the second implementation, a more balanced partitioning approach is used. In this implementation, all processors are allocated $k = n/(N_pN_d)$ columns in all divisions. If N_pN_d does not divide n exactly, in division 1, the remaining $n-kN_p(N_d-1)$ columns are divided again equally among all processors and the remaining columns are allocated to P_1. For example, in the case of N_p=16 and N_d=50, each processor will be allocated 3 columns in each division except division 1. In division 1, each processor

Table 2. Speedup for N_d=50 to 90 and row width R=10 to 40 with N_p=2, 4, 8 and 16 processors. The DP matrix is of size 3000×3000.

	N_d=50	60	70	80	90		N_d=50	60	70	80	90
	N_p=2						N_p =4				
R=10	1.56	1.55	1.65	1.59	1.43		2.39	2.76	2.72	2.60	2.92
20	1.63	1.58	1.62	1.49	1.41		2.93	2.20	2.79	2.60	2.74
30	1.46	1.47	1.55	1.57	1.53		**3.15**	2.79	2.69	2.54	2.84
40	1.50	1.61	1.40	**1.66**	1.60		2.43	2.96	2.63	2.60	2.71
	N_p =8						N_p =16				
R=10	4.46	4.56	4.09	3.43	4.71		3.46	3.81	2.55	3.75	4.74
20	3.77	3.98	4.38	3.48	4.68		3.57	4.42	2.96	3.57	**6.60**
30	4.13	**4.94**	4.45	3.13	4.92		3.27	5.81	2.70	3.26	6.47
40	4.37	4.77	4.18	3.45	4.63		3.15	6.10	2.49	3.59	5.77

Table 3. Number of extra columns allocated to processor 1 in division 1

N_d=50	60	70	80	90		N_d=50	60	70	80	90
N_p =2						N_p =4				
0	0	60	120	120		0	120	200	120	120
N_p =8						N_p =16				
200	120	200	440	120		600	120	760	440	120

Table 4. Speedup using BSP algorithm for N_d=50 to 90 and row width R = 10 to 40 with N_p= 2, 4, 8 and 16 processors using improved partitioning. The DP matrix is of size 3000×3000.

	N_d=50	60	70	80	90		N_d=50	60	70	80	90
	N_p =2						N_p =4				
R=10	1.59	1.58	1.64	1.60	1.60		2.82	3.16	3.04	2.92	**3.24**
20	1.62	1.56	1.59	1.62	1.64		2.65	2.99	3.09	3.11	3.20
30	1.44	1.64	**1.66**	1.56	1.71		3.17	2.52	2.67	2.99	2.68
40	1.58	1.61	1.61	1.62	1.66		3.12	2.55	2.97	2.31	3.08
	N_p =8						N_p =16				
R=10	**5.98**	5.78	5.52	5.23	5.64		9.09	7.40	7.12	8.14	8.33
20	5.62	5.79	5.84	5.15	5.62		7.58	**9.36**	7.43	8.70	8.19
30	4.96	5.79	5.58	5.05	5.41		6.65	5.40	7.98	6.34	7.68
40	5.45	5.20	5.17	5.18	5.20		8.02	7.68	3.77	3.76	6.59

will be allocated 40 columns each and processor 1 will receive 48 columns. Another alternative partitioning approach is to allocate the remaining columns equally across all divisions. This will be investigated in our future implementation.

Table 4 shows the experimental results using the improved partitioning. For N_p=16, there is clearly a substantial improvement in performance and the difference in performance between different N_d is reduced. The results show that a balanced partitioning approach is crucial to the performance of the BSP algorithm for irregular dynamic programming.

6 Conclusions and Future Work

In this paper, we have described a tunable BSP algorithm for irregular DP algorithms of type 2D/1D. In the BSP algorithm presented in Figure 6, communication is initiated through the BSP shared memory primitive `bsp_put()` invoked by each sender processor. The receiving part of the communication is automatically handled by the BSPonMPI library and is carried out in bulk at the end of every superstep. This makes the code simple and easy to understand. Note that such one-sided communication primitive is also available in MPI 2.0. An MPI algorithm for irregular DP applications similar to the one presented in [16] that uses matching *send* and *receive* primitive for inter-processor communication can sometime lead to code that is hard to understand and debug.

The experimental results also show that good partitioning approach is essential to achieving high parallel efficiency for this BSP algorithm. The corresponding parallel efficiency for P = 2, 4 and 8 ranges from 75% to 83%. For P = 16, the parallel efficiency drops to 58%. Table 4 shows that the selection of N_d and R has a more significant effect on the performance N_p=16. This could be due to (1) the relatively high barrier synchronization cost L for 16 processors; and (2) the scheduling of tasks between each of the two processors in each node of the Linux cluster.

With improved performance of future versions of the BSPonMPI library, the effects of barrier synchronization cost will be minimized accordingly. We will explore different processor mapping and data partitioning strategies to resolve the issue of scheduling dual-processor nodes in a cluster. Our future work also includes benchmarking the communication and synchronization cost of different processor configurations for our system. This will allow us to predict the performance of different combinations of N_d and R and determine the combination that will yield the best performance. We will also explore how the BSP algorithm can be adapted to other type of DP applications such as 2D/2D and 3D/1D. Such applications are frequently used in the field of computational biology.

References

1. Alves, C.E.R., Cáceres, E.N., Dehne, F.: Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In: Proc. of the fourteenth annual ACM symposium on Parallel algorithms and architectures, Winnipeg, Manitoba, Canada (2002)
2. Alves, C.E.R., Cáceres, E.N., Dehne, F., Song, S.W., Parallel, A.: Wavefront Algorithm for Efficient Biological Sequence Comparison. In: Kumar, V., Gavrilova, M., Tan, C.J.K., L'Ecuyer, P. (eds.) ICCSA 2003. LNCS, vol. 2667, pp. 249–258. Springer, Heidelberg (2003)

3. Anson, E.L., Myers, G.W.: Realigner: A Program for Refining DNA Sequence Multi-Alignments. In: 1st Conference on Computational Molecular Biology, pp. 9–16 (1997)
4. Birney, E., Durbin, R.: Dynamite: A Flexible Code Generating Language for Dynamic Programming Methods. In: Proc. Intelligent Systems for Molecular Biology, pp. 56–64 (1997)
5. Bonorden, O., Juurlink, B., von Otte, I., Rieping, I.: The Paderborn University BSP (PUB) Library. Parallel Computing 29(2), 187–207 (2003)
6. Bowie, J., Luthy, R., Eisenberg, D.: A Method to Identify Protein Sequences That Fold Into A Known Three-dimensional Structure. Science 253, 164–170 (1991)
7. Ciressan, C., Sanchez, E., Rajman, M., Chappelier, J.C.: An FPGA-based coprocessor for the parsing of context-free grammars. In: IEEE Symposium on Field-Programmable Custom Computing Machines (April 2000)
8. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological Sequence Analysis-Probabilistic Models of Protein and Nucleic Acids. Cambridge University Press, Cambridge (1998)
9. Farach, M., Thorup, M.: Optimal evolutionary tree comparison by sparse dynamic programming. In: 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, November 20-22, 1994, pp. 770–779 (1994)
10. Galil, Z., Park, K.: Dynamic Programming with Convexity, Concavity and Sparsity. Theoretical Computer Science 92, 49–76 (1992)
11. Gelfand, M.S., Mironov, A.A., Pevzner, P.A.: Gene Recognition Via Spliced Sequence Alignment. Proc. Natl. Acad. Sci. 93, 9061–9066 (1996)
12. Gelfand, M.S., Roytberg., M.A., Dynamic, A.: Programming Approach for Prediction the Exon-Intron Structure. Biosystems 30, 173–182 (1993)
13. Hill, J., McColl, B., Stefanescu, D., Goudreau, M., Lang, K., Rao, S., Suel, T., Tsantilas, T., Bisseling, R.: BSPlib: The BSP programming library. Parallel Computing 24(14), 1947–1980 (1998)
14. Huang, X., Chao, K.M.: A Generalized Global Alignment Algorithm. Bioinformatics 19(2), 228–233 (2003)
15. Kumar, V., Grama, A., Gupa, A., Karypis, G.: Introduction to Parallel Computing. Cummings Publishing Company Inc., The Benjamin (1994)
16. Liu, W., Schmidt, B.: A Tunable Coarse-Grained Parallel Algorithm for Irregular Dynamic Programming Applications. In: Bougé, L., Prasanna, V.K. (eds.) HiPC 2004. LNCS, vol. 3296, Springer, Heidelberg (2004)
17. Mount, D.W.: Bioinformatics-Sequence and Genome Analysis. Cold Spring Harbor Laboratory Press (2001)
18. Needleman, S.B., Wunsch., C.D., General, A.: Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. J. Mol. Biol. 48, 443–453 (1970)
19. Ney, H.: The Use of a One-Stage Dynamic Programming Algorithm for Connected Word Recognition. IEEE Trans. on Acoustic, Speech and Signal Processing ASSP-32(2), 263–271 (1984)
20. Schmidt, B., Schroder, H., Schimmler, M.: Massively Parallel Solutions for Molecular Sequence Analysis. In: Proc. of IPDPS 2002 (2002)
21. Schmidt, B., Schroder, H., Schimmler, M.: A Hybrid Architecture for Bioinformatics. Future Generation Computer System 18, 855–862 (2002)
22. Smith, T.F., Waterman, M.S.: Identification of Common Subsequences. Journal of Molecular Biology 147, 195–197 (1981)
23. Valiant, L.G.: A Bridging Model for Parallel Computation. Communications of the ACM 33(8), 103–111 (1990)
24. Zuker, M., Stiegler, P.: Optimal Computer Folding of Large RNA Sequences Using Thermodynamics and Auxiliary Information. Nucleic Acids Research, 9 (1981)