Locality and Parallelism Optimization for Dynamic Programming Algorithm in Bioinformatics

Guangming Tan[1,2]
Shengzhong Feng[1] and Ninghui Sun[1]
{tgm, fsz, snh}@ncic.ac.cn
1. Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences
2. Graduate School of Chinese Academy of Sciences

Abstract

Dynamic programming has been one of the most efficient approaches to sequence analysis and structure prediction in biology. However, their performance is limited due to the drastic increase in both the number of biological data and variety of the computer architectures. With regard to such predicament, this paper creates excellent algorithms aimed at addressing the challenges of improving memory efficiency and network latency tolerance for nonserial polyadic dynamic programming where the dependences are nonuniform. By relaxing the nonuniform dependences, we proposed a new cache oblivious scheme to enhance its performance on memory hierarchy architectures. Moreover we develop and extend a tiling technique to parallelize this nonserial polyadic dynamic programming using an alternate block-cyclic mapping strategy for balancing the computational and memory load, where an analytical parameterized model is formulated to determine the tile volume size that minimizes the total execution time and an algorithmic transformation is used to schedule the tile to overlap communication with computation to further minimize communication overhead on parallel architectures. The numerical experiments were carried out on several high performance computer systems. The new cache-oblivious dynamic programming algorithm achieve 2-10 speedup and the parallel tiling algorithm with communication-computation overlapping shows a desired potential for fine-grained parallel computing on massively parallel computer systems.

Keywords: dynamic programming, cache-oblivious, tiling, locality, parallelism

1 Introduction

Dynamic programming (DP) is a common technique to solve a wide variety of discrete optimization problems such as scheduling, string editing, packaging and inventory management. More recently, researchers have extended its applications to the realm of bioinformatics. For instance, the Smith-Waterman [Smith and Waterman 1981] algorithm for matching sequences of amino-acids and necleotides, Zuker's [Lyngso and Zuker 1999] algorithm for predicting RNA secondary structures. In many DP algorithm researches, it is considered as a class of multistage problems. Grama, et.al. [A. Grama and Kumar 2003] model the dependences in DP formulations as a directed graph and classify them into four classes of DP formulation: serial monadic (single source shortest path problem, 0/1 knapsack problem), serial polyadic (Floyd all pairs shortest paths algorithm), nonserial monadic (longest common subsequence problem, Smith-Waterman algorithm) and nonserial polyadic (optimal matrix parenthesization problem, binary search tree and Zuker algorithm). In the previous classification, the term serial/monadic represents uniform data dependences, which has been studied extensively and optimized efficiently on current architectures in the last decade. All DP of this kind are multistage problems where there are only two sets dependences due to the DP recurrences: in the stages themselves and among the consecutive stages only. The term nonserial polyadic represents the another more complicated family of DP problems whose dependences are nonuniform, specifically, new dependences appearing among non-consecutive stages are observed are observed besides the dependences mentioned above.

In order to focus on nonuniform dependences analysis, we first give a general formulation for this nonserial polyadic DP algorithm. Derived from DP formulation in RNA secondary structure prediction algorithm and optimal matrix parenthesization problem, we present an abstract DP formulation 1, where $a(i)$ is the initial value.

$$m[i,j] = \begin{cases} min_{i \le k < j}\{m[i,j], m[i,k] + m[k+1,j]\} & 0 \le i < j < n \\ a(i) & i = j \end{cases} \quad (1)$$

In most applications, this formulation as a computational kernel mainly involves float operations (Such as Zuker algorithm and optimal binary trees). However, the standard implementation of the nonserial polyadic dynamic programming algorithm cannot exploit a reasonable fraction of the floating point peak performance of current microprocessors. Even worse, the performance always drops dramatically as size of the problem grows(See Figure 2).

The reason for the disappointing performance is the well-known memory wall [Wulf and McKee 1995] in deep memory hierachy architecture. The DP algorithms are easily implemented as loop nests iterations, thus, it is also considered as an iteration domain/space problem [Irigoin and Triolet 1988] for optimizing its locality and parallelism. Tiling the iteration domain (loop blocking, partitioning) [Irigoin and Triolet 1988][Xue 1997b] is a well-known technique used by compilers and programmers to improve data locality and to control parallel granularity to increase the computation to communication ratio [Coleman and McKinley 1995][Wolfe 1987][Ramanujam and Sadayappan 1991][Xue and Huang 1998]. However, most of the work on optimal tiling only considers perfect loop nests with parallelepiped shaped iteration domain and uniform dependences [Xue 1997b][Ramanujam and Sadayappan 1991][Xue 1997a][Xue 1997c], and unfortunately it is invalid when it comes to noserial polyadic DP formulation. This paper presented a novel solution to optimal tiling to triangular iteration domain and nununiform dependeces which focus on the nonserial polyadic DP formulation in Zuker RNA secondary structure prediction algorithm, and

The work is supported by Youth Fund of ICT,CAS(20056600-22) and NNSF of China(60573163)

0-7695-2700-0/06 $20.00 *2006 IEEE*

the results show our new approach of significant implementation in terms of performance.

dp_standard(matrices m, int n)
for $(j = 1; j \leq n; j++)$
  for $(i = j; i \geq 1; i--)$ {
    indxj = indx[j];
    ij = i+indxj;
    $t = m[ij]$;
    for $(k = i; k < j; k++)$
      t=min2(t, m[i+indx[k]]+m[k+1+indxj])
    $m[ij] = t$
  }

Figure 1: The program for dp_standard with vertical traverse

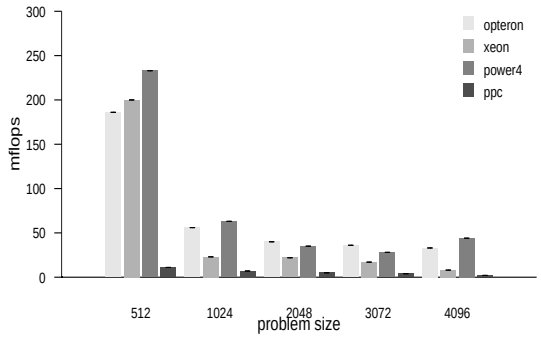


Figure 2: MFLOPS of a straightforward 3-loops iteration implementation of nonserial polyadic dynamic programming algorithm

What has been ignored by previous work is communication advantage that current high performance computer architecture can provide. One of the keys to improve communication performance is to overlap communication with computation and remove the burden of communication from the main processor. One enabling technology for this effort is memory mapped network interfaces [M. A. Blumrich and Sandberg 1994] such as Myrinet [N. J. Boden and Su 1995], Infinband [inf] and Quadrics [F. Petrini and Frachtenberg 2002], which support Virtual Interface Architecture (VIA) [D. Dunning and Dodd 1998] and Remote Direct Memory Access (RDMA). In memory-mapped networks, the data can be transferred between main memory and the network interface through the use of DMA. This mechanism relieves the host CPU from the responsibility for moving data from memory and eliminates the need of unnecessary copying from user buffers to kernel buffers (Zero Copy [F. O. Carroll and Ishikawa 1998]). More aggressively, BlueGene/L [Adiga and et al 2002] provides additional dual-processor I/O node which handles communication between a compute node and other systems, including the host and file servers. Based on the innovative hardware, various programming environments and system softwares also have been developed. In the recent, MPI 2.0 [mpi] has support one-sided communication. Some emerging explicitly parallel programming paradigms using a global address space model, including UPC [upc], Titanium [K. A. Yelick and Aiken 1998] and Co-Array Fortran [Numrich and Reid 1998][caf], exploit communication-computation overlap through one-sided communication.

The goal of this paper is to develop methods for meeting these challenges. And previous researches in this filed show thta an algorithm in a divide-and-conquer fashion often has better cache performance than the one in simple iteration because divide-and-conquer recursively solves subproblems which are small enough to fit in the cache and in this sense only cache misses that occur in the DP are the compulsory misses. Thereby, the algorithm has a good temporal locality which consequently can enhance the performance of DP to certain large extent. In this paper, we propose a cache-oblivious [M. Frigo and Ramachandran 1999] nonserial polyadic dynamic programming algorithm using divide-and-conquer technique. Also, a tiling optimization is introduced based on the cache oblivious algorithmic transformation. In order to exploit communication-computation overlapping. The new tile scheduling strategy results in following scene: a processor computes its tile at k time step and concurrently receives data from other processors to use them at $k+1$ time step and sends data produced at $k-1$ time step. We investigate the application of our optimization methods to serveral modern architectures. All experimental results show that the total execution time is reduced. Our specific contributions are as follows:

- We have developed a cache-oblivious algorithm for nonserial polyadic DP in the first time which runs much faster than the loop nests iteration implementation on deep memory hierachies.
- We have proposed a new tiled parallel scheme for triangular iteration domain with nonuniform dependences which can promise computation and memory workload balance. Besides, We formulated and analytically solved an optimization problem to determine the tile volume size that minimizes the total execution time of the tile parallel algorithm.
- We have proposed a new tile scheduling strategy to exploit the inherent overlapping between communication and computation among tile executions, then surveyed the application of our schedule to modern communication architecture.

The rest of the paper is organized as follows. In section 2, we summarize the previous work on locality and parallelism optimizaton for DP algorithm. Section 3 discuss our cache oblivious algorithm aimed to improve the cache performance of nonserial polyadic DP. In section 4, we use a tiling approach to develop a parallel DP algorithm, then introduce the tile scheduling algorithm to overlap communication with computation. In section 5, we report experimental resutls to validat our claims. Finally, the conclusions are presentd in section 6.

2 Related Work

A number of groups have been doing research in the area of memory hierarchy performance analysis and optimizations in recent years. One of the most successful cases is the optimization of dense numerical computation [J. Demmel 2005][Frigo and Johnson 2005][M. Puschel and Rizzolo 2005]. One characteristic that all these problems share is very regular memory access which are known in advance at compile time. The SUIF [M. W. Hall and Lam 1996] compiler framework includes a large set of libraries for performing data dependency analysis and loop transformations such as tiling. However, we should note that the dependences involved in SUIF is different from that of nonserial polyadic dynamic programming and SUIF will not perform transformations without algorithmic intervention. For dynamic programming algorithms, Venkataraman et al. [G. Venkataraman and Mukhopadhyaya 2003] presented a tiled implementation of the Floyd-Warshall algorithm and derived an upper bound on achievable speedup of 2 for state-of-the-art architectures. The common feature of these work is cache-aware, that is, the optimized algorithm contains parameters (set at

either compile-time or runtime) that can be tuned to optimize the cache complexity for the particular cache size and line. Recently, cache oblivious algorithms have been proved to be able to obtain the same bound of cache misses for many applications [Frigo and Johnson 2005]. Similar to the tile implementation of Floyd-Warshall algorithm in [G. Venkataraman and Mukhopadhyaya 2003], Park et al. [J. S. Park 2004] developed a cache oblivious algorithm and achieved more speedup by combining tiling/blocking technique sensitive to cache parameters. From the forgoing analysis, we noted that the DP algorithm in Floyd-Warshall problem is serial polyadic, where the dependences are uniform. However nonserial polyadic dynamic programming algorithm poses unique challenges to improving cache performance due to their nonuniform dependences. Thereby, optimizations such as tiling can be applied to nonserial polyadic DP only after considering the specific details individually.

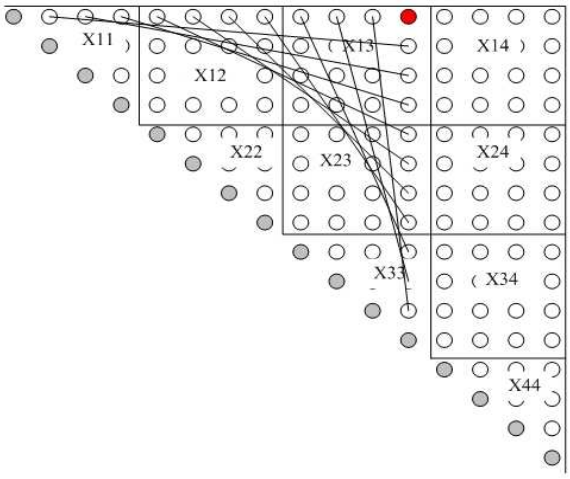


Figure 3: The blocked DP table. The size is $n' \times n'$,$X_{11},X_{22},X_{33},X_{44}$ are three triangular matrices,whose size are $\frac{n'}{4} \times \frac{n'}{4}$, where $X_{12},X_{13},X_{14},X_{23}X_{24},X_{34}$ are six rectangular matrices, whose size are $\frac{n'}{4} \times \frac{n'}{4}$

R. Andonov et.al. [Andonov and Rajopadhye 1997] first applied orthogonal tiling approach to sequence alignment DP algorithm where the tiling problem was considered as a 2D uniform dependence iteration space tiling. Their further optmization [Andonov and Balev 2003] yielded an improvement of a factor of 2.5 over orthogonal tiling for a sequence global alignment DP algorithm. For DP algorithms with nonuniform dependences, the complicated dependences obviously make the parallelization harder. It is not surprising that a lot of work has been done in developing an efficient parallel algorithm. Unfortunately, most of previous work deals with this problem in the context of communication costless models. Bradford [Bradford 1992] described several algorithms solving optimal matrix chain multiplication parenthesizations using the CREW PRAM model. Edmonds et al. [P. Edmonds and George 1993] and Galil et al. [Galil and Park 1994] presented several parallel algorithms on general shared memory multiprocessor systems. Another important research is in the systolic framework, Guibas et al. [L. Guibas and Thomson 1979] focused on designing triangular systolic arrays. The main difficulty for obtaining an efficient parallel implementation on distributed memory multicomputers system is to find a good balance between communication and computation costs. [J. H. Chen and Maizel 1998][F. Almeida and Gonzalez 2002] represent parallel implementations of RNA secondary structure prediction DP algorithm. In [J. H. Chen and Maizel 1998] the computational load balance is satisfactory but their designs do not optimize the communication cost. The authors proved experimentally that the communications took about 50% of the execution time for a sequence with length of more than 9212. In [F. Almeida and Gonzalez 2002] the authors described blocked parallel implementation on a ring of processors and showed the usefulness of the tiling technique for this nonuniform dependences DP. They focused on reducing the communication costs. However, in [J. H. Chen and Maizel 1998] their analytical formulas they didn't take into account the value of the startup latency and the processors were assumed to be permanently busy. But for current machines it is an unrealistic approximation. In [F. Almeida and Gonzalez 2002] the authors ignored that the computation of each iteration point was different, so their algorithm can not attain computation load balance. They overestimated the benefit of achieving the communication only between two neighbors and consequently kept the entire iteration space in each processor node. Thereby, this method has unavoidable limitations on the problem size because of the physical available memory.

As for how to utilize the enable technology of communication-computation overlap on modern network architectures, Danalis et al. [A. Danalis and Swany 2005] presented program transformations in parallel programs which use MPIs collective operations. However, its transformation targeted uniform dependence iteration domain which is easily implemented as a perfect loop nest.

3 Cache-oblivious Algorithm for Nonserial Polyadic DP

The most important technique to develop cache-oblivious algorithm is divide-and-conquer. Before presenting the detailed recursive divide-and-conquer algorithm, we transform the equation 1.Assume (i, j) is the original coordinate in the original domain $\mathscr{D} = \{(i, j)|0 \leq i \leq j < n\}$, where $n = |\mathscr{D}|$ is the original problem size, (i', j') is the new coordinate in the transformed domain $\mathscr{D}' = \{(i', j')|0 \leq i' \leq j' < n'\}$, where $n' = n+1 = |\mathscr{D}'|$ is the new problem size. The iteration domain transformation is defined as follows:

$$(i', j') = f(i, j) : i' = i, j' = j+1$$

Thus, in the transformed domain formulation 1 is rewritten as the new formulation 2, where $a(i)$ is the known initial value (the values on the new diagonal also can be any values).

$$m[i', j'] = \begin{cases} min_{i'+1 \leq k' < j'} \{m[i', j'], m[i', k'] + m[k', j']\} & 0 \leq i' < j' < n' \\ a(i) & j' \leq i'+1 \end{cases} \quad (2)$$

In the new domain, the entries on the new diagonal do not contribute to the computation. We claim that except for the unused values on the new diagonal in the new domain, the transformed formulation 2 gains the same dynamic programming matrices as the original formulation 1 in the original domain. In fact, the orignal domain $\mathscr{D}$ is a subset of the transformed domain $\mathscr{D}'$, $\mathscr{D} \subset \mathscr{D}'$ which is visualized as adding a new diagonal to the original DP matrices(See the gray point along the diagonal in Figure 3

Thus, a recursive divide-and-conquer algorithm is considered within the transformed domain $\mathscr{D}'$. Without loss of generality, assume that n' is a power of two (If n' is not a power of two, additional unused entries are added. In implementation, only logical entries are added and several branch instructions can avoid the unused computation). The DP matrices is partitioned into ten sub-matrices, which consist of three triangular matrices and six rectangular matrices (See Figure 3). X is the original DP matrices with 2^k size,

then it is partitioned into ten submatrices:

$$\mathbf{X} = \begin{pmatrix} X_{11} & X_{12} & X_{13} & X_{14} \\ & X_{22} & X_{23} & X_{24} \\ & & X_{33} & X_{34} \\ & & & X_{44} \end{pmatrix}$$

The sub-matrices along diagonal $X_{11}, X_{22}, X_{33}, X_{44}$ are self-contained, that is, each entries only depends on other entries in the same sub-matrices. In addtion, X_{12} only depends on X_{11} and X_{22}, X_{34} only depends on X_{33} and X_{44}. If combining X_{11}, X_{12}, X_{22} and X_{33}, X_{34}, X_{44} into two larger sub-matrices, respectively, we get two independent DP matrices with 2^{k-1} size and can be divided recursively. Thus, recursive function $\mathscr{G}$ is defined:

$$\begin{pmatrix} \mathrm{A} & \mathrm{C} \\ & \mathrm{B} \end{pmatrix} = \mathscr{G}\begin{pmatrix} \mathrm{A} & \mathrm{C} \\ & \mathrm{B} \end{pmatrix}$$

where A and B are triangular matrices, and C is a rectangular matrices. All entries in three matrices are unknown. Thus, the two DP sub-matrices are computed recursively using $\mathscr{G}$

$$\begin{pmatrix} X_{11} & X_{12} \\ & X_{22} \end{pmatrix} = \mathscr{G}\begin{pmatrix} X_{11} & X_{12} \\ & X_{22} \end{pmatrix}$$

$$\begin{pmatrix} X_{33} & X_{34} \\ & X_{44} \end{pmatrix} = \mathscr{G}\begin{pmatrix} X_{33} & X_{34} \\ & X_{44} \end{pmatrix}$$

After $X_{11}, X_{12}, X_{22}, X_{33}, X_{34}, X_{44}$ are computed, we can solve the remainder four rectangular sub-matrices. Because of the data dependencies, X_{23} should be computed firstly and can only depends on X_{22} and X_{33}. Then, once X_{22} and X_{33} have been computed, X_{23} can be computed immediately. We define another recursive function $\mathscr{F}$:

$$\begin{pmatrix} \mathrm{A} & \mathrm{C} \\ & \mathrm{B} \end{pmatrix} = \mathscr{F}\begin{pmatrix} \mathrm{A} & \mathrm{C} \\ & \mathrm{B} \end{pmatrix}$$

where A and B are triangular matrices, and C is a rectangular matrices. A and B have been computed and C is an unknown matrices only depends on A and B. Then X_{23} is computed using recursive $\mathscr{F}$:

$$\begin{pmatrix} X_{22} & X_{23} \\ & X_{33} \end{pmatrix} = \mathscr{F}\begin{pmatrix} X_{22} & X_{23} \\ & X_{33} \end{pmatrix}$$

Now, we define two tensor operations $\otimes$ and $\oplus$. Let matrices $A = (a_{ij})_{s\times s}$,$B = (b_{ij})_{s\times s}$,$C = (c_{ij})_{s\times s}$.

Definition 1. $\forall a_{ij} \in A, b_{ij} \in B, c_{ij} \in C,\ 1 \le i, j \le s$, if $c_{ij} = min_{k=1}^{n}\{c_{i,j}, a_{i,k} + b_{k,j}\}$, then $C = A \otimes B$.

Definition 2. $\forall a_{ij} \in A, b_{ij} \in B, c_{ij} \in C,\ 1 \le i, j \le s$, if $c_{ij} = min\{a_{i,j}, b_{i,j}\}$, then $C = A \oplus B$.

After X_{23} has been computed, both X_{13} and X_{24} can be computed. Assume that we first compute X_{13}. Since X_{12} and X_{23} are known rectangular sub-matrices, we can compute the partial results of sub-matrices X_{13} using equation (3)

$$X_{13} = X_{13} \oplus (X_{12} \otimes X_{23}) \tag{3}$$

then, we complete the computation of X_{13} by recursive function $\mathscr{F}$:

$$\begin{pmatrix} X_{11} & X_{13} \\ & X_{33} \end{pmatrix} = \mathscr{F}\begin{pmatrix} X_{11} & X_{13} \\ & X_{33} \end{pmatrix}$$

With the same method, X_{24}, X_{14} also can be computed:

$$X_{24} = X_{24} \oplus (X_{23} \otimes X_{34}) \tag{4}$$

$$\begin{pmatrix} X_{22} & X_{24} \\ & X_{44} \end{pmatrix} = \mathscr{F}\begin{pmatrix} X_{22} & X_{24} \\ & X_{44} \end{pmatrix}$$

$$X_{14} = X_{14} \oplus (X_{12} \otimes X_{24}) \tag{5}$$

$$X_{14} = X_{14} \oplus (X_{13} \otimes X_{34}) \tag{6}$$

$$\begin{pmatrix} X_{11} & X_{14} \\ & X_{44} \end{pmatrix} = \mathscr{F}\begin{pmatrix} X_{11} & X_{14} \\ & X_{44} \end{pmatrix}$$

Divid the DP matrices recursively into smaller sub-matrices until the size of sub-matrices is small enough to be contained in cache totally. For these small sub-matrices, **dp_standard** is called to finish computing in recursive function $\mathscr{G}$, where all entries in the sub-matrices are unknown. For the small sub-matrices in recursive function $\mathscr{F}$, only one of the three sub-matrices need to be computed. So **dp_return** is called in such case (See Figure 4).

Equations (3)(4)(5)(6) are four elementary operations in the recursive implementation and have an uniform formal: $C = C \oplus (A \otimes B)$, which is implemented as **dp_base** (See Figure 5). It is obvious that **dp_base** is analogous to dense matrix multiplication, thus most of the optimization in dense matrix multiplication also can be applied to **dp_base**. Procedures **dp_tri** and **dp_rect** in Figure 6 implement the recursive algorithm. In the implementation, the return of one recursive is relaxed when the size of submatrices can entirely be fit into the cache totally.

A direct tiled implementation of DP iteration as shown in only improves little performance. We now apply a tiling approach to the cache oblivious recursive DP algorithm. In order to better match the recursive algorithm, it seems that Morton data layout is more effective. The DP matrices is partitioned into two triangles and one rectangles. These submatrices are laid out contiguously in the memory. Each of these submatrices is further recursively divided and laid out in the same way. At the end of recursion, elements of the submatrices are stored contiguously. As shown in the experiments presented in section 5, the tiling to the new cache oblivious algorithm improves the performance greatly.

```
dp_return(matrices m, int n)
for (j = n/2; j < n; j++)
    for (i = n/2 - 1; i ≥ 0; i--) {
        t = m[ij]
        indxj = indx[j];
        ij = i+indxj;
        t = m[ij]
        for (k = i; k < j; k++)
            t=min2(t, m[i+indx[k]]+m[k+1+indxj])
        m[ij] = t
    }
```

Figure 4: Pseudocode procedure for the base case of recursive algorithm.

```
dp_base(matrices A, matrices B, matrices C, int n)
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        t = C[i][j]
        for (k = 0; k < n; k++)
            t=min2(t, A[i][k]+B[k][j])
        C[i][j] = t
    }
```

Figure 5: Pseudocode procedure for the basic operations of recursive algorithm.

```
dp_tri(matrices A, matrices B, matrices C, int n)
{
if(n ≤ b)
    dp_standard(C,n);
sub_size=n/2;
dp_tri(A11, A12, A22, sub_size);
dp_tri(B11, B12, B22, sub_size);
dp_rect(A, B, C, n)
}

dp_rect(matrices A, matrices B, matrices C, int n)
{
if(n ≤ M)
    dp_return(C,n);
sub_size=n/2;
dp_rect(A22,B11,C21,sub_size);
dp_base(A12,C21,C11,sub_size);
dp_rect(A11,B11,C11,sub_size);
dp_base(C21,B12,C22,sub_size);
dp_rect(A12,B22,C22,sub_size);
dp_base(A12,C22,C12,sub_size);
dp_base(C11,B12,C12,sub_size);
dp_rect(A11,B22,C12,sub_size);
}
```

Figure 6: Pseudocode implementation for two recursive procedures. $|A| = |B| = |C| = n/2$

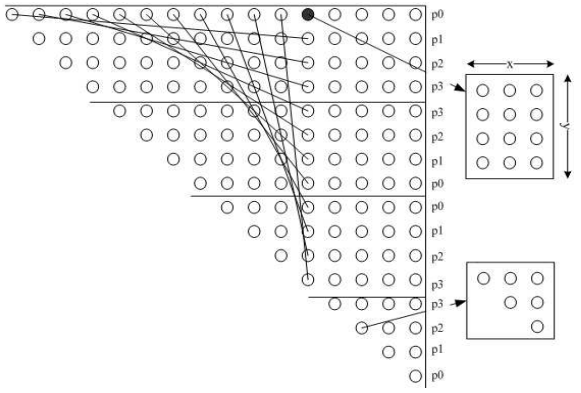


Figure 7: The tiled triangular domain. In the transformed domain the dependences are the same those of the original domain. The tile along the diagonal is triangle and others are rectangle whose width and height are x and y respectively. This figure illustrates the case $p = 4$ and the size of tiled space is 16.

4 Parallelism Optimization for Tiling Algorithm

Parallelization using a tiling approach for the nonserial polyadic DP algorithm is a problem of tiling the triangular iteration space with nonuniform dependence problem. The computation of a point (i, j) depends on $O(j-i)$ points, that is, all the points in the domain from the ith row and all the points from the jth column have something to do with the value of the function in the point (i, j). The dependence vectors are variable with (i, j) (See Figure 7). Along the diagonal in triangular domain, the computation can be partitioned into multistages and obviously the dependences appear among nonconsecutive stages. The computation in stage k depends on stages from $1, 2, ..., k-1$, not only stage $k-1$ for the uniform dependence problems. We first describe a new tile-processor mapping algorithm for parallel tiled DP algorithm, then give a new tile schedule strategy to overlap communication with computation.

4.1 A Tile-processor Mapping Algorithm

Assume that we have p processors and the size of tiled triangular iteration space is n. First we address the problem of tile-processor mapping for load balance. For other similar matrix problem, block-cyclic distribution has been proven to be an efficient method [lin].

In previous work [J. H. Chen and Maizel 1998][F. Almeida and Gonzalez 2002] for parallelizing the tile graph, the simple block-cyclic distribution maps tiles to the processors in a column/row wise modulo p fashion; i.e. row i is allocated to i mod p processor. This simple block-cyclic mapping of tiles to processors ensures that vertical/horizontal successive tiles are mapped to the same processor and no communication is involved. We note that the computation of each point is different and the cost of computation is more expensive when the points are closer to the right-up part in the iteration space. Thus, this causes a substantial workload unbalance among processors. To address this problem, an alternate block-cyclic distribution is presented in Figure 7. This alternate block-cyclic mapping distributes the same number of iteration points to all processors. That is to say, each processor has the same memory complexity of $O(m^2/p)$, where m is the size of original iteration domain.

A column/row of tile is called macro column/row. The entire iteration space is partitioned into n/p strips containing p macro rows. For each strip, only the first tile on each processor is triangular and the others are rectangular. Each strip can be visualized as trapezoid. The strips are alternatively reorganized and coalesced into a parallelogram (See Figure 8). The parallel algorithm proceeds in a pipeline way from strip 0 to n/p where p processors compute their own tiles along diagonal in parallel within one strip. In each strip the computational cost becomes larger from the bottom macro row to the upper one. In our proposed alternate block-cyclic mapping, processor i which computes the bottom macro row in strip k will compute the upper one in strip $k+1$. Processor i can immediately begin to compute the macro row in strip $k+1$ in that computation of its dependent points have been finished in the previous strips. In this sense, a pipeline is formed among the strips. Transparently, our alternate block-cyclic distribution achieves computation load balance.

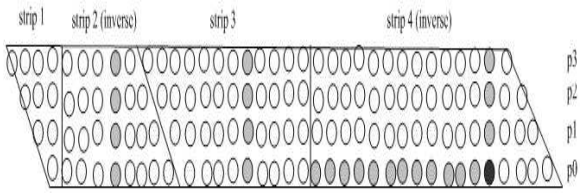


Figure 8: The coalesced iteration space. The trapezoid strips are alternately inverse. The dark point (corresponding to the dark point in Figure 7) depends on all the gray points.

4.2 Optimial Tile Parameters

We call the parallel step of computing a tile as a macrostep. In this section we give an analytical solution concerning the optimal tile volume $v = x \times y$. Let us consider the parallelogram iteration space

as affine to the original iteration space (m) with width w and height h. With our alternate block-cyclic distribution on p processors, we obtain h/py passes and the size of each of size $py \times w$. Without loss of generality, we assume the span of each pass is divided by x. Since the last period of a pass is the w/x macrosteps and requires p time steps to finish, we get total parallel time steps T_m:

$$T_m = \frac{h}{py}\sum_{i=1}^{w}\frac{i}{x} + p = \frac{m(m+1)+2}{4pv} + p \tag{7}$$

Although the arithmetic complexity of calculating each point is different, the total arithmetic complexity on each processor is the same using our alternate block-cyclic distribution. Let us denote γ and ω as the average time and number of tiles used to execute a single point in the original iteration domain, α, β is network communication startup and transfer latency. The execution time of one macrostep is given by P_m:

$$P_m = (\alpha + (p-1)\beta\omega v) + \gamma v \tag{8}$$

Combing 7 and 8, we therefore obtain the total execution time T:

$$T = T_m \times P_m = \frac{(m^2+m+2)\alpha}{4pv} + ((p-1)\omega\beta + \gamma)pv + \sigma \tag{9}$$

where $\sigma = \frac{(m^2+m+2)((p-1)\omega\beta+\gamma)}{4p} + p\alpha$. Therefore, in order to find the optimal tile size, we need to minimize the running time T in 9 is a convex function and the volume $v_\star$minimizing this function is given by:

$$v_\star = \frac{1}{2p}\sqrt{\frac{(m^2+m+2)\alpha}{((p-1)\omega\beta+\gamma)}} \tag{10}$$

4.3 Overlapping Communication with Computation

Because of the dependences between successive phases, the computation of one tile is restricted to a serialized process: $communication, compute, communication$. Total execution tiles consists of successive phases interleaved with communication ones. A processor receives the data needed to execute a tile at time step k, and performs the computations and then sends to other processors, which will be used for tile calculation in time step k+1. The total execution time is given by:

$$T = T_m \times P_m = T_m \times (T_{comm} + T_{comp})$$

where T_{comm} and T_{comp} are the communication and computation time, respectively. Each time step contains a triplet of receive-compute-send for each tile. Since it exploits all inherent parallelism at the tile level where all processors concurrently either compute or communicate with the dependent processors, the parallel algorithm can achieve reasonable speedup. However, each processor inevitably has to wait for essential data before starting the computation of a certain tile and wait for the transmission of the results to other processors, thus resulting in significant idle processor time when sizes of messages are becoming larger.

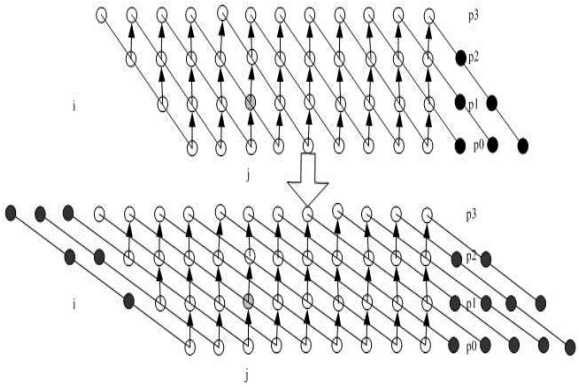


Figure 9: Non-overlap (upper) and overlap (bottom) tile schedule. The dark point is unused tile representing processor idle. The arcs only depict the actual data flow involved with communication. This figure only depicts the tile schedule on one strip. Actually, in parallelogram representing the entire iteration space these idle points are replaced by the points in successive strip with a pipeline way.

The communication time can be divided into startup time $T_{startup} = \alpha$ and messages transfer time $T_{trans} = (p-1)\beta\omega v$. The burden of actual data transmission is removed from CPU on modern high performance network architectures. So it is reasonable to perform an algorithmic transformation to schedule the execution of tiles so that a processor node can receive, compute and send data at the same time. Figure 3 depicts two kinds of tile schedule strategy: non-overlapping and overlap schedule. In the overlap schedule, assume that processor p_k is computing tile (i, j), then p_{k+d} is computing tile $(i-d, j-2*d)$ (i.e. p_{k-1} is computing tile $(i+1, j+2)$, p_{k+1} is computing tile $(i-1, j-2)$) whereat time step t. Thus, the tile data at time step t on each processor is delayed to be needed by other processors at time step $t+2$. Therefore, processor p_k computes its tile at t time step and concurrently receives data from other processors to use them at $t+1$ time step and sends data produced at $t-1$ time step.

Although the idle points to the right of a strip can be eliminated with a pipeline way, the overlap schedule results in pipeline startup overhead, that is, it increases the number of macrosteps by $p-1$, where $T'_m = T_m + p - 1$. However, the cost of each macrostep is given by:

$$P'_m = max\{T_{comm}, T_{comp}\} = max\{\alpha + (p-1)\beta\omega v), \gamma v\}$$

When $T_{comp} = T_{comm}$, it can achieve a theoretical maximum reduction in time by a factor of 1. In fact, we should note that only some communication can be overlapped by tile computation because the communication startup is handled by CPU, and in this sense the theoretical speedup is less than 2.

5 Performance Results and Analysis

5.1 Memory Hierarchy Performance

We experimented on four commonly used modern computing platforms-Opteron, Xeon, Power4, PowerPC. The configuration of the platforms is listed in Table 1, which demonstrates the various parameters of the processor, the underlying memory system and the compiler flags.Primarily we are interested in execution time or MFLOPS of the algorithms. Table 2 shows the running time on six different platforms with increasing problem sizes. The number of

Table 1: Machine configuration for the various platform used for experiments. The cache are of the form Capacity/Line size(C/L).

Parameter	Opteron	Xeon(P4)	Power4	PowerPC
clock rate	1.6Ghz	2.4GHz	1.3Ghz	400MHz
L1 data cahce	64KB/32B	8KB/64B	32KB/128B	32KB/128B
L2 cache	1MB/64B	512KB/64B	1MB/128B	1MB/128B
data TLB entries	32	64	1024	128
TLB associativity	direct	direct	128	16
VM page size	4KB	4KB	64KB	64KB
Compiler	pgcc	icc	xlc	xlc
Option	-O3	-O3	-O5	-O5
Operating system	SuSE	Redhat	AIX	AIX

Table 2: Running time of three implementations on six different platforms. The notations used in table are explained: s: standard implementation, r: recursive without tiling implementation, r+t: recursive with tiling implementation. The time of recursive with tiling implementation is the best instance for all different tiling sizes.

size	Opteron			Xeon(P4)			Power4			PowerPC		
	s	r	r+t	s	r	r+t	s	r	r+t	s	r	r+t
1000	2.93	1.92	1.01	2.96	1.17	0.69	2.08	1.48	0.87	10.42	5.23	1.38
1500	12.44	5.68	2.73	10.48	7.50	3.64	9.64	6.04	3.34	52.60	28.25	4.92
2000	32.94	18.11	7.15	26.29	19.72	8.60	28.95	18.23	8.47	142.86	86.98	12.07
2500	68.17	40.43	25.25	59.66	40.63	27.69	65.64	39.46	24.41	296.93	194.45	30.64
3000	122.17	77.17	32.61	113.27	72.54	40.54	127.20	74.94	33.81	533.11	368.11	43.10
3500	199.22	134.22	78.47	212.58	128.44	92.67	219.57	148.05	74.04	908.97	626.97	86.19
4000	304.03	211.63	89.53	376.92	203.90	106.81	366.40	241.89	87.84	1466.87	982.94	105.85

L1 cache misses, L2 cache misses and TLB misses on Opteron are used to explain the trends in execution time.

Figure 10 shows that L1 data cache misses for the three implementations on Opteron. The number of cache misses for two optimized algorithms are much lower than the standard algorithms. The pure recursive algorithm reduces the number of cache misses by 1-2 times, and the hybrid recursive and tiled algorithm reduces the number of cache misses by 2-3 times. The plots show an increase in the number of L1 data misses as the size of the problems becomes larger, which is due to an increase in the capacity and conflict misses. In the experiment, the problem size varies with 500. The number of L1 data cache misses in recursive without tiling increases with the increase problem size by 500. However, when the problem size is increased by 500, the number of L1 data cache misses in recursive with tiling almost doesn't increase. Only when the problem size is increased by 1000, we notice the notable increase in cache misses. The conflict misses are due to cross interference among the auxiliary array, where the tiling technique removes some conflict misses.

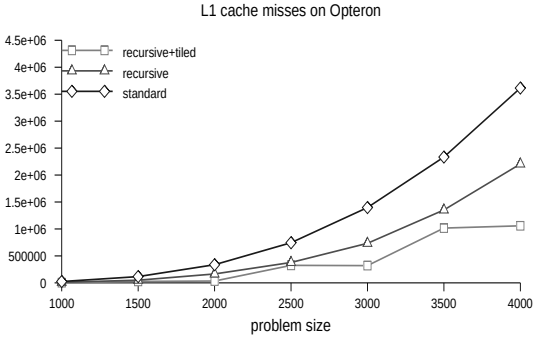


Figure 10: L1 Cache misses comparison on Opteron

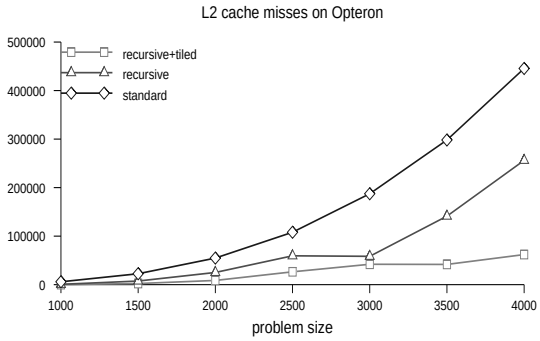


Figure 11: L2 Cache misses comparison on Opteron

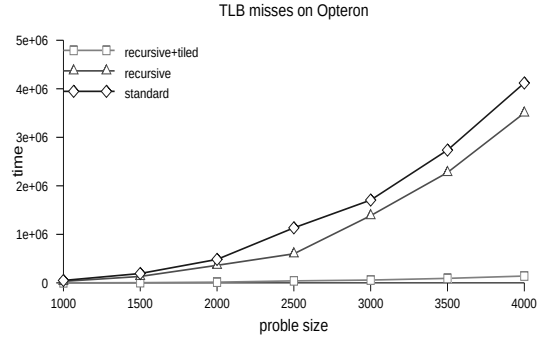


Figure 12: TLB misses comparison on Opteron

Figure 11 exhibits the L2 cache misses for three implementations on the Opteron. The recursive algorithm without tiling reduces the number of L2 data cache misses by 1-2 times when compared with the standard implementation. But the L2 cache misses have a notable increase when the problem size is larger than 3000 because of capacity misses. The recursive algorithm with tiling reduces the number of L2 data cache misses by about 5 times comparing the standard implementation. Although the number of L2 cache misses is much smaller than that of L1 data cache misses, the miss latency of L2 cache misses is 2-3 times longer than that of L1 cache misses. L2 cache miss also plays an important on reducing the running time.

Another important reason for decreasing in running time for recursive algorithm is explained by TLB misses. Figure 12 shows the TLB misses of three implementations on the Opteron (There are two levels TLB on Opteron, Figure 12 gives the sum of two levels TLB misses). In the recursive implementation, TLB thrashing is avoided by tiling the sub-matrices, so the number of TLB misses is reduced greatly. Comparing the above running time plots with cache and TLB performance plots, we find that the trend in running time accords with the trend in TLB misses. Although we don't measure the TLB misses on the PowerPC, the larger 10 times reductions in running time can be explained by the larger page size 64KB. The DP formulation only needs to compute a triangular matrices, the data is laid out by column/row index by an additional index array. So in the standard and recursive without tiling implementation cause irregular memory accesses. In this case, larger page size results in more TLB thrashing. The blocking recursive implementation rearranges the data accesses at the cost of sub-matrices copy operations, however, it improves the TLB performance greatly.

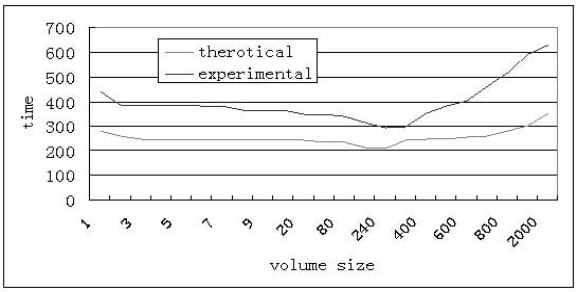


Figure 13: Comparison of experimental and theoretical minimum.

5.2 Communication Peformance

The performance of the parallel algorithm will be tested through experimentation on a cluster with 16 Opteron processors running at 2.2GHz, each with 3GB RAM. The processors are connected to Infiniband network. The parallel programming environment is MVAPICH2-0.9.2 which is a high performance MPI on Infiniband designed by OSU [iba][J. Liu and Panda 2004]. The used communication operators are MPI_Get/MPI_Put one-sided communication.

In order to validate the tile parameter formula, we estimate the overhead through a ping-pong test benchmark. Table 3 shows the average results in terms of the latency and bandwidth. The arithmetic time for computing a single instance (i, j) of the triangular iteration is an $O(j-i)$ function. Like the approach in [F. Almeida and Gonzalez 2002], we approximate it by the average running time of the sequential algorithm over the whole iteration space, i.e. $\gamma = R(n)/\frac{n^3-n}{6}$ (which is more reasonable than the equation in [F. Almeida and Gonzalez 2002]), which denotes the time for

Table 3: Latency and bandwidth for MVAPICH2 on Opteron. The message size unit is byte.

type	message size	MVAPICH2 on Opteron
latency	0	3.21 *us*
bandwidth	1	1.22MB/s
	16	19.85MB/s
	512	429.02MB/s
	1024	638.91MB/s
	16384	893.367MB/s
	65536	801.540MB/s

computing the entire iteration space. In Figure 13, the experimental result is compared to our analytical calculations deriving from 9. Although the theoretical analysis includes a detailed actual time delay parameters, there is a gap between the theoretical and experimental time. This gap results from our approximating the running time and message sizes of computing a single tile. Our theoretical formula uses an average value instead of a variable value with the position of a tile. However, we observe that the theoretical formula is used to find an optimal tile volume in our model. The trendline for the theoretical function is close to the experimental time measured. The difference between experimental minimum and theoretical minimum is rather small.

We are interested in the speedup factor, which is a measure of the performance of a parallel algorithm. Figure 14(a) and Figure 14(b) show the parallel execution time and the speedup obtained as the number of processors increase for overlapping and non-overlapping parallel tiled algorithm. It is transparently to all that overlapping executions which take advantage of the higher performance communication architecture are faster than the non-overlapping ones. The reduction of execution time is more drastic when the computation-communication ratio is closer to 1.

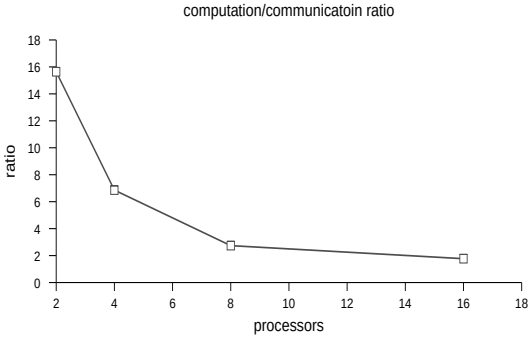
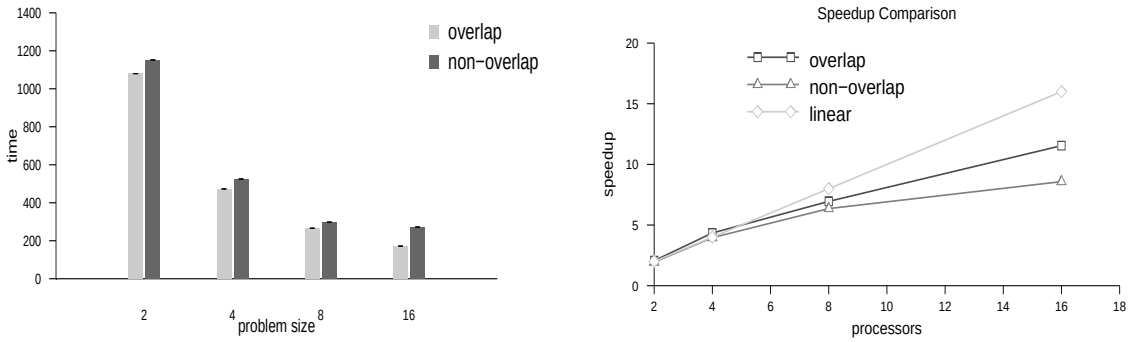


Figure 15: The ratio of computation to communication

Inevitably, There is a gap between the speedup of the parallel algorithm and linear speedup. Since the processor idle at the startup or end of pipeline can be negligible, a more reasonable explain is that the decrease of the ratio of computation to communication. Figure 15 plots the computation to communication ratio obtained by the parallel algorithm without communication-computation overlapping, from which one can easily draw a significant conclusion. An important observation is that communication startup is handled by CPU. When the number of processors becomes larger, the size of strips on each processor becomes little, that is, the size of communication messages becomes little. At the same time, the cost of communication startup increases with the number of processors



(a) (b)

Figure 14: Comparison of the total parallel execution time and speedup

due to all-pairs communication for each tile. Therefore, the speedup of overlapping over non-overlapping can not obtain the theoretical maximal speedup 2.

6 Conclusions

In recent years many research groups have systematically study the course of optimization for the algorithms used in bioinformatics. In order to apply a quantitative approach in computer architecture design, optimization and performance evaluation, workload benchmark suits of representative applications from the biology and life sciences community, where the codes are carefully selected to span a breadth of algorithms and performance characteristics are proposed [D.A. Bader and Singh 2005][K. Albayraktaroglu and Yeung 2005][Y. Li and Fortes 2005]. In our previous work, we have investigated the runtime performance for several commonly used algorithms on memory hierarchy architectures in order to develop optimization techniques for optimizing applications in bioinformatics [G. Tan and Sun 2006]. A common features of these work is to focus on the dynamic programming algorithms.

In this paper, we have demonstrated decreased running times for nonserial polyadic dynamic programming algorithm by improving locality using combination of algorithmic ideas and architectural capabilities. The cache oblivious technique for optimizing DP algorithms has reduced the running time sharply in current deep memory hierarchy architectures. An important fact is that we have to perform algorithmic transformation to use general optimization techniques such as tiling and blocking. On parallel architectures we addressed and resolved the problem of optimally tiling the iteration domain with nonuniform dependences for parallelizing nonserial polyadic dynamic programming algorithm. Although the benefits of such tiling have been well-known in the past decade, there was no systematic method to choose tiling parameters with computation and memory load balance. We have proposed an alternate block-cyclic distribution to map the triangular iteration space to processors. An optimal tile volume size to minimize the total execution time has also been shown in an analytical and experimental way. As far as we known, another key aspect is that we have proposed an algorithmic transformation to overlap communication with computation for such tiling problem in the first time (In fact, this method also is applicable to uniform iteration space tiling problem). The technique of our algorithmic transformation a successful attempt and guidance for developers to optimize an application code in order to exploit the communication-computation overlapping. Confining to the available experimental platform to us, we can not run our test on a larger size of processors. However, we can derive that the communi-cation-computation overlap parallel algorithm will perform better on a large scale parallel system in a fine grained mode, which will be mainstream parallel computing to achieve petaflops [pet] performance in the future.

References

A. DANALIS, K. Y. KIM, L. P., AND SWANY, M. 2005. Transformations to parallel codes for communication-computation overlap. In *Supercomputing*.

A. GRAMA, A. GUPTA, G. K., AND KUMAR, V. 2003. *Introduction to Parallel Computing*. Addison Wesley.

ADIGA, N. R., AND ET AL. 2002. An overview of the bluegene/l supercomputer. In *Supercomputing*.

ANDONOV, R., AND BALEV, S. 2003. Optimal semi-oblique tiling. *IEEE Transaction on Parallel and Distributed Systems 14*, 9, 944–960.

ANDONOV, R., AND RAJOPADHYE, S. 1997. Optimal orthogonal of 2d iterations. *Journal of Parallel and Distributed Computing 45*, 159–165.

BRADFORD, P. G. 1992. Efficient parallel dynamic programming. In *In 30th Annual Allerton Conference on Communication, Control and Computing*, 186–194.

http://lacsi.rice.edu/software/caf/.

COLEMAN, S., AND MCKINLEY, K. S. 1995. Tile size selection using cache organization and data layout. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

D. DUNNING, G. REGNIER, G. M. D. C. B. S. F. B. A. M. E. G., AND DODD, C. 1998. The virtual interface architecture. *IEEE Micro 18*, 2, 66–67.

D.A. BADER, V. SACHDEVA, V. A. G. G., AND SINGH, A. 2005. Bioperf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *IEEE International Symposium on Workload Characterization*.

F. ALMEIDA, R. A., AND GONZALEZ, D. 2002. Optimal tiling for rna base pairing problem. In *ACM Symposium on Parallel Architecture and Algorithm*, 173–182.

F. O. CARROLL, H. TEZUKA, A. H., AND ISHIKAWA, Y. 1998. The design and implementation of zero copy mpi using commodity hardware with a high performance network. In *Proceedings of the International Conference on Supercomputing*, 243–249.

F. PETRINI, W. FENG, A. H. S. C., AND FRACHTENBERG, E. 2002. The quadrics network: High performance clustering technology. *IEEE Micro 22*, 1, 46–57.

FRIGO, M., AND JOHNSON, S. G. 2005. The design and implementation of fftw3. In *Proceedings of the IEEE special issue on Program Generation, Optimization, and Adaptation*, vol. 93, 216–231.

G. TAN, L. XU, S. F., AND SUN, N. 2006. An experimental study of optimizing bioinformatics applications. In *Proceddings of IEEE International Parallel & Distributed Processing Symposium (HiCOMB)*.

G. VENKATARAMAN, S. S., AND MUKHOPADHYAYA, S. 2003. A blocked all-pairs shortest-paths algorithm. *ACM Journal of Experimental Algorithmics 8*.

GALIL, Z., AND PARK, K. 1994. Parallel algorithm for dynamic programming recurrences with more than o(1) dependency. *Journal of Parallel and Distributed Computing 21*, 213–222.

nowlab.cse.ohio-state.edu/projects/mpi-iba/.

Infiniband trade association, release 1.0. Tech. rep.

IRIGOIN, F., AND TRIOLET, R. 1988. Supernode partitioning. In *15th ACM Symposium on Principles of Programming Languages*, 319–329.

J. DEMMEL, J. DONGARRA, V. E. E. 2005. Self-adapting linear algebra algorithms and software. In *Proceedings of the IEEE special issue on Program Generation, Optimization, and Adaptation*, vol. 93, 293–312.

J. H. CHEN, S. Y. LE, B. A. S., AND MAIZEL, J. V. 1998. Optimization of an rna folding algorithm for parallel architectures. *Parallel Computing 24*, 1617–1634.

J. LIU, J. W., AND PANDA, D. K. 2004. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming 32*, 3, 167–198.

J. S. PARK, M. PENNER, V. K. P. 2004. Optimizing graph algorithms for improved cache performance. *IEEE Trans. on Parallel and Distributed Systems 15*, 9.

K. A. YELICK, L. SEMENZATO, G. P. C. M. B. L. A. K. P. N. H. S. L. G. D. G. P. C., AND AIKEN, A. 1998. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience 10*, 11–13.

K. ALBAYRAKTAROGLU, A. JALEEL, X. W. B. J. M. F. C.-W. T., AND YEUNG, D. 2005. Biobench: A benchmark suite of bioinformatics applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*.

L. GUIBAS, H. K., AND THOMSON, C. 1979. Direct vlsi implementation of combinatorial algorithms. In *Caltech Conference on VLSI*, 509–525.

http://www.netlib.org/linpack.

LYNGSO, R. B., AND ZUKER, M. 1999. Fast evaluation of internal loops in rna secondary structure prediction. *Bioinformatics 15*, 6, 440–445.

M. A. BLUMRICH, K. LI, R. A. C. D. E. W. F., AND SANDBERG, J. 1994. Virtual memory mapped network interface for the shrimp multicomputer. In *International Conference on Computer Architecture*, 120–129.

M. FRIGO, C. E. LEISERSON, H. P., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithm. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, 285–297.

M. PUSCHEL, J. M. F. MOURA, J. J. D. P. M. V. B. S. J. X. F. F. A. G. Y. V. K. C. R. W. J., AND RIZZOLO, N. 2005. Spiral: Code generation for dsp transforms. In *Proceedings of the IEEE special issue on Program Generation, Optimization, and Adaptation*, vol. 93, 232–275.

M. W. HALL, J. M. ANDERSON, S. P. A. B. R. M. S.-W. L. E. B., AND LAM, M. S. 1996. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*.

http://www.mpi-forum.org.

N. J. BODEN, D. COHEN, R. E. F. A. E. K. C. L. S. J. N. S., AND SU, V. K. 1995. Myrinet: A gigabit-per-second local area net work. *IEEE Micro 15*, 1, 29–36.

NUMRICH, R. W., AND REID, J. 1998. Co-array fortran for parallel programming. *ACM SIGPLAN Fortran Forum 17*, 2, 1–31.

P. EDMONDS, E. C., AND GEORGE, A. 1993. Dynamic programming on a shared memory multiprocessor. *Parallel Computing 19*, 9–22.

http://www.hq.nasa.gov/hpcc/petaflops/.

RAMANUJAM, J., AND SADAYAPPAN, P. 1991. Tiling multidimensional iteration spaces for non share memory machines. In *Supercomputing*, 111–120.

SMITH, T. F., AND WATERMAN, M. S. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology 147*, 1, 195–197.

Upc language specifications, v1.2. Tech. Rep. LBNL-59208, Lawrence Berkeley National Lab.

WOLFE, M. 1987. Iteration space tiling for memory hierarchies. *Parallel Processing for Scientific Computing*, 357–361.

WULF, W., AND MCKEE, S., 1995. Hitting the memory wall: Implications of the obvious. ACM Computer Architecture News.

XUE, J., AND HUANG, C. 1998. Reuse driven tiling for improving data locality. *International Journal of Parallel Programming 26*, 6, 671–696.

XUE, J. 1997. Communication-minimal tiling of uniform dependence loops. *Journal of Parallel and Distributed Computing 42*, 1, 42–59.

XUE, J. 1997. On tiling as loop transformation. *Parallel Processing Letters 7*, 4, 490–424.

XUE, J. 1997. Unimodular transformations of nonperfectly nested loops. *Parallel Computing 22*, 12, 1621–1645.

Y. LI, T. LI, T. K., AND FORTES, J. A. B. 2005. Workload characterization of bioinformatics applications. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*.