A Parallel Dynamic Programming Algorithm on a Multi-core Architecture

Guangming Tan[†,‡], Ninghui Sun
[†]Key Lab of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[‡]Graduate School of Chinese Academy of Sciences, Beijing, China
{tgm, snh}@ncic.ac.cn

Guang R. Gao
Computer Architecture and Parallel Systems Lab
Department of Electrical&Computer Engineering
University of Delaware, Newark, DE, USA
ggao@capsl.udel.edu

ABSTRACT

Dynamic programming is an efficient technique to solve combinatorial search and optimization problem. There have been many parallel dynamic programming algorithms. The purpose of this paper is to study a family of dynamic programming algorithm where data dependence appear between non-consecutive stages, in other words, the data dependence is non-uniform. This kind of dynnamic programming is typically called *nonserial polyadic dynamic programming*. Owing to the non-uniform data dependence, it is harder to optimize this problem for parallelism and locality on parallel architectures. In this paper, we address the chanllenge of exploiting fine grain parallelism and locality of nonserial polyadic dynamic programming on a multi-core architecture. We present a programming and execution model for multi-core architectures with memory hierarchy. In the framework of the new model, the parallelism and locality benifit from a data dependence transformation. We propose a parallel pipelined algorithm for filling the dynamic programming matrix by decomposing the computation operators. The new parallel algorithm tolerates the memory access latency using multi-thread and is easily improved with tile technique. We formulate and analytically solve the optimization problem determing the tile size that minimizes the total execution time. The experiments on a simulator give a validation of the proposed model and show that the fine grain parallel algorithm achieves sub-linear speedup and that a potential high scalability on multi-core arichitecture.

Categories and Subject Descriptors: F.2.3 Theory of Computation: Analysis of Algorithms and Problem Complexity

General Terms: Algorithm, Performance, Measurement

Keywords: Dynamic Programming, Data Dependence, Multi-core, Memory Hierarchy, Scalabilitiy

1. INTRODUCTION

Combinatorial search and optimization is used to look for a solution to a problem among many potential ones. For many search and optimization problems, dynamic programming (DP) is a classical, powerful and well-known technique for solving large kinds of optimization problems. There are many applications such as scheduling, inventory management, automatic control, and VLSI design, etc [17]. More recently, it has been found useful towards solving many problems in bioinformatics. For example, the two most important application is Smith-Waterman algorithm [26] for matching sequences of amino-acids/necleotides and Zuker's algorithm [23] for predicting RNA secondary structures. However, a combinatorial explosion limits this method's chance of being widely used because the CPU time and storage requirements can be so high. Parallel processing could be an efficient tool to solve large-scale DP problems. In fact, parallelization of DP algorithm has been a classical problem in parallel algorithm research in the last decade. In order to find efficient parallel algorithms for implementing DP, Grama, et.al. [17] present a classification of DP formulation: DP can be considered as a multistage problem composed of many sub-problems. If sub-problems at all levels depend only on the results of the immediately preceding levels, it is called a *serial* DP formulation; otherwise, it is called a *nonserial* DP formulation. Typically, there is recursive equation called a *functional equation*, which represents the solution to optimization problem. If a functional equation contains a single recursive term, the DP formulation is *monadic*; otherwise, if it contains multiple recursive terms, we call is *polyadic* formulations. Based on this classification criteria, four classes of DP formulations can be defined: 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 parenthesizeation problem and Zuker algorithm). From the view point of data dependence [31], serial DP formulation shows a uniform dependence because between subproblems is consecutive. The data dependence in nonserial DP formulation appears among non-consecutive levels, meaning that it is non-uniform. This non-uniform data dependence make the parallelization harder on current memory hierarchy and network latency computer architecture.

The rest of this paper is organized as follows: Section 2 presents a DP problem formulation and the challenges of parallelization on emerging multi-core architectures. Section 3 summarizes previous work on parallelizing the nonserial DP algorithm. In section 4, for the memory hierarchy on multi-core architecture, we construct a preliminary programming model and execution model. In order to exploit better parallelism, we perform a transformation of the data dependence for nonserial DP algorithm. Then based on the models, we proposed a parallel pipelined algorithm with load balancing for transformed nonserial polyadic DP. Furthermore, a tiling technique [21] is used to improve the performance further. Section 5 develops an analytical model for the proposed parallel algorithm. Section 6 presents the experimental results on the C64 simulator-FAST [10], which is execution driven cycle-by-cycle simulator. We conclude this paper in Section 7.

2. PROBLEM FORMULATION

A typical example of the nonserial polyadic DP is the base pair maximization algorithm for predicating RNA secondary structure. Given an RNA sequence, let $S(i,j)$ denote the folding of sub-sequence of the RNA strand from index i to j which results in the highest number of base pairs or minimized free energy. The calculation is represented as a DP formulation:

$$S(i,j) = \begin{cases} S(i+1, j-1) + 1 \\ S(i+1, j) \\ S(i, j-1) \\ max_{i<k<j} S(i,k) + S(k+1, j) \end{cases}$$

In this formulation, the last equation evaluating maximum is an nonserial polyadic dynamic programming. The data dependency and the flow of computation is depicted in Figure 1.

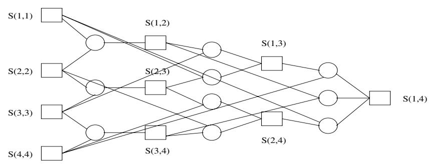


Figure 1: The flow of computing the optimal folding of an RNA sequence $s_1, ..., s(n)$, where $n = 4$

As an abstract representation, the nonserial polyadic DP formulation is defined by following recurrence:

$$c\begin{pmatrix} i \\ j \end{pmatrix} = min_{k=1}^{t(i,j)} \{ g_k (c \begin{pmatrix} i + a_1(i,j) \\ j - b_1(i,j) \end{pmatrix}, ..., c \begin{pmatrix} i + a_w(i,j) \\ j - b_w(i,j) \end{pmatrix}) \}$$

This formulation defines a $n \times n$ triangular domain with dependence vectors $\delta_l(i,j) = \begin{pmatrix} -a_l(i,j) \\ b_l(i,j) \end{pmatrix}$. Typically, the value of $t(i,j)$ is an $O(j-i)$ function which means the computation of a entry (i,j) in DP domain(matrix) may depend on several entries which has been computed. In order to simplify the presentation of the proposed algorithm, without disturbing the dependence, we instantiate the general formulation by the DP formulation appearing in RNA secondary structure prediction. In fact, our research on this instantiation also applies to the general problem because our proposed algorithm only depends on the data dependence which is not been changed. The DP matrix can be filled using following recursive formulation:

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

Larg scale multi-core architectures, which have been mainstream, have been used to build a petaflops supercomputer. There are several prototypes or real products of multi-core chips, such as IBM's Cell [18]/Cyclops64 [13], Cray's new XMT [2] and GRAPE-DR [1]. To some extent, some common features of these large scale multi-core architectures are their small on-chip memory (no data cache), explicit memory hierarchy to programmer and many threads. The memory access latency can be tolerated by multi-threads. However, to exploit locality and data reuse in the on-chip memory while achieving maximum parallelism is a challenging problem. In this work, we present our parallel algorithm based on the emerging multi-core architecture.

3. RELATED WORKS

For this family of DP algorithms with non-uninform data dependence, which obviously make the parallelization harder, there has been a lot of work on exploiting the parallelism. Bradford [7] described several algorithms, which solve optimal matrix chain multiplication parenthesizations using the CREW PRAM model. Edmonds et al. [14] and Galil et al. [16] presented several parallel algorithms on general shared memory multiprocessor systems. Another important research area is in the systolic framework, for example, Guibas et al. [19, 22] focuses on designing triangular systolic arrays. These works focus on how to reduce the complexity of arithmetic cost on different theoretical parallel models. On distributed memory multi-computer systems, the main difficulty for obtaining an efficient parallel implementation is to find a good balance between communication and computation cost. In [8, 15, 25], the authors represented parallel implementations of RNA secondary structure prediction DP algorithm. The computational load balance is satisfactory, however, the algorithm do not optimize the communication cost. The authors proved experimentally that the communication take about 50% of the execution time for a sequence of length 9212. Although a simple blocking method was used, they did not take in account the value of the startup latency, and furthermore, the processors are assumed to be permanently busy. For current machines it is an unrealistic approximation. Inspired by the blocking technique, F. Almeida [4] proposed a parallel implementation with tiling on a ring of processors. They showed the usefulness of the tiling technique for this nonuniform dependence DP. However, like the algorithms in [25], this parallel tiling algorithm can not achieve computational load balance. In their performance analytical model, the authors ignored the fact that the computation of each iteration point is different. Besides, in order to only achieve communication between two neighbors tiles, they have to keep the entire iteration in each processor. G. Tan [27] proposed a fine-grain parallel algorithm which overlapping computation with communication on cluster system connected by high performance network with RDMA mechanism. W. Zhou [33] presented a parallel out-of-core [29] algorithm for this dynamic programming problem under the conventional out-of-core model. Their research is to find a replacement strategy for in-core buffer. They used a load balance algorithm which is similar to the method in [28], but this method only can promise the number of entries on each processor is the same, the arithmetic cost on each processor is not the same because of the non-uniform data dependence.

4. THE PROPOSED ALGORITHM

Like memory hierarchy on general computer systems, it is a great challenge to to exploit parallelism while keeping locality. A general strategy on a cache memory model is to develop parallel out-of-core algorithm. For example, on IBM Cyclops64 the latency of access to each memory segment is different. However, there are many cheap hardware thread units on this multi-core architecture, which permits the memory access latency to be tolerated by use of multi-threading (that is same with other multi-core architecture). In order to facilitate the study of the methodology for designing algorithms on such a large scale multi-core architecture, it is necessary to build a programming/execution model. Because the memory hierarchy plays an important role in achieving performance, we reconstruct the conventional out-of-core model. The most important feature in this new model are the *helper threads*, which are used to tolerate memory access latency. In our proposed parallel algorithm, there are only two helper threads, one of which is used to load data from DRAM to SRAM, the other transfers data from SRAM to DRAM.

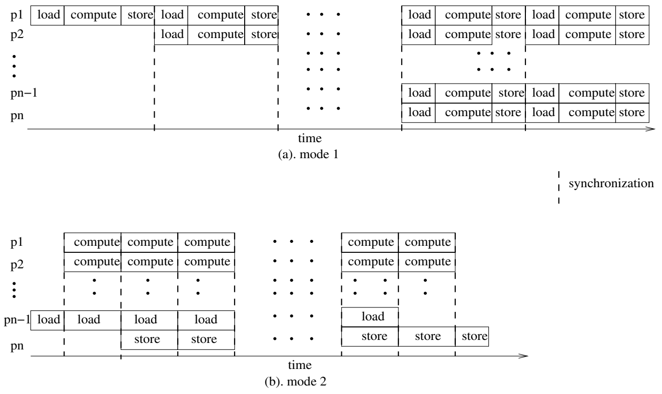


Figure 2: (a). The execution model of previous out-of-core model. (b). The execution model of out-of-core model on multi-core architecture. The number of *helper thread* depends on the architecture parameters such as bandwidth.

4.1 Programming Model

In order to exploit the locality, we refer to out-of-core programming model which is inspired by data parallel programming paradigm [6]. In fact, we can consider the based multi-core architecture-IBM Cyclops64 as a new data parallel architecture. In the out-of-core programming model mapped on IBM Cyclops64, a large array is declared with full size stored in DRAM. Consider an array that is too large to fit in SRAM/SPM on chip, called *Out-of-Core Arrays* or *OCAs*. Each time only a small section can fit in SRAM/SPM. The memory pieces in SRAM/SPM is called *In-Core Arrays* or *ICAs*. In this programming model, the locality means that operations should access *ICAs* that are in SRAM/SPM. Another important indication of this out-of-core model is that *ICAs* should be shared so that other helper threads move the data between *ICAs* and *OCAs*. So, in this model, we can use helper threads to tolerate the latency of access to *OCAs*, then release the burden on maximize locality.

4.2 Execution Model

In the framework of out-of-core model, each work thread should follow the sequential steps: *load-compute-store*. R. Bordawekar [6] proposed a *Local Placement Model* in which a worker can compute the elements in *ICAs* until it load the data from *OCAs*. At the end of each synchronization step, each thread perform a store to flush *ICAs* to *OCAs*. In their model, all operations are serialized (See Figure 2). On multi-core architecture, some threads (or idle threads) can be excluded as helper threads to overlap load/store operations with computation. However in this new execution model double *ICAs* should be available. Thus, the computation of elements in it ICAs and load/store between *ICAs* and *OCAs* are parallelized. The execution is visualized as Figure 2. In the next sections, we address the challenge of developing an efficient fine-grained parallel algorithm for non-serial polyadic dynamic programming on multi-core architecture with memory hierarchy.

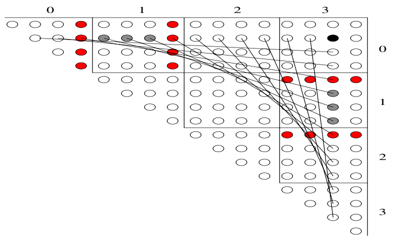


Figure 3: The blocked original DP matrix (size $n = 16$). The row and column where red elements locate need cross block reference.

4.3 Parallel Algorithms on Memory Hierarchy

Blocking is an efficient technique to exploit locality on memory hierarchy model. We use blocking strategy to exploit not only locality, but also fine grain parallelism. However, the parallelism is not enough if the DP matrix is simply blocked, because of data dependence. We apply a data dependence transformation to the original problem so that the data dependence is partially smoothed when the DP matrix is blocked.

4.3.1 A Transformation of Data Dependence

The purpose of the computation during dynamic programming algorithms is to fill a dynamic programming matrix, which can be easily implemented as a simple three nested loops. We consider this as an iteration domain problem, which can be optimized using blocking/tiling techniques. Figure 3 gives a original blocked DP matrix. The blocked matrix B does not change the data dependence. For example, $B_{0,3}$ depends on $B_{0,0}$,$B_{0,1}$,$B_{0,2}$,$B_{3,1}$,$B_{3,2}$,$B_{3,3}$. According to the programming model on memory hierarchy, only a limited number of sub-blocks are loaded into lower level memory and computed because of the small size of memory. Without loss of generality, we assume that we can load three blocks: the computed block and two other blocks which it depends on. When $B_{0,3}$ is being computed using $B_{0,1}$, $B_{1,3}$,only $B_{0,3}$, $B_{0,1}$, $B1,3$ are loaded. Following equation 1, each element pair between $B_{0,1}$ and $B_{1,3}$ is accumulated. However, the corresponding elements on the right border of $B_{0,1}$ are in $B_{2,3}$ and the corresponding elements on upper border of $B_{1,3}$ are in $B_{0,0}$ (See Figure 3), which are not in lower level memory. We call this case *cross block reference*, where the depended elements are reloaded and the parallelism within blocks decreases. Therefore, in order to achieve more blocked data reuse and parallelism, a data dependence transformation is applied to the original

DP domain.

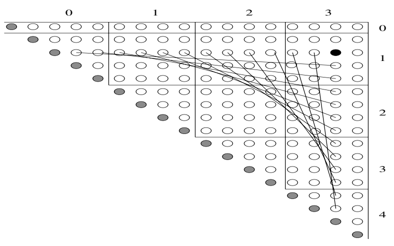


Figure 4: The blocked transformed DP matrix (size $n = 16$) where the gray points along the diagonal do not contribute to computation, the *cross block reference* is eliminated.

Assume (i, j) is the original coordinate in the original domain $\mathcal{D} = \{(i, j)|0 \le i \le j < n\}$, where $n = |\mathcal{D}|$ is the original problem size, (i', j') is the new coordinate in the transformed domain $\mathcal{D}' = \{(i', j')|0 \le i' \le j' < n'\}$, where $n' = n + 1 = |\mathcal{D}'|$ 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 equation 1 is rewritten as the new equation 2, where $a(i)$ is the known initial value (the values on the new diagonal also can be any values).

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

In the new domain, the entries on the new diagonal does 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 gets the same dynamic programming matrices with the original formulation 1 in the original domain. Thus, we have corollary 1.

Corollary 1. *$\forall (i, j) \in \mathcal{D}$ and $\forall (i', j') = (i, j + 1) \in \mathcal{D}'$, after formulation 1 and 2 are used in domain $\mathcal{D}$ and $\mathcal{D}'$, respectively, $m[i, j] = m'[i', j']$ or $m[i, j] = m'[i, j + 1]$.*

Proof: See Appendix 1.

This domain transformation ensures that the new DP formulation 2 gets the correct results. In fact, the original domain $\mathcal{D}$ is a subset of the transformed domain $\mathcal{D}'$, $\mathcal{D} \subset \mathcal{D}'$. It can be viewed as adding a new diagonal to the original DP matrices (See the gray point along the diagonal in Figure 4. Thus, the *cross block reference* is eliminated. Our parallel algorithm is considered within the transformed domain $\mathcal{D}'$.

4.3.2 Parallel Pipelined Algorithm

Let us assume that we have $p + 2$ threads, two of which are helper threads, and the size of transformed domain (DP matrix) is n. The DP matrix is divided by a block size $2\sqrt{p}$. For any block $A(i, j)$ in the blocked transformed domain, it depends on the blocks on the same row $A(i, i...j)$ and column $A(i...j, j)$. The blocks along the diagonal are triangles and they are self-contained, but there exits good parallelism for computing the triangular blocks in a diagonal-wise way. Besides, the execution time of the triangles occupies a little in the total execution time, so we focus on other rectangular blocks. Because there is data dependence between two consecutive entries in the same row and column, we can not get efficient parallelism. However, through decomposing the computation, we can exploit higher fine grain parallelism.

Based on equation 2, we define two tensor operations $\otimes$ and $\oplus$ for the blocked matrices operation. 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 $C = A \oplus B$.*

Thus, we get a formulation to compute any block $A(i, j)$:

$$\begin{aligned} A(i, j) &= \oplus_{k=i}^{j}(A(i, k) \otimes A(k, j)) \\ &= (\oplus_{k=i+1}^{j-1}(A(i, k) \otimes A(k, j))) \\ &\quad \oplus (A(i, i) \otimes A(i, j)) \oplus (A(i, j) \otimes A(j, j)) \end{aligned} \tag{3}$$

In equation 3, the computation of a block $A(i, j)$ $(i \ne j)$ is divided into two parts. The first one depends on rectangular blocks on the same row/column:

$$\oplus_{k=i+1}^{j-1}(A(i, k) \otimes A(k, j))$$

the second one depends on a triangular block and itself:

$$(A(i, i) \otimes A(i, j)) \oplus (A(i, j) \otimes A(j, j))$$

Let us take computation of $A(0, 3)$ for example in Figure 5, the first part is $(A(0, 1) \otimes A(1, 3)) \oplus (A(0, 2) \otimes A(2, 3))$, the second part is $(A(0, 0) \otimes A(0, 3)) \oplus (A(0, 3) \otimes A(3, 3))$ We observe that parallelism can be exploited at two levels for the first part. The first level is $O(j-i-1)$ $\oplus$ operations;The second level is each $\otimes$ operation. The parallelism in the second part is low because of the data dependence between two consecutive entries. However, our decomposition algorithm leverages the computation in the second part and reduces the proportion of this part. For computing any block $A(i, j)$, the number of operators ($\otimes$ and $\oplus$) is $O(j - i - 1)$ in the first part, but it is only $O(2)$ in the second part.

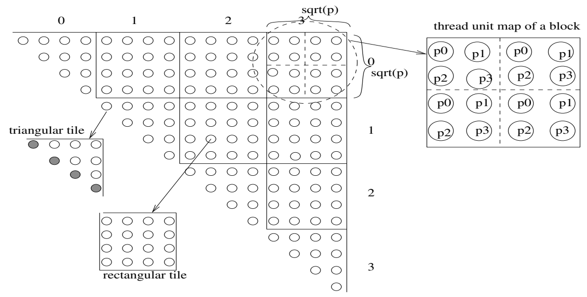


Figure 5: Each block size is $4p$. The first block in each row strip is triangular, others are rectangular. Each block is partitioned into 4 sub-blocks with size of $p = \sqrt{p} \times \sqrt{p}$ when computing a block. The elements are mapped to threads as a 2-D mesh fashion. The tile along the diagonal 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.

During computation of the second part, the sub-matrices $A(i, i)$ and $A(j, j)$ are triangular. The two operations $A(i, i) \otimes A(i, j)$, $A(i, j) \otimes A(j, j)$ depend on the final results of $A(i, j)$, so $A(i, i)$, $A(j, j)$, $A(i, j)$ are integrated into one sub-matrices, where the parallelism can be exploited along the diagonal. Now, we focus on the part of $\oplus_{k=i+1}^{j-1}(A(i, k) \otimes A(k, j))$. Obviously, each $\otimes$ for the depended blocks can be executed in parallel, which is the idea similar to previous coarse grained parallel algorithm. However, we noted that the memory access latency is different for each memory segment even though the memory address is uniformly arranged. So, in our fine grained parallel algorithm we need to find a strategy to tolerate memory access latency so that the parallel algorithm can achieve fine scalability.

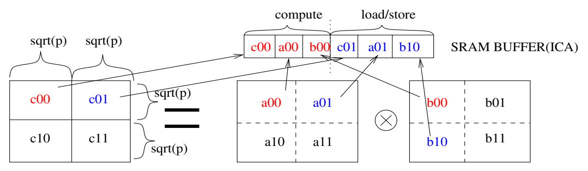


Figure 6: The computation of one block which is divided into 4 sub-blocks. The sub-blocks from the operand matrices also are divided into 4 sub-blocks. The sub-blocks are the images in SRAM buffers when computing sub-block $C(0,0)$ and loading sub-blocks $A(0,1), B(1,0)$ in *step 1*.

Because the block size is $2\sqrt{p}$, each block is divided into 4 sub-blocks with size of $p = \sqrt{p} \times \sqrt{p}$ (See Figure 6). Each element in one sub-block is mapped to one thread(See Figure 5). According to the *definition 1*, there is no dependence among all elements in a block for $\otimes$ operation, so all threads proceed in parallel. For any $i+1 \leq k \leq j-1$, we need compute $A(i,k) \otimes A(k,j)$. Let C, A and B denote $A(i,j)$, $A(i,k)$ and $A(k,j)$, respectively. The data dependence shown in equation 2 indicates that the computation of one sub-block of C needs the sub-blocks in the same row and column in B and C, respectively (This is the same with the blocked matrix multiplication). A simple strategy to implement $\otimes$ can be derived from matrix multiplication. Typically, it needs $(2\sqrt{p})^3 \times (3+1) = 32p\sqrt{p}$ (3 loads and 1 store), therefore, there exists data reuse for each sub-block.
Our algorithmic framework is to partition the threads into *computation* and *helper* threads. The *helper* threads are used to load/store between on-chip memory and off-chip memory, while *computation* threads only access on-chip memory to perform calculations. In order to reduce the number of read sub-block from DRAM, we adopt double-buffering approach allocating three SRAM buffers which contain half of each sub-block, respectively. There are two helper threads used to load/store data between DRAM and SRAM. One half of each of the three buffers is used for computation, the other half is used to transfer data. The basic idea is that the helper threads can load/store data that is used to compute the next C sub-block while the computation threads compute the current C sub-block. The computation of 4 sub-blocks of C can proceed in a pipeline style. The pipeline algorithm consists of 8 parallel steps which are described in Figure 7.

```
ParllelSteps
startup: LOAD $C00$, $A00$, $B00$;
step 1: COMPUTE $C00$; LOAD $A01$, $B10$;
step 2: COMPUTE $C00$; LOAD $C01$, $B01$;
step 3: COMPUTE $C01$; LOAD $B11$; STORE $C00$;
step 4: COMPUTE $C01$; LOAD $C11$, $A10$;
step 5: COMPUTE $C11$; LOAD $A11$; STORE $C01$;
step 6: COMPUTE $C11$; LOAD $C10$, $B00$;
step 7: COMPUTE $C10$; LOAD $B10$; STORE $C11$;
step 8: COMPUTE $C10$;
end: STORE $C10$;
```

Figure 7: The eight pipelined parallel steps for computing one block. The memory access is overlapped with computation by multi-thread, that is, the *computation* and *helper* threads proceed in parallel.

The pipeline algorithm *ParalelStpes* in Figure 7 needs 4 loads/stores from/to C, 4 loads from A, 6 loads from B, therefore the number of memory access is only $18p$. Although the memory access complexity is not optimal, we have exploited a fine parallel algorithm to overlap data transfer with computation, and thus, the memory access latency is tolerated.
For each block $A(i,j)$, the number of $\otimes$ operations required is $O(j-i-1)$. In fact, while *step 8* is computing $C10$, one of the helper threads can load the $C00$, $A00$, $B00$ for the next $\otimes$ operation. Thus, the *startup* step is removed to *step 8* so that a pipeline is reformed among the $\otimes$ operations for block $A(i,j)$.

4.3.3 Tiling

Tiling iteration domain (loop blocking)[9, 30, 24, 32] is a well-known technique used by compilers and programmers to improve data locality and to control parallel granularity in order to increase the computation to communication ratio. In our parallel algorithm based on the modified out-of-core programming model, the "*communication*" is the data transfer between DRAM and SRAM/SPM, while the locality in SRAM/SPM also should be accounted. In this case, tiling is used to minimize the total execution time of parallel program on out-of-core programming model on multi-core architecture.
We now apply a tiling approach to this parallel pipelined algorithm, which fills the transformed domain $\mathcal{D}'$. Each tile has two parameters x and y, which are called tile *height* and *width* respectively (see Figure 5). In this current work, we only consider a square tile with $x = y$ (in the rest of this paper, we only use tile parameters referring to tile width/height). In the tiled domain, each tile can be considered as a element in this new domain. In order to keep the dependence, the tiles along diagonal are triangles, the other tiles are rectangle, and both tile parameters are x. Because the data dependence in the tiled domain is the same as that equation 2, the tiled DP matrix can be filled using the proposed parallel pipelined algorithm.

5. PERFORMANCE MODELING

The study of performance modeling is confined to parallel algorithm with tiling. The basic operation is blocked $\otimes$ which contains eight parallel steps. Assume that the size of the original transformed domain is n, tile parameters is x, the number of computational threads is p. Then, the size of the tiled domain is $m = \frac{n}{x}$, which is blocked with block size of $4p = 2\sqrt{p} \times 2\sqrt{p}$. According to the proposed parallel pipelined algorithm, there are $m' = \frac{m}{2\sqrt{p}}$ row strips. In row strip i, there are $m' - i$ blocks to be filled. Because of the data dependence shown in transformed DP formulation 2, for any block $A(i,j)$ $(i \leq j < m' - i)$ in row strip i, it needs $j-i-1$ blocked $\otimes$ operations. Let $I_\otimes$ denotes the number of blocked $\otimes$ operations for filling the entire tiled transformed DP domain.

$$I_\otimes = \sum_{i=1}^{m'-2} \sum_{j=i+1}^{m'-i-1} j$$

Because $m = \frac{n}{x}$, we get the nubmer of blocked $\otimes$ operations:

$$I_\otimes = \frac{1}{24}\left[\frac{n^3}{x^3 p^{\frac{3}{2}}} - 6\frac{n^2}{x^2 p} + 8\frac{n}{x\sqrt{p}}\right] \tag{4}$$

5.1 Memory-traffic Complexity

The programming model that is used for designing algorithms that deal with these problems is similar to the out-of-core model. In the out-of-core model, an important performance measurement is I/O complexity [3, 20]. On IBM Cyclops64 multi-core system, there is no data cache, but the access latency for each memory segment is different, so this memory system can also be consider as a memory hierarchy. For example, we refer to SPM closest to hardware thread unit as *level 1*, on-chip SRAM as it level 2 and off-chip DRAM as *level 3*. However, SPM is mainly used to keep the private data for each thread, so we only use SRAM for *In-Core Arrays*. In the new out-of-core model, we refer to *memory-traffic complexity*. This is defined as the amount of memory traffic between on-chip SRAM that is smaller than problem size and off-chip DRAM that is larger than the problem size.

Lemma 1. *For the parallel pipelined algorithm, tiling with parameter x reduces the memory-traffic complexity by a factor of x, where $x = O(\sqrt{C})$ and C is the size of on-chip SRAM.*

Proof: For the parallel algorithms without tiling, the memory-traffic complexity of non-pipelined and pipelined is:

$$M_{non-pipeline} = I_{\otimes} \times 32p\sqrt{p} = O(n^3)$$
$$M_{pipeline} = I_{\otimes} \times 18p = O(\frac{n^3}{\sqrt{p}})$$

For the tiling version, the element of each single $\otimes$ operation is tile with parameter x and the volume of a tile is x^2. Then each single $\otimes$ operation needs x^2 memory traffic, so the amount of memory traffic of the blocked $\otimes$ operation is $18px^2$. Because the number of blocked $\otimes$ operations is $I_{\otimes}$, combining equation 4, the *memory-traffic complexity* is shown that:

$$\begin{aligned} M_{tile} &= I_{\otimes} \times 18p \\ &= \tfrac{1}{24}[\tfrac{n^3}{x^3 p^{\frac{3}{2}}} - 6\tfrac{n^2}{x^2 p} + 8\tfrac{n}{x\sqrt{p}}] \times 18p \\ &= O(\tfrac{n^3}{x\sqrt{p}}) \end{aligned}$$

Lemma 1 gives the upper bound of *memory-traffic complexity*. The $\otimes$ operation is similar to the basic operation in matrix multiplication, and as a result, we can use the similar technique [20] to prove the lower bound of *memory-traffic complexity* is $\Omega(\frac{n^3}{x\sqrt{p}})$, which gives us the following theorem:

Theorem 1. *The parallel tiled pipelined algorithm, which is tiled with parameter x, is asymptotically optimal with respect to memory-traffic complexity.*

The term *memory-traffic complexity* only shows the amount of memory access similar to the case on general memory hierarchy. However, we noted that there are helper threads to tolerate memory access latency on multi-core architecture. That is, besides the memory accesses are overlapped with computation, they also can be parallelized within memory bandwidth limitation using helper threads. In this performance model, assume that there is no bandwidth limitation and that the memory access for load and store is the same. We refer to another measure called *memory-traffic efficiency*. It is defined as a ratio of the time reduction percent of memory access to the number of helper threads. In our proposed parallel algorithm, we use two helper threads. If one helper thread is used for load, the other is used for store, then the 4 store operations are completely overlapped and the time reduction percent for a $\otimes$ operation is 4/18, therefore, the *memory-traffic efficiency* is 11% for 2 helper threads. However, as shown in figure 7, each parallel step only needs two memory accesses, therefore, we can schedule one idle load thread to store and the time reduction percent for a $\otimes$ operation is 8/18 and *memory-traffic efficiency* is 22%. If the number helper threads is 3, then in each parallel step all memory accesses are parallel and the time reduction percent is 10/18, but *memory-traffic efficiency* is 19%. In fact, the *memory-traffic efficiency* is determined by the parallelism in memory access. In all practical architectures there exists memory bandwidth limitation, so more helper threads do not means higher efficiency.

5.2 Execution Time

Under the execute model we now determine a analytic formulation of the execution time of our parallel program. In this work, we only use square tile. Let us denote the time to execute a single instance of equation 2 as α and , the latency of one memory access as β. Each step in *ParallelSteps* needs $\sqrt{p}$ instance of $\otimes$ operation for each thread. For the tiled algorithm, since the element of each operation is a tile with volume x^2, the execution time of computation in each parallel step is: $T_{comp} = \alpha\sqrt{p}x^3$

In each parallel step, there are only two memory accesses that are parallelized by two helper threads, so the data transfer time is: $T_{tran} = \beta px^2$

Because the helper and computation threads proceed in parallel, the time for transferring data and executing $\otimes$ operation for a tile is overlapped (the execution time should be determined by the longer one). In the startup and end of the pipeline, two extra load/store are required. Therefore, the execution time *ParallelSteps* is

$$T_{\otimes} = max\{T_{comp}, T_{tran}\} = max\{\alpha\sqrt{p}x^3, \beta px^2\} \tag{5}$$

Combining equation 4 with 5, we get the execution time of all parallel pipelined steps:

$$\begin{aligned} T_0(x) &= \tfrac{n}{2x\sqrt{p}} \times 2\beta px^2 + I_{\otimes} \times 8 \times T_{\otimes} \\ &= n\beta\sqrt{p}x + 8 \times I_{\otimes} \times max\{\alpha\sqrt{p}x^3, \beta px^2\} \\ &= \begin{cases} T_1(x) = n\beta\sqrt{p}x + 8I_{\otimes} \times \beta px^2 & x < \frac{\beta\sqrt{p}}{\alpha} \\ T_2(x) = n\beta\sqrt{p}x + 8I_{\otimes} \times \alpha\sqrt{p}x^3 & x \geq \frac{\beta\sqrt{p}}{\alpha} \end{cases} \end{aligned} \tag{6}$$

The triangular blocks on the diagonal is self-contained and their running time is:

$$T_3(x) = (\sum_{i=1}^{m'} i \times \sum_{j=1}^{4\sqrt{p}} + m' \sum_{j=1}^{2\sqrt{p}})x^3\alpha = n\alpha px^2 + \frac{n^2\alpha(4\sqrt{p}+1)}{4}x \tag{7}$$

So the optimal x is selected to minimize following formulation:

$$\begin{aligned} \mathcal{P} : Minimize \quad & T(x) = T_0(x) + T_3(x) \\ s.t. \quad & \tfrac{\beta\sqrt{p}}{\alpha} \leq x < min\{\sqrt{\tfrac{C}{48p}}, \tfrac{n}{4\sqrt{p}}\} \end{aligned} \tag{8}$$

Therefore, the objective is to select a optimal tile parameter x to minimize the function $T(x)$.

Theorem 2. *The optimal tile parameter of parallel tiling pipelined algorithm is selected by the rule:*
if $2 < p < \frac{\alpha}{\beta} min\{\sqrt{\frac{C}{48}}, \frac{n}{4}\}$, $x^* = \frac{\beta\sqrt{p}}{\alpha}$;
otherwise,

$$x^* = \begin{cases} \lfloor\frac{n}{4\sqrt{p}}\rfloor - const & n \leq \sqrt{\frac{C}{3}} \\ \lfloor\sqrt{\frac{C}{48p}}\rfloor - const & n \geq \sqrt{\frac{C}{3}} \end{cases}$$

Proof: See Appendix 2.

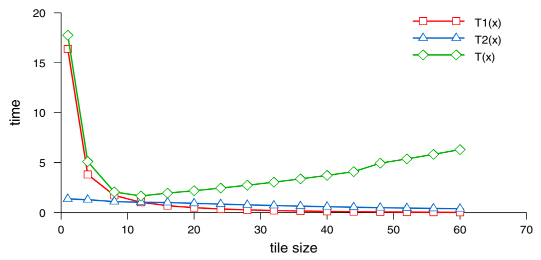


Figure 8: Finding the global minimum of the tile parameter x according to Theorem 2, $p = 16$, $n = 1024$, $x^*_{mid} = 12$ $min\{\sqrt{\frac{C}{48p}}, \frac{n}{4\sqrt{p}}\} = 64$, $x^* = 12$

We have some observations for solving this non-linear optimization problem. According to the solution to $T_0(x)$, x is expected to be larger, however the portion of computing triangular blocks is more with increase of x–that is, the portion of parallelism decreased even though the execution time of parallel pipeline algorithm is reduced. An important implication from the solving this optimal tiling problem is the scalability of the parallel algorithm. The whole solution space is partitioned by x^*_{mid}. The case that the optimal solution falling into the left of x^*_{mid} means $\frac{n}{p} < \frac{\beta}{\alpha}$.That is, the execution time is determined by the memory data transfer when the number of thread is larger than some value. Corollary 2 shows that the optimal solution locates the right of x^*_{mid}, which means the scalability of our algorithm is determined by the arithmetic operation instead of memory latency. So our proposed parallel algorithm on multi-core architecture has fine scalability with large scale processors. The solution for the global minimum in case of $n = 1024$ and $p = 16$ is shown in Figure 8

6. NUMERICAL EXPERIMENTS

IBM Cyclops64 supercomputer is an on-going project and there is no real machine to date. The simulation tool, named Functionally Accurate Simulation Toolset (FAST) [10, 12], is designed for the purpose of architecture design verification and software development. Based on the FAST simulator, a thread virtual machine (TNT) [11] is implemented to support a multi-thread programming environment. The parallel algorithms are implemented using TNT library on the simulator. Because the DP algorithm only needs to fill an upper triangular matrix, the data layout is very important towards improve its locality. However, this topic is beyond the scope of this paper. We use a linear array to store the triangular DP matrix with row-wise order. For the *ICAs* for row and column data that is depended on by other entities, the data layouts are row-wise and column-wise respectively.

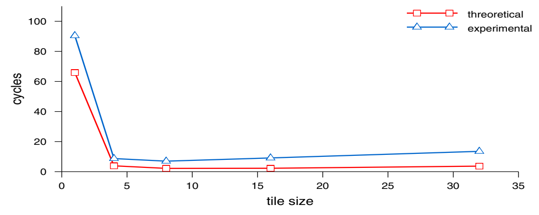


Figure 9: The comparison of theoretical and experimental execution time. $p = 16$, $n = 1024$,$x^*_{mid} = 12$ $x^* = 12$

6.1 Model Validation

We validated the performance model by comparing the theoretical to experimental execution time, which was measured on the FAST simulator. Figure 9 plots the trends of the theoretical and experimental execution time. The performance model accurately predicts the trend of execution time and gets the correct the optimal tile parameter x^*. Because the model does not take the synchronization into account, the theoretical execution time is less than the experimental execution time. When the tile size increases, the number of *parallel steps* for a given problem size decrease, thus the synchronization overhead becomes less because there is a synchronization at the end of each step. The plots in figure 9 demonstrates that the difference between theoretical and experimental execution time is reduced with the increasing tile size. Because the synchronization on the C64 is implemented by hardware efficiently [34], the performance model, which does not consider the synchronization overhead, can simulate the trend of running time and the optimal tile parameter. However, as shown in the next performance evaluation experiment, it is important to reduce synchronization overhead in order to achieve better scalability.

Table 1: The execution time of different problem size. The first row represents the running time of the serial algorithm which is implemented as three nested loops iteration. Time: seconds

#threads	256	512	1024	2048	4096
serial	1.40	11.28	90.43	226.54	1738.88
4	0.36	2.46	17.94	42.72	275.82
16	0.16	1.01	6.99	15.75	106.28
64	0.12	0.62	4.57	7.57	44.06

6.2 Performance

In this test, the execution time is obtained at the optimal tile parameter for different cases. For emphasizing the importance of locality optimization, we keep the initialized DP matrix in off-chip DRAM. Table 1 presents the running time of the original serial and optimized parallel algorithm. This work attempts to demonstrate some optimization schemes on multi-core architectures. So the naive serial algorithm is is implemented as a three nested loops iteration. The proposed parallel algorithm achieves sub-linear speedup. The locality and scalability of the algorithms are evaluated:

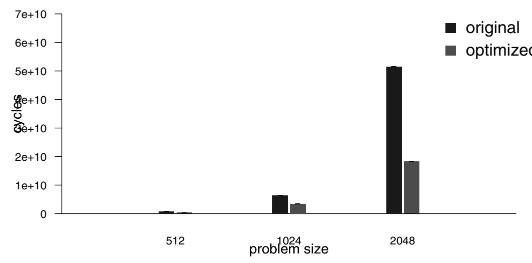


Figure 10: Comparison of the cost of off-chip memory access for different problem size.

Locality. The computation strategy in *ParallelSteps* improves data reuse, which reduces the amount of off-chip memory access, reducing the overhead of memory access. Figure 10 plots the distribution of computation and off-chip memory access time for problem size 256, 512 and 1024. Even though we do not take helper thread into account in this experiment, the cost of off-chip memory access is reduced greatly. The number of computation threads is 4, so we estimate the number of off-chip memory accesses is approximate 3 times less than the naive implementation according to the algorithmic analysis in *section 3.2.3*. In our implementation, tiling is also used to improve the locality, so the real cost of off-chip memory access is reduced by more than 3 times. In other words, the pipeline algorithm actually reduced the DRAM bandwidth through the on-chip data reuse. When the algorithm is implemented in IBM Cyclops64-like multi-core architecture, an more aggressive optimization trick is to use *LDM/STM* composed of four

LDD/STD(load/store double word) instructions to aggregate multiple memory access. Hence, DRAM requests are reduced by $\frac{1}{4}$ times so that the utilization of DRAM bandwidth is improved.

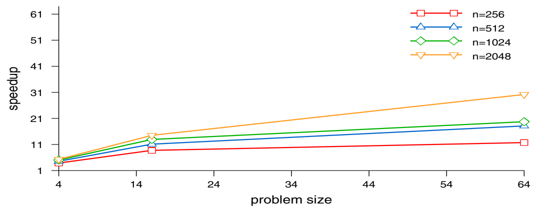
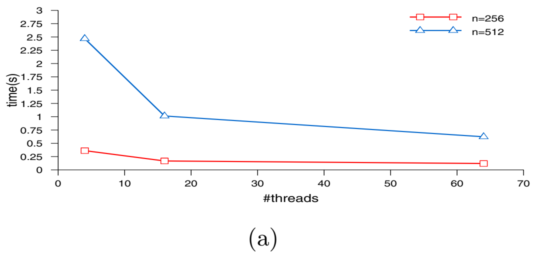


Figure 11: The speedup of our proposed parallel pipelined algorithm on different number of threads



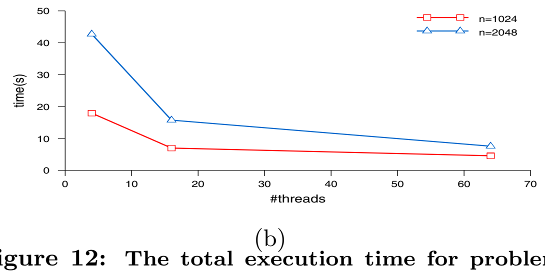


Figure 12: The total execution time for problem size 256, 512, 1024 and 2048

Scalability. First, we have measured the scalability of the proposed parallel pipelined algorithm in weak scaling experiments. In a weak scaling study, we increase the problem size as the number of compute threads increases. The speedup is defined as the ratio of the execution time of parallel program to the execution time of the original serial program. Figure 11 clearly demonstrates the scalability of our parallel DP scheme. For all problem sizes, the parallel algorithm achieves sub-linear speedups because of the greatly improvement of locality and the pipeline scheduling scheme which hide the off-chip memory access latency. This is most evident in the case where the number of threads is less than 16, and the parallel algorithm get linear speedups. The plots show that the algorithm has a fine scalability that means higher speedup for a larger scale problem size on a larger scale processor size.

Second, we have conducted strong scaling experiments. In contrast to weak scaling, we fixed the size of the problem size while increasing the number of processors in the strong scaling experiments. Figure 12 presents the strong scaling experiments results. As shown in this experiment, for a given problem size, as the number of threads increase, the reduction in execution time becomes less significant. Although there is an efficient hardware synchronization on C64, the overhead becomes significant when the cost of arithmetic and memory latency is greatly reduced for a given problem size on large scale threads. Figure 13 and 14 plot the synchronization overhead trends. For the small problem sizes such as 256 and 512, the percentage of synchronization overhead determines the the total execution time.

A barrier synchronization is inserted at the end of each step *ParallelSteps* to implement the above pipelined scheme. This guarantees that computation happens after loading all the required data, and that storing follows the corrresponding computation stage. Although a barrier can be finished in as little as dozens of cycles, since the pipeline algorithm hides the memory access latency and the time of arithmetic operations is reduced greatly by parallel thread units, the overhead of barrier synchronization become significant (see Figure 13). Because of the limitation of the simulator, we can not test larger problem sizes in time, but the experiments give some reasonable implications. It is certain that the cost of one synchronization operation increases with the larger scale of threads, but the percentage of the overhead of synchronization decreases with increasing of problem size in Figure 13. This implicates that the algorithm has a fine scalability with problem size. An interesting case in Figure 14 is that the total synchronization time decreases with the larger scale of threads. This performance benefits from the optimal tiling parallel technique. On one hand, while a larger scale of threads results in more synchronization time, it reduces the number of synchronization operations. On the other hand, the volume of a tile determines the number of *ParallelSteps* to fill the DP matrix. However, although most of the barrier synchronizations occur in *ParallelSteps*, a proper tile parameters can reduce the synchronization overhead. This causes the parallel algorithm to have reasonable scalability with the number of threads.

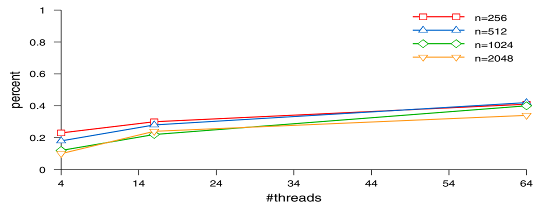


Figure 13: The synchronization overhead percentage in total execution time

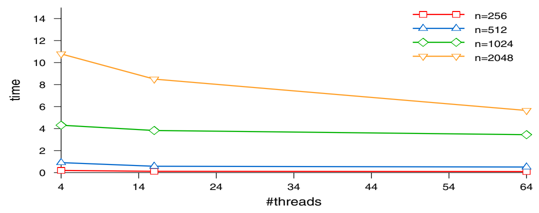


Figure 14: The synchronization overhead time

7. CONCLUSION AND FUTURE WORK

We have demonstrated an efficient scheme to exploit fine grain parallelism and locality of a dynamic programming algorithm with non-uniform data dependence on a multi-core architecture. In order to generalize program optimization technique, we have presented a programming and execution model for C64-like multi-core architectures. Moreover, this model is an extension conventional out-of-core model, therefore our proposed algorithm can be adapted to achieve high performance on conventional out-of-core model. Because experiments have shown that our proposed performance model is reasonable, we can apply a similar technique to optimize other algorithm on multi-core architecture. In fact, our solution of the optimal parameter can be incorporated into the development of automatic optimization tools or runtime functions in compilers. Besides, if we ignore the *helper threads*, the decomposition and pipeline technique in the parallel algorithm can be efficiently Obviously, In order to achieve high scalability with parallel algorithms on large scale threads, it is necessary to optimize synchronization overhead further. Another challenge is to develop a method to analytically determine the optimal number of helper threads which is used to tolerate memory access on multi-core architecture. This topic is very important to port more applications to the emerging multi-core architectures. In our on-going work, we are optimizing a graph theory algorithms benchmark SSCA#2 [5] on C64 platform. Under the framework of our proposed execution/programming model on multi-core architecture, the preliminary results show that the optimized algorithms achieve $2-6$ speedups for the original benchmark. Since SSCA#2 benchmark is memory intensive and its memory access is irregular, the determination of the optimal helper threads plays a very important role in achieving better performance.

8. ACKNOWLEDGMENTS

This work is partially supported through the support from NSFC (60633040), IBM, ET International, the Department of Defense, the Department of Energy (DE-FC02-01ER25503), the National Science Foundation (CNS-0509332), and other government sponsors. We would like to thank many CAPSL member Weirong Zhu, Andrew Russo, Ziang Hu for helpful discussions.

9. REFERENCES

[1] http://grape-dr.adm.s.u-tokyo.ac.jp/system-en.html.

[2] http://www.cray.com/products/xmt/.

[3] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1998.

[4] F. Almeida, R. Andonov, and D. Gonzalez. Optimal tiling for rna base pairing problem. acm symposium on parallel architecture and algorithm. In *ACM Symposium on Parallel Architecture and Algorithm*, pages 173–182, 2002.

[5] D. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *The 12th International Conference on High Performance Computing (HiPC 2005)*, pages 465–476, 2005.

[6] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–10, 1995.

[7] P. G. Bradford. Efficient parallel dynamic programming. in 30th annual allerton confer-ence on communication. In *Control and Computing*, pages 185–194, 1992.

[8] J. H. Chen, S. Y. Le, B. A. Shapiro, and J. V. Maizel. Optimization of an rna folding algorithm for parallel architectures. *Parallel Computing*, 24(1617-1634), 1998.

[9] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[10] J. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. In *Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*, 2005.

[11] J. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Tiny threads: a thread virtual machine for the cyclops-64 cellular architecture. In *Fifth Workshop on Massively Parallel Processing (WMPP), held in conjunction with the 19th International Parallel and Distributed Processing System*, 2005.

[12] J. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Towards a software infrastructure for the cyclops-64 cellular architecture. In *The 20th International Symposium on High Performance Computing Systems and Applications (HPCS'06)*, 2006.

[13] M. Denneau and H. S. Warren. 64-bit cyclops priciples of operation. Technical report, IBM Watson Research Center, 2005.

[14] P. Edmonds, E. Chu, and A. George. Dynamic programming on a shared memory multiproc-essor. *Parallel Computing*, 19(1):9–22, 1993.

[15] I. H. M. Fekete and P. Stadler. Prediction of rna base pairing posibilities for rna secondary structure. *Biopolymers*, 9:1105–1119, 1990.

[16] Z. Galil and K. Park. Parallel algorithm for dynamic programming recurrences with more than o(1) dependency. *Journal of Parallel and Distributed Computing*, 21:213–222, 1994.

[17] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.

[18] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, pages 10–24, March 2006.

[19] L. Guibas, H. Kung, and C. Thomson. Direct vlsi implementation of combinatorial algorithms. In *Caltech Conference on VLSI*, pages 509–525, 1979.

[20] J. Hong and H. Kong. I/o complexity: The red blue pebble game. In *Proceedings of ACM Symposium on Theory of Computing*, 1981.

[21] F. Irigoin and R. Triolet. Supernode partitioning. In *proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 319–329, 1988.

[22] B. Louka and M. Tchuente. Dynamic program-ming on two-dimensional systolic arrays. *Information Processing Letters*, 29:97–104, 1988.

[23] R. B. Lyngso and M. Zuker. Fast evaluation of internal loops in rna secondary structure prediction. *Bioinformatics*, 15(6), 440-445 1999.

[24] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for non share memory machines. In *Supercomputing*, pages 111–120, 1991.

[25] B. A. Shapiro, J. C. Wu, D. Bengali, and M. J. Potts. The massively parallel genetic algorithm for rna folding: Mimd implementation and population variation. *Bioinformatics*, 17(2):137–148, 2001.

[26] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[27] G. Tan, S. Feng, and N. Sun. Locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*.

[28] G. Tan, S. Feng, and N. Sun. Load balancing algorithm in cluster-based rna secondary structure prediction. In *proceedings of the 4th International Symposium on Parallel and Distributed Computing*, pages 91–96. IEEE Computer Society, 2005.

[29] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[30] M. Wolfe. Iteration space tiling for memory hierarchies. *Parallel Processing for Scientific Computing*, pages 357–361, 1987.

[31] M. Wolfe. Data dependence and program restructuring. *The Journal of Supercomputing*, 4:321–344, 1990.

[32] J. Xue and C. Huang. Reuse driven tiling for improving data locality. In *International Journal of Parallel Programming*, volume 26, pages 671–696, 1998.

[33] W. Zhou and D. K. Lowenthal. A parallel, out-of-core algorithm for rna secondary structure prediction. In *Proceedings of the 2006 International Conference on Parallel Processing*, pages 74–81, 2006.

[34] W. Zhu and V. C. Sreedhar and Z. Hu and G. R. Gao. Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures. The 34th International Symposium on Computer Architecture. 2007.

APPENDIX

A. PROOF OF COROLLARY 1

Proof: Diagonal traverse, horizontal traverse and vertical traverse can be used to fill the DP matrices, for simplicity, we only give the proof with diagonal traverse. Because the diagonal in the transformed domain $\mathcal{D}'$ is unused, it is ignored.

Base case: When diagonal $d = j - i = 0$, $m[i, j] = a(i)$ is the initial value and $m'[i', j'] = m'[i, j+1] = a(i)$ also is the initial value, so $m[i, j] = m'[i', j']$ or $m[i, j] = m'[i, j+1]$, where $(i', j') = (i, j+1)$

Induction step: Assume $d = j - i < p$, $m[i, j] = m'[i', j']$ where $(i', j') = (i, j+1)$, it has to be proven true for $d = j - i = p$. For $(i', j') = (i, j+1)$ and $d > 0$, we have

$$\begin{aligned} m'[i', j'] &= min_{i'+1 \le k' < j'}\{m'[i', j'], m'[i', k'] + m'[k', j']\} \\ &= min_{i+1 \le k' < j'}\{m'[i, j'], m'[i, k'] + m'[k', j']\} \\ &= min_{i+1 \le k' < j+1}\{m'[i, j+1], m'[i, k'] + m'[k', j+1]\} \end{aligned}$$

and

$$m[i, j] = min_{i \le k < j}\{m[i, j], m[i, k] + m[k+1, j]\}$$

According to the definition of domain transformation function, the initial value of $m'[i, j+1]$ equals to that of $m[i, j]$. So we only have to proof that $m'[i, k'] + m'[k', j+1] = m[i, k] + m[k+1, j]$ for $i+1 \le k' < j+1$ and $i \le k < j$. In the process of calculating DP formulation, k and k' is increased by 1 from $i+1$ and i, respectively, that is to say, $k' = k+1$, so we have

$$m'[i, k'] + m'[k', j+1] = m'[i, k+1] + m'[k+1, j+1]$$

Since $k < j \Rightarrow k - i < j - i < p$ and $i \le k \Rightarrow j - k - 1 < j - i < p$, according to the induction hypothesis, we have

$$\begin{aligned} m[i, k] &= m'[i, k+1] \\ m[k+1, j] &= m'[k+1, j] \end{aligned}$$

Thus, for $i+1 \le k' < j+1$ and $i \le k < j$, $m'[i, k'] + m'[k', j+1] = m[i, k] + m[k+1, j]$. Furthermore:

$$\begin{aligned} & min_{i \le k < j}\{m[i, j], m[i, k] + m[k+1, j]\} \\ &= min_{i+1 \le k' < j+1}\{m'[i, j+1], m'[i, k'] + m'[k', j+1]\} \end{aligned}$$

That is, when $d = j - i = p$, $m[i, j] = m'[i', j']$ or $m[i, j] = m'[i, j+1]$, where $(i', j') = (i, j+1)$.
This finishes the proof for Corollary 1.

B. PROOF OF THEOREM 2

Before we give a proof of theorem 2, we prove the following corollary 2 and 3. Note that the following properties hold:

$$\begin{aligned} T_1(x) \ge T_2(x) \quad & x \le \tfrac{\beta\sqrt{p}}{\alpha} \\ T_1(x) = T_2(x) \quad & x = \tfrac{\beta\sqrt{p}}{\alpha} \\ T_1(x) \le T_2(x) \quad & x \ge \tfrac{\beta\sqrt{p}}{\alpha} \end{aligned}$$

Therefore we need to solve the following optimization problem:

$$\begin{aligned} \mathcal{P}: \quad & Minimize \quad T_0(x) = 8 \times min\{max\{T_1(x), T_2(x)\}\} \\ & s.t. \quad x = O(\sqrt{C}) \end{aligned} \tag{9}$$

In our parallel algorithm, there are at least six SRAM buffers with sizes of p tiles, whose data type is *double*, and therefore, we get the first constraint condition:

$$x < \sqrt{\frac{C}{48p}} \tag{10}$$

Next, noting that $I_{\otimes} > 0$, we get the second constraint condition:

$$x < \frac{n}{4\sqrt{p}} \tag{11}$$

Therefore, by combining equations 4, 9, 10, 11 an instance of the optimization problem is produced as follows:

$$\begin{aligned} \mathcal{P}': \quad & Minimize \quad T_0(x) \\ &= \frac{1}{24}\begin{cases} T_1(x) = \frac{n^3\beta}{\sqrt{p}x} + 9n\beta\sqrt{p}x - 6n^2\beta & x \le \frac{\beta\sqrt{p}}{\alpha} \\ T_2(x) = 8n\alpha x^2 + (n\beta\sqrt{p} - \frac{6n^2\alpha}{\sqrt{p}})x + \frac{n^3\alpha}{p} & x \ge \frac{\beta\sqrt{p}}{\alpha} \end{cases} \\ & s.t. \quad x < \sqrt{\tfrac{C}{48p}} \\ & \qquad\ \ x < \tfrac{n}{4\sqrt{p}} \end{aligned} \tag{12}$$

By denoting x^* as the solution of problem $\mathcal{P}'$, we have following corollary:

Corollary 2. *Given the problem size n and SRAM size C, the optimal tile parameter x^* for problem $\mathcal{P}'$ is:*

$$x^* = \begin{cases} \lfloor \frac{n}{4\sqrt{p}} \rfloor - const & n \le \sqrt{\frac{C}{3}} \\ \lfloor \sqrt{\frac{C}{48p}} \rfloor - const & n \ge \sqrt{\frac{C}{3}} \end{cases}$$

where const is positive integer which satisfies $x^* > 0$

Proof: If we denote by x_1^* and x_2^* as the solutions of $T_1(x)$ and $T_2(x)$ respectively, we get:

$$x_1^* = \frac{n}{3\sqrt{p}} \qquad x_2^* = \frac{6n\alpha - p\beta}{16\alpha\sqrt{p}}$$

If we denote $x_{mid}^* = \frac{\beta\sqrt{p}}{\alpha}$, it partitions the solution space into the two intervals: $(0, x_{mid}^*]$ and $[x_{mid}^*, min\{\sqrt{\frac{C}{48p}}, \frac{n}{4\sqrt{p}}\})$. However, it is obviously that $x_1^* > \frac{n}{4\sqrt{p}}$ and $x_2^* > \frac{n}{4\sqrt{p}}$, that is, x_1^* and x_2^* are out of the valid solution space. In the solution space to the left of x_1^* and x_2^*, $T_1(x)$ and $T_2(x)$ are descending and they reach a minimum point at $min\{\sqrt{\frac{C}{48p}}, \frac{n}{4\sqrt{p}}\} - const$.

Corollary 3. *Given the problem size n and $2 < p < \frac{\alpha}{\beta}min\{\sqrt{\frac{C}{48}}, \frac{n}{4}\}$*, the optimal solution to problem $\mathcal{P}$ is $x^* = \frac{\beta\sqrt{p}}{\alpha}$

Proof: Let $a = (8n\alpha + n\alpha p)$, $b = (n\beta\sqrt{p} - \frac{6n^2\alpha}{\sqrt{p}} + \frac{n^2\alpha(4\sqrt{p}+1)}{4})$, the global minimum of $T(x)$ is obtained at $x^* = \frac{-b}{2a}$. However, the solution space is confined within the interval $[\frac{\beta\sqrt{p}}{\alpha}, min\{\sqrt{\frac{C}{48p}}, \frac{n}{4\sqrt{p}}\})$. Assume that $x^* > \frac{\beta\sqrt{p}}{\alpha}$. we have

$$n < \frac{4\beta p(2p+17)}{24\alpha - \alpha\sqrt{p}(4\sqrt{p}+1)} \tag{13}$$

According to equation 13, $n > 0$ if and only if $p \le 2$. That is, when $p > 2$, $x^* < \frac{\beta\sqrt{p}}{\alpha}$. The property of quadratic function shows that $T(x)$ is increasing for $x > x^*$. So the solution to problem $\mathcal{P}$ is $\frac{\beta\sqrt{p}}{\alpha}$.

Combining Corollary 2 and 3, we can determine the optimal tile parameter x^* using following theorem:

Theorem 2. *The optimal tile parameter of parallel tiling pipelined algorithm is selected by the rule:*
if $2 < p < \frac{\alpha}{\beta}min\{\sqrt{\frac{C}{48}}, \frac{n}{4}\}$, $x^* = \frac{\beta\sqrt{p}}{\alpha}$;
otherwise,

$$x^* = \begin{cases} \lfloor \frac{n}{4\sqrt{p}} \rfloor - const & n \le \sqrt{\frac{C}{3}} \\ \lfloor \sqrt{\frac{C}{48p}} \rfloor - const & n \ge \sqrt{\frac{C}{3}} \end{cases}$$