Improving Performance of Dynamic Programming via Parallelism and Locality on Multicore Architectures

Guangming Tan, *Member*, *IEEE*, Ninghui Sun, *Member*, *IEEE*, and Guang R. Gao, *Fellow*, *IEEE*

Abstract—Dynamic programming (DP) is a popular technique which is used to solve combinatorial search and optimization problems. This paper focuses on one type of DP, which is called nonserial polyadic dynamic programming (NPDP). Owing to the nonuniform data dependencies of NPDP, it is difficult to exploit either parallelism or locality. Worse still, the emerging multi/many-core architectures with small on-chip memory make these issues more challenging. In this paper, we address the challenges of exploiting the fine grain parallelism and locality of NPDP on multicore architectures. We describe a latency-tolerant model and a percolation technique for programming on multicore architectures. On an algorithmic level, both parallelism and locality do benefit from a specific data dependence transformation of NPDP. Next, we propose a parallel pipelining algorithm by decomposing computation operators and percolating data through a memory hierarchy to create just-in-time locality. In order to predict the execution time, we formulate an analytical performance model of the parallel algorithm. The parallel pipelining algorithm achieves not only high scalability on the 160-core IBM Cyclops64, but portable performance as well, across the 8-core Sun Niagara and quad-cores Intel Clovertown.

Index Terms—Dynamic programming, memory hierarchy, latency tolerant, percolation, multicore.

1 Introduction

For many search and optimization problems, dynamic programming (DP) is a classical, powerful, and well-known technique that is used to find an optimal solution among many potential ones. This technique is used in many applications such as scheduling, inventory management, automatic control, and VLSI design [1]. More recently, it has been found useful toward solving many problems in bioinformatics. For example, the two most important applications are the Smith-Waterman algorithm [2], which is used for matching sequences of amino acids/nucleotides, and Zuker algorithm [3], which is used for predicting RNA secondary structures. However, a combinatorial explosion limits the chance of this method to be widely used because CPU time and storage requirements can be very high. Parallel processing can be used as an efficient tool to solve large-scale DP problems. In fact, parallelization of the DP algorithm has been a classical problem in parallel algorithm research in the last decade [4], [5], [6], [7], [8].

Due to the wide variety of problems solved using DP, it is difficult to develop generic parallel algorithms for them. In order to find efficient parallel algorithms for implementing DP, Grama et al. [1] presents a classification of DP: DP is considered as a multistage problem composed of many subproblems. If the subproblems located on all levels depend only on the results from the immediately preceding levels, it is called *serial*; otherwise, it is called *nonserial*. There is recursive equation called a *functional equation*, which represents the solution to an optimization problem. If a functional equation contains a single recursive term, the DP is *monadic*; otherwise, if it contains multiple recursive terms, it is *polyadic*. Based on this classification criteria, four classes of DP are defined: serial monadic DP (SMDP), used in single source shortest path and 0/1 knapsack problems, serial polyadic DP (SPDP), used in Floyd all pairs shortest paths problem, nonserial monadic DP (NMDP), used in longest common subsequence problem and the Smith-Waterman algorithm, and nonserial polyadic (NPDP), used in both the optimal matrix parenthesization problem and Zuker algorithm. The DP formulation is outlined in Table 1. In this work, we focus on the NPDP appearing in the Zuker algorithm, which is used for RNA secondary structure prediction. This algorithm searches an optimal structure with a minimal free energy, and the data type for each element in the matrix is float.

The DP classification shows all data dependence which occurs during the computation in the DP matrix. However, there are two distinct features which make a difference between NPDP and the other three DP problems. First, the DP matrix for NPDP is triangular, or rather, the iteration domain is a triangle, as opposed to being rectangular as in the other DP problems. The triangular iteration domain makes the optimization of memory accesses and load balancing difficult. Second, data dependence in NPDP is dynamic. The number of elements which are depended upon for computing $m[i,j]$ is $O(j-i)$. Furthermore, the data dependence appears among nonconsecutive levels. The nonconsecutive data dependence has negative effects

• *G. Tan and N. Sun are with NCIC, Institute of Computing Technology, Chinese Academy of Sciences, No. 6, Ke Xue Yuan South Road, Zhongguancun, PO Box 2704, Beijing 100190, P.R. China. E-mail: {tgm, snh}@ncic.ac.cn.*
• *G.R. Gao is with the Department of Electrical and Computer Engineering, University of Delaware, 140 Evans Hall, Newark, DE 19716. E-mail: ggao@capsl.udel.edu.*

Manuscript received 18 Sept. 2007; revised 2 Feb. 2008; accepted 5 May 2008; published online 15 May 2008.
Recommended for acceptance by F. Petrini.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2007-09-0321.
Digital Object Identifier no. 10.1109/TPDS.2008.78.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

TABLE 1
Classification and Formulation of DP

Type	DP formulation
SMDP	$m[i,j] = min_{1 \le j \le i}\{m[i-1,j], m[i-1,j-c]\}$
SPDP	$m[i,j] = min\{m[i,j-1], m[i-1,j]\}$
NMDP	$m[i,j] = min_{1 \le k \le n}\{m[i,j], m[i,k] + m[k,j]\}$
NPDP	$m[i,j] = min_{i \le k < j}\{m[i,j], m[i,k] + m[k+1,j]\}$

TABLE 2
Architectural Features of Current Multicore Architectures

Type	#cores	on-chip memory	mem/core
Sun Niagara-1	8 (32 threads)	3MB	400KB(100KB)
Intel TeraScale	80	160KB	2KB
IBM CELL	8	2MB	256KB
IBM Cyclops64	160	2.5MB	16KB

on blocking the iteration domain because it violates the regularity of the dependence between two blocks. We refer to this violation as *cross block reference* (see Section 3).

Because of the challenges of the NPDP algorithm itself, it is not a surprise that there have been a lot of research on its optimization and parallelization for different architectures. On the other hand, with the rapid advance of multicore/many-core chip technology, we have recently witnessed many proposals from both the industry and academia which actively exploit the design space of multicore chip design. However, there are common features which exist in multicore architectures and they are as follows: cache size or local memory per core (we refer to both terms as on-chip memory in the following text) are very small and there is an increasing number of cores per chip. Table 2 summarizes the local memory sizes on several multicore chips commercially available and under development. Many cores on-chip provide a chance to exploit more parallelism in algorithms, but the small on-chip memory forces us to exploit more fine-grain parallelism on chip level and maximize data reuse in the on-chip memory. Therefore, unlike the coarse parallel algorithm design on conventional parallel computers, we have to face the challenge of parallelism and locality on-chip level for developing high-performance algorithms on multicore architectures. Multicore architectures with an explicit memory hierarchy are the most critical when it comes to rethinking algorithmic techniques which are used to address parallelism and locality issues in the emerging multicore/many-core architecture world.

In this paper, we address the challenges of parallelism and locality for NPDP algorithm on multicore architectures. We propose a new programming model—latency-tolerant model—to create just-in-time locality using a percolation programming technique [9]. The efficient fine-grain parallel algorithm with temporal locality is leveraged by a novel transformation of data dependencies in NPDP. The rest of this paper is organized as follows: In Section 2, we describe the latency-tolerant model and explain how to create just-in-time locality using percolation programming. Section 3 gives an analysis of the effect of data dependence on blocking the DP matrix, then proposes a novel transformation of data dependence, which is the cornerstone of the fine-grain parallel algorithm. In Section 4, we present the fine-grain parallel NPDP algorithm, which focuses on exploiting locality on multicore architectures with small on-chip memory. Section 5 formulates a performance modeling of the parallel algorithm in the latency-tolerant model and analytically solves the problem which minimizes the total execution time. In Section 6, we show a detailed performance evaluation of our proposed parallel algorithm on the IBM Cyclops64 and the Sun Niagara. Section 7 summarizes the previous work on parallelizing NPDP. Section 8 concludes this paper.

2 Latency-Tolerant Model

The use of a memory hierarchy is a general technique used to deal with the gap between faster processing elements (PEs) and slow memory access. Currently, several important models, such as I/O [10], cache-ware [11], and cache-oblivious [12] model have been proposed to algorithmic design and analysis. Contemporarily, parallel programming models like BSP [13], LogP [14], and CGP [15] have been widely used in developing parallel algorithms. In these models, locality and parallelism for an algorithm are exploited in a separate way. As mentioned in Section 1, many-core chip design is increasing the number of cores which share a limited on-chip memory. Algorithms designed for multicore systems should focus more on locality while the parallelism is exploited in on-chip level. For example, when developing parallel algorithms on the STI CELL, we have to restructure these algorithms, which run well on conventional architecture, in order to improve data reuse in local storage of each SPU.

One important observation for current multicore architecture is that *memory access operations can proceed asynchronously through multithreading or DMA*. The latency-tolerant model is inspired by the architectural support for separating computation from memory operations through memory hierarchy in user level. In the abstract latency-tolerant model, some features are highlighted as follows:

- **Two-level memory model:** according to latency, the memory space is partitioned into *in-core memory (ICM)* with low latency and *out-of-core memory (OCM)* with high latency.
- **Stronger execution model:** A computation PE is active only when the depended data is in *ICM*, and only if the computation PE needs the data, they will be in *ICM*.
- **Fine-grain parallelism:** The accesses to *ICM* and *OCM* proceed in parallel.

In a real multicore architecture, on-chip memory (local storage of each SPU in the STI CELL) is mapped to *ICM*, and off-chip memory (system memory of PPU) is mapped to *OCM*. For the convenience of description, we define three specific operations according to the scope and direction of memory operations: LOADLT/STORELT for transferring data from *OCM* to *ICM* and EXECLT for operating data only in *ICM*.

The stronger execution model leads to *just-in-time locality*—data is local to a computation PE just before the computation PE are scheduled to operate on the data. We

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

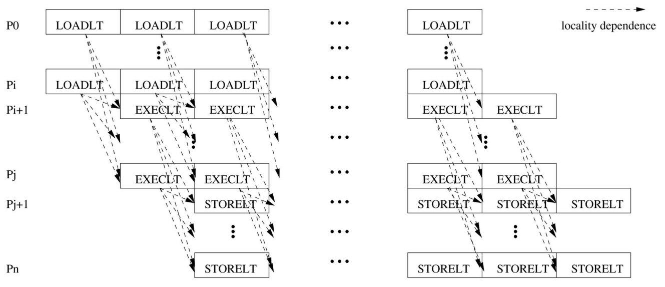

Fig. 1. Pipelining of LOADLT-EXECLT-STORELT. $P_{0,\ldots,i}$ and $P_{j+1,\ldots,n}$ are percolation PEs. The locality dependence implies that the control and data dependence are already satisfied in the latency-tolerant model.

```
BASEPERCOLATION
1. while the set of not enabled memory tasks is not empty;
   /*LOADLT: inward percolation*/
2. A memory tasks becomes enabled;
3. Data is transformed in off-chip memory;
4. The transformed data are transferred to on-chip memory;
   /*EXECLT*/
5. The computation tasks meeting locality requirement is enabled;
   /*STORELT: outward percolation*/
6. Results are tranferred to off-chip memory;
7. Transformations are performed in the outward
   percolated data in off-chip memory;
```

Fig. 2. Outline of steps in the percolation process.

refine a percolation technique [9] to achieve just-in-time locality. In a conventional (*weaker*) execution model, as soon as the data and control dependencies are satisfied, a computation proceeds without being aware of where the data lies, i.e., *ICM* or *OCM*. In latency-tolerant model, the conditions of enabling a computation are not only data and control dependent but also locality dependent. In other words, a computation PE is active only when the data, which it needs to carry out those computations, are in *ICM*. This is different from a conventional cache locality mechanism. A cache line may contain unused data that incur a number of cache misses, which degrades performance. A feasible way to satisfy locality dependence is to decouple computation operations with memory operations so that the memory operations are executed by the separated PEs, which are called *percolation* PEs. Initially, the tasks of percolation may involve either collecting the data and moving them toward computation PEs, or sending/migrating the data away from computation PEs. Note that the data in off-chip memory may not be contiguous. For example, if the DP matrix in this study are not stored in a blocked layout, the accesses to each subblocks are not contiguous. Although various blocking layout techniques [16] are effective to exploit locality with a little preprocessing overhead, we claim that there is an alternative way to replace or implement the traditional data layout methods without initial preprocessing by utilizing the massive cores on multicore architectures. During data movement though memory hierarchy, percolation also transforms off-chip noncontiguous memory access to on-chip contiguous memory access. The transformation is finished at little cost because it is only involved with indexing computation. However, note that we exploit parallelism between computation and percolation PEs so that the overhead of "creating" just-in-time locality is hidden. For example, when computation PEs are processing the data at block i, some percolation PEs gather/transform the data in block $i+1$ and other percolation PEs scatter/transform the results that are generated using the data in block $i-1$. The basic steps of the percolation process are illustrated in Fig. 2. These steps are executed in such a pipelining way that computation tasks and percolation tasks are overlapped to hide the latency caused by the memory hierarchy (see Fig. 1). The percolation consists of three concurrent processes: inward percolation (transforming and transferring data from *OCM* to *ICM*), execution in on-chip memory, and outward percolation (transforming and transferring data from *ICM* to *OCM*). When the computation task is processing data blocks, the memory tasks respond to requests for more inflow and outflow of data across the memory hierarchy. Notice that sometimes it is not necessary to percolate results outward to the off-chip memory after every execution in the on-chip memory. Also, Fig. 2 lists transformation and inward/outward percolation in two different steps for ease of understanding. In our implementation, the data transformation performs the calculation of affine function between the two memory levels, and during inward and outward percolation, the data is then transformed according to the affine function. Finally, once a computation task is enabled, it will execute on a core until it finishes, and it will never be preempted. Therefore, it is up to the programmer to exploit data reuse, which exists in the percolated data that is available in the on-chip memory. We will show an example of such data reuse in Section 4.

In a summarization, latency-tolerant model requires just-in-time locality for high performance, and percolation is a critical technique to achieve just-in-time locality. Two key steps involved in developing an algorithm used for a latency-tolerant model are 1) separating computation from memory operations for just-in-time locality and 2) orchestrating the sequence of computation and memory operations for pipelining. More percolation operations require more reserved PEs as well as a higher bandwidth requirement. It is important to increase data reuse for the just-in-time locality by percolation.

3 Transformation of Data Dependence

Blocking or tiling is a well-known technique for improving locality. However, a straightforward blocking of the DP matrix has negative effects on the parallelism and memory traffic. In Fig. 3a, the matrix is partitioned into several triangular and rectangular blocks. According to data dependencies for NPDP, the calculation of a block depends on blocks on the same row and column. For example, $B_{0,3}$ depends on $B_{0,0}$, $B_{0,1}$, $B_{0,2}$, $B_{1,3}$, $B_{2,3}$, $B_{3,3}$. Because *ICM* is too small to contain all the of blocks which are depended on, we observe a critical performance problem. Assume block $B_{0,1}$, $B_{1,3}$, and $B_{0,3}$ are loaded into *ICM* and used to calculate a partial $B_{0,3}$. Recall the data dependence in NPDP, one element $b_{i,j}$ in $B_{0,3}$ is calculated by operations between ones in the ith row in $B_{0,1}$ and ones in the jth column in $B_{1,3}$. Note that elements which combine with the rightmost elements in $B_{0,1}$ fall into block $B_{2,3}$. However, $B_{2,3}$ is not loaded into *ICM*

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

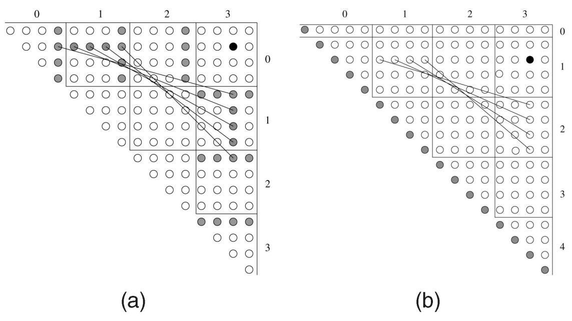
Fig. 3. Blocking with/without *cross block reference*.

so that the arithmetic operations on the rightmost elements in $B_{0,1}$ are stalled, therefore the available parallelism is cut down. The same case arises in the topmost elements in $B_{1,3}$. We refer to this mismatch as *cross block reference* (see the red points in Fig. 3a).

The cross block reference results in low data reuse, low parallelism, and waste of memory bandwidth, which degrades performance of a program in multicore architectures. The elements involved in cross block reference have to be loaded into *ICM* from *OCM* more than twice, the waste of memory bandwidth places a severe burden on the low bandwidth of *OCM*. We propose a data dependence transformation to eliminate the cross block reference. The DP matrix is easily expressed as an iteration domain. 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, the NPDP formulation is rewritten as

$$m[i', j'] = min_{i'+1 \le k' < j'}\{m[i', j'], m[i', k'] + m[k', j']\}. \quad (1)$$

In the new domain, elements on the diagonal do not contribute to computation. We claim that except for the unused values on the new diagonal in the new domain, both formulation (1) and the original one produces the same DP matrix.

Proposition 1. *$\forall (i, j) \in \mathcal{D}$ and $\forall (i', j') = (i, j+1) \in \mathcal{D}'$, after the original NPDP formulation and formulation (1) are used to compute domain $\mathcal{D}$ and $\mathcal{D}'$, respectively, $m[i, j] = m'[i', j']$ or $m[i, j] = m'[i, j+1]$.*

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 Fig. 3b). The transformation eliminates *cross block reference*. The proposed parallel algorithm in the next section calculates the transformed domain $\mathcal{D}'$.

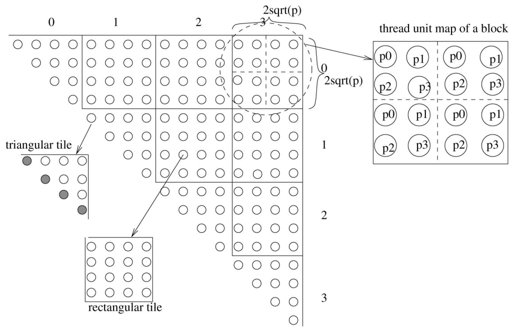

Fig. 4. The blocking and mapping of the DP matrix. It illustrates the case $p = 4$ and the size of tiled space is 16.

4 Parallel Nonserial Polyadic DP Algorithm

Let us assume that there are p threads and the size of transformed domain is n. Based on the transformed DP formulation, the cross block reference does not happen in the blocked DP matrix. An important observation is that the blocked algorithm is composed of many basic matrix block operations. Let matrices $A = (a_{ij})_{s \times s}, B = (b_{ij})_{s \times s}, C = (c_{ij})_{s \times s}$, we define two tensor operations $\otimes$ and $\oplus$ for the block matrix operations.

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 $C = A \oplus B$.*

We partition the transformed DP matrix into a 2D mesh as shown in Fig. 4. For any block $B(i, j)$, it depends on the blocks $B(i, i \dots j)$ on the same row and $B(i \dots j, j)$ on the same column. Combining with two tensor operations, the calculation of any block $B(i, j)$ is expressed as

$$\begin{aligned} B(i,j) &= \oplus_{k=i}^{j} (B(i,k) \otimes B(k,j)) \\ &= \left(\oplus_{k=i+1}^{j-1} (B(i,k) \otimes B(k,j))\right) \\ &\quad \oplus (B(i,i) \otimes B(i,j)) \oplus (B(i,j) \otimes B(j,j)). \end{aligned} \quad (2)$$

With this partition, the computation of a block $B(i, j)$ $(i \ne j)$ is divided into two parts. The first one depends on rectangular blocks on the same row/column:

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

The second one depends on a triangular block and itself:

$$(B(i,i) \otimes B(i,j)) \oplus (B(i,j) \otimes B(j,j)).$$

Let us take computation of $B(0, 3)$ for example, the first part is $(B(0,1) \otimes B(1,3)) \oplus (B(0,2) \otimes B(2,3))$, the second part is $(B(0,0) \otimes B(0,3)) \oplus (B(0,3) \otimes B(3,3))$.

4.1 Parallelism

During the computation of the second part, the submatrices $B(i, i)$ and $B(j, j)$ are triangular. The two operations of $B(i,i) \otimes B(i,j)$, $B(i,j) \otimes B(j,j)$ depend on the final results of $B(i, j)$. If $B(i, i)$, $B(j, j)$, $B(i, j)$ are integrated into one

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

submatrix, the parallelism can be exploited along the diagonal. But for any block $B(i,j)$, the computation of the second part is finished in a constant time because it only depends on two triangular blocks.

The parallelism in the blocking algorithm is observed at two levels for the first part. First, each $B(i,k) \otimes B(k,j)$ operation computes a partial result of $B(i,j)$, then all partial results are accumulated through multiple $\oplus$ operations. During the calculation of $B(i,j)$, the $O(j-i-1)$ partial results are obtained in parallel, then all partial results are accumulated using a reduction tree, also in parallel. This level of parallelism needs a lot of memory space because it needs to use all blocks in a row and column. The high memory usage easily exceeds the small size of on-chip memory (*ICM* in the latency-tolerant model). We refer to this type of parallelism as coarse-grain parallelism. The small on-chip memory obviously cuts down the feasibility of exploiting coarse-grain parallelism. Second, within any $B(i,k) \otimes B(k,j)$ operation, each $c_{ij} = min_{k=1}^{n}\{c_{i,j}, a_{i,k} + b_{k,j}\}$ (see Definition 1) can be executed in parallel. This level of parallelism needs only the current computed block and two other blocks, whose data is depended on. The block size may be adaptive to the size of on-chip memory. According to the data dependence, the number of operations increases as the computation moves to top and to the right, in a block. The load balance problem is overcome through an elaborate mapping between tasks and threads. Without loss of generality, the DP matrix is divided into a 2D triangular mesh with each block size of $2\sqrt{p}$. Each block is further divided into four subblocks with size of $p = \sqrt{p} \times \sqrt{p}$. Each element in a subblock is mapped to a thread unit. Between two subblocks, the mapping produces an alternate cycling, which achieves a load balancing among all thread units (see Fig. 4).

A special case is the triangular block along the diagonal. The computation is self-contained and also easily parallelized in a diagonal-wise way. However, we note that the both the second part and special case only occupy a small portion of the entire execution time. Thus, the optimization of the parallel algorithm focuses on the first part, where all blocks are rectangles.

4.2 Locality

The fine-grain parallel algorithm divides a block into four subblocks. In a simple parallel algorithm, all threads compute their own elements in parallel using (1), which needs three loads and one store. Thus, for one $\otimes$ operation, it needs $(2\sqrt{p})^3 \times (3+1) = 32p\sqrt{p}$ operations between two-level memory. In order to exploit locality, which will reduce memory traffic, we partition each block with a finer granularity. For any three blocks A, B, C, where C depends on A and B, they are further partitioned into four subblocks, respectively. Fig. 5 shows an example. Based on the fine-grain blocking strategy, we are able to determine the pattern of data reuse during the computation of a block. Each subblock of C depends on both two subblocks in a row of A and two subblocks in a column of B. We arrange a sequence of the computation C as $c(0,0)$, $c(0,1)$, $c(1,1)$, $c(1,0)$. By doing this, not only the two $\otimes$ between two subblocks of A and B reuse the subblock of C but also the subblocks of A and B also are reused when computing the next subblocks of C. For example, in Fig. 5, $a(0,0)$, $a(0,1)$ are reused when $c(0,0)$ and $c(0,1)$ are computed. Our parallel algorithm achieves data reuse by scheduling operations. The pipeline algorithm with temporal locality needs four loads/stores from/to C, four loads from A, and six loads from B, therefore the number of memory access is only $18p$. For each block, the number of $\otimes$ operations required is $O(j-i-1)$.

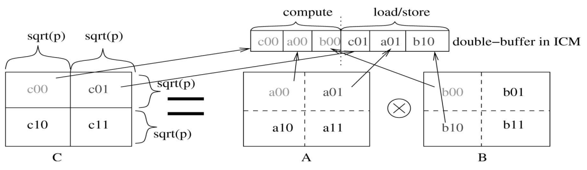

Fig. 5. The computation of one block which is divided into four subblocks. The subblocks from the operand matrices also are divided into four subblocks. The subblocks shown in the images in *ICM* are of subblocks when computing subblock $C(0,0)$ and loading subblocks $A(0,1)$, $B(1,0)$ in *step 1*.

Based on the parallel algorithm with temporal locality, we exploit additional parallelism to create just-in-time locality using percolation under the latency-tolerant model. According to percolation programming, the program needs to separate computation from memory operations. In the parallel NPDP algorithm, the memory operations include the loading of the blocks that are depended upon into *ICM* and writing back to *OCM*. After the memory operations are completed, the additional parallelism consists of overlapping them with computation. For example, when a thread unit is computing $c(0,0)$, the operations of loading $C00$, $A00$, $B00$ for the next $\otimes$ operation can be performed at the same time. A parallel pipelining algorithm is outlined in Algorithm 1 implementing the percolation in eight steps. The implementation groups the threads into *computation* and *percolation* threads. The *percolation* threads are used to load/store between *ICM* and *OCM*, while *computation* threads only access on-chip memory to perform calculations. Because the algorithm needs to access subblocks which elements are not stored contiguously, it may not achieve spatial locality. In order to achieve spatial locality in on-chip memory, the percolation threads gather the noncontiguous data in *OCM* into a contiguous space in *ICM*. In order to hide the overhead of percolation, we adopt double-buffering approach, where one buffer is used for current computation, and the other is used for the next computation. The percolation threads load/store data which is used to compute the next C subblock while the computation threads compute the current C subblock. The spatial locality is created in the buffer for next computation when percolation threads load/store data between two levels of memory. Therefore, since the access to *OCM* is overlapped with computation, the memory access latency is hidden.

Algorithm 1: The eight pipelined parallel steps.

```
ParallelSteps
startup: LOADLT $C00$, $A00$, $B00$;
step 1: EXECLT $C00$; LOADLT $A01$, $B10$;
step 2: EXECLT $C00$; LOADLT $C01$, $B01$;
step 3: EXECLT $C01$; LOADLT $B11$; STORELT $C00$;
step 4: EXECLT $C01$; LOADLT $C11$, $A10$;
step 5: EXECLT $C11$; LOADLT $A11$; STORELT $C01$;
```

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

step 6: EXECLT $C11$; LOADLT $C10$, $B00$;
step 7: EXECLT $C10$; LOADLT $B10$; STORELT $C11$;
step 8: EXECLT $C10$;
end: STORELT $C10$;

Note that the unit of a percolation is a block. Tiling iteration domain [17], [18], [19], [20] 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 the parallel pipelining algorithm based on the latency-tolerant model, the "*communication*" is the data transfer between two levels of memory, while the locality in *ICM* should also be accounted for. In this case, tiling is used to minimize the total execution time of parallel programs with a latency-tolerant model, which have been written for multicore 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 Fig. 4). In this current work, we only consider a square tile with $x = y$ (in the rest of this paper, we use tile parameters referring to tile width/height). In order to satisfy dependencies, 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 inherited, the tiled DP matrix is filled using the proposed parallel pipelining algorithm.

5 Performance Modeling

The study of performance modeling is confined to the parallel pipelining algorithm with tiling. The basic operation $\otimes$ 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 (1), for any block $B(i,j)$ $(i \leq j < m' - i)$ in row strip i, it needs $j - i - 1$ blocked $\otimes$ operations. Let $I_\otimes$ denote 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 number of blocked $\otimes$ operations:

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

5.1 Memory-Traffic Complexity

In the latency-tolerant model, *memory-traffic complexity* is used to measure the performance of an algorithm. This is defined as the amount of memory traffic between *ICM* that is smaller than problem size and *OCM* 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* OCM.

Proof. See Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2008.78. □

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 [21] 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 for a general memory hierarchy. However, we noted that helper threads can be used tolerate memory access latency on multicore architectures. In other words, besides overlapping memory access with computation, memory accesses can be parallelized within the memory bandwidth limitation using helper threads. In the performance model, assume that there is no bandwidth limitation and that the memory access latency for loads and stores 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 loads, and the other is used for stores, then the four store operations are completely overlapped and the time reduction percent for a $\otimes$ operation is 4/18; therefore, the *memory-traffic efficiency* is 11 percent for two helper threads. However, as shown in Algorithm 1, 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 percent. If the number of 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 percent. In fact, the *memory-traffic efficiency* is determined by the parallelism in memory access. In all practical architectures, there exists a memory bandwidth limitation, so more helper threads do not mean higher efficiency.

5.2 Execution Time

Under the execution model, we now determine an analytic formulation for 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 (1) 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

$$T_{tran} = \beta p x^2.$$

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

Because the helper and computation threads proceed in parallel, the time for transferring data and executing the $\otimes$ 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

$$T_{\otimes} = max\{T_{comp}, T_{tran}\} = max\{\alpha\sqrt{p}x^3, \beta p x^2\}. \tag{4}$$

Combining (3) with (4), we get the execution time of all parallel pipelined steps:

$$\begin{aligned} T_0(x) &= \frac{n}{2x\sqrt{p}} \times 2\beta p x^2 + I_{\otimes} \times 8 \times T_{\otimes} \\ &= n\beta\sqrt{p}x + 8 \times I_{\otimes} \times max\{\alpha\sqrt{p}x^3, \beta p x^2\} \\ &= \begin{cases} T_1(x) = n\beta\sqrt{p}x + 8I_{\otimes} \times \beta p x^2 & x < \frac{\beta\sqrt{p}}{\alpha} \\ T_2(x) = n\beta\sqrt{p}x + 8I_{\otimes} \times \alpha\sqrt{p}x^3 & x \geq \frac{\beta\sqrt{p}}{\alpha}. \end{cases} \end{aligned} \tag{5}$$

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

$$\begin{aligned} T_3(x) &= \left(\sum_{i=1}^{m'} i \times \sum_{j=1}^{4\sqrt{p}} + m' \sum_{j=1}^{2\sqrt{p}} \right) x^3 \alpha \\ &= n\alpha p x^2 + \frac{n^2 \alpha (4\sqrt{p} + 1)}{4} x. \end{aligned} \tag{6}$$

So the optimal x is selected to minimize the following formulation:

$$\begin{aligned} \mathcal{P} : Minimize \quad & T(x) = T_0(x) + T_3(x) \\ \text{s.t.} \quad & \frac{\beta\sqrt{p}}{\alpha} \leq x < min\left\{ \sqrt{\frac{C}{48p}}, \frac{n}{4\sqrt{p}} \right\}. \end{aligned} \tag{7}$$

Therefore, the objective is to select an 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

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

Proof. See Appendix B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2008.78. □

We have obtained observations which help in solving this nonlinear optimization problem. According to the solution to $T_0(x)$, x is expected to be larger; however, the portion of computing triangular blocks is greater due to the 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 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 falls 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 1 shows that the optimal solution located at the right of x^*_{mid}, which means the scalability is determined by the arithmetic operation instead of memory latency. So our proposed parallel algorithm on a multicore architecture has fine scalability with large-scale processors. The solution for the global minimum in the case of $n = 1{,}024$ and $p = 16$ is shown in Fig. 6.

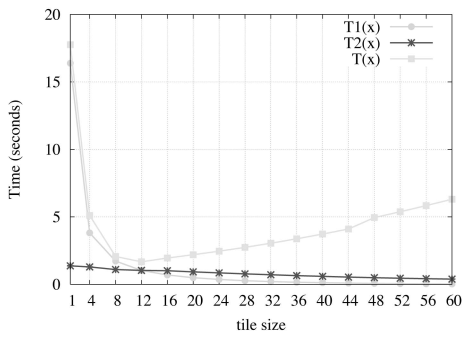

Fig. 6. Finding the global minimum of the tile parameter x according to Theorem 2, $p = 16$, $n = 1{,}024$, $x^*_{mid} = 12$ $min\{\sqrt{\frac{C}{48p}}, \frac{n}{4\sqrt{p}}\} = 64$, $x^* = 12$.

6 Numerical Experiments

We evaluate the proposed parallel NPDP algorithm based on a latency-tolerant model on two multithreading multicore platforms—IBM Cyclops64 and Sun Niagara-1. We also report results from running experiments on Intel Clovertown. These are representative multicore designs of software-managed and hardware-managed memory hierarchies, respectively. The experiments on IBM Cyclops64 show that the parallel pipelined NPDP algorithm, through percolation, achieves fine scalability for explicit memory hierarchy architectures. Although the percolation technique on a latency-tolerant model is inspired by IBM Cyclops64-like architecture with an explicit memory hierarchy, it also demonstrates the potentials of improving locality through parallelism on cache-based multicore architecture like Sun Niagara-1. The experiments on Sun Niagara-1 shows that the optimization technique greatly reduces the number of the shared cache misses (L2 cache) for computation tasks.

6.1 IBM Cyclops64

The main target of many-core architecture in this work is the IBM Cyclops64 (C64) cellular architecture., which is designed to serve as a dedicated Petaflops supercomputer. The C64 chip architecture represents a major departure from mainstream microprocessor design. The C64 chip integrates multiple (160) PEs, embedded memory and communication hardware in the same piece of silicon. A thread unit (TU), the C64 computational cell, is a simple 64-bit, single issue, in-order RISC processor with a small instruction set architecture (60 instruction groups) operating at a moderate clock rate. The C64 incorporates efficient support for thread-level execution. For instance, a thread can stop executing instructions for a number of cycles or indefinitely; however, it can be awoken by another thread through a hardware interrupt. All the thread units within a chip connect to a 16-bit signal bus, which provides a means to efficiently implement barriers. The C64 features a three-level (Scratchpad (SP) memory, on-chip SRAM, off-chip

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

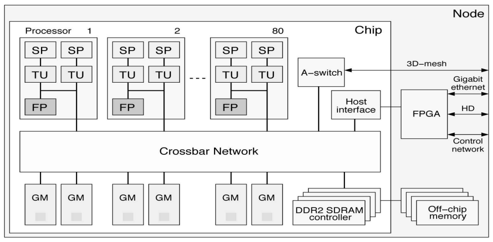

Fig. 7. IBM Cyclops64 chip architecture.

DRAM) memory hierarchy without data cache. Instead, a portion of each thread unit's corresponding on-chip SRAM bank is configured as the scratchpad memory (SP). Therefore, the thread unit can access to its own SP with very low latency, which provides a fast temporary storage to exploit locality under software control. The remaining sections of all on-chip SRAM banks together form the global memory (GM) that is uniformly addressable from all thread units. There are four off-chip memory controllers connected to four off-chip DRAM banks. The current design size for DRAM is 1 Gbyte. The thread execution is nonpreemptive. It means one single application can run at a given time and the microkernel will not interrupt the user application unless an exception occurs. Fig. 7 shows the C64 chip architecture and its memory hierarchy.

The 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) [22], [23], is designed for the purpose of architecture design verification and software development. In this experiment, the simulator is configured with parameters as shown in Table 3. Based on the FAST simulator, a thread virtual machine (TNT) [24] is implemented to support a multithread programming environment. The TNT library provides user and library developers with an efficient Pthread-like API for thread-level parallel programming purpose. The parallel algorithms are implemented using the TNT library.

6.2 Sun Niagara-1

Sun Niagara-1 (T1) is a multicore architecture integrating SMT and CMP technique. As shown in Fig. 8, there are eight cores, each of which runs four thread streams. A Niagara processor runs up to eight cores simultaneously for a total of 32 concurrent threads or strands. The OS provides a view of 32 cores to programmers. A strand that stalls for any reason is switched off and its slot on the pipeline is given to the next strand automatically. The stalled strand is inserted back in the queue when the stall is complete. All cores are connected by a high-speed, low-latency crossbar in silicon. Each core is a single issue, in-order execution unit. However, a single floating-point unit (FPU) is shared by all cores. Each core has a private L1 data/instruction cache. There is a 12-way associative unified 3-Mbyte L2 on-chip cache. All cores share the entire L2 cache. The memory model supports uniform memory access (UMA) across all cores. Table 4 summarizes the main architectural parameters of Sun T1. We implement the parallel algorithm using POSIX Pthread library.

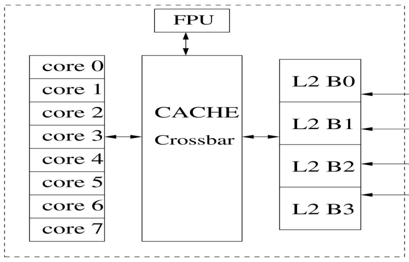

Fig. 8. Sun Niagara-1 chip architecture.

TABLE 4
Sun T1 Architectural Parameters

Component	# of units	Params./unit
Threads	8 (x4)	single in-order issue, 1000MHz
FPUs	1	floating point/MAC, divide/square root
I-cache	8	16KB
D-cache	8	8KB
L2 cache	1	3MB
DRAM (off-chip)	4	2GB
Crossbar	1	90GB/s

6.3 Performance on IBM Cyclops64

According to the analysis of performance model in Section 5, a minimal execution time is obtained when we use the optimal tile size specified by Theorem 2. First, we validated the performance model by comparing the theoretical to experimental execution time, which was measured on the FAST simulator. Fig. 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 optimal tile parameter x^* ($p = 16, n = 1{,}024, x^*_{mid} = 12, x^* = 12$). Because the model does not take the synchronization into account, the theoretical execution time is less than the experimental execution time. Notice that this difference increases beyond tile size 8. The performance

TABLE 3
FAST Simulation Parameters

Component	# of units	Params./unit
Threads	64	single in-order issue, 500MHz
FPUs	32	floating point/MAC, divide/square root
I-cache	16	32KB
SRAM (on-chip)	160	32KB(load 20cycls,store 10cycles)
DRAM (off-chip)	4	256MB(load 36cycls,store 18cycles)
Crossbar	1	384GB/s

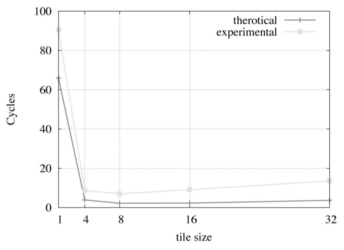

Fig. 9. Model validation.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

TABLE 5
The Execution Time (t) in Seconds and Parallel Efficiency (e) of Different Problem Size (n) on IBM C64

#cores	1	4		16		64	
n	t	t	e	t	e	t	e
256	1.40	0.46	97%	0.16	54%	0.11	18%
512	11.28	2.46	114%	1.01	70%	0.62	28%
1024	90.43	17.94	126%	6.99	81%	4.57	31%
2048	226.54	42.72	126%	15.75	91%	7.57	48%
4096	1738.88	275.82	195%	106.28	100%	44.06	60%

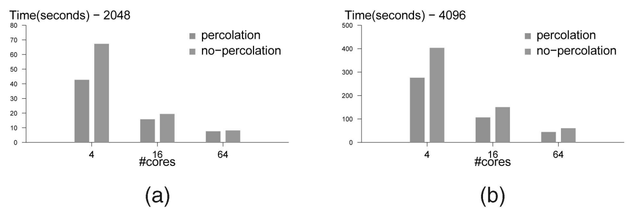

Fig. 11. The performance of percolation programming. (a) Problem size $= 2{,}048$. (b) Problem size $= 4{,}096$.

model does not evaluate bandwidth of off-chip memory, but latency. The increasing tile sizes place higher requirement on bandwidth.

The runtime system on IBM C64 provides two kinds of memory allocation functions: SRAM_MALLOC/ DRAM_MALLOC. SRAM_MALLOC allocates a memory region in on-chip SRAM, and DRAM_MALLOC does in off-chip DRAM. The initialized DP matrix is in off-chip DRAM allocated by DRAM_MALLOC, the double-buffer is in on-chip SRAM allocated by SRAM_MALLOC. Table 5 summarizes the running time of the original serial and parallel pipelining algorithm. Fig. 10 depicts the achieved speedups with different problem sizes. The parallel pipelining algorithm achieves superlinear speedups for less than (equal to) 16 threads, but scale to 64 threads sublinearly. Table 5 also presents the parallel efficiency $(\frac{speedup}{\#cores})$ of the parallel algorithm. The scalability is measured as *strong* and *weak* scalability. A strong scaling is achieved when the number of processors increased while the overall problem size is kept constant. The parallel efficiency decreases with the number of threads (cores) in the strong scaling measurement. The efficiency, however, is improved when the problem sizes become larger. In other words, the parallel pipelining algorithm achieved high weak scalability, and speed is achieved when the number of processors are increased while the problem size per processor is kept constant.

6.3.1 Locality

Percolation programming on a latency-tolerant model is a key technique which is used to improve locality for the parallel pipelining NPDP algorithm. The program is composed of explicit fragmentation of computation and memory operations, which are executed on the specified computation and memory threads. While the memory threads transform noncontiguous off-chip memory access to linear on-chip memory access, the pipeline scheduling algorithm hides the off-chip memory operations by exploiting the parallelism between computation and memory threads. Fig. 11 proves that percolation programming, which separates computation from memory operation, reduces the overall execution time. This is most evident in the case where the number of threads is less than 16, in which case the parallel algorithm get linear speedups.

The percolation creates just-in-time locality in on-chip memory and hides the overhead of the "creation" operations through pipelining. In the latency-tolerant model, the complexity of memory traffic is an important measurement of a parallel algorithm, so therefore, we should increase reuse of the percolated data. The computation strategy in *ParallelSteps* improves data reuse, thus reducing the amount of off-chip memory access. Fig. 12 plots the distribution of computation and off-chip memory access time required for problem sizes 256, 512, and 1,024. As a result, the requirement on off-chip DRAM bandwidth is reduced through the on-chip data reuse. On the IBM Cyclops64, a more aggressive optimization trick is to used *LDM/STM*, which is composed of four load/store double word (*LDD/STD*) instructions, to aggregate multiple memory access. Hence, the DRAM requests are reduced by 75 percent times so that the utilization of DRAM bandwidth is improved.

6.3.2 Synchronization

Note that the parallel algorithm cannot achieve good strong scalability. In order to figure out factors that limit the scalability, we profile the distribution of execution time. The overall execution time is broken into four parts of *load*, *store*, *barrier*, *compute* operations. Fig. 13 presents a time comparison of the four operations. Although the computation operations scale well with the number of threads, the overhead of barrier becomes significant with the increasing number of threads. For example, for the smaller problem sizes 2,048, the percentage (70 percent) of synchronization overhead determines the total execution time.

A barrier 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

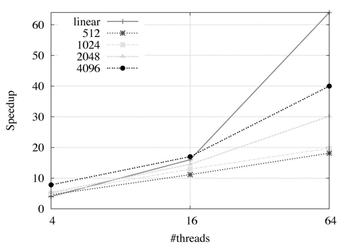

Fig. 10. Scalability on IBM C64.

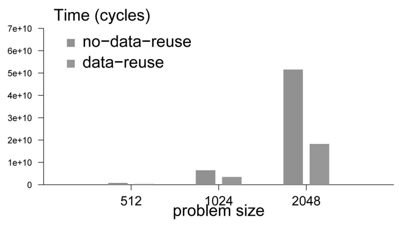

Fig. 12. Performance of temporal locality.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

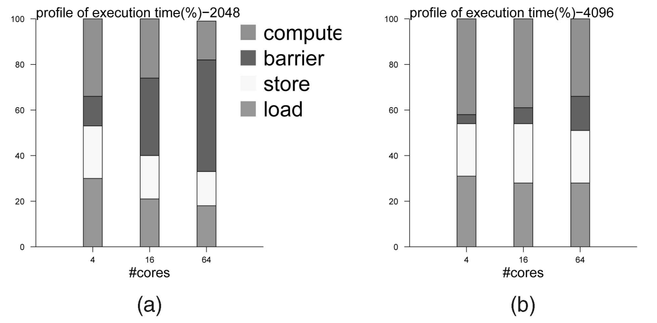

Fig. 13. The profiling of execution time on IBM C64. (a) Problem size = 2,048. (b) Problem size = 4,096.

that storing follows the corresponding computation stage. Although a barrier can be finished in as little as dozens of cycles, and 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 becomes a significant factor in decreased execution time. It is certain that the cost of one synchronization operation increases with the larger number of threads, but the percentage of the overhead of synchronization decreases with increasing of problem size. This implies that the algorithm can scale well to larger problem size. An interesting case is that the total synchronization time decreases with the larger number of threads. Although a larger number of threads results in more synchronization time itself, the optimal tiling selection reduces the number of synchronization operations because more threads need less *ParallelSteps*.

6.4 Performance on Sun Niagara

In the Sun Niagara-1 (T1) chip, the on-chip memory has both an L1 and L2 cache. Since the L1 cache is a private cache for each core, in the latency-tolerant model, we map the shared L2 cache to *ICM*, the *OCM* is system memory. Since user/programmer cannot manage data placement and movement between cache and memory explicitly, as is done in the IBM Cyclops64-like architecture, we emulate a contiguous memory region in system memory as a double-buffer in cache. With the percolation programming, the computation tasks only read/write the data confined in the emulated memory region.

Table 6 summarizes the execution time and parallel efficiency on Sun T1. The parallel algorithm does not achieve the same scalability as that on IBM C64. Both strong and weak scalability decrease with the increasing number of threads and problem size. The poor performance comes from two sides.

TABLE 6
The Execution Time (t) in Seconds and Parallel Efficiency (e) of Different Problem Size (n) on Sun T1

#cores	1	4		8		32	
n	t	t	e	t	e	t	e
1024	35.63	18.01	50%	5.66	78%	4.62	24%
2048	258.55	146.05	44%	41.74	78%	33.54	24%
4096	1715.71	1173.68	36%	325.21	65%	252.89	22%

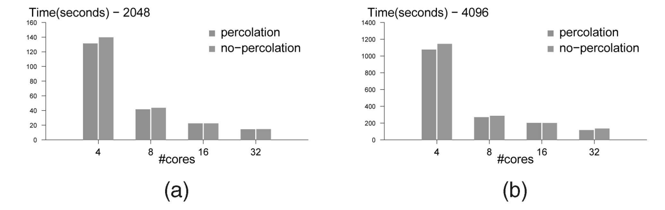

Fig. 14. The performance of percolation on Sun T1. (a) Problem size = 2,048. (b) Problem size = 4,096.

First, latency-tolerant model and percolation technique is useful for an explicit memory hierarchy architecture. With this methodology, programmers have total control of the data layout and movement through memory hierarchy. On a cache-based architecture, this mechanism only takes effect for memory operations on user virtual space. Thus, the data in the emulated memory region may not have the just in locale effect when placed in the shared L2 cache. Fig. 14 shows that the percolation technique reduces the overall execution a little. This performance result is different from that on IBM C64.

Second, a more detailed profiling of the execution time reveals that the load/store tasks occupy a small portion, even though the computation tasks do not scale well. The L27 cache size on Sun T1 is 3.2 Mbytes, we only report the problem size more than (equal to) 2,048 so that the advantage of percolation is highlighted since the memory requirements are more than 3 Mbytes. Fig. 15 presents the categories of execution time for the different problem size and number of threads. The percolation tasks simply move data from/to the emulated and other memory regions. In the Sun T1, because of low memory latency and switching of multiple strands for tolerating memory access stall, the data movement is finished quickly. Thus, the overlapping between computation and memory tasks in the parallel pipelining algorithm has shown little improvement. Note that a single FPU is shared by all cores on a Sun T1 chip and the latency of accessing to FPU is about 40 cycles, then the maximum floating-point throughput is $25MFlops = \frac{1GFlops}{40}$. Half of NPDP's instructions are float arithmetic operations, then its performance is at most $12.5MFlops = \frac{1}{2} \times 25MFlops$. The number of floating-point operations required in NPDP program is about $\Theta(n^2)$, where n is problem size, thus the larger problem size requires higher floating-point capability.

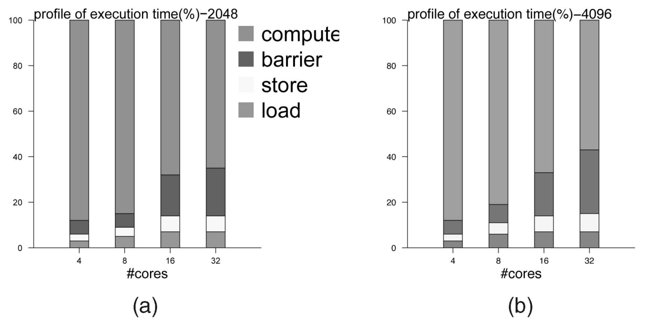

Fig. 15. The profiling of execution time on Sun T1. (a) Problem size = 2,048. (b) Problem size = 4,096.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

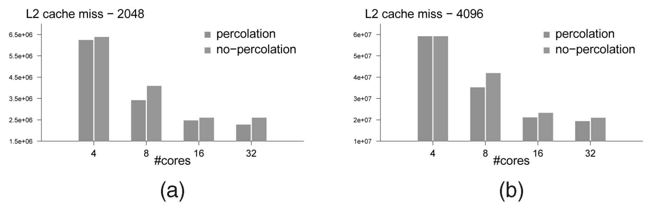

Fig. 16. The comparison of L2 cache misses on Sun T1. (a) $\text{Problem size} = 2{,}048$. (b) $\text{Problem size} = 4{,}096$.

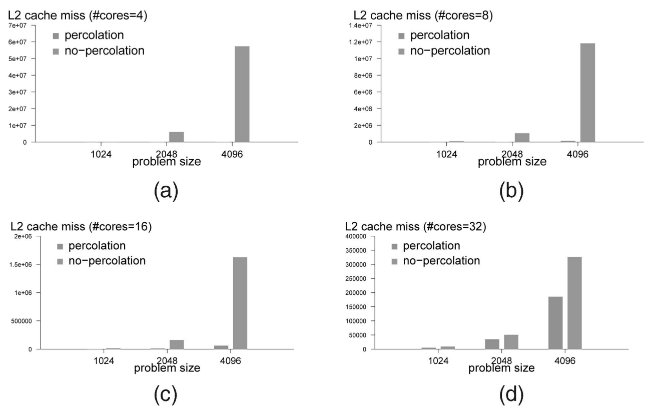

Fig. 17. The comparison of L2 cache misses for computation threads on Sun T1. (a) $\#\text{cores} = 4$. (b) $\#\text{cores} = 8$. (c) $\#\text{cores} = 16$. (d) $\#\text{cores} = 32$.

However, the parallel program cannot achieve good scalability to arrive the maximum performance. Except for the overhead of memory copy, the contention of multiple threads to access FPU is an important reason. In Fig. 14, we see that the execution time is reduced when the number of threads is 4. When the number of threads increases, the competition for a single FPU leads to poor scalability. Therefore, the scalability of the parallel program is limited by the number of FPUs.

Although the parallel pipelining algorithm does not achieve as good performance on Sun T1 as it does on IBM C64, the percolation programming based on the latency-tolerant model indeed demonstrates the potential of improving the constructive cache sharing for the shared L2 cache. We profiled the number of L2 cache misses for each thread. The multithreading programming on Sun T1 allows user to bind a thread to a core; however, our experiments evaluate the shared L2 cache misses, which are not affected by binding between threads and physical cores. Fig. 16 compares the total number of L2 cache misses for all threads. The percolation technique appreciably reduces the number of L2 cache misses. Note that the percolation programming transform the noncontiguous memory access into a linear memory access in *ICM* just before a computation task needs the data. Fig. 17 details the number of L2 cache misses for computation tasks. Since we have a load-balancing algorithm for mapping computation to threads, we observe the variance of L2 cache misses among all threads is less than 5 percent. Therefore, the values in Fig. 17 are the maximum among that of all threads. The experimental result indicates that the percolation technique also creates just-in-time locality on cache-based architecture. However, because the performance bottleneck of the multithreading program is the number of FPUs on Sun T1, the significant reduction of L2 cache misses unfortunately plays a minor role in improving the performance.

6.5 Discussion

Both IBM C64 and Sun T1 fit into the category of multithreading multicore architectures. In the emerging multicore architectures, the speed (capability) of single core is lower than that of traditional processors. The technical trend of multicore is scaling performance by scaling parallelism. However, the shared on-chip resources make the exploitation of parallelism more challenging than SMP architecture. The key idea of percolation programming on the latency-tolerant model is to create just-in-time locality using multithreads. Although the original work of this paper is inspired by IBM C64-like architectures with a software-managed memory hierarchy, the ideas can also be applied to more general multicore architectures. For instance, in the previous section, we report the detailed experimental result on Sun T1.

Table 7 reports the execution time and efficiency of our parallel program on Intel Clovertown, which is a quad-core chip. Due to the limitation space, we do not present the detailed profiling/analysis of the Intel Clovertown. Compared with the performance on the Sun T1, where the parallel efficiency decreases with the increasing problem size because of the low performance of float point operations, on the Intel Clovertown, each core has its own FPU so that it achieves a comparable scalability with that in IBM C64. It easy to see that the capability of single core on Intel quad-core is much higher than that on IBM C64 by comparing the execution time. Their performance is comparable when the number of cores on IBM C64 is as high as 64. Currently, however, it is difficult to scale hundreds to thousands of cores on one chip using the chip design technology in Intel Clovertown. In other words, our experimental results provide advantageous implications on IBM C64-like many-core architecture, which is an alternative way to scale hundreds to thousands of cores in the future.

The parallel pipelining algorithm incorporates just-in-time locality and temporal locality (data reuse) to improve performance. The temporal locality is exploited through a finer-grain parallelism within a subblock. It seems that the fine-grain blocking is specific to NPDP. Note that a lot of existing scientific computing programs adopt a blocking technique, therefore, it is very possible to refine these programs in such way. For example, the $\otimes$ operation is similar to matrix multiplication. The just-in-time locality is created by percolation programming. The necessary condition needed is the separation of computation from memory

TABLE 7
The Execution Time (t) in Seconds and Parallel Efficiency (e) of Different Problem Size (n) on Clovertown

#cores	1	4	
n	t	t	e
1024	2.93	1.06	69%
2048	35.69	7.67	116%
4096	429.43	59.51	180%

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

operations and pipeline of these two operation groups. The percolation approach obviously makes programming more complex. This work implements percolation by hardcoding it into our parallel program. It is worth developing a runtime system or high-level programming language to support percolation programming. In our vision, task level parallel programming may be an appropriate way to success because it provides a flexible mechanism to specify different tasks.

Moreover, if we regard network communication as memory access (the case in MPI-2 [25] and UPC [26]), the latency-tolerant model is extended to comprise distributed memory systems in parallel architecture with an interconnect network. For example, in the parallel pipelining algorithm shown in Algorithm 1, we implement the LOADLT and STORELT operations using remote memory *put* and *get* functions, respectively. Because of one-sided communication in MPI-2/UPC, the communications are overlapped with computation. However, sometimes the percolation is involved with gather/scatter operations as well, which cannot be overlapped with other operation within the same step.

7 Related Works

For this family of DP algorithms with nonuniform data dependence, which obviously makes the parallelization harder, there has been a lot of work on exploiting the parallelism. Bradford [27] described several algorithms which solve optimal matrix chain multiplication parenthesizations using the CREW PRAM model. Edmonds et al. [28] and Galil and Park [29] presented several parallel algorithms on general shared memory multiprocessor systems. Another important research area is in the systolic framework. For example, Guibas et al. [30] and Louka and Tchuente [31] focus on designing triangular systolic arrays. These works all focus on how to reduce the complexity of the arithmetic cost on different theoretical parallel models.

On distributed memory multicomputer systems, the main difficulty for obtaining an efficient parallel implementation is finding a good balance between communication and computation cost. In [32], [33], and [34], the authors presented a parallel implementations of RNA secondary structure prediction DP algorithm. The computational load balance is satisfactory, however, the algorithm does not optimize the communication cost. Although a simple blocking method was used, they did not take into 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, Almeida et al. [35] proposed a parallel implementation with tiling on a ring of processors. They show the usefulness of the tiling technique for this nonuniform dependence DP. However, like the algorithms in [34], this parallel tiling algorithm cannot 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 limit communication down to two neighbor tiles only, they have to keep the entire iteration in each processor. Tan et al. [36] proposed a fine-grain parallel algorithm which overlaps computation with communication on a cluster system connected by a high-performance network with an RDMA mechanism. Zhou and Lowenthal [37] presented a parallel out-of-core [38] algorithm for this DP problem under the conventional out-of-core model. Their research involves finding a replacement strategy for in-core buffer. They used a load-balancing algorithm which is similar to the method in [39]; however, this method only ensures that the number of entries on each processor is the same—the arithmetic cost on each processor is not the same because of the nonuniform data dependence.

As regards data dependence, SUIF [40] compiler framework includes a large set of libraries which are used for performing data dependency analysis and loop transformations such as tiling. However, we should note that the dependency involved in SUIF is different from that of NPDP and, furthermore, SUIF will not perform transformations without algorithmic intervention because this nonuniform data dependency from one iteration to the next makes automatic optimization through compiler significantly more difficult.

8 Conclusion

This paper has introduced a latency-tolerant model for designing parallel algorithms on the emerging multicore/many-core architectures. The key idea is to create locality by percolation and exploit additional parallelism which is used to tolerate the overhead of percolation. An important step involved is splitting the algorithm into multiple explicit phases of computation and memory operations and then schedule them to achieve an overlap in the execution between them.

Based on the latency-tolerant model, we proposed a parallel pipelined NPDP algorithm, which exploits fine-grain parallelism and data reuse, which in turn, improves on-chip memory usage and memory bandwidth. A main obstacle in improving parallelism and locality is the nonuniform data dependence in an NPDP algorithm. The novel data dependence transformation eliminates the cross block reference and paves the way for the optimization of parallelism and locality. The parallel pipelined algorithm naturally incorporates percolation to create spatial locality and tolerate the off-chip memory access latency. Our experimental result show that the proposed algorithm achieves reasonable performance on multicore architectures.

Acknowledgments

The authors would like to thank the editor, all the reviewers, Russo Andew, and Yungang Bao for the help in improving this paper. This work is partially supported through the support from NSFC (60633040), IBM, ET International, the Department of Energy (DE-FC02-01ER25503), the US National Science Foundation (NSF) (CNS-0509332). A preliminary version appeared in SPAA '07. The IBM Cyclops64 research has been performed during the first author's academic visit to University of Delaware under Professor Guang R. Gao.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

REFERENCES

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

[2] T. Smith and M. Waterman, "Identification of Common Molecular Subsequences," *J. Molecular Biology,* vol. 147, no. 1, pp. 195-197, 1981.

[3] R.B. Lyngso and M. Zuker, "Fast Evaluation of Internal Loops in RNA Secondary Structure Prediction," *Bioinformatics,* vol. 15, no. 6, pp. 440-445, 1999.

[4] V. Viswannathan, S. Huang, and H. Liu, "Parallel Dynamic Programming," *Proc. Second IEEE Symp. Parallel and Distributed Processing (SPDP '90),* pp. 497-500, 1990.

[5] H. Xu, F. Hanson, and S. Chung, "Data Parallel Solutions of Dimensionality Problems in Stochastic Dynamic Programming," *Proc. 30th IEEE Conf. Decision and Control (CDC '91),* pp. 1717-1723, 1991.

[6] G. Karypis and V. Kumar, "Efficient Parallel Mappings of a Dynamic Programming Algorithm: A Summary of Results," *Proc. Seventh Int'l Conf. Parallel Processing (ICPP '93),* pp. 563-568, 1993.

[7] S. Huang, H. Liu, and V. Viswannathan, "Parallel Dynamic Programming," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 3, pp. 326-328, Mar. 1994.

[8] G. Lewandowski, A. Condon, and E. Bach, "Asynchronous Analysis of Parallel Dynamic Programming Algorithms," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 4, pp. 425-438, Apr. 1996.

[9] G.R. Gao and K.B. Theobald, "Programming Models and System Software for Future High-End Computing Systems: Work-in-Progress," *Proc. 17th IEEE Symp. Parallel and Distributed Processing (SPDP),* 2003.

[10] A. Aggarwal and J.S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Comm. ACM,* vol. 31, no. 9, pp. 1116-1127, 1998.

[11] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1999.

[12] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-Oblivious Algorithms," *Proc. 40th Ann. Symp. Foundations of Computer Sciences (FOCS '99),* pp. 285-297, 1999.

[13] J.M. Hill, B. McColl, D. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, and R. Bisseling, "BSPLIB: The BSP Programming Library," *Parallel Computing,* 1998.

[14] D. Culler, R. Karp, D.A. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. Eicken, "LOGP: A Practical Model of Parallel Computation," *Comm. ACM,* vol. 39, no. 11, pp. 78-85, 1996.

[15] F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scalable Parallel Computational Geometry for Coarse-Grained Multicomputers," *Proc. Ninth Symp. Computational Geometry (SCG '93),* pp. 298-307, 1993.

[16] F.G. Gustavson, A. Henriksson, I. Jonsson, B. Km, and P. Ling, "Recursive Blocked Data Formats and BLAS's for Dense Linear Algebra Algorithms," *Proc. Fourth Int'l Workshop Applied Parallel Computing (PARA '98),* pp. 195-206, http://citeseer.ist.psu.edu/gustavson98recursive.html, 1998.

[17] S. Coleman and K.S. McKinley, "Tile Size Selection Using Cache Organization and Data Layout," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* 1995.

[18] M. Wolfe, "Iteration Space Tiling for Memory Hierarchies," *Parallel Processing for Scientific Computing,* pp. 357-361, 1987.

[19] J. Ramanujam and P. Sadayappan, "Tiling Multidimensional Iteration Spaces for Nonshared Memory Machines," *Proc. ACM/IEEE Conf. Supercomputing (Supercomputing '91),* pp. 111-120, 1991.

[20] J. Xue and C. Huang, "Reuse Driven Tiling for Improving Data Locality," *Int'l J. Parallel Programming,* vol. 26, no. 6, pp. 671-696, 1998.

[21] J. Hong and H. Kong, "I/O Complexity: The Red Blue Pebble Game," *Proc. 13th Ann. ACM Symp. Theory of Computing (STOC),* 1981.

[22] J. Cuvillo, W. Zhu, Z. Hu, and G.R. Gao, "Fast: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture," *Proc. First Ann. Workshop Modeling, Benchmarking and Simulation (MoBS '05),* held in conjunction with the 32nd Ann. Int'l Symp. Computer Architecture (ISCA), 2005.

[23] J. Cuvillo, Z. Hu, W. Zhu, and G.R. Gao, "Towards a Software Infrastructure for the Cyclops-64 Cellular Architecture," *Proc. 20th Int'l Symp. High Performance Computing Systems and Applications (HPCS),* 2006.

[24] J. Cuvillo, W. Zhu, Z. Hu, and G.R. Gao, "Tiny Threads: A Thread Virtual Machine for the Cyclops-64 Cellular Architecture," *Proc. Fifth Workshop Massively Parallel Processing (WMPP '05),* held in conjunction with the 19th Int'l Parallel and Distributed Processing System, 2005.

[25] *Mpi-2 Specification,* http://www.mpi-forum.org/, 2008.

[26] *Berkeley UPC—Unified Parallel c,* http://upc.lbl.gov/, 2008.

[27] P.G. Bradford, "Efficient Parallel Dynamic Programming," *Proc. 30th Ann. Allerton Conf. Comm. Control and Computing,* pp. 185-194, 1992.

[28] P. Edmonds, E. Chu, and A. George, "Dynamic Programming on a Shared Memory Multiprocessor," *Parallel Computing,* vol. 19, no. 1, pp. 9-22, 1993.

[29] Z. Galil and K. Park, "Parallel Algorithm for Dynamic Programming Recurrences with More Than O(1) Dependency," *J. Parallel and Distributed Computing,* vol. 21, pp. 213-222, 1994.

[30] L. Guibas, H. Kung, and C. Thomson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proc. First Caltech Conf. VLSI,* pp. 509-525, 1979.

[31] B. Louka and M. Tchuente, "Dynamic Programming on Two-Dimensional Systolic Arrays," *Information Processing Letters,* vol. 29, pp. 97-104, 1988.

[32] J.H. Chen, S.Y. Le, B.A. Shapiro, and J.V. Maizel, "Optimization of an RNA Folding Algorithm for Parallel Architectures," *Parallel Computing,* vol. 24, pp. 1617-1634, 1998.

[33] I.H.M. Fekete and P. Stadler, "Prediction of RNA Base Pairing Possibilities for RNA Secondary Structure," *Biopolymers,* vol. 9, pp. 1105-1119, 1990.

[34] 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,* vol. 17, no. 2, pp. 137-148, 2001.

[35] F. Almeida, R. Andonov, and D. Gonzalez, "Optimal Tiling for RNA Base Pairing Problem," *Proc. 14th Ann. ACM Symp. Parallel Architecture and Algorithm (SPAA '02),* pp. 173-182, 2002.

[36] G. Tan, S. Feng, and N. Sun, "Locality and Parallelism Optimization for Dynamic Programming Algorithm in Bioinformatics," *Proc. ACM/IEEE Conf. Supercomputing (Supercomputing),* 2006.

[37] W. Zhou and D.K. Lowenthal, "A Parallel, Out-of-Core Algorithm for RNA Secondary Structure Prediction," *Proc. 35th Int'l Conf. Parallel Processing (ICPP '06),* pp. 74-81, 2006.

[38] J.S. Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data," *ACM Computing Surveys,* vol. 33, no. 2, pp. 209-271, 2001.

[39] G. Tan, S. Feng, and N. Sun, "Load Balancing Algorithm in Cluster-Based RNA Secondary Structure Prediction," *Proc. Fourth Int'l Symp. Parallel and Distributed Computing (ISPDC '05),* pp. 91-96, 2005.

[40] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murph, S.W. Liao, E. Bugnion, and M.S. Lam, "Maximizing Multiprocessor Performance with SUIF Compiler," *Computer,* 1996.

Guangming Tan received the PhD degree in computer science from the Chinese Academy of Science, Beijing. He is an assistant professor in the Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Science. From 2006 to 2007, he was a visiting researcher in the Computer Architecture and Parallel Systems Laboratory (CAPSL), University of Delaware. His main research interests include parallel algorithm and programming, performance modeling and evaluation, and computer architecture. He has published several papers in important conferences and journals, such as SC, SPAA, EuroPar, the *Journal of Supercomputing*, and *International Journal of High Performance Computing Applications*. He is a member of the IEEE.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.

Ninghui Sun received the bachelor degree from Peking University, in 1989 and the master's and PhD degrees from the Chinese Academy of Science (CAS), Beijing, in 1992 and 1999, respectively. He is a professor in the Institute of Computing Technology (ICT), CAS. He is the architect and main designer of Dawning2000, Dawning3000, Dawning4000, and Dawning5000 high-performance computers. His major research interests include computer architecture, operating system, and parallel algorithm. He is a member of the IEEE.

Guang R. Gao received the master's and PhD degrees in electrical engineering and computer science from Massachusetts Institute of Technology, in 1982 and 1988, respectively. He is currently an endowed distinguished professor of electrical and computer engineering at the University of Delaware, Newark. Prior to joining the University of Delaware, he served on the faculty of the School of Computer Science, McGill University, Montreal. He has founded and directed the Computer Architecture and Parallel Systems Lab (CAPSL), University of Delaware. His main research interests include high-performance computer systems: architectures, programming models, compilers, and system software, and their applications. He has devoted much of his time in researching scalable parallel program execution and architecture models that can serve as a basis for high-end parallel supercomputers. To this end, he has pioneered a fine-grained multithreaded execution and architecture model based on his early work on static dataflow models and their extensions. He has also pioneered in applying dataflow models in optimizing compiler technology, e.g., dataflow software pipelining and register allocation. He has led numerous research programs in parallel computing architectures and software sponsored by the US National Science Foundation (NSF), DARPA, DOE, DOD, and other US and Canadian government agencies and private organizations. In addition, he has been involved and contributed in researching several high-end computer architecture designs. He has published more than 200 papers in international journals and conferences. He has co-initiated several important conferences (such as PACT and CASES) and has served as a program committee member in many high-quality international conferences and workshops. He has been a keynote speaker/invited speaker in a number of international meetings. He has served as an editor or as a member of the editorial boards of the *IEEE Transaction on Computers*, *IEEE Concurrency*, and *IFIP Parallel Processing Letters*. He is a fellow of the IEEE and the Association for Computing Machinery (ACM).

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on January 11, 2009 at 21:04 from IEEE Xplore. Restrictions apply.