EasyPDP: An Efficient Parallel Dynamic Programming Runtime System for Computational Biology

Shanjiang Tang, Ce Yu, Jizhou Sun, Bu-Sung Lee, Tao Zhang, Zhen Xu, and Huabei Wu

Abstract—Dynamic programming (DP) is a popular and efficient technique in many scientific applications such as computational biology. Nevertheless, its performance is limited due to the burgeoning volume of scientific data, and parallelism is necessary and crucial to keep the computation time at acceptable levels. The intrinsically strong data dependency of dynamic programming makes it difficult and error-prone for the programmer to write a correct and efficient parallel program. Therefore, this paper builds a runtime system named EasyPDP aiming at parallelizing dynamic programming algorithms on multicore and multiprocessor platforms. Under the concept of software reusability and complexity reduction of parallel programming, a DAG Data Driven Model is proposed, which supports those applications with a strong data interdependence relationship. Based on the model, EasyPDP runtime system is designed and implemented. It automatically handles thread creation, dynamic data task allocation and scheduling, data partitioning, and fault tolerance. Five frequently used DAG patterns from biological dynamic programming algorithms have been put into the DAG pattern library of EasyPDP, so that the programmer can choose to use any of them according to his/her specific application. Besides, an ideal computing distribution model is proposed to discuss the optimal values for the performance tuning arguments of EasyPDP. We evaluate the performance potential and fault tolerance feature of EasyPDP in multicore system. We also compare EasyPDP with other methods such as Block-Cycle Wavefront (BCW). The experimental results illustrate that EasyPDP system is fine and provides an efficient infrastructure for dynamic programming algorithms.

Index Terms—Dynamic programming, Easypdp, DAG data driven model, fault tolerance, DAG pattern, multicore, block cycle.

1 Introduction

Dynamic programming (DP) is a popular algorithm design technique for the solution to many decision and optimization problems. It solves the problem by decomposing it into a sequence of interrelated decision or optimization steps, and then solving them one after another. It has been widely applied in many scientific applications such as computational biology. Typical applications include RNA and protein structure prediction [1], genome sequence alignment [43], context-free grammar recognition [7], string editing, optimal static search tree construction [8], and so on. Indeed, dynamic programming realizes both of optimality and efficiency of the computed results in comparison to other methods for these applications, but the computing cost is still too high when data sharply increase. Therefore, the parallelization for the dynamic programming becomes crucial and necessary. However, by virtue of the strong data dependency of the dynamic programming, it is difficult and error-prone for the programmer to write a correct and efficient parallel program. Moreover, designing highly efficient parallel programs that effectively exploit multiprocessor computer systems is a daunting task that usually falls on a small number of experts, since the traditional parallel programming techniques such as message passing and shared-memory threads are often cumbersome for most developers. They require the programmer to manage concurrency explicitly by creating and synchronizing multithreads through messages or locks, which is difficult and error-prone especially for the inexperienced programmer.

To simplify parallel programming, we need to develop two components [2]: an *abstract programming model* that allows users to describe applications and specify concurrency from the high level, and an *efficient runtime system* which handles low-level thread creating, mapping, resource management, and fault-tolerance issues automatically regardless of the system characteristics or scale. Indeed, the two components are closely related. Recently, there has been a research trend toward these goals by using approaches such as streaming [3], [41], memory transactions [4], [5], data-flow-based schemes [6], and so on.

This paper presents EasyPDP, a runtime system based on DAG Data Driven Model for dynamic programming. The DAG Data Driven Model consists of three modules: user application module, DAG pattern module, and DAG runtime system module. The user application module describes the critical steps that the programmer needs to follow. The DAG pattern module establishes a DAG pattern library, in which there are many DAG patterns provided by the system or defined by users. The DAG runtime system

- *S.J. Tang, C. Yu, J. Sun, T. Zhang, Z. Xu, and H. Wu are with the School of Computer Science and Techonology, Tianjin University, Weijin Road 92#, Nankai District, Tianjin, China 300072. E-mail: {tashj, yuce, jzsun, whb}@tju.edu.cn, {slnazhangtao, xuzhen_0126}@163.com.*
- *B.-S. Lee is with the School of Computer Engineering, Nanyang Technological University, #N4-B2A-03(PDCC), 50 Nanyang Avenue, Singapore 639798. E-mail: ebslee@ntu.edu.sg.*

Manuscript received 22 Apr. 2010; revised 2 Aug. 2011; accepted 3 Aug. 2011; published online 5 Aug. 2011.
Recommended for acceptance by S. Aluru.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-04-0242.
Digital Object Identifier no. 10.1109/TPDS.2011.218.

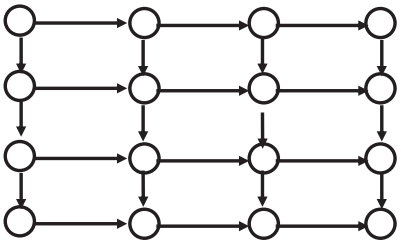


Fig. 1. A DAG Diagram.

module implements the static and dynamic task allocation and scheduling algorithms according to the specific applications. The program starts from the user application module. It gets the selected DAG pattern from the DAG pattern library, and initializes the pattern by setting the pattern size, configuring the data mapping for each DAG node and so on. After that, the DAG runtime system starts to do parallel computation automatically. The final result is obtained when the runtime parallel computing is done.

EasyPDP is aimed at shared-memory systems such as multicore chips and symmetric multiprocessors. It uses threads to spawn parallel data tasks, and further facilitates communication through shared-memory buffers without excessive data copying. The runtime schedules data tasks dynamically to worker threads to achieve a good load balance. The fault-tolerance and recovery mechanism detects and recovers faults automatically during the task execution by reassigning data tasks. Overall, the messy details of parallelization, fault tolerance, data distribution, and load balancing are hidden from the programmer and are handled by the runtime system automatically. However, it also allows the programmer to provide the application-specific knowledge such as functions defined by the user (if the system does not provide).

We evaluate EasyPDP on multicore systems and demonstrate that it leads to scalable performance in this environment. An ideal computing distribution model is proposed to discuss the optimal values for the performance tuning arguments (*DataSize*, *BlockSize*, *ThreadNum*, *Timeout*) of EasyPDP. We empirically demonstrate the EasyPDP dependency on each performance argument. We discuss the EasyPDP overhead by comparing EasyPDP with the sequential iterative code and its cache miss ratio. We also compare EasyPDP to a static parallel scheduling method named Block-Cycle Wavefront (BCW) [9] through some regular and irregular DP applications. The experimental results demonstrate that EasyPDP outperforms BCW for the DP algorithm parallelization. Through fault-injection experiment, we show that the EasyPDP fault-tolerance and recovery mechanism can detect and handle faults during runtime execution.

The remainder of this paper is organized as follows. Section 2 provides an overview of the DAG Data Driven Model. The DP algorithm and its classification are introduced in Appendix A of *supplemental material*, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.218. Section 3 summarizes common types of DAG patterns for DP algorithms. Section 4 presents our EasyPDP implementation. The performance evaluation of EasyPDP is presented

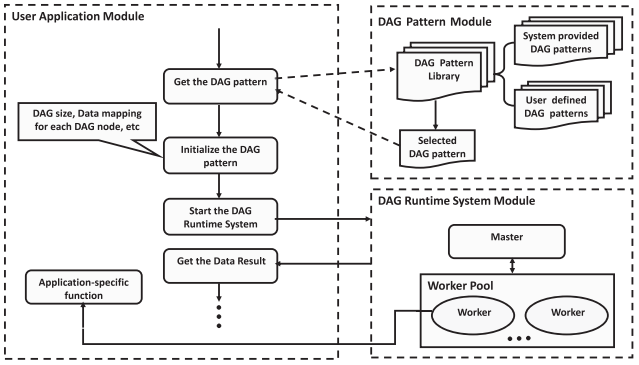


Fig. 2. The DAG Data Driven Model Diagram.

in Appendix C, which is available in the online *supplemental material*. Section 5 reviews related work. Section 6 concludes the paper and gives out future work.

2 DAG Data Driven Model Overview

2.1 Programming Model

Data or tasks with data dependencies and precedence relationships are modeled as Directed Acyclic Graph (DAG), such as Genome alignment, RNA secondary structure prediction, Gene finding, etc. Moreover, there are some applications whose modeled DAG diagrams are almost the same, except for their sizes, as shown in Fig. 1. In view of the reuse concept, we could make those frequently used DAGs as DAG Patterns and establish a DAG pattern library to classify and store them. Besides, lots of static and dynamic task allocation and scheduling algorithms are based on DAG. For the simplicity of parallel programming and the purpose of reusability, we could summarize one or more frequently used algorithms according to the specific application fields and implement them as code skeletons so that the programmer could call them.

Inspired by this idea, we present *DAG Data Driven Model*, as shown in Fig. 2. It is made up of three modules: User Application Module, DAG Pattern Module, and DAG Runtime System Module. The user application module presents critical steps that the programmer needs to follow. The DAG pattern module establishes a DAG pattern library, in which lots of DAG patterns provided by the system or defined by the user are stored. With regard to the DAG runtime system module, it implements the static and dynamic task allocation and scheduling algorithms according to a specific application. The three modules are closely corelated. The following are detailed descriptions of the three modules.

2.2 User Application Module

The user application module is a trigger module which presents the basic steps that should be concerned and done by the programmer. It consists of five steps. According to a specific application, the programmer first chooses one DAG pattern from the DAG pattern library. If there are no suitable DAG patterns for his or her applications, he or she could define a new DAG pattern and add it into the DAG pattern library. The next step is the DAG pattern initialization. For a selected DAG pattern, the programmer should determine its

size (width, height) by setting the corresponding arguments and therefore the total number of DAG nodes can be calculated. For each DAG node, the programmer should map it to the application data block. Before scheduling the DAG runtime system, some arguments must be initialized by the programmer, including input data, block size, the number of threads, and the application-specific function, where the programmer implements the application algorithms and functionality, etc. When the DAG runtime system starts, the application-specific function will be called by the worker threads simultaneously. All the details of parallel programming parts are transparent to the programmer, which are implemented and encapsulated in the DAG runtime system module. After computation, the final data result is returned and can be gained by the programmer.

In the user application module, the programmer just needs to do simple initialization work and focus all of his or her attention on the algorithm or functionality of an application rather than on parallelization. The frequently used DAG patterns are stored in the DAG pattern library so that the programmer can choose to use one of them by specifying the argument, which simplifies the programming and reflects the reuse concept.

2.3 DAG Pattern Module

A DAG is denoted as $D = \{V, E\}$, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of n nodes, E represents the communication relationship and the precedence constraints among data nodes, and $e_{pq} = (v_p, v_q) \in E$ represents a data message sent from data node v_p to v_q, which suggests that v_q can start computing only after v_q is completed.

A DAG pattern is a regular DAG that defines the basic dependence relationships among data nodes, while its size (width, height, etc.) is changeable, set by arguments. By setting different sizes, we can make a DAG pattern suitable for different kinds of applications. For instance, Fig. 1 is a regular DAG, and each of its nodes only depends on its left and upper nodes. It can turn to be a DAG pattern by making its size (width and height) changeable as arguments. For a DAG pattern, each of its DAG nodes maps to a block of data, and the dependency relationships between nodes can be obtained from the DAG pattern, while the computation workload for each DAG node cannot be obtained, which is excluded from the DAG pattern.

For various application fields, we could summarize frequently used DAG patterns and categorize them for each. In order to organize and manage the DAG patterns well, a DAG pattern library should be established. Each DAG pattern in the DAG pattern library has a unique identifier. There are two types of DAG patterns, i.e., one is the system provided, the other is the user defined. The system provided DAG patterns are those frequently used DAG patterns summarized from the applications, while the user defined patterns, which are defined and added to the DAG library by the programmer, are the application-specific DAG patterns instead of the frequently used ones.

2.4 DAG Runtime System Module

The DAG runtime system module is responsible for DAG operations, parallelization, and concurrency control. The runtime system adopts the master-slave pattern. Its master part is used for DAG operations, data tasks allocation, and fault-tolerance control. The DAG operations include the DAG parsing and updating. On one hand, the DAG parsing operation aims at discovering new computable data nodes: it traverses every DAG node and gets all those nodes whose in-degrees are zero. By parsing the DAG, the master allocates those new computable node tasks to workers by putting the tasks into worker pool buffer. On the other hand, the DAG updating operation updates the DAG by removing those completed DAG nodes.

The fault-tolerance and recovery mechanism is necessary and crucial for the DAG Data Driven Model. When a computing DAG node fails, the other nodes that depend on it directly and indirectly will always be incomputable. After a while, there will be no computable data nodes and the whole computation stops thereafter without any results returned. Taking Fig. 1 for instance, without fault-tolerance and recovery mechanism, all nodes that depend on $v_{(2,2)}$ directly and indirectly will finally be incomputable nodes if the computing node $v_{(2,2)}$ fails unexpectedly. For the runtime system, when detecting a failed DAG node, it will reassign the DAG node, clean dirty results as well as workers, and recover the data computing.

In order to manage slave workers, the slave part adopts the structure of the worker pool. It has a pool buffer, which is a task interface between the master and workers. The master puts the computable node tasks into the buffer and workers get the tasks from the buffer. Both static and dynamic worker pools are supported here.

For the static worker pool, each worker has its own buffer. Tasks and workers are bound together according to a certain static data allocation method. Once the master puts a task into the pool buffer, the static worker pool will figure out which worker it belongs to and distributes the task to that worker's own buffer.

For the dynamic worker pool, tasks and workers are not bound. The workers dynamically get the data tasks from the pool buffer. If the pool buffer is not empty, all the workers must be busy working. Compared with the static worker pool, the load balancing for the dynamic worker pool is better.

The whole DAG runtime system execution process is as follows: the program in the user application module starts the runtime system, and the master begins to work. It first gets the user selected DAG and starts the worker pool. Thereafter, the master parses the DAG and puts the computable DAG data node tasks to the pool buffer. The worker pool allocates tasks in the pool buffer to its workers. The worker calls the programmer's application-specific function for computing. When a data node task is completed, the master updates and parses the DAG to find new computable DAG nodes. Once a fault is detected, the master will reassign the data node. The whole runtime process continues until all the DAG node tasks are completed.

3 The DAG Patterns for DP Algorithms

In Section 2, we categorize DAG patterns according to various application fields and further establish the DAG pattern library to manage them. Here for DP applications, we present five common frequently used DAG patterns derived from the DP algorithms shown in Fig. 3. Each DAG

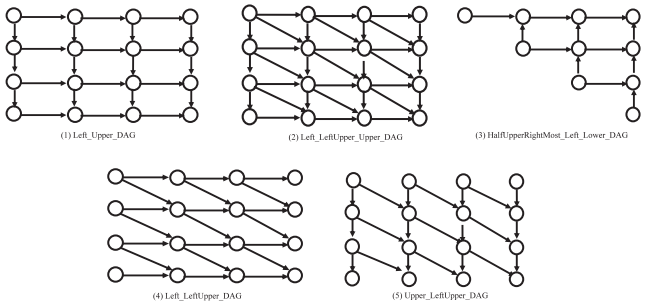


Fig. 3. Some DAG patterns for DP algorithms.

pattern is given a unique identifier according to its data dependency relationship. The *Left_LeftUpper_Upper_DAG* pattern, *Left_Upper_DAG* pattern and *HalfUpperRightMost_Left_Lower_DAG* pattern are directly derived from (a), (b), (c) of Fig. 1 available in the online *supplemental material*, respectively, while the *Left_LeftUpper_DAG* and *Upper_LeftUpper_DAG* are extracted from (a), (b) of Fig. 2 available in the online *Supplemental Material*.

Although DAG patterns are often summarized from the regular DP algorithms, they can be used in many irregular ones. Taking the *Left_Upper_DAG* pattern for example, in addition to fitting for regular DP algorithms like (b) of Fig. 1 available in the online *Supplemental Material*, it is also well suited to irregular DP algorithms such as (d) of Fig. 1 available in the online *Supplemental Material*, (c) and (d) of Fig. 2 available in the online *Supplemental Material*. With the same DAG pattern, the difference between the regular and irregular DP algorithms lies in the pattern-independent workload for each DAG node that represents a block of data.

Essentially, there exist intrinsic connections between different patterns. For example, *Left_Upper_DAG* pattern and *Left_LeftUpper_Upper_DAG* pattern are topologically equivalent. Both the *Left_LeftUpper_DAG* pattern and *Upper_LeftUpper_DAG* pattern can be extracted from *Left_LeftUpper_Upper_DAG* pattern by eliminating all its upper/left dependencies. To put it another way, we could use the *Left_LeftUpper_Upper_DAG* pattern instead of *Left_LeftUpper_DAG* pattern and *Upper_LeftUpper_DAG* pattern in some cases, except that it decreases the parallelization degree.

4 The EasyPDP System

EasyPDP implements the DAG Data Driven Model for shared-memory systems. Its goal is to support efficient execution on multiple cores without burdening the programmer with concurrency management for DP algorithms. EasyPDP consists of a simple API that is visible to application programmers, runtime functions that are invisible to application programmers and an efficient runtime that handles parallelization, DAG operations and fault recovery. The example and usability of EasyPDP is explained in Appendix B available in the online *Supplemental Material*. Moreover, the experimental performance evaluation of EasyPDP is discussed in Appendix C available in the online *Supplemental Material*.

4.1 The EasyPDP Functions

The current EasyPDP implementation provides four types of functions for C and C++, i.e., *user programming API*, *DAG operation function*, *worker pool function*, and *fault-tolerance function*. However, similar functions can be defined for other languages such as Java or C#. The details are summarized in Table 1.

The *user programming API*, which is visible to application programmers, includes two sets of functions. One set is provided by EasyPDP, but used in the programmer's application code to initialize the system (1 required function), and the other set is the user-defined functions (1 required and 2 optional functions). Apart from the process function that takes on the actual computation for the application algorithms, the user could provide a DAG pattern initialization function for the user-defined DAG pattern as well as the data mapping function to map the DAG nodes to the application data blocks. For the EasyPDP API, it neither relies on any specific compiler options nor requires a parallelizing compiler. However, it assumes that its functions can freely use stack-allocated and heap-allocated structures for private data on demand. It also assumes that there is no communication through shared-memory structures other than the input/output buffers for these functions. For C/C++, we cannot check these assumptions statically for arbitrary programs. Although there are stringent checks within the system to ensure that valid data are communicated between the user and the runtime code, eventually we trust the user to provide functionally correct code. For Java and C#, static checks that validate these assumptions are possible.

For the *DAG operation function*, it has two optional default functions that initialize the system-provided DAG patterns and map DAG nodes to data blocks. The *DAG pattern handle function* can parse the DAG for finding new computable DAG nodes and update the DAG pattern by deleting completed node from current DAG pattern.

To the *worker pool function*, it provides some basic thread pool operation functions. The *pool initialization function* plays the role of initializing the pool queue together with queue lock and creating threads; *pool destroy function* is used to destroy threads and free the memory space accordingly; *pool queue tasks adding function* and *runtime thread routine function* take on the work of actual computation.

The EasyPDP provides support for the fault tolerance. It detects faults through timeouts. The critical *fault-tolerance function* includes *DAG node adding functions* and *DAG node removing functions* for the *timeoutQueue*, *timeout checking function* that detects the timeout DAG nodes from *timeoutQueue*. Once detected, the timeout DAG node is removed from the *timeoutQueue*, the timeout worker thread is cleaned up and then the removed nodes are redistributed.

4.2 The EasyPDP Runtime

In order to obtain a good load balance for both regular and irregular DP algorithms, the EasyPDP runtime adopts the dynamic worker pool, which uses dynamic allocation and scheduling algorithms. Moreover, the EasyPDP runtime is developed on top of Pthreads [26], but can be easily ported to other shared-memory thread libraries.

4.2.1 Basic Operation and Control Flow

Fig. 4 shows the basic data flow for EasyPDP runtime system. The runtime is controlled by the scheduler (master) and initialized by the user program. The programmer

TABLE 1
The Functions in the EasyPDP

Function Description	R/O
User Programming API	
`int EasyPDP_scheduler(scheduler_args_t * arg)` Initializes the EasyPDP runtime system. The *scheduler_args_t* provides the needed function & data pointers.	R
`void (*process_t)(void *)` The application process function, defined by the user, called by the pool workers.	R
`void (*DAG_Pattern_init_t)(void *)` DAG pattern initialization function, defined by the user, in order to support the user defined DAG patterns. EasyPDP provides a default DAG pattern initialization function where there are lots of system provided DAG patterns.	O
`data_blocks* (*DAG_pattern_node_data_mapping_t)(void* arg,int DAG_pattern_node_id)` Maps the DAG node with application data block, defined by the user. If not specified, EasyPDP uses a default mapping function.	O
DAG Operation Related Function	
`void default_DAG_pattern_init(scheduler_args_t* arg)` The default DAG pattern initialization function, where lots of system provided DAG patterns are initialized.	O
`void DAG_pattern_handle(int DAG_pattern_node_id,DAGPattern_args_t* arg)` The DAG pattern operation function. It can parse the DAG pattern to discover current new computable DAG nodes, and can update the DAG pattern by deleting a DAG node from DAG pattern.	R
`data_blocks* default_DAG_pattern_node_data_mapping(scheduler_args_t* arg,int DAG_pattern_node_id)` The default DAG pattern node mapping function.	O
Worker Pool Related Function	
`void pool_init (int thread_num)` The worker pool initialization function. It initializes the pool queue, queue_lock and creates *thread_num* threads.	R
`int pool_destroy ()` Destroys the pool and frees the memory space.	R
`int pool_add_worker (process_t process, void *arg)` Adds a new task into the pool queue.	R
`void *thread_routine (void *arg)` The pool threads runtime routine function.	R
Fault Tolerance Related Function	
`void add_timeoutQueue(int DAG_pattern_node_id)` Adds a new computable DAG node into timeoutQueue.	R
`void remove_timeoutQueue(int DAG_pattern_node_id)` Removes the timeout DAG node or finished DAG node from timeoutQueue.	R
`void timeout_check_timeoutQueue(scheduler_args_t* arg)` Checks the timeoutQueue to see whether there are timeout DAG nodes. If existing, it removes the timeout DAG nodes from timeoutQueue, cleans the timeout worker thread and redistributes the timeout DAG node.	R

R and O identify required and optional fuctions, respectively.

provides the scheduler with all the required data and function pointers in terms of the `scheduler_args_t` structure, which is the only data structure used for the basic function and buffer allocation information to be communicated between the user program and the runtime. The fields of `scheduler_args_t` are presented in Table 2. The basic fields provide both pointers to DP data buffers and user-provided functions. For the user's DAG pattern, the EasyPDP runtime system provides a basic data structure for user to define his own DAG pattern and an interface (callback function) for adding the pattern into the DAG pattern library. When configured with user's DAG pattern, the runtime will automatically call the user's callback function to initialize the DAG pattern. The performance tuning fields present some key arguments that affect the system performance. All the fields should be properly set by the programmer before calling `EasyPDP_scheduler`. After initialization, the master scheduler calls the `DAG_pattern_init` function to initialize the DAG pattern and `pool_init` function to startup the worker pool. After that, the master scheduler calls `DAG_pattern_handle` function to discover new computable DAG nodes whose in-degrees are zero and then `DAG_pattern_node_data_mapping` function to map DAG nodes to data blocks before sending them into the pool buffer.

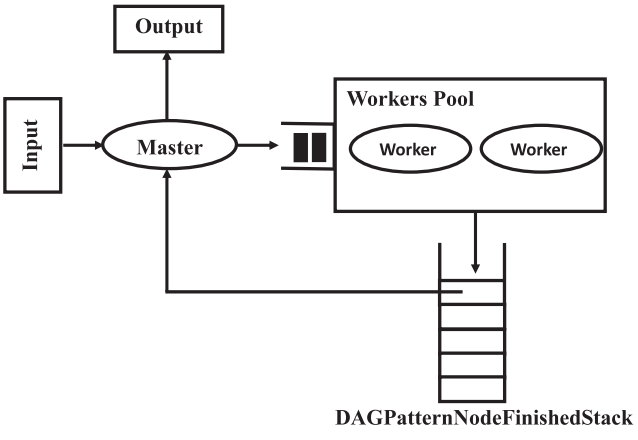


Fig. 4. The basic data flow for the EasyPDP runtime.

Once the pool buffer is not empty and there are idle workers, the worker pool will distribute data tasks in the pool buffer to idle workers. The `Process` is called by worker threads to do DP algorithm computation. When a worker thread completes a DAG node task, it pushes the corresponding DAG node id into *DAGPatternNodeFinishedStack* for notifying the master.

The master checks the *DAGPatternNodeFinishedStack* in a small regular time for completed DAG nodes. Once getting a DAG node, the master will call `DAG_pattern_handle` to update DAG and parse the DAG to find new computable DAG nodes. The whole process continues until all the DAG node tasks are completed. Finally, the output results return.

TABLE 2
The Fields of *scheduler_args_t* Data Structure

Field	Description
Basic Fields	
dp_data	The matrix DP data. All the data computations are based on it.
data_row	The number of rows for matrix dp_data.
data_col	The number of columns for matrix dp_data.
DAG_pattern_id	The identity of user selected DAG pattern.
process	Pointer to DP computation function.
DAG_pattern_init	Pointer to the user defined DAG pattern initialization function.
DAG_pattern_node_data_mapping	Pointer to the user Map function.
Performance Tuning Fields	
block_row	The number of rows for data block.
block_col	The number of columns for data block.
thread_num	The number of threads.
timeout	The value of timeout. If timeout≤0, the fault recovery mechanism doesn't work. Otherwise, it works.

4.2.2 Fault Tolerance

Recall in Section 2.4 that the failure of a computing DAG node can cause all other nodes that depend on it directly and indirectly to be incomputable, and the whole computations will eventually pause at a place forever without fault-tolerance and recovery mechanism. Therefore, it is critical and necessary to build a fault-tolerance and recovery mechanism to detect and recover from faults.

EasyPDP detects faults through timeout. If a worker does not complete a task within a reasonable amount of time, then a failure is assumed. Of course, a fault may cause a task to complete with incorrect or incomplete data instead of failing completely. EasyPDP has no way of detecting this case on its own and cannot stop an affected task from potentially corrupting the shared memory. To address this shortcoming, one should combine the EasyPDP runtime with other known error detection techniques [27], [28]. Two kinds of faults that cause timeout are considered here. One is caused by the death of a worker thread. The other case is that a computing worker thread goes into the dead-loop or deadlock for some reasons.

Fig. 5 presents the overall flow of EasyPDP fault-tolerance mechanism. When the user program calls `EasyPDP_scheduler`, the following sequence of actions occur (the numbered labels in Fig. 5 correspond to the numbers in the list below).

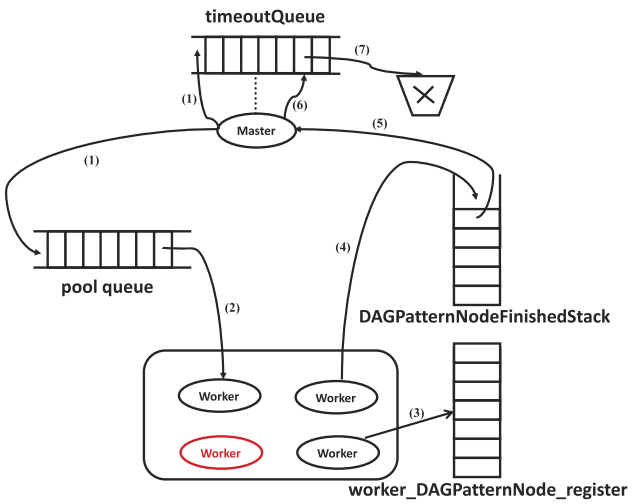


Fig. 5. The overall flow of EasyPDP fault-tolerance mechanism.

1. The master distributes computable DAG nodes discovered by parsing the DAG to both the *timeoutQueue* and *pool queue* simultaneously. For the *timeoutQueue*, it has a *time_start* field that records the current time for each DAG node when it is put into the *timeoutQueue*.
2. The worker pool gets computable DAG data tasks from the pool buffer and distributes them to its idle worker threads dynamically.
3. When an idle worker thread obtains a DAG node task, it will register its thread id and the DAG node id in the *worker_DAGPatternNode_register* before doing DP algorithm computation.
4. Once a worker completes a DAG node task, it will push the DAG node id to the *DAGPatternNodeFinishedStack* for notifying the master.
5. The master fetches the finished DAG node id from the *DAGPatternNodeFinishedStack* in a small regular time. Then, it goes to Step 7.
6. The master checks the *timeoutQueue* to see whether there are timeout DAG nodes. Note that the value of *time_start* for each DAG node in the *timeoutQueue* strictly increases from queue front to queue rear. Thereby the master need not check all nodes in the *timeoutQueue* every time. Instead, it just needs to check nodes from the queue rear to queue front in order until a nontimeout node is found. If a timeout DAG node is detected, the master looks up the corresponding timeout thread id through the *worker_DAGPatternNode_register*, and then makes the worker pool kill that thread and instead create a new one, and go to Step 7 to remove the timeout DAG node from the *timeoutQueue*. After that, the master goes to Step 1 to redistribute the timeout DAG node.
7. The master removes a DAG node from the *timeoutQueue*. Then, it goes to Step 1.

The current EasyPDP does not provide fault recovery for the master scheduler itself. The master scheduler runs only for a very small fraction of the time and has a small memory footprint, hence it is less likely to be affected. On the other hand, a fault in the master scheduler has more serious

implications for the correctness of the program. We can use known techniques such as the redundant execution or checkpointing to address this shortcoming.

4.2.3 Buffer Operation and Management

Four types of temporary buffers shown in Fig. 5 are necessary to store data and support fault tolerance. All buffers are allocated in shared memory but are accessed in a well-specified way by a few functions, and are not directly visible to user code.

The *pool queue* buffer is the only data interface between the master and the worker pool. The master sends the computable data tasks into the *pool queue* buffer, and the worker pool fetches data from it. The queue lock is used to guarantee that only one access exists every time.

In order to notify the master to update the DAG in real time, the *DAGPatternNodeFinishedStack* buffer is adopted. Every time the worker finishes the DAG node task, it writes the DAG node id into the *DAGPatternNodeFinishedStack* buffer so that the master could know it at once.

The *worker_DAGPatternNode_register* buffer and *timeoutQueue* buffer are two critical parts of fault-tolerance mechanism. For the *timeoutQueue* buffer, it is only visible to master and has a *time_start* field that records the distributed time for each DAG node. The master repeatedly checks it with the current time to see whether it exceeds the *timeout* in a regular time. If a DAG node is assumed to be timeout, the master will notify the worker pool to kill the dirty worker just in case there are dead-loop threads or deadlock threads. Since the EasyPDP adopts dynamic worker pool, the master cannot know which worker thread did the timeout DAG node task without *worker_DAGPatternNode_register* buffer. Every time a worker gets a DAG node task, it will register its thread id in *worker_DAGPatternNode_register* buffer for that DAG node.

4.2.4 Refinements

Table 2 shows the performance tunable arguments that the user could use to optimize his/her application. Some optimization topics about these arguments are described below.

Block size. The user setting of arguments `block_row` and `block_col` determines the size of a data block. Note that each block size setting will directly affect the size of the corresponding DAG pattern for a DP application, which in turn affects the parallelization degree indirectly. For the irregular DP algorithms such as (d) of Fig. 1 available in the online *Supplemental Material*, since computation workloads of matrix cells are unequal (irregular), the workload of each DAG node is sharply unequal (irregular) when the size of data block is larger.

Number of threads. In systems with multiple cores, since DP applications are data intensive, it had better set the value of argument `thread_num` as the number of available cores in order to typically maximize the system throughput even if an individual task takes longer time.

Timeout. If a failure occurs during the runtime execution, the timeout value will be a critical criterion for the fault-tolerance and recovery mechanism to detect the fault in real time. On one hand, too large value of timeout will make the fault-tolerance and recovery mechanism obtuse to discover faults; on the other hand, too small value of timeout will make the fault-tolerance and recovery mechanism wrongly assume that a being computed task is failed and recompute that task, which also adversely influences the performance. Therefore, the user's proper value setting of timeout is important according to the specific characteristics for various DP applications. For the irregular DP algorithms, the workload for each DAG node task may be unequal, which means that the user's timeout value setting may not suit most irregular DAG node tasks. To address this shortcoming, we present a self-adjusted/adaptive mechanism for timeout. That is, according to the successfully completed DAG node task from *DAGPatternNodeFinishedStack*, the total execution time for that DAG node task can be calculated by subtracting *time_start* for that DAG node in *timeoutQueue* from the current time. If the total execution time is less than *timeout*, but greater than 80 percent of *timeout*, it indicates that the *timeout* is a bit small at present and then our self-adjusted/adaptive mechanism will double the current *timeout* value.

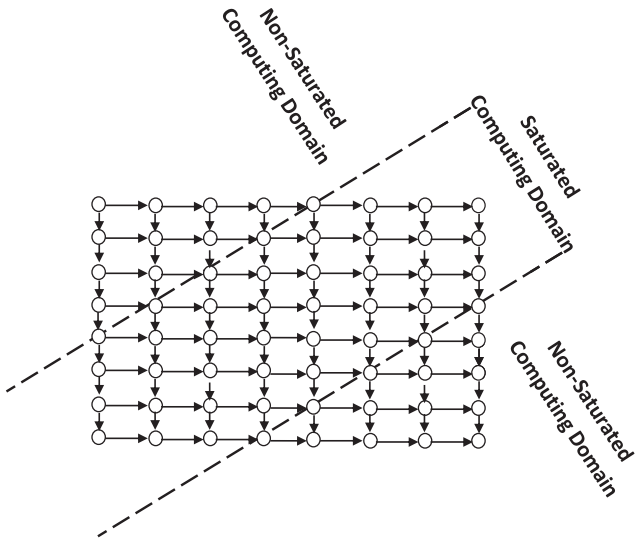


Fig. 6. The ideal computing distribution model for a regular DP algorithm.

4.2.5 Discussion on Performance Arguments

The proper setting for each performance argument mentioned above is critical for EasyPDP runtime performance. However, what is the relationship between these arguments for performance optimization? Are these arguments independent of each other or not? In order to answer these questions, we take the regular DP application as an example for analysis.

Fig. 6 is an ideal computing distribution model for a common DAG of a regular DP application. It is made up of three computing domains: two nonsaturated computing domains and one saturated computing domain. Here, we assume that the computing process goes along the anti-diagonal step by step in the EasyPDP system. On one hand, in the nonsaturated computing domain, its maximum parallelization degree is less than the number of computing workers. It implies that there must be some idle workers when computation is going on in the domain. On the other hand, the saturated computing domain indicates that the

maximum parallelization degree is greater than or equal to the number of computing workers. All the workers should be busy and no idle workers exist during the computation in this domain. Variable definitions for further discussion are listed as follows:

- d : the size of the targeted matrix data. $d = n \times n$ when it is a $n \times n$ matrix.
- b : the size of a block data. $b = m \times m$ when the block is a $m \times m$ matrix.
- t : the number of threads.
- r : the initial value of *timeout*.
- k_{ij} : the calculation cost for a matrix cell (i, j).
- k : the average calculation cost for each matrix cell.
- c_{ij} : the communication cost for a matrix cell (i, j).
- c : the average communication cost for each matrix cell.
- c_0 : the average cost of all other overheads for a block data, such as DAG node parsing and updating cost.

The average calculation cost for a block data should be $b \times k$. There is no communication cost for a matrix cell among the worker threads in the shared-memory environment in contrast to the message passing in the distributed memory environment. So the average total computation cost for a block data should be $b \times k + c_0$. Then, we could get

1. The total nonsaturated computing domain cost is
$$2 \times (t-1) \times (b \times k + c_0). \tag{1}$$
2. The total saturated computing domain cost is
$$\frac{\frac{d}{b} - 2 \times \frac{(t-1)\times t}{2}}{t} \times (b \times k + c_0). \tag{2}$$
3. Therefore, the total cost for a regular DP algorithm is
$$\begin{aligned} S &= 2 \times (t-1) \times (b \times k + c_0) \\ &\quad + \frac{\frac{d}{b} - 2 \times \frac{(t-1)\times t}{2}}{t} \times (b \times k + c_0) \\ &= (b \times k + c_0) \times \left(t - 1 + \frac{d}{b \times t} \right). \end{aligned} \tag{3}$$

Based on the results above, we could obtain the following conclusion.

Theorem 1. *When $t = t_0 (t_0 \geq 1)$, the optimal value of b for the minimal value S is*

$$b = \begin{cases} d & (t_0 = 1), \\ \sqrt{\dfrac{c_0 \times d}{k \times t_0 \times (t_0 - 1)}} & (t_0 > 1). \end{cases} \tag{4}$$

Therefore, the optimal minimal value S is

$$S = \begin{cases} d \times k + c_0 & (t_0 = 1), \\ \dfrac{\sqrt{t_0 \times (t_0-1) \times c_0 \times d \times k} + d \times k \times \sqrt{t_0}}{t_0} & \\ \quad + c_0 \times (t_0 - 1) + \sqrt{c_0 \times d \times k \times (t_0 - 1)} & (t_0 > 1). \end{cases} \tag{5}$$

Proof. See Appendix D available in the online *Supplemental Material*. □

Theorem 2. *When $b = b_0 (0 > b_0 \leq d)$, the optimal value of t for the minimal value S is*

$$t = \sqrt{\frac{d}{b_0}} \quad (0 < b_0 \leq d), \tag{6}$$

in this case, the optimal minimal value S is

$$S = (b_0 \times k + c_0) \times \left(2 \times \sqrt{\frac{d}{b_0}} - 1 \right) \quad (0 < b_0 \leq d). \tag{7}$$

Proof. See Appendix E available in the online *Supplemental Material*. □

Theorem 3. *The optimal value of timeout r for the DP algorithm is*

$$r = \begin{cases} \left(\left\lceil \dfrac{\sqrt{\frac{d}{b}} - t}{t} \right\rceil + 1 \right) & \\ \quad \times (b \times k + c_0) + \varepsilon & \left(t < \sqrt{\dfrac{d}{b}}, 0 < b \leq d \right), \\ (b \times k + c_0) + \varepsilon & \left(t \geq \sqrt{\dfrac{d}{b}}, 0 < b \leq d \right). \end{cases} \tag{8}$$

where $\varepsilon (\varepsilon > 0)$ represents a small extra necessary delayed time for the fault-tolerance and recovery mechanism.

Proof. See Appendix F available in the online *Supplemental Material*. □

5 Related Work

As an efficient algorithm design technique, DP has been widely applied in many scientific applications such as computational biology. With the burgeoning amount of scientific data, the DP computation cost is still too high, and it is necessary and meaningful to parallelize it. Lots of work has been done to exploit the parallelization of DP algorithms. Edmonds et al. [29] and Galil and Park [30] described several parallel algorithms on general shared-memory multiprocessor systems. Bradford [31] presented several algorithms that solve optimal matrix chain multiplication parenthesization using the CREW PRAM model. Tan et al. [33], [34] focused on a specific type of nonserial polyadic DP with triangular matrix, for which he introduced some optimization algorithms and theoretical models on multicore architectures. All these works above are concerned with how to reduce the complexity of the arithmetic cost on varied theoretical parallel models, while the parallel implementation complexities for their approaches are not well considered. For the distributed memory multiprocessor system, Almeida et al. [35] presented a parallel implementation with tiling on a ring processors, whereas this parallel tiling algorithm cannot achieve load balance. Zhou and Lowenthal [36] proposed a parallel out-of-core [37] algorithm based on the conventional out-of-core model. Although a load-balancing algorithm was used, the method only ensures that the number of entries for each processor is the same, however, it cannot be satisfied for the irregular DP since the arithmetic cost on each processor is not the same because of the irregular data dependence. Liu

and Schmidt presented a static parallel scheduling strategy named block-cycle-based wavefront method and did some work including giving some algorithms, making a pattern-based prototype system afterward, in [9], [38], and [39]. Despite some optimization work they have done, their static scheduling methods still cannot achieve a better load balance especially for irregular DP applications compared with the dynamic scheduling as the EasyPDP runtime has adopted. In contrast, some other schedulers such as work-stealing (WS) scheduler [45] and parallel depth-first search (PDF) scheduler [47], [48], [49], which take data locality into account, can have a good performance for DP algorithms as well due to the reduced cache and TLB misses. We explained in Section 4.2.2 that the fault tolerance is crucial and necessary for parallel DP algorithms, whereas there are no fault-tolerance and recovery mechanisms which have been considered and supported in previous work except our EasyPDP. Chowdhury [11], [12], [13], [14] proposed a cache-efficient divide-and-conquer algorithm which divides the DP problem into lots of subproblems and solves them concurrently in one direction. It can achieve a good cache efficiency and space complexity. In contrast, our EasyPDP automatically partitions the DP into lots of blocks represented with DAG according to the argument *BlockSize* set by the user, and solves each by working threads iteratively. The argument *BlockSize* of EasyPDP could be used to reduce the cache miss (see discussion in Appendix C.4 available in the online *Supplemental Material*) in some cases.

For simplifying parallel programming, we propose EasyPDP based on our DAG Data Driven Model for DP applications. Recently, there is a significant approach that is developing two components: a parallel programming model and runtime system for parallel program simplification. MapReduce [40] proposed by google for simplifying the parallel programming and data processing on clusters is a parallel programming model as well as a runtime system at the same time. Phoenix [2] is an efficient runtime system based on MapReduce model for shared-memory systems. StreamIt uses a synchronous data-flow model that allows a compiler to automatically map a streaming program to a multicore system [41].

6 Conclusion and Future Work

In the paper, a parallel dynamic runtime system named EasyPDP, based on DAG Data Driven Model, is proposed and implemented for parallelization of DP algorithms in shared-memory systems. However, it can also be applied to other applications, especially those with strong data or task dependencies. With EasyPDP, the programmer only needs to concern about the specific DP formulas, provides some application-related arguments, and leaves parallelization and scheduling tasks to the EasyPDP runtime system. EasyPDP automatically partitions the DP matrix into blocks and allocates those computable block data to idle threads dynamically during the parallel execution. It can also recover from runtime faults through its timeout fault-tolerance mechanism. An ideal computing distribution model is proposed to discuss the performance tuning arguments (*DataSize*, *BlockSize*, *ThreadNum*, *Timeout*) of EasyPDP (see Section 4.2.5). We empirically demonstrate the dependency on each performance argument by experiments. We discuss EasyPDP overhead by comparing EasyPDP with the sequential code and its cache miss. We also compare EasyPDP to a static parallel scheduling method named Block-Cycle Wavefront. Experimental results show that EasyPDP outperforms BCW for DP algorithms parallelization.

Indeed, there are still some shortcomings in our current version of EasyPDP, which will be addressed in our future work. For instance, space complexity is a major issue in the usability of many DP algorithms, and there are some methods and technologies [42], [43], [44], can be used to reduce it. Moreover, the argument *BlockSize* of EasyPDP can be used to reduce the cache miss (see discussion in Appendix C.4 available in the online *Supplemental Material*) for some DP algorithms, but is not sufficient enough. However, the schedulers such as work-stealing scheduler for distributed caches [45], which take data locality into account, can reduce cache miss significantly. We plan to incorporate it into our future version of EasyPDP by using Cilk [46].

In addition, our future work also include expanding EasyPDP to support other multiprocessor/many-core architectures such as CELL BE and GPU, exploring more commonly used DAG patterns for DP algorithms, and extending current EasyPDP so that it can be applied to other applications. Moreover, in order to achieve a good reusability, we plan to optimize our EasyPDP system and provide some basic APIs needed for biological programming further in future.

The EasyPDP source code is publicly available for downloading at http://easypdp.sourceforge.net/.

Acknowledgments

The authors would like to thank the editor, all the reviewers, Chao Sun, Jun Du, Liya Fan, Libo Sun, and Tingxu Yan for the help in improving this paper. This work is sponsored by the National Natural Science Foundation of China (10978016, 11003027), and the Key Technologies R & D Program of Tianjin, China (09ZCKFGX00400, 11ZKFGX01000). C. Yu (yuce@tju.edu.cn) is the corresponding author.

References

[1] J. Bowie, R. Luthy, and D. Eisenberg, "A Method to Identify Protein Sequences that Fold into a Known Three-Dimensional Structure," *Science,* vol. 253, no. 5016, pp. 164-170, 1991.

[2] C. Ranger et al., "Evaluating MapReduce for Multi-Core and Multiprocessor Systems," *Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture,* pp. 13-24, 2007.

[3] E. Kohler, R. Morris, and B. Chen, "Programming Language Optimizations for Modular Router Configurations," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 251-263, 2002.

[4] B.D. Carlstrom et al., "The ATOMO Transactional Programming Language," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 1-13, June 2006.

[5] T. Harris and K. Fraser, "Language Support for Lightweight Transactions," *Proc. 18th Ann. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications,* Oct. 2003.

[6] S. Balakrishnan and G.S. Sohi, "Program Demultiplexing: Data-Flow Based Speculative Parallelization of Methods in Sequential Programs," *Proc. 33rd Ann. Int'l Symp. Computer Architecture (ISCA '06),* June 2006.

[7] C. Ciressan, E. Sanchez, M. Rajman, and J.C. Chappelier, "An FPGA-Based Coprocessor for the Parsing of Context-Free Grammars," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* 2000.

[8] M. Farach and M. Thorup, "Optimal Evolutionary Tree Comparison by Sparse Dynamic Programming," *Proc. 35th Ann. Symp. Foundations of Computer Science,* pp. 770-779, 1994.

[9] W.G. Liu and B. Schmidt, "Parallel Design Pattern for Computational Biology and Scientific Computing Applications," *Proc. IEEE Int'l Conf. Cluster Computing,* pp. 456-459, 2003.

[10] V. Kumar, A. Grama, A. Gupa, and G. Karypis, *Introduction to Parallel Computing.* Benjamin/Cummings Publishing Company, Inc., 1994.

[11] R.A. Chowdhury and V. Ramachandran, "Cache-Efficient Dynamic Programming Algorithms for Multicores," *Proc. 20th Ann. Symp. Parallelism in Algorithms and Architectures,* pp. 207-216, 2008.

[12] R.A. Chowdhury, H.S. Le, and V. Ramachandran, "Cache-Oblivious Dynamic Programming for Bioinformatics," *IEEE/ACM Trans. Computational Biology and Bioinformatics,* vol. 7, no. 3, pp. 495-510, July-Sept. 2009.

[13] R.A. Chowdhury and V. Ramachandran, "Cache-Oblivious Dynamic Programming," *Proc. 17th Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 591-600, 2006.

[14] R.A. Chowdhury, H. Le, and V. Ramachandran, *Efficient Cache-Oblivious String Algorithms for Bioinformatics,* Technical Report TR-07-03, Dept. of Computer Sciences, Univ. of Texas, Feb. 2007.

[15] G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably Good Multicore Cache Performance for Divide-and-Conquer Algorithms," *Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 501-510, 2008.

[16] Z. Galil and K. Park, "Dynamic Programming with Convexity, Concavity and Sparsity," *Theoretical Computer Science,* vol. 92, pp. 49-76, 1992.

[17] X. Huang and K.M. Chao, "A Generalized Global Alignment Algorithm," *Bioinformatics,* vol. 19, no. 2, pp. 228-233, 2003.

[18] N. Futamura, S. Aluru, and X. Huang, "Parallel Syntenic Alignments," *HiPC '02: Proc. Ninth Int'l Conf. High Performance Computing,* pp. 420-430, 2002.

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

[20] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Protein and Nucleic Acids.* Cambridge Univ. Press, 1998.

[21] R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman, "Algorithms for Loop Matchings," *SIAM J. Applied Math.,* vol. 35, no. 1, pp. 68-82, 1978.

[22] A. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Trans. Information Theory,* vol. TIT-13, no. 2, pp. 260-269, Apr. 1967.

[23] D.W. Mount, *Bioinformatics-Sequence and Genome Analysis.* Cold Spring Harbor Laboratory Press, 2001.

[24] M.S. Gelfand, A.A. Mironov, and P.A. Pevzner, "Gene Recognition via Spliced Sequence Alignment," *Proc. Nat'l Academy of Sciences of USA,* vol. 93, no. 17, pp. 9061-9066, 1996.

[25] M. Zuker and P. Stiegler, "Optimal Computer Folding of Large RNA Sequences Using Thermodynamics and Auxiliary Information," *Nucleic Acids Research,* vol. 9, no. 1, pp. 133-148, 1981.

[26] B. Lewis and D.J. Berg, *Multithreaded Programming with Pthreads.* Prentice Hall, 1998.

[27] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K.S. Kim, "Robust System Design with Built-In Soft-Error Resilience," *Computer,* vol 38, no. 2, pp. 43-52, 2005.

[28] J.C. Smolens et al., "Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth," *Proc. 11th Int'l Conf. Architectural Support for Programming, Languages and Operating Systems,* pp. 224-234, Oct. 2004.

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

[30] Z. Galil and K. Park, "Parallel Algorithm for Dynamic Programming Recurrences with More than O(1) Dependency," *J. Parallel and Distributed Computing,* vol. 21, no. 2, pp. 213-222, 1994.

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

[32] A. Mark and S. Ramesh, "PC Software Performance Tuning," *Computer,* vol. 29, no. 8, pp. 47-54, 1996.

[33] G.M. Tan, H.N. Sun, and R.G. Gao, "A Parallel Dynamic Programming Algorithm on a Multi-Core Architecture," *Proc. 19th Ann. ACM Symp. Parallel Algorithms and Architectures,* pp. 135-144, 2007.

[34] G.M. Tan et al., "Locality and Parallelism Optimization for Dynamic Programming Algorithm in Bioinformatics," *Proc. ACM/IEEE Conf. Supercomputing (SC '06),* pp. 11-17, 2006.

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

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

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

[38] W. Liu and B. Schmidt, "A Generic Parallel Pattern-Based System for Bioinformatics," *Proc. EURO-PAR,* pp. 989-996, 2004.

[39] W. Liu and B. Schmidt, "Parallel Pattern-Based Systems for Computational Biology: A Case Study," *IEEE Trans. Parallel and Distributed Systems,* vol. 17, no. 8, pp. 750-763, Aug. 2006.

[40] J. Dean and J. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. ACM,* vol. 51, no. 1, pp. 107-113, 2008.

[41] M.I. Gordon et al., "A Stream Compiler for Communication-Exposed Architectures," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 291-303, Oct. 2002.

[42] D. Hirschberg, "A Linear Space Algorithm for Computing Maximal Common Subsequences," *Comm. ACM,* vol. 18, no. 6, pp. 341-343, 1975.

[43] X. Huang, "A Space-Efficient Parallel Sequence Comparison Algorithm for a Message-Passing Multiprocessor," *Int'l J. Parallel Programming,* vol. 18, no. 3, pp. 223-239, 1989.

[44] S. Rajko and S. Aluru, "Space and Time Optimal Parallel Sequence Alignments," *IEEE Trans. Parallel and Distributed Systems,* vol. 15, no. 12, pp. 1070-1081, Dec. 2004.

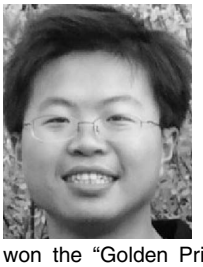
[45] U.A. Acar, G.E. Blelloch, and R.D. Blumofe, "The Data Locality of Work Stealing," *Theory of Computing Systems,* vol. 35, no. 3, pp. 321-347, 2002.

[46] M. Frigo, C.E. Leiserson, and K.H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 212-223, 1998.

[47] G.E. Blelloch and P.B. Gibbons, "Effectively Sharing a Cache among Threads," *Proc. 16th Ann. ACM Symp. Parallelism in Algorithms and Architectures,* pp. 235-244, 2004.

[48] G.E. Blelloch, P.B. Gibbons, and Y. Matias, "Provably Efficient Scheduling for Languages with Fine-Grained Parallelism," *J. ACM,* vol. 46, no. 2, pp. 281-321, 1999.

[49] G.E. Blelloch, P.B. Gibbons, G.J. Narlikar, and Y. Matias, "Space-Efficient Scheduling of Parallelism with Synchronization Variables," *Proc. Ninth Ann. ACM Symp. Parallel Algorithms and Architectures,* pp. 12-23, 1997.

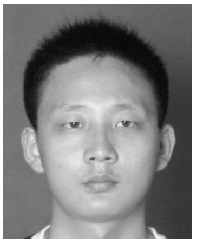


Shanjiang Tang received the bachelor's and master's degrees from Tianjin University (TJU), China, in July 2008 and January 2011, respectively. Currently, he is working toward the PhD degree in the School of Computer Engineering, Nanyang Technological University, Singapore. He has worked at the IBM China Research Lab (CRL) in the area of performance analysis of multicore oriented Java multithreaded program as an intern for four months in 2009. In 2006, he won the "Golden Prize" in the 31th ACM/ICPC Asia Tournament of National College Students. He was awarded the "Talents Science Award" from Tianjin University in 2007. His research interests include parallel algorithms and programming model, parallel program performance analysis, and computational biology.

Ce Yu received the BS and MS degrees in 1992 and 2005 in the Tianjin University (TJU), respectively, and the PhD degree in computer science from the same University in 2009. Currently, he is working as an instructor and director of High Performance Computing Lab (HPCL) of Computer Science and Technology in Tianjin University. He holds three patents, one software copyright, and published more than 17 academic papers in peer-reviewed journals and conferences. His main research interests include parallel computing, astronomy computing, cluster technology, cell BE, multicore, and grid computing.

Jizhou Sun received the master degree in computer science from Tianjin University (TJU), China, in 1982, and the PhD degree in electrical engineering and computer science from Sussex University, United Kingdom, in 1995. Currently, he is working as a professor in computer science and technology, Tianjin University. His main research interests include parallel computing: parallel algorithms and architectures, and high-performance computing; computer graphics: scientific visualization, image synthesis, and image processing; network information security: network intrusion detection, security software. He has got three patents, eight software copyrights, and published more than 120 papers in international journals and conferences. He is a senior member of Chinese Society of Image and Graphics, and a member of Tianjin Software community.

Bu-Sung Lee received the BSc (Hons) and PhD degrees from the Electrical and Electronics Department, Loughborough University of Technology, United Kingdom, in 1982 and 1987, respectively. Currently, he is working as an associate professor in the School of Computer Engineering, Nanyang Technological University, Singapore. He was elected the inaugural president of Singapore Research and Education Networks (SingAREN), 2003-2007, and has been an active member of several national standards organizations, such as Board member of Asia Pacific Advanced Networks (APAN) Ltd. In 2010, he holds a joint appointment as director, Service Platform Lab, HP Labs Singapore. His research interests include computer networks protocols, distributed computing, network management, and Grid/Cloud computing.



Tao Zhang received the bachelor degree in the School of Software, Tianjin University (TJU), China, in July 2011. Currently, he is working toward the PhD degree with the Department of Computer Science, Tsinghua University, China. His research interests include the high-performance climate system model coupler, programming model, and parallel program performance analysis.

Zhen Xu received the bachelor's, master's, and PHD's degrees from the Tianjin University (TJU), China, in 2005, 2007, and 2010, respectively. She has worked at the High Performance Computing Lab, University of Tianjin City in the area of hybrid parallel programming, performance analysis, supported tools and environment. She has participated in several research projects in parallel computing sponsored by the National Natural Science Foundation of China and Tianjin Municipal Science and Technology Committee, as important participant and accomplisher. Her research interests include distributed algorithms and systems, applied parallel computing, and high-performance computing.

Huabei Wu received the bachelor's and master's degree from Zhenzhou University, China, in 1999 and 2004, respectively, and the PHD degree from Tianjin University (TJU), China, in 2009. He has participated and completed several research projects in parallel computing sponsored by the National Natural Science Foundation of China and Tianjin Municipal Science and Technology Committee. His research interests include parallel programming model and design pattern, distributed algorithms and systems, high-performance computing, and computational biology.