# EasyPDP: A Parallel Dynamic Programming Runtime System for Computational Biology and Scientific Computing Applications

<table>
<thead>
<tr>
<th>Journal:</th>
<th>Transactions on Parallel and Distributed Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manuscript ID:</td>
<td>TPDS-2009-11-0545</td>
</tr>
<tr>
<td>Manuscript Type:</td>
<td>Regular</td>
</tr>
<tr>
<td>Keywords:</td>
<td>Dynamic Programming, EasyPDP, DAG Data Driven Model, fault tolerance, DAG pattern, multi-core, block-cycle</td>
</tr>
</tbody>
</table>

Note: The following files were submitted by the author for peer review, but cannot be converted to PDF. You must view these files (e.g. movies) online.

EasyPDP_Latex_source.rar
EasyPDP: A Parallel Dynamic Programming Runtime System for Computational Biology and Scientific Computing Applications

Shanjiang Tang, Jizhou Sun, Ce Yu, Huabei Wu, Zhen Xu

Abstract—Dynamic programming is a popular and efficient design technique in computational biology and scientific computing. Nevertheless, their performance is limited due to the burgeoning volume of scientific and genome data, parallelism is necessary and crucial to make at least certain application instances tractable in practice and to keep the computation time at acceptable levels. This paper built a runtime system named EasyPDP aiming at parallelizing dynamic programming algorithms on multi-core and multiprocessor platforms. Under the concept of software reusing and complexity reduction of parallel programming, a DAG Data Driven Model was proposed, which supports for those applications with strong data interdependence relationship. Based on the model, EasyPDP runtime system was designed and implemented. It automatically handles thread creation, dynamic data task allocation and scheduling, data partitioning, and fault tolerance. Moreover, five frequently used DAG patterns from biological dynamic programming algorithms have been put into the DAG pattern library of EasyPDP, so that the user could choose to use it according to his/her specific application. We evaluate the performance potential and fault tolerance feature of EasyPDP in multi-core 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, multi-core, block-cycle.

1 INTRODUCTION

Dynamic programming (DP), which is a popular algorithm design technique for the solution of many decision and optimization problems by decomposing the problem at hand into a sequence of interrelated decision or optimization steps that are solved one after the others, is widely applied in lots of areas such as computational biology and scientific computing. Typical applications are RNA and protein structure prediction[1], genome sequence alignment[8], context-free grammar recognition[3], string editing, optimal static search tree construction[4], and so on. Many of the computational biology applications are compute intensive and computational biology research is now faced with the burgeoning number of genome data. Indeed, dynamical programming realizes both of optimality and efficiency of the computed results in contrast to other methods for these applications, but the computing cost is still too high when the compute-intensive applications become large. Therefore, the parallelization for the dynamical programming becomes crucial and useful. However, designing efficient, highly 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 too cumbersome for most developers. They require that the programmer manages concurrency explicitly by creating threads and synchronizing them through messages or locks, which is difficult and error-prone especially for the non-experience 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 relevant. Recently, there has been a research trend towards these goals using approaches such as streaming, memory transactions, data-flow based schemes and so on.

This paper presents EasyPDP, a programming API and runtime system based on Directed Acyclic Graph (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 program starts from the user application module, it firstly gets the user selected DAG pattern from the DAG pattern library in DAG pattern module through the identifier of each DAG pattern system provided or user defined, then initializes the DAG pattern including DAG pattern size, data mapping for each DAG node, etc, and starts the DAG runtime system from runtime system module to do parallel computation automatically. Finally, it gets the data result after the runtime parallel computing.

The authors are with the School of Computer Science&Technology, Tianjin University, Tianjin 300072, China. E-mail: {tashj, jzsun, yuce}@tju.edu.cn.
EasyPDP is aimed at shared-memory systems such as multi-core chips and symmetric multiprocessors and uses threads to spawn parallel data tasks. It also uses shared-memory buffers to facilitate communication without excessive data copying. The runtime schedules data tasks dynamically to worker threads in order to achieve load balance. The fault tolerant and recovery mechanism could be used to automatically detect and recover faults during task execution by re-assigning 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 automatically. However, it also allows the programmer to provide the application specific knowledge such as user DAG pattern defined function (if system not provided).

We evaluate EasyPDP on multi-core systems and demonstrate that it leads to scalable performance in this environment. Through fault injection experiments, we show that the EasyPDP fault tolerant and recovery mechanism could detect and handle faults during runtime execution. Finally, we compare the performance of EasyPDP with other static data partitioning methods (column-based partitioning, block-cycle based partitioning [5]) through some regular and irregular DP applications.

The remainder of this paper is organized as follows. Section 2 provides an overview of DAG Data Driven Model. Section 3 introduces the DP algorithm and its classification. Section 4 summarizes the common types of DAG patterns for DP algorithms. Section 5 presents our EasyPDP implementation and Section 6 presents the evaluation results. Section 7 reviews related work. Section 8 concludes the paper and gives out future work.

2 DAG DATA DRIVEN MODEL OVERVIEW

2.1 Programming Model

![A DAG Diagram.](http://mc.manuscriptcentral.com/tpds-cs)

Many applications in computational Biology and scientific fields consisting of a set of data or tasks with the data dependencies and precedence relationships are modeled as a Directed Acyclic Graph (DAG), such applications as Genome alignment, RNA secondary structure prediction, Gene finding, etc. And there are often some applications whose modeled DAG diagrams are almost the same, except for their sizes, such as Figure 1. In consideration of the reuse concept, we could make those common frequently used DAGs as DAG Patterns and establish a DAG pattern library to classify and store them. Moreover, lots of static and dynamic task allocation and scheduling algorithms are based on DAG. For the simplicity of parallel programming and reuse concept, 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 by the arguments.

Inspired by this idea and consideration, we present DAG Data Driven Model, as illustrated in Figure 2. It makes up of three modules: User Application Module, DAG Pattern Module and DAG Runtime System Module. The user application module presents the critical steps that the programmer needs concern. The DAG pattern module establishes a DAG pattern library, in which lots of DAG patterns provided by system or defined by user are stored. To the DAG runtime system module, it implements the static and dynamic task allocation and scheduling algorithms according to the specific applications. The three modules are closely correlated. 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 in terms of the specific applications. It consists of five steps. According to the specific application, the programmer firstly chooses one DAG pattern from the DAG pattern library. However, if the programmer finds no suitable DAG patterns for his or her applications, he or she could define a new DAG pattern and add it into DAG pattern library in advance. Then next is the DAG pattern initialization step. For a selected DAG pattern, the programmer should determine its size (width, height) by the 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 scheduler arguments must be firstly initialized by the programmer, including input data, block size, the number of thread and the data calculation function where the programmer implements the application algorithms and functionality, etc. When to start the DAG runtime system, the application data calculation function will then 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 DAG runtime system module. After computing, the final data result returns and can be got by the programmer.

In the user application module, the programmer just needs to do some simple initialization work and focuses all of his or her attention on algorithm or functionality of the application, not on parallelization. The common
2.3 DAG Pattern Module

A DAG is denoted as $D=(V,E)$, where $V=\{v_1,v_2,...,v_n\}$ is a set of $n$ nodes, $E$ represents the communication relationship and the precedence constraints among data nodes, $e_{pq}=(v_p,v_q) \in E$ represents a data message sent from data node $v_p$ to $v_q$, which suggests that $v_q$ can start computing only after $v_p$ 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 non-determined, set by arguments. With different sizes set, we can make a DAG pattern suit for one kind of applications. For instance, Figure 1 is a regular DAG, each of its nodes only depends on its left and upper nodes if its left and upper nodes exist. It can become a DAG pattern by making its size (width and height) changeable as arguments. For a DAG pattern, each of its DAG nodes maps with a block of data, and the nodes dependency relationships could be obtained from the DAG pattern, while the workload for each DAG node couldn’t be obtained, which is excluded from the DAG pattern.

Regarding the various applications fields, we could summarize those common frequently used DAG patterns for each. In order to organize and manage the DAG patterns well, a DAG pattern library can be established. Each DAG pattern in the DAG pattern library has a unique identifier, and can be selected by its identifier. In the DAG pattern library, there are two types of DAG patterns, one is system provided DAG patterns, another is user defined DAG patterns. The system provided DAG patterns are those common frequently used DAG patterns summarized from the applications, whereas the user defined patterns, which are defined and added to the DAG library by the programmer, are the application-specific DAG patterns, not the common frequent 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 token for DAG operations, data tasks allocation and fault-tolerance control. The DAG operations include DAG parsing and updating. The DAG parsing operation aims for discovering current new computable data nodes, it travels every DAG node and gets all those nodes whose in-degree is zero. By parsing the DAG, the master allocates those new computable node tasks to worker by putting the tasks into workers pool buffer. The DAG updating operation updates the DAG by removing those completed DAG nodes.

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

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

For the static workers pool, each worker in the workers pool has its own buffer. The tasks and the workers are bound together according to a certain static data allocation method. Once the master put a task into the pool buffer, the static workers pool will calculate which worker it belongs to and put the task to that worker’s own buffer.

To the dynamic workers pool, the tasks and the 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. In contrast, if there is at least a idle worker, the pool buffer must be empty. Compared with static workers pool, the load balance for the dynamic workers pool is better.

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

3 THE DP ALGORITHM AND CLASSIFICATION

DP is an important algorithm design technique in computational biology and scientific computing. It solves the problem by decomposing the problem into a set of interdependent subproblems. After the subproblems was solved, it uses the result to solve larger subproblems until the entire problem is solved. In general, the solution to a DP problem is expressed as a minimum(or maximum) of possible alternative solutions. Each of these alternative solutions is constructed by composing one or more subproblems. If \( r \) represents the cost of a solution composed of subproblems \( x_1, x_2, \ldots, x_l \), then \( r \) can be written as \( r = g(f(x_1), f(x_2), \ldots, f(x_l)) \), where the function \( g \) is called the composition function, and its nature depends on the problem. If the optimal solution to each problem is determined by composing optimal solutions to the subproblems and selecting the minimum(or maximum), the formulation is said to be a DP formulation[6].

The DP algorithms can be classified in terms of the matrix size and the dependency relationship of each cell on the matrix[18]: A DP algorithm is called a 1D/0D algorithm if its matrix size is \( t \) and each matrix cell depends on \( O(n^t) \) other cells. If a DP algorithm is a 1D/0D algorithm, it takes time \( O(n^{t+e}) \) provided that the computation of each term takes constant time. For example, three DP algorithms are defined as follows:

**Algorithm 1. (2D/0D):**

Given \( F[i,0] \) and \( F[0,j] \) for \( 0 \leq i,j \leq n \),

\[
F[i, j] = \min\{F[i-1, j] + x_i, F[i, j-1] + y_j, F[i-1, j-1] + z_{ij}\},
\]

where \( x_i, y_j \) and \( z_{ij} \) are computed in constant time.

**Algorithm 2. (2D/1D):**

Given \( f(i,j) \) for \( 1 \leq i \neq j \leq n \), \( F[i,j] \)
=0 for $1 \leq i \leq n$.

$$F[i, j] = c(i, j) + \min\{F[i, k - 1] + F[k, j]\},$$

where $1 < k \leq j$ and $c(i, j)$ is computed in constant time.

**Algorithm 3.** (2D/2D): Given $c(i, j)$ for $1 \leq i, j \leq 2n$, $F[i,0]$ and $F[0,j]$ for $1 \leq i, j \leq n$,

$$F[i, j] = \min\{F[i', j'] + c(i' + j', i + j)\},$$

where $0 \leq i' < i, 0 \leq j' < j$ and $c(i, j)$ is computed in constant time.

For the DP algorithm, it usually exhibits the characteristics of the wavefront computation, that is, each element computes a value that depends on the computation of a set of previous elements. The wavefront moves along anti-diagonals and the wavefront shift direction depends on the dependency relationship. For a wavefront computation, if each matrix cell is computed from the same number of other matrix cells, then the wavefront computation is regular one, otherwise, we call it irregular.

There are many DP algorithms in Computational Biology and Scientific Computing. Some popular DP algorithms for computational biology are shown in Table 1. DP is applied for sequence comparison with numerous variations, determining the intron/exon structure of genes and to assemble DNA sequences from overlapping fragments. As shown in Figure 3, it is the computation dependency relationship and distribution of load computation density along computation shift direction for some popular algorithms. By using increasingly blacking shades to indicate computational load density changes, we could notice that the 2D/0D DP algorithms are regular ones, while the 2D/ID ($i \geq 1$) DP algorithms are irregular. In this paper, we concentrate on the parallelization of DP algorithms of the type 2D/ID ($i \geq 0$), which are important DP algorithms for many applications.

### 4 THE DAG PATTERNS FOR DP ALGORITHMS

Recall in the DAG Data Driven Model that we summarize the DAG patterns according to the application fields and establish the DAG pattern library to store and manage them. Here for the DP applications, we summarize five common frequently used DAG patterns derived from lots of DP algorithms as shown in Figure 5. Each DAG pattern is given an unique identifier name according to its data dependency relationship. The Left_Upper_DAG pattern, Left_Upper_DAG pattern and HalfUpperRightMost_Left_Lower_DAG pattern are directly derived from (a),(b),(c) in Figure 3 respectively. The Left_Upper_DAG and Upper_Left_Upper_DAG are from (a),(b) as shown in Figure 4.

Although the DAG patterns are often summarized from the regular DP algorithms, they can sometimes be used in many irregular DP algorithms. Token the Left_Upper_DAG pattern for example, apart from the regular DP algorithms as (b) in Figure 3, it also well suits for the irregular DP algorithms as (d) in Figure 3, (c) and (d) in Figure 4. With the same DAG pattern, the difference between regular DP algorithm and irregular
Fig. 5. Some DAG Patterns for DP Algorithms.

DP algorithm lies in the workload for each DAG node that represents a block of data, which is excluded from DAG pattern.

In deed, it is equivalent between the Left_Upper_DAG pattern and Left_LeftUpper_Upper_DAG pattern. The Left_LeftUpper_Upper_DAG pattern and Upper_LeftUpper_DAG pattern can be derived from Left_LeftUpper_Upper_DAG pattern by erasing all its upper or left dependencies, that’s mean, we could use the Left_LeftUpper_Upper_DAG pattern instead of Left_LeftUpper_Upper_DAG pattern and Upper_LeftUpper_DAG pattern in some cases, except that it lowers the parallelization degree.

5 THE EASYPDP SYSTEM

EasyPDP implements 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.

5.1 The EasyPDP Functions

The current EasyPDP implementation provides four types of functions for C and C++, which are user programming API, DAG operation function, workers pool function and fault tolerance function, respectively. However, similar functions can be defined for languages like Java or C#. The functions are summarized in Table 2.

The user programming API, which is visible to application programmers, includes two sets of functions. The first set is provided by EasyPDP and is used by the programmer’s application code to initialize the system (1 required function). The second set includes the functions that the programmer defines (1 required and 2 optional functions). Apart from the process function that does application algorithm computation, the user provides functions to initialize the DAG pattern for the user defined DAG pattern and to map the DAG nodes with the application data blocks. For the EasyPDP API, it doesn’t rely on any specific compiler options and doesn’t require a parallelizing compiler. However, it assumes that its functions can freely use stack-allocated and heap-allocated structures for private data. 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 can not check these assumptions statically for arbitrary programs. Although there are stringent checks within the system to ensure valid data are communicated between user and 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 the DAG nodes with data blocks. The DAG pattern handle function can parse the DAG for finding current computable DAG nodes and update the DAG pattern by deleting the given completed node from the current DAG pattern.

To the workers pool function, the basic thread pool operations are provided. It includes the pool initialization function that initializes the pool queue, queue lock, and creates threads, pool destroy function that destroy threads and frees the memory space, pool queue tasks adding function, and the runtime thread routine function that do task computation.

The EasyPDP provides support for fault tolerance. It
TABLE 2
The functions in the EasyPDP. \( R \) and \( O \) identify required and optional functions respectively.

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

detects faults through timeouts. The critical fault tolerance function includes the DAG node adding functions and DAG node removing functions for the timeoutQueue, timeout checking function that detects the timeout DAG nodes from timeoutQueue. If the timeout DAG nodes detected, it removes the timeout nodes from the timeoutQueue, cleans up the timeout worker thread and redistributes the timeout DAG nodes.

5.2 The EasyPDP Runtime

In order to obtain a good load balance both for regular and irregular DP algorithms, the EasyPDP runtime adopts the dynamic workers pool, which uses the dynamic allocation and scheduling algorithm. Moreover, the EasyPDP runtime was developed on top of Pthreads[14], but can be easily ported to other shared-memory thread packages.

5.2.1 Basic Operation and Control Flow

Figure 6 shows the basic data flow for the runtime system. The runtime is controlled by the scheduler(master) and initialized by the user program. The programmer provides the scheduler with all the required data and function pointers through the scheduler_args_t structure, which is the only data structure used to communicate basic function information and buffer allocation between user program and runtime. The fields of scheduler_args_t are summarized in Table 3. The basic fields provide pointers to DP data buffers and to the user-provided func-

![Fig. 6. The basic data flow for the EasyPDP runtime.](http://mc.manuscriptcentral.com/tpds-cs)
TABLE 3
The fields of scheduler_args_t data structure.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dp_data</td>
<td>The matrix DP data. All the data computations are based on it.</td>
</tr>
<tr>
<td>data_row</td>
<td>The number of rows for matrix dp_data.</td>
</tr>
<tr>
<td>data_col</td>
<td>The number of columns for matrix dp_data.</td>
</tr>
<tr>
<td>DAG_Pattern_id</td>
<td>The identity of user selected DAG pattern.</td>
</tr>
<tr>
<td>process</td>
<td>Pointer to DP computation function.</td>
</tr>
<tr>
<td>DAG_Pattern_init</td>
<td>Pointer to the user defined DAG pattern initialization function.</td>
</tr>
<tr>
<td>DAG_pattern_node_data_mapping</td>
<td>Pointer to the user Map function.</td>
</tr>
<tr>
<td>block_row</td>
<td>The number of rows for data block.</td>
</tr>
<tr>
<td>block_col</td>
<td>The number of columns for data block.</td>
</tr>
<tr>
<td>max_thread_num</td>
<td>Maximum number of threads.</td>
</tr>
<tr>
<td>timeout</td>
<td>The value of timeout. If timeout≤0, the fault recovery mechanism doesn't work, otherwise, it works.</td>
</tr>
</tbody>
</table>

Performance Tuning Fields

The performance tuning fields give out the key arguments that greatly affects 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 workers pool. After that, the master scheduler calls the DAG_pattern_handle function to discover the current computable DAG nodes whose in-degrees are zero and DAG_pattern_node_data_mapping function to map the DAG nodes with data blocks before sending them into the pool buffer.

Once the pool buffer is not empty and there are idle workers, the workers pool will distribute the data tasks from pool buffer to the idle workers. The Process is called by the worker threads to do DP algorithm computation. When a worker thread completes a DAG node data task, it sends the DAG node id into DAGPatternNodeFinishedStack for notifying the master. The master checks the DAGPatternNodeFinishedStack in a small regular time for the completed DAG nodes. Once getting a DAG node, the master will call DAG_pattern_handle to update DAG and parse the DAG to find current new computable DAG nodes. The whole process continues until all the DAG node tasks are completed. Finally, the output results return when all the tasks are finished.

5.2.2 Fault Tolerance

Recall in section 2.4 that since a computing DAG node failed often causes all other nodes that dependent on it directly and indirectly always incomputable, the whole computations will eventually pause at a place forever without fault recovery. Therefore, it is critical and necessary to have the fault tolerance mechanism to detect and recovery faults.

EasyPDP detects faults through timeouts. 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 known error detection techniques[15][16]. Two faults that cause timeout are considered here. One fault is the computing worker thread died because of something occurred. The other case is that the computing worker thread goes into deadloop or deadlock for some reasons.

Figure 7 presents the overall flow of EasyPDP fault tolerance mechanism. When the user program calls the EasyPDP_scheduler(), the following sequence of actions occurs (the numbered labels in Figure 7 correspond to the numbers in the list below):

1) The master distributes the 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 was
put into the timeoutQueue.

2) The workers pool gets the 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 DAG node
id in the worker_DAGPatternNode_register before do
DP algorithm computation.

4) Once the worker completes a DAG node task, it
will send the DAG node id to the DAGPatternN
dodeFinishedStack 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 timeoutQueue
is strictly increasing from right (queue front) to
left(queue rear). Thereby the master needn’t check
all nodes in timeoutQueue every time; in contrast,
it just need to check nodes from right to left
until a non-timeout node found. If a timeout
DAG node discovered, the master firstly finds out
the corresponding timeout thread id through the
worker_DAGPatternNode_register, then makes the
workers pool kills that thread and instead creates
a new thread, and goes 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 the DAG node from the time-
outQueue. Then it goes to step 1.

The current EasyPDP code 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
program correctness. We can use known techniques such
as redundant execution or checkpointing to address this
shortcoming.

5.2.3 Buffer Operation and Management

Four types of temporary buffers as shown in Figure 7 are
necessary to store data and support for 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
master and workers pool. Each time the master sends the
computable data tasks into the pool queue buffer and the
workers pool fetches from it. The queue lock is used to
make sure only one access exists every time.

In order to notify the master updates the DAG in real
time, the DAGPatternNodeFinishedStack buffer is used.
Every time the worker finishes the DAG node task, it
writes the DAG node id into the DAGPatternNodeFin-
ishedStack buffer and the master could know it at once.

The worker_DAGPatternNode_register buffer and time-
outQueue buffer are two critical parts of fault tolerance
mechanism. For the timeoutQueue buffer, it is only vis-
ible to master, and has a time_start field that records
the distributed time for each DAG node. The master
frequently 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 work-
ers pool to kill the dirty worker in case of dead-loop and
deadlock worker thread. Since that the EasyPDP adopts
dynamic workers pool, the master can not know which
worker thread did the timeout DAG node task without
worker_DAGPatternNode_register buffer. Every time when
a worker gets a DAG node task, it will register its thread id in worker_DAGPatternNode_register buffer for
that DAG node.

5.2.4 Refinements

Table 3 presents the performance tunable arguments that
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 then affects the parallelization degree
indirectly. For the irregular DP algorithms as (d) in
Figure 3, since computation workload for each matrix
cell is unequal(irregular), thereby the bigger the data
block size set, the more irregular for the workload of
each DAG node is.

Number of Threads: In systems with multiple cores,
since DP applications are data-intensive, in order to
typically maximizes the system throughput even if an
individual task takes longer, it is better to set the value of
argument max_thread_num as the number of available
cores.

Timeout: If a failure occurs during runtime execution,
the value of timeout is to be a critical criterion for fault
tolerance mechanism to detect the fault in real time. Too
big value of timeout will make the fault tolerance mech-
anism obtuse to discover faults, in contrast, too small
value setting for timeout will make the fault tolerance
mechanism wrongly assumes a being computed task
failed and recomputing that task, which also directly
influences the performance. Therefore, user’s proper
value setting for timeout is important according to the
specific characteristics for various DP applications. To
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 for the most irreg-
ular DAG node task. To address this shortcoming, we
present a self-adjustment mechanism for timeout. That
is, according to the successful completed DAG node task
from DAGPatternNodeFinishedStack, the total execution
time for that DAG node task could be calculated by subtracting \( \text{time}_{\text{start}} \) for that DAG node in \( \text{timeoutQueue} \) from the current time. If the total execution time is less than \( \text{timeout} \) but greater than eighty percent of \( \text{timeout} \), it suggests that the \( \text{timeout} \) is a bit small at present and then our self-adjustment mechanism will double the current \( \text{timeout} \) value.

6 PERFORMANCE EVALUATION

This section presents the performance evaluation results for EasyPDP running on Dell PowerEdge 2950 Dual Quad Core server with Xeon E5310 processors. Four popular DP algorithms token from Table 1 for Computational Biology are evaluated, for which the Smith-Waterman algorithm with linear and affine gap penalty (SWLAG), and Syntenic alignment(SA) algorithm are regular DP algorithms, while the Smith-Waterman algorithm with general gap penalty (SWGG), and Viterbi Algorithm(VA) are irregular DP algorithms. The computation dependency relationships for SWLAG, SA, VA, SWGG are corresponding to (a),(b),(d) of Figure 3, and (c) of Figure 4 respectively.

6.1 Dependency to Data Size

Figure 8 presents the run time results with EasyPDP both for regular and irregular DP algorithms as we vary the input data size when the block size is 50, and the number of worker threads is 8. For regular DP algorithms (a) and (b), it is obvious that the run time with different data sizes almost increases with liner speed, since for the regular DP algorithms, the workloads for each matrix cell are equal and determined, which says that the total computational volume is proportional to the data size. Reversely, (c) and (d) show that the run time for the irregular DP algorithms is larger and increases greatly with increasing big time interval compared to regular DP algorithms. To the irregular DP algorithm, the workloads for each its matrix cell are undetermined and often increases greatly along the diagonal, thereby the total computational volume increases sharply as we enlarge the input data size. The more irregular the DP algorithm is, the more sharper the run time diagram is. Moreover, it also suggests that the parallelization for irregular DP algorithm is critical and necessary.

6.2 Dependency to Block Size

The argument block size is a critical performance metric in DP algorithm parallelization. Its value setting is a
tradeoff between the load balancing and communication time. Too big or small value of block size will both make the program performance bad. And often the value of the most suitable block size for regular DP algorithm is much more bigger than that of the irregular DP algorithm, which is mainly due to workload of the data blocks. In general, the real computation workload for an irregular data block is often many times or more than that for a regular data block with the same block size. As illustrated in Figure 9, they are run time results for varied block sizes when we set the application data size to be 10000 and the number of worker threads to be 8. We can notice that they both have a most suitable block size, and for regular SWLAG DP algorithm, its most suitable block size is between 400 and 500, whereas the most suitable block size for irregular SWGG DP algorithm is between 8 and 20.

6.3 Dependency to Number of Threads

![Fig. 10. Speedup with EasyPDP for DP algorithms as we scale the number of worker threads.](image)

Figure 10 presents the speedup results with EasyPDP as we scale the number of worker threads for four popular DP algorithms in dual quad cores system. It is obvious that all the speedup lines are much close to the ideal speedup line except their last points that the number of worker threads is 8. As we know that when the number of threads equals with the number of system processor cores, the speedup is often to be the best. Because our EasyPDP was implemented as master-slave model, and the number of application threads in fact should be 9 when we set the number of worker threads to be 8, which just exceeds the number of processor cores.

6.4 Comparison to BCW

In contrast with dynamic runtime system EasyPDP, the Column based Wavefront(CW)[6] and Block-Cycle based Wavefront(BCW)[5] are those static data partitioning methods to parallelize DP algorithm. The CW algorithm can be viewed as a special BCW algorithm when setting the argument $\text{block}\_\text{col}$ as a result of that $\text{data}\_\text{col}$ divided by $\text{thread}\_\text{num}$ for BCW, thereby it only needs to compare the performance with BCW for EasyPDP. To the BCW algorithm implementation, we also adopt the DAG Data Driven Model, whereas the static workers pool was considered here. In order to compare the EasyPDP with BCW thoroughly and completely, we define the metric $\text{BCW/EasyPDP rate}$ as BCW divided by EasyPDP with their run times in the same condition, and do comparisons from three perspectives: data size, block size and the number of threads respectively.

Figure 11 presents the compared rate results between EasyPDP and BCW. The baseline $1.00 \text{ LINE}$ is given, so it says EasyPDP is better when the rate points is above the baseline, otherwise it is BCW. A conclusion drawn from the diagram is that EasyPDP is more efficient than BCW both for regular and irregular DP algorithms, as it can be observed that the experimental rate lines both for regular and irregular DP algorithms are all above the
Fig. 11. The compared BCW/EasyPDP rate results from three different aspects, where (a), (b), (c) are from data size aspect, block size aspect and the number of threads aspect respectively. In Figures, the 1.00 LINE is baseline. We say EasyPDP is more efficient when the experimental rate point is above it, otherwise the BCW it is.

1.00 LINE. The primary reason is, for the DP algorithms, its data dependency is strictly strong, which means that only a few DAG block nodes are computable at the beginning, and more and more new computable nodes are spawn during the running process. Compared with EasyPDP, the BCW, which uses the static data allocation and scheduling method, has a fatal situation case during the runtime that there are some computable DAG nodes as well as some idle threads at the same time, whereas the case will never occur for EasyPDP, a dynamic data allocation and scheduling runtime system. Moreover, for the BCW, the case often occurs many times and even more, especially for irregular DP algorithms.

6.5 Fault Recovery

Fig. 12. Normalized execution time in the presence of different number of faults for EasyPDP.

Figure 12 presents the results for fault injection experiments for EasyPDP. The graphs represent normalized execution time. To an error, it occurs at arbitrary point within the program execution, which may affect the execution and buffers for the tasks, but does not corrupt the runtime or its data structures. The EasyPDP runtime detects faults through timeouts and recovers to complete the execution correctly. Fault recovery is completely transparent to the programmer. The initial value for the argument timeout is set by user and dynamically changed at runtime. Its value can greatly affect the system performance (See discussion in Section 5.2.4). Here the timeout value was initialized as 10 seconds. Note that the fault impacts for the regular DP algorithms(SWLAG and SA) are much higher than the irregular DP algorithms(SWGG and VA). Since that with the same number of data size, the execution times for the irregular DP algorithms are far greater than the regular DP algorithms. Here the timeout value of 10 is suitable for the irregular DP algorithms, while it is relatively too large compared with the execution times of the regular DP algorithms and impacts the performance greatly.

7 Related Work

As an efficient algorithm design technique, DP has been widely applied in computational biology and scientific computing. With the burgeoning number of science and genome data, the DP computation cost is still too high, and it is necessary and meaningful to parallelize it. Lots of work has been done on exploiting the parallelization of DP algorithms. Edmonds et al.[17] and Galil[18] described several parallel algorithms on general shared memory multiprocessor systems. Bradford[19] presented several algorithms that solve optimal matrix chain multiplication parenthesization using the CREW PRAM model. Tan[20][21] focused on a specific type of nonserial polyadic DP with triangular matrix, for which he introduced some optimization algorithms and theoretical models on multi-core architectures. All these works above are aimed at how to reduce the complexity of the arithmetic cost on varied theoretical parallel models, while the parallel implementation complexities for their approaches are all out of consideration. For the distributed memory multiprocessor system, Almeda et al.[22] presented a parallel implementation with tiling on
a ring processors, whereas this parallel tiling algorithm cannot achieve load balance. Zhou[23] proposed a parallel
dataflow model on conventional systems. 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 irregular DP since the arithmetic cost on
each processor is not the same because of the irregular
data dependence. In [5][25][26][27], the authors just presented a static parallel partitioned strategy named block-cycle based wavefront method firstly and did some work including giving some algorithms, making a
pattern-based prototype system afterward. Despite some optimization work they did yet, their static partitioning methods still cannot achieve a better load balance especially for irregular DP applications compared with the dynamic partitioning as the EasyPDP runtime adopted. 
Recall what we have explained in section 5.2.2 that the fault tolerance is critical and necessary for parallel DP
algorithms, but there are no fault tolerance mechanisms considered and supported by all previous works and
systems except our EasyPDP here. 

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[28] 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 multi-core system[29].

8 Conclusion and Future Work

In the paper, a parallel dynamic runtime system named EasyPDP based on our DAG Data Driven Model, was
proposed and implemented for DP algorithms parallelization in shared-memory systems. With EasyPDP, the programmer just needs to concern about the specific DP formulas, provide some runtime needed
application-related arguments, and leave parallelization and scheduling to the EasyPDP runtime system. EasyPDP automatically partitions the DP matrix data and allocates the block data to those current idle threads dynamically during parallel execution. It can also recover from runtime faults through its timeout fault
tolerance mechanism. We compared the performance of EasyPDP to that of Block-Cycle Wavefront(BCW). And experiment results show that EasyPDP is superior to BCW for DP algorithms parallelization.

Our future work includes migrating EasyPDP from
shared-memory system to distributed multi-processors
system with MPI implementation instead, exploring more common used DAG patterns for DP algorithms,
and extending current EasyPDP so that it can be applied to other applications, etc.

Acknowledgement

The work is sponsored by the National Natural Science Foundation of China(10778623), Natural Science Foundation of Tianjin, China (08JCJZDJC19700), and the Key Technologies R & D Program of Tianjin, China (09ZCKFGX00400).

References


Shanjing Tang received the bachelor degree from the Tianjin University(TJU), China, in 2008. He is a master candidate in the School of Computer Science &Technology, Tianjin University (TJU), China. 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. He has worked at the IBM China Research Lab(CRL) in the area of performance analysis of multi-core oriented Java multi-threaded program as an intern for four months in 2009. His research interests include parallel algorithms and programming model, parallel program performance analysis, and computational biology.

Huabei Wu received the PHD degree from Tianjin University(TJU), China, in 1999 and 2004, respectively. He has participated in 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.

Jizhou Sun received his master degree in computer science from Tianjin University(TJU), China, in 2002, and the PhD degree in electrical engineering and computer science from Sussex University, United Kingdom, in 1995. He is currently a professor and dean of computer science&technology and national pilot school of computer software in Tianjin University. He has founded and directed the High Performance Computing Lab(HPCL), University of Tianjin. 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 conducted and completed numerous research projects in parallel computing architectures and computer graphics sponsored by the National Science Foundation of China. He has got 3 patents, 8 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, a member of Tianjin Software community. He is a fellow of the IEEE and the Association for Computing Machinery (ACM). His homepage is http://cs.tju.edu.cn/faculty/jzsun/, and the HPCL homepage is http://cs.tju.edu.cn/orgs/hpclab/.

Zhen Xu received the bachelor’s and master’s degree from the Tianjin University(TJU), China, in 2005 and 2007, respectively. She is a PhD candidate in the School of Computer Science & Technology, Tianjin University(TJU), China. 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 are distributed algorithms and systems, applied parallel computing, high performance computing.

Ce Yu received his Ph.D degree of Computer Science from Tianjin University(TJU) in 2009, MS, BS degree in 2005 and 1992 in the same University respectively. He is currently an instructor and director of High Performance Computing Lab(HPCL) of Computer Science&Technology in Tianjin University. He holds 3 patents, 1 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, Grid computing. His homepage is http://cs.tju.edu.cn/faculty/yuce/.