Chapter

PARALLEL DYNAMIC PROGRAMMING FOR LARGE-SCALE DATA APPLICATIONS

Chen Wang, Shanjiang Tang*[*] *and Ce Yu
School of Computer Science and Technology,
Tianjin University, Tianjin, China

Abstract

Dynamic programming is an important technique widely used in many scientific applications. In this chapter, we first motivate the importance of parallelizing DP algorithms for large-scale data applications. Next, we have an extensive review for accelerating DP algorithms on specific applications and on different computing platforms. As a concrete example, we present our research work on implementing generous parallel DP frameworks both for shared memory and distributed memory. We also discuss some related issues such as fault tolerance, stragglers and load balance. Finally, we give some open problems in this area, and a summary for this chapter.

Keywords: Parallel computing, Dynamic programming, Programming model

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 decisions 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 [2], context-free grammar recognition [3], string editing, optimal static search tree construction [4], and so on.

In contrast to other methods like the recursive method, dynamic programming can achieve both optimality and efficiency for application results. Nevertheless, its computing cost is still too high especially when the application data is large. The parallelization of

[*]E-mail address: tashj@tju.edu.cn (Corresponding author).

dynamic programming is a promising approach to alleviate it [5–7]. However, by virtue of the strong data dependency of the dynamic programming, it's often difficult and error-prone for programmers to write a correct and efficient parallel DP program. Moreover, designing highly efficient parallel programs that effectively exploit parallel computing 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 data communication, are often cumbersome for most developers. They require the programmer to manage concurrency explicitly by creating and synchronizing multi-processes/multi-threads through messages or locks, which is difficult and error-prone especially for the inexperienced programmer.

To simplify parallel programming, many recent studies, such as MapReduce [8] and Pregel [9], have shown that the model-based approach is a practical and effective solution. It consists of two key parts [10]: an abstract *programming model* that allows users to describe applications and specify concurrency at the high level, and an efficient *runtime system* which handles low-level threads/process creating, mapping, resource management, and fault tolerance issues automatically regardless of the system characteristics or scale.

In this chapter, we consider the parallelism of DP applications on two classical parallel computing systems, i.e., a single-node multi-core platform and a cluster system consisting of multiple machines. For the multi-core platform, we introduce a DAG data driven programming model and a multi-core runtime system called EasyPDP. In contrast, for the distributed cluster system, we introduce DPX10, which is a distributed DP parallel computing framework based on X10 language [11] and APGAS (Asynchronous Partitioned Global Address Space) model [12].

The chapter is organized as follows. We first give the background information about parallel programming models and dynamic programming algorithms in Section 2. In Section 3, we introduce DAG data driven model, which is a programming model that abstracts a dynamic programming application as a directed acyclic graph. Section 4 presents EasyPDP, a multi-core parallel programming system for large-scale DP applications, followed by a distributed parallel DP system called DPX10 in Section 5. For the completeness of discussion, Section 6 reviews existing work on accelerating dynamic programming algorithms. A discussion of open problems is provided in Section 7. Finally, we conclude the chapter in Section 8.

2. Overview of Parallel Programming Models and DP Algorithms

In this section, we review parallel programming models and give the definition and classification of DP algorithms.

2.1. Parallel Programming Models

In traditional parallel computing, there are two classic programming models, i.e., distributed memory model (e.g., MPI), and shared memory model (e.g., Pthread, OpenMP).

Distributed memory refers to a multiprocessor computer system in which no processor has direct access to all the system's memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors. In contrast, a shared memory multiprocessor offers a single memory space used by all processors. Processors do not have to be aware of where data reside.

There's another model called distributed shared memory. In distributed shared memory model, each node of a cluster has access to a large shared memory in addition to each node's limited non-shared private memory.

Each model has its own merit. Specifically, shared memory model is good for communication-intensive computing, but its scalability is a big problem. Distributed memory model has good scalability but the cost of data communication is high.

Many studies [12–16] have been presented to take advantage of both models and at the same time, make it easier for developers to write efficient parallel and distributed applications. Partitioned global address space (PGAS) [16] is a parallel programming model that attempts to combine the advantages of a SPMD programming style for distributed memory systems with the data referencing semantics of shared memory systems. It assumes a global memory address space that is logically partitioned, and a portion of it is local to each process or thread. The novelty of PGAS is that the parts of the shared memory space may have an affinity for a particular process, thereby exploiting locality of reference.

2.2. Dynamic Programming Algorithms

Dynamic programming is a powerful technique widely used for many scientific applications. DP problem is solved by decomposing the problem into a set of interdependent subproblems, and using their results to solve larger subproblems until the entire problem is solved [5]. There are two key attributes that a problem must have in order to make dynamic programming applicable: optimal substructure and overlapping sub-problems. Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. Such optimal substructures are usually described by means of recursion. Overlapping sub-problems means that the space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems.

DP problems can be classified in terms of the matrix dimension and the dependency relationship of each cell on the matrix [6]: A DP algorithm is called a tD/eD algorithm if its matrix dimension is t and each matrix cell depends on $O(n^e)$ other cells. 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 3.1 ($2D/0D$): Given $D[i,0]$ and $D[0,j]$ for $1 \le i,j \le n$,

$$D[i,j] = \min\{D[i-1,j]+x_i, D[i,j-1]+y_i\}$$

where x_i, y_i are computed in constant time.

Algorithm 3.2 ($2D/1D$): Given $w(i,j)$ for $1 \le i,j \le n; D[i,i]=0$ for $1 \le i$,

$$D[i,j] = w(i,j) + \min_{i \le k \le j}\{D[i,k-1]+D[k,j]\}$$

Algorithm 3.3 ($2D/2D$): Given $w(i,j)$ for $1 \le i,j \le 2n; D[i,0]$ and $D[0,j]$ for $0 \le i,j \le n$,

$$D[i,j] = \min_{0 \le j' \le j, 0 \le i' \le i}\{D[i',j'] + w(i'+j', i+j)\}$$

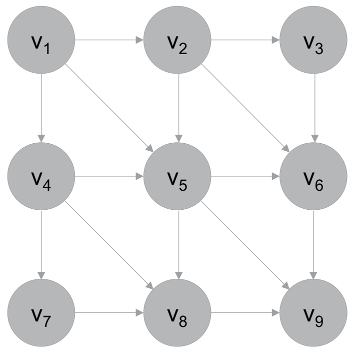


Figure 1. An example DAG.

3. Parallel Programming Model for DP Applications

The computation of a DP algorithm is a process of filling the DP matrix. The dependency between cells in the matrix can be different in different DP algorithms. So we use the directed acyclic graph (DAG) to represent the DP matrix and the dependency relationship between cells.

3.1. DAG Data Driven Model

A directed Acyclic graph (DAG) is denoted as $D = \{V, E\}$, where $V = \{V_1, V_2, \cdots, V_n\}$ is a set of n vertices and E is a set of directional edges, as shown in Figure 1. In the DAG, each vertex represents a cell on the matrix as discussed above. The edge describes the dependency between cells and determines the execution order of them. For example, $e_{pq} = (v_p, v_q) \in E$ suggests that v_q can start computing only when v_p completes.

Moreover, there are many DP applications whose modeled DAG diagrams are almost the same, except for their sizes. For the reuse purpose, we could make those frequently used DAGs as DAG Patterns and establish a DAG pattern library to classify and store them.

The computation of a DAG-represented DP application consists of instantiating a DAG pattern, where the vertices are distributed and initialized, followed by an execution phase where all vertices are scheduled and computed until the algorithm terminates, and the final stage for users to process the result.

In the execution phase, the vertices with in-degree of zero compute in parallel, each executing the same user-defined **compute** method that expresses the logic of a given algorithm. When a vertex completes, the in-degree of each of its children decreases by one. The whole execution continues until all vertices completed.

We use the longest common substring(LCS) problem to illustrate these concepts. Given two strings S and T, the LCS problem is to find their longest common substring. Its DP formulation is:

$$F[i,j] = \begin{cases} F[i-1,j-1]+1 & x_i = y_j \\ \max\{F[i-1,j], F[i,j-1]\} & x_i \neq y_j \end{cases}$$

where $F[i,j]$ records the length of LCS of $S_{0\ldots i}$ and $T_{0\ldots j}$. So $F[m,n]$ would be the length of LCS of S and T, where m and n are the length of S and T.

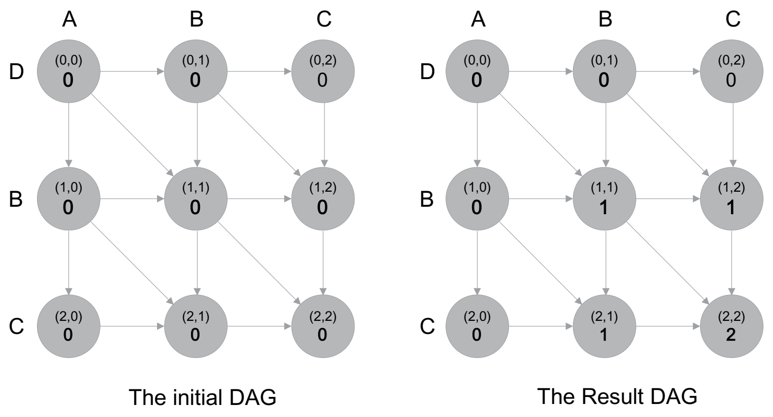


Figure 2. An example DAG of longest common substring problem.

Figure 2 shows a simple example: finding the LCS of string *ABC* and string *DBC*. At the initial stage, the DAG is constructed and nine vertices are initialized with zero. The computation starts from the zero in-degree vertex $(0,0)$ and terminates when all vertices are completed. The sequence of computation may be different since the vertices without dependency relationship execute in parallel. For example, vertex $(0,2)$ can be computed before vertex $(0,1)$. Finally, the result can be processed by using backtracking method to get the substring *BC*.

3.2. The DAG Patterns for DP Algorithms

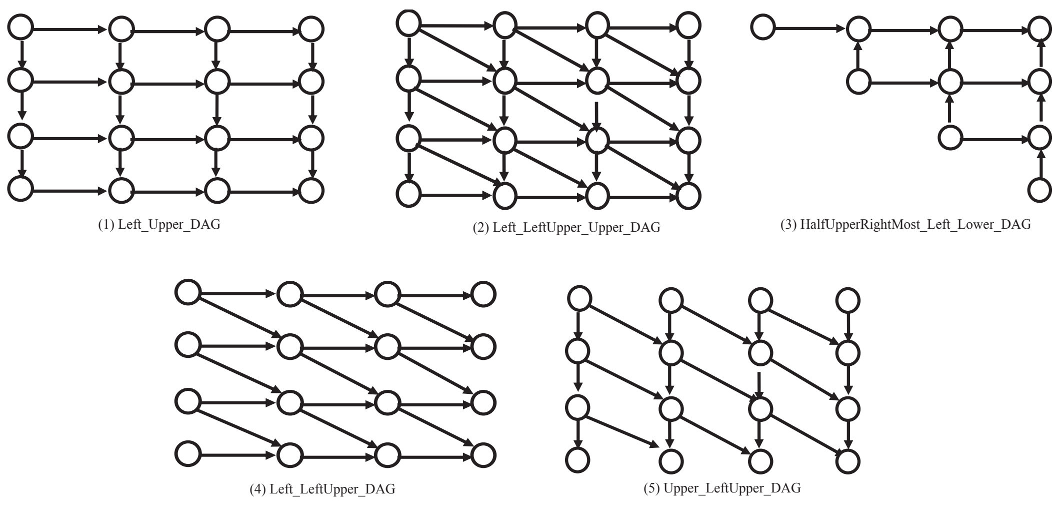

(1) Left_Upper_DAG
(2) Left_LeftUpper_Upper_DAG
(3) HalfUpperRightMost_Left_Lower_DAG
(4) Left_LeftUpper_DAG
(5) Upper_LeftUpper_DAG

Figure 3. Some DAG Patterns for DP Algorithms.

There are often some applications whose DAG diagrams are almost the same except for their sizes. For simplicity and reuse purposes, we could make those frequently used DAGs as DAG patterns and establish a DAG pattern library to classify and store them.

Here we present five common frequently used DAG patterns derived from the DP algorithms shown in Figure 3. Each DAG pattern is given a unique identifier according to its data dependency relationship.

Although DAG patterns are often summarized from the regular DP algorithms, they can be used in many irregular ones. 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. EasyPDP: A Multi-Core Runtime System for DP Applications

In this section, we introduce EasyPDP [5], a multi-core programming system for DP applications. 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.

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 can not 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

Table 1. The functions in the EasyPDP. *R* and *O* identify required and optional functions, respectively.

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

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 System

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, but can be easily ported to other shared-memory thread libraries.

4.2.1. Basic Operation and Control Flow

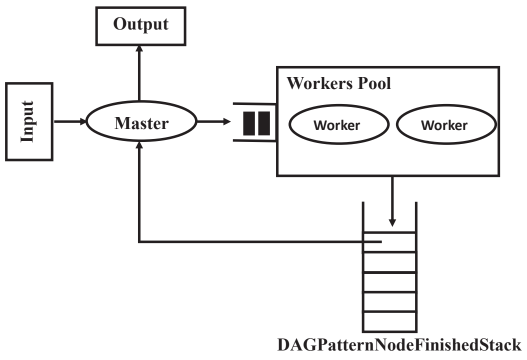


Figure 4. The basic data flow for the EasyPDP runtime.

Figure 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 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 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,

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$\leq$0, the fault recovery mechanism doesn't work. Otherwise, it works.

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.

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.

4.2.2. Fault Tolerance

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.

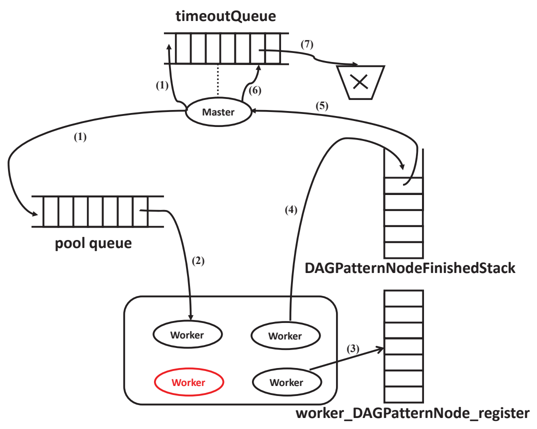


Figure 5. The overall flow of EasyPDP fault tolerance mechanism.

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 [17] [18]. 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.

Figure 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 Figure 5 correspond to the numbers in the list below):

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 *DAGPatternNodeFinished-Stack* 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 needn't 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 non-timeout 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 Figure 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 Figure 1 of *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'd 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 eighty 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.

4.3. Performance Evaluation

This section presents the performance evaluation results for EasyPDP running on Dell PowerEdge 2950 Dual Quad Core server with Xeon E5310 processors of 64K L1 cache and 4096K L2 cache. Four popular DP algorithms are evaluated. Specifically, the Smith-Waterman algorithm with linear and affine gap penalty (SWLAG), and Syntenic alignment(SA) algorithm are regular DP algorithms, whereas the Smith-Waterman algorithm with general gap penalty (SWGG), and Viterbi Algorithm(VA) are irregular DP algorithms.

4.3.1. Dependency on Block Size

The *BlockSize* is a critical performance argument in DP algorithm parallelization. Its setting is a tradeoff between the load balancing and communication time. Both too big and too small values of *BlockSize* will adversely affect the program performance. And often the

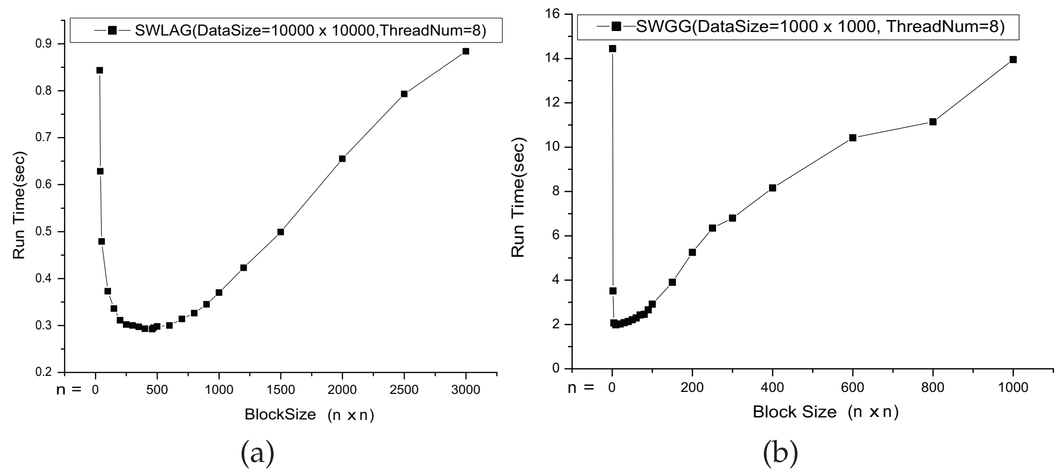


Figure 6. The EasyPDP run time results with different block sizes. In Figure (a), the most suitable block size is between 400 × 400 and 500 × 500. While in Figure (b), the most suitable block size is between 8 × 8 and 20 × 20.

value of the most suitable block size for regular DP algorithm is much bigger than that of the irregular DP algorithm, due to workload of the data blocks. In general, the practical computation workload for a irregular data block is often many times or more than that for a regular data block with the same block size. The running time results for various block sizes are illustrated in Figure 6. We can observe that each of them has a most suitable block size, and for regular SWLAG DP algorithm, its most suitable block size is between 400 × 400 and 500 × 500, whereas the most suitable block size for irregular SWGG DP algorithm is between 8 × 8 and 20 × 20.

4.3.2. Dependency on Number of Threads

Figure 7(a) presents the speedups and comparisons against EasyPDP when 'the number of worker threads' is set to be one as we scale the number of worker threads for four popular DP algorithms in the dual quad cores system. It is obvious that all the speedup curves are much close to the ideal speedup curve except their last points for which the number of worker threads is 8. The phenomenon illustrates that the EasyPDP has a good scalability in its performance improvement. We know that when the number of threads is equal to the number of system cores, the speedup is often the best. Since our EasyPDP is implemented as the master-slave model, 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 by one.

Figure 7(b) gives out the comparisons between the sequential iterative code and EasyPDP when we scale the number of EasyPDP worker threads. We can note that the curve of SWGG is above the ideal speedup line, while the others are not. The reason is that the affection of cache miss for the algorithm SWGG in EasyPDP is non-negligible, while other algorithms are not. By comparing curves between (a) and (b) of Figure 7 for

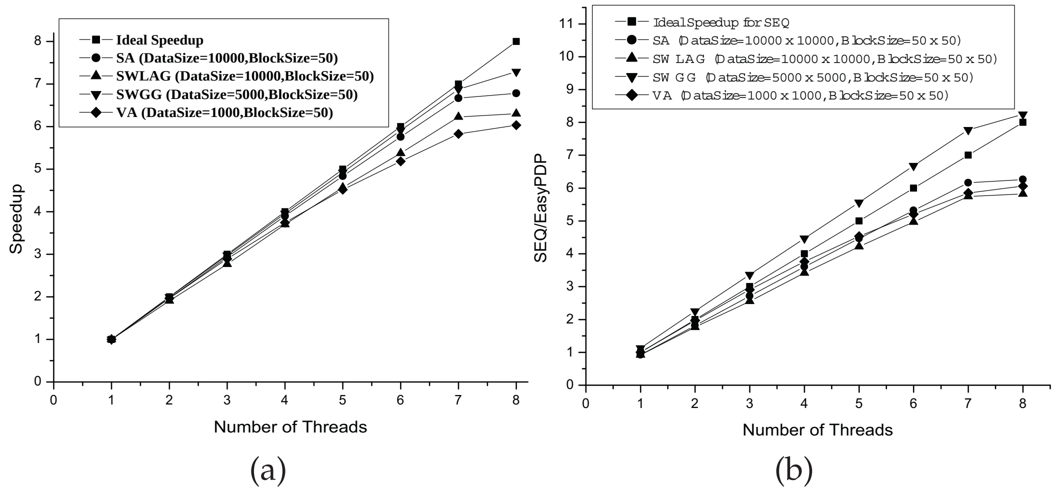


Figure 7. The dependency on the number of threads. (a) The speedups and comparisons against EasyPDP when 'the number of worker threads' is set to be one for four DP algorithms as we scale the number of worker threads. (b) The comparisons against the sequential iterative code as we scale the number of EasyPDP worker threads for four DP algorithms.

algorithms SWLAG, SA, and VA, it is apparent that the curves in (b) of Figure 7 are a bit further away from the ideal speedup curve. This is due to the influence from the EasyPDP overhead.

5. DPX10: A Distributed Parallel DP Computing System

In this section, we propose DPX10 [19, 20], a DAG-based X10 [11] framework for DP applications. DPX10 is a vertex-centric system for the simplicity, reliability, efficiency and scalability of parallel DP programming and execution. It is based on X10 language and APGAS (Asynchronous Partitioned Global Address Space) model [12]. In the following, we start by introducing the interface of DPX10. Then we describe the system design and implementation details.

5.1. The Programming Interface

Writing a DPX10 application involves implementing the predefined **DPX10App** interface (see Figure 8). Its template argument defines the value type associated with vertices. Each vertex has an associated computing result of the specified type.

The **compute** method should be implemented by users. It performs on each vertex at runtime. Parameter **(i, j)** is a unique identifier indicating which vertex is computing. The communication between vertices is hidden from users. The dependencies are resolved automatically by DPX10 and passed as a parameter **vertices**. Users can inspect the value associated with these vertices via **getResult** method in **Vertex** class.

```
public interface DPX10App[T] {
    def compute(i:Int, j:Int, vertices:Rail[Vertex[T]]):T;
    def appFinished(dag:Dag[T]):void;
}

public class Vertex[T] {
    val i:Int, j:Int;
    def getResult():T;
}
```

Figure 8. The DPX10App and Vertex API foundations.

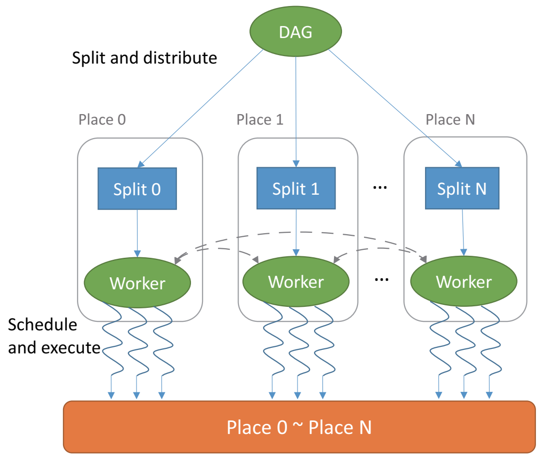


Figure 9. Logical flow of DPX10's execution.

When the program terminates, the **appFinished** method is invoked where the final result should be processed. The argument **dag** can be used to access the results of all vertices.

5.2. System Design and Implementation

The goal of DPX10 is to support efficient execution on multiple nodes and multiple cores without burdening programmers with concurrency management. DPX10 consists of a DAG pattern library to represent a DP algorithm, some useful APIs and the runtime that handle distribution, scheduling and fault recovery.

5.2.1. Execution Overview

Figure 9 shows the overall flow of a DPX10 operation in our implementation. The gray curves with double-headed arrow indicate data communications between workers in different places. In the absence of faults, the execution of a DPX10 program consists of several stages:

1. The DPX10 runtime first distributes and initializes all vertices of the input DAG across available places in parallel. Then it examines vertices on each place and inserts those with zero in-degree into a local ready list to wait for scheduling.

```
public abstract class Dag[T] {
    val width:Int;
    val height:Int;
    def this(width:Int, height:Int);
    def getVertex(i:Int, j:Int):Vertex[T];

    abstract def getDenpendency(i:Int, j:Int):Rail[VertexId];
    abstract def getAntiDenpendency(i:Int, j:Int):Rail[VertexId];
}
```

Figure 10. The DAG API foundations.

2. DPX10 spawns one worker on each place. Each worker is responsible for scheduling local vertices and executing users **compute** method on vertices. Once all local vertices are finished the worker exits.

3. When all workers complete, the computation is finished. DPX10 then invokes the user-defined **appFinished** method to notify the user.

5.2.2. DAG Pattern Library

The DAG pattern is an abstract for a set of DP algorithms which excepts the size, has the same data dependency between vertices, as shown in Section 3.2. Some important APIs of DAG operations are shown in Figure 10. Its template argument is the same as **Vertex** class. The constructor takes two parameters **height** and **width** to determine the size of the DAG.

Two key methods are **getDependency** and **getAntiDependency** which describe the dependency between vertices. They are used by DPX10 runtime to resolve the dependencies automatically. They need to be implemented by the user when creating a custom DAG pattern. The **getDependency** method returns a list of identifiers that represent vertices that should be completed before the vertex **(i, j)**. Another method returns a list of identifiers of vertices that is dependent on the given vertex **(i, j)**. The indegree of these vertices will decrement when vertex **(i, j)** is finished.

Each vertex in a DAG has a unique 2D coordinate marked as (i, j), and an indegree field indicates the unfinished number of its predecessors. Vertices with zero in-degree are schedulable. In addition, a finish flag is kept for each vertex to identify its status and to help recover the result after a node failure.

Users can define the partition and distribution of the DAG through a *Dist* structure to achieve a better locality. At the current stage, three type of distributions are supported by DPX10, which can be configured by command line. Figure 11 demonstrates these three distributions on 4 places. The first two are BLOCK_COLUMN and BLOCK_ROW, which split vertices into columns and rows. The last one is BLOCK_BLOCK, which divides vertices into blocks of equal size. The number of partitions is equal to the number of places. The DAG distribution and assignment play a vital role in achieving high performance. It involves many factors, including the dependencies between the vertices, the dimensions of the graph and the number of computing nodes.

We use the LCS example again to demonstrate the DAG distribution and communications between workers. Figure 12 shows a DAG consisting of nine vertices which are distributed (BLOCK_COLUMN) into three places. Vertices with a check marker below their

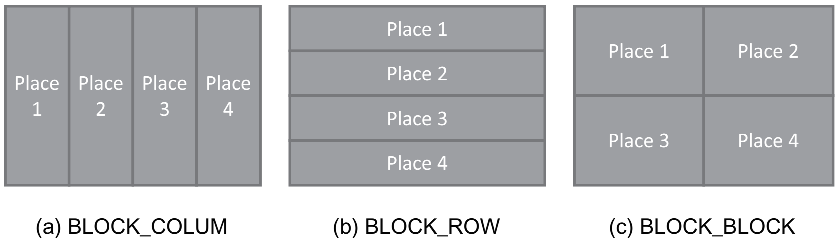


Figure 11. Three type of distributions.

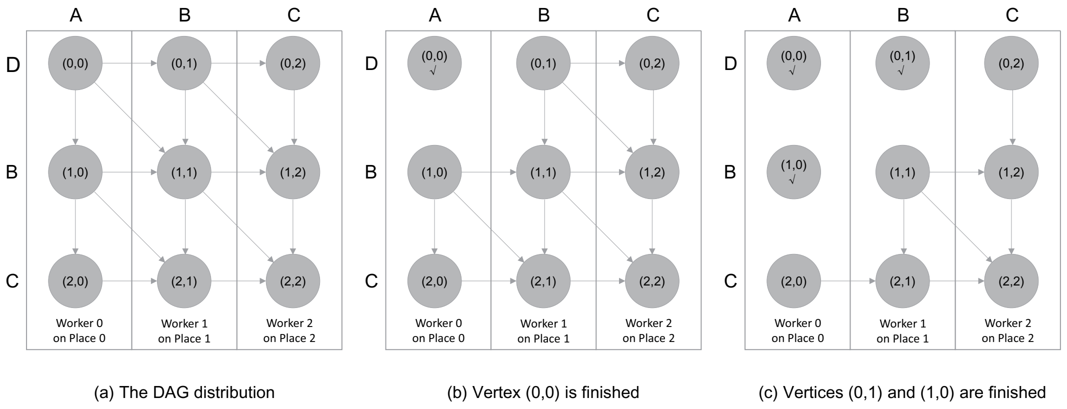


Figure 12. An LCS example to show the DAG distribution and communications between workers. All vertices are divided by columns and distributed into three places. Vertices with a check marker below their coordinates are finished vertices.

coordinates are finished vertices. And those without a parent indicate an in-degree of zero. As shown in Figure 12 (b), when vertex (0,0) completes, the in-degrees of vertices(0,1) and (1,0) decrease to zero. To compute vertex (0,1), worker 1 needs to communicate with worker 0 and copy the result of vertex (0,0) from it. When computations of vertices (0,1) and (1,0) are done (shown in Figure 12 (c)), vertices (0,2), (1,1) and (2,0) become schedulable and the program goes on as the same.

The default initialization method can be overridden to initialize the vertices on demand such as setting the unneeded vertices as finished. For example, as in the longest palindromic subsequence problem (will be discussed in Section 5.3), all the vertices below the diagonal are useless. Consequently, these vertices are marked as finished at the initialization phase.

5.2.3. Worker Computation

On each place, a portion of vertices are assigned in the initial stage. The worker on each place is responsible for computing all its local vertices. There is a ready list that contains executable and uncompleted vertices. Workers repeatedly pull vertices from the list and execute them until all local vertices are finished. A *finished vertices counter* is used to determine the termination of the worker.

When a vertex is ready for computation, the worker spawns a new activity which is parallel with the current one. In this activity, the worker first retrieves its parent vertices through **getDependency** method that we discussed in Section 5.2.2 and passes them along with the identifier of the current vertex to user-defined **compute** method. So users can implement the logic of the algorithm without considering dependencies and communications. After the compute method returned, the worker updates the value of the computing vertex and decreases the in-degree of vertices which rely on the current one. If a vertex's in-dgree goes to zero, it is then ready for computation and is inserted into the ready list on its local place. Finally, the worker marks the vertex as finished and increases the *finished vertices counter*.

The dependent vertices retrieved before calling the compute method may be located at remote places, which means network communications may occur. To reduce the overhead of data transmission, the worker maintains a cache list that caches recently transmitted vertices. For efficiency, the cache list is implemented by a static array and its size can be specified by users. We adopt a simple FIFO replacement mechanism for the cache, considering that the DP algorithm normally has a regular DAG pattern and each vertex may only be needed for a short period.

5.2.4. Fault Tolerance

Fault tolerance is important because hardware and software faults are ubiquitous [21]. The X10 team has been extending X10 to "Resilient X10", where a node failure is reported as a *DeadPlaceException*.

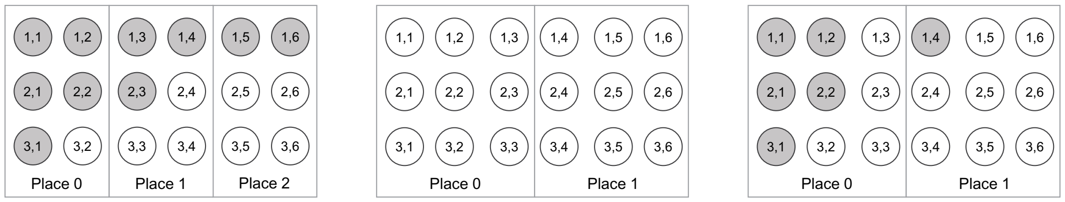
Three basic methods are introduced by X10 to handle the node failures: (a) Ignoring failures and using the results from the remaining nodes, (b) Reassigning the failed nodes work to the remaining nodes, or (c) Restoring the computation from a periodic snapshot [22]. The first method is suitable for problems where the loss of some portion of results may only have minor impacts on accuracy, which is unacceptable for our scenario since users usually need all data to compute the final result accurately. The second method is often adopted in iterative computations, such as the KMeans algorithm [23], for which in each iteration step the master dispatches tasks to workers. The master maintains the computation status and the intermediate results. Once a worker node fails, the master can dispatch tasks to remaining workers. But this method is not fit for DPX10. The reason is that the intermediate result isn't possessed by the master. In contrast, every worker in DPX10 holds a partition of the DAG and is responsible for scheduling the local vertices. The third method is checkpoint, which uses a periodic snapshot to rearrange and restore the distributed array among remaining places after a node failure. The *ResilientDistArray* class implements this function as a fault-tolerant extension of the *DistArray* [22]. However, the checkpoint mechanism is infeasible because a large volume of intermediate results may be produced in the progress of computing. To address it, we propose a new fault tolerant approach as follows.

Algorithm 1 is a pseudo-code that demonstrates this recovery process. Once a *DeadPlaceException* raised, the program stops and enters the recovery mode. Data stored in dead nodes is now inaccessible. DPX10 then creates a new distributed array among remaining places (Line 1), which has the same distribution manner as the old one. We denote the new distributed array as newArray and the old distributed array as oldArray. DPX10 visits all accessible vertices (stored in living places) of oldArray and copies the results of finished vertices into newArray (Line 4 - 10). Then we initialize all unfinished vertices in newArray by estimating in-degrees of them with the **getDependency** method (Line 12 - 14). Next, we revisit the finished vertices and decrease the in-degrees of unfinished vertices by the size of vertices returned by the **getAntiDependency** method (Line 16 - 23). Finally, we replace ODA with NDA (Line 24 - 25).

Algorithm 1: Recovery Procedure

```
1  newArray ← create a new distributed array;
2  oldArray ← the old distributed array;
3  // Resotre accessible and finished vertices from oldArray
4  foreach accessible vertex of oldArray do
5      if vertex is finished then
6          $i$ ← vertex.$i$;
7          $j$ ← vertex.$j$;
8          newArray($i, j$) ← vertex;
9      end
10 end
11 // Set in-degree of vertices in newArray
12 foreach vertex of newArray do
13     vertex.indegree ← getDependency( vertex).size() ;
14 end
15 // Decrease the in-degree of unfinished vertices in newArray
16 foreach vertex of newArray do
17     if vertex is finished then
18         depVertices ← getAntiDependency( vertex);
19         foreach $v$ of depVertices do
20             decreaseIndegree( $v$);
21         end
22     end
23 end
24 // Replace oldArray with newArray
25 oldArray ← newArray;
```

Figure 13 demonstrates this recovery process with an example containing a DAG matrix of 3×6 partitioned into 3 parts for three places. As shown in Figure 13(a), the old distributed array (the old DAG) divides the vertices by the column and distributes them into 3 places (0, 1, 2). Gray vertices denote completed tasks, whereas white vertices represent pending tasks. Assuming place 2 failed, DPX10 creates a new DAG with the same size as the old one and creates a new distributed array to store the new DAG. The vertices are also divided by the column and distributed into remaining places (0, 1), as shown in Figure 13(b). The vertices of oldArray stored in place 2 become inaccessible, which means the results of finished vertex (1, 5) and vertex (1, 6) are lost. Other finished vertices of oldArray stored in place 0 and place 1 can be copied to newArray, as shown in Figure 13(c). Notice



(a) The oldArray before place 2 fails

(b) Create a newArray to store the new DAG when place 2 fails

(b) The newArray after recovery process

Figure 13. An example of recovery process. Gray vertices denote completed vertices. When a failure occurs, a new DAG with the same size as the old one is constructed. All completed vertices except the inaccessible or remote ones are restored into the new DAG.

that the results of vertex (1, 3) and vertex (2, 3) are not restored. The vertices of the third column was stored in place 1 before the failure occurs. And now they are stored in place 0. By default, DPX10 will not restore finished vertices in remote places, such as in this case, the results of vertex (1, 3) and (2, 3) are dropped. The recovery process performs in parallel on all alive places.

5.2.5. Straggler Mitigation Strategy

In practice, due to many unexpected factors such as faulty hardware and software misconfiguration, it often occurs that some tasks run much slower than other tasks (named as straggler tasks). These tasks can slow down the whole process since other tasks have to wait for the result of them. The straggler is prone to occur and become a thorny issue when a program is executed on a heterogeneous cluster or a cloud environment, such as Amazon's Elastic Compute Cloud(EC2) [24]. These environments offer an economic advantage - the ability to own large amounts of computing power only when needed - but they come with the caveat of having to run on virtualized resources with potentially uncontrollable variance.

We classify the straggler tasks into two types, namely, *Hard Straggler* and *Soft Straggler*, defined as follows:

- **Hard Straggler:** A task that goes into a deadlock status due to endless waiting for certain resources (e.g. the network is broken). It cannot stop and complete unless we kill it.
- **Soft Straggler:** A task that can complete its computation successfully, but will take much longer time than common tasks.

For a hard straggler, we should kill it and run another equivalent task, or called backup task, immediately once it was detected. And for a soft straggler, there are two possibilities:

- P1). Soft straggler completes before its backup task, which means there is no need to run a backup task at the beginning.
- P2). Soft straggler finishes later than its backup task. So we should kill it when the backup task is completed. In that way, the straggler task would not occupy the resources to do useless work.

In Hadoop [25], each task keeps track of a progress score, which then can be used to estimate the finish time and to determine straggler tasks. Once a straggler task is detected, a backup task is spawned to run concurrently with the straggler (i.e., speculative execution) [24]. The task killing operation occurs when either of two tasks complete. The drawback of this solution is, no matter which possibility (P1 or P2) takes place, the backup task and the straggler are always running concurrently for a period of time. In other words, it leads to some unnecessary cost, especially for the case of P2. Instead, in DPX10 we do not run a backup task right away. We put it in the end of the ready list, which means some extra time is given to allow the straggler task to earn its second chance. Moreover, a task in DPX10 is a *block* of vertices. And due to the characteristics of DP applications, the execution time of a vertex in the DAG is usually very short, which makes it expensive for a node to keep a progress score and notify other nodes during the computations.

Adaptive Timeout-based Straggler Mitigation Strategy. The intuitive way to detect a straggler is to use a time-out mechanism. Once the execution time of a currently running block exceeds the time limit, the block is marked as a straggler task. However, it is hard for users to choose a proper time-out value since the execution time of a block differs in different computing resources. The time-out value that is too large or too small could fail to detect a straggler or detect too many unnecessary straggler tasks. In DPX10, we adopt an adaptive time-out mechanism. Every node keeps track of three values $t_{i,b}, T_i$ and T_{avg}.

- $t_{i,b}$ is the elapsed time of the current running block b on node i.
- T_i is the average computation time of the latest m blocks completed on node i; $T_i = \frac{T_{i1}+T_{i2}+\cdots+T_{im}}{m}$, where T_{ij} $(1 \leq j \leq m)$, is the execution time of the latest jth block. The reason that we use the average time instead of a single latest execution time is based on the consideration that the computing power might have a sudden change in a short moment.
- T_{avg} is the average of T_i; $T_{avg} = \frac{T_1+T_2+\cdots+T_n}{n}$, where n is the number of computing nodes.

For each node, these three values are updated after a block completes. During the computation of a block b on node i, there are two possibilities: (a) $t_{i,b} > xT_{avg}$; (b) $t_{i,b} \leq xT_{avg}$, where x is an empirical value which is set to 1.4 in DPX10. We consider b as a straggler task only for case (a) since it has slowed down other computing nodes.

5.3. Experiments

In this section, we evaluate the performance of DPX10 by running four different DP applications on Tianhe-1A [26]. Each computing node of Tianhe-1A system is a multi-core SMP server which has dual 2.93Ghz Intel Xeon 5670 six-core processors (total 12 cores per node/24 hardware threads). Each node has 24GB memory and 120GB SSD, connected with Infiniband QDR. The Kylin Linux system is deployed. We used the latest X10 release version, X10 2.5.1. The X10 distribution was built to use Socket runtime.

Two important environment variables needed to be set. *X10_NPLACES* specifies the number of places, which usually equals to the number of processors. And *X10_NTHREADS* indicates the number of threads, which usually equals to the number of cores. So here we set *X10_NTHREADS* to 6 in all our experiments. And *X10_NPLACES* was twice the number of computing nodes used in the experiment.

We carried out four DP applications with a different number of places and graph sizes to show the simplicity, scalability, and efficiency of DPX10. The four DP applications were: (a) Smith-Waterman algorithm with linear and affine gap penalty (SW), (b) Manhattan Tourists Problem (MTP), (c) Longest Palindromic Subsequence(LPS), and (d) 0/1 Knapsack Problem (0/1KP). Moreover, SW was utilized to demonstrate the performance of our new recovery method and the straggler strategy.

The Smith-Waterman algorithm and Knapsack problem are already discussed. The recursive formulation of another two applications is as following.

- The Manhattan Tourists Problem:

$$D(i,j) = \max \begin{cases} D(i-1,j) + w(i-1,j,i,j) \\ D(i,j-1) + w(i,j-1,i,j) \end{cases}$$

  where $w(i_1, j_1, i_2, j_2)$ is the length of the edge from (i_1, j_1) to (i_2, j_2).

- Longest Palindromic Subsequence:

$$D(i,i) = 1$$

$$D(i,j) = \begin{cases} 2, & x_i = x_j, \\ & j = i+1 \\ D(i+1,j-1)+2, & x_i = x_j, \\ & j \neq i+1 \\ \max\{D(i+1,j), D(i,j-1)\}, & x_i \neq x_j \end{cases}$$

  where x_i, x_j is the ith and jth character of the string.

The time for initializing the cluster, generating test graphs, and verifying results were not included in the measurements.

5.3.1. Line of Code

One goal of DPX10 is to provide an easy way for developers to write distributed DP programs. Thus, we use the line of code(LOC) to evaluate the simplicity of writing DP programs with DPX10. We compared the same applications written with DPX10 and with X10 directly. The codes of pre-processing, post-processing, comments and blank lines were not included.

The result is showing at Table 3. With X10 used directly, a distributed program is about 4 times more than the LOC a serial version. And there are many repeated codes in different distributed DP programs such as the distribution of vertices and the communication between workers. DPX10 tries to handle these parts of work automatically and let developers focus on the logic of the algorithm. As we can see, the LOC of four DPX10 programs are about one-third of the same programs written with X10 directly. Moreover, the first three programs nearly have the same LOC with their serial versions. Unlike the first three applications, 0/1KP wrote with DPX10 has fewer more lines. The reason is that the DAG of

Table 3. The LOC of four DP applications

Applications	X10 (Serial)	X10 (Distributed)	DPX10
SW	44	147	42
MLP	29	146	46
LPS	28	172	33
0/1KP	21	154	76

0/1KP is not provided by DPX10. So users need to implement it by themself, which costs 45 extra lines. Even though, implementing a custom DAG is much easier since it doesn't involve any parallel programming.

5.3.2. Scalability

As an indication of how DPX10 scales with places, Figure 14 shows the runtime for SW, MTP, LPS, and 0/1KP with 1 billion vertices. We run our experiments on up to 20 nodes(40 places) because that is all that we have permission to access. In future work, we hope to study DPX10 using larger system/partition sizes to better understand its scalability.

The execution time goes down quickly at first and then reaches a plateau as the number of places increases. The increase of places can reduce the time for executing non-dependent vertices but can also increase the cost of data transmission. Because of the strong data dependency, the speedup curves are not ideal. Figure 14(a) to Figure 14(c) reveal a speedup of about 4 for a 5 fold increase in nodes and Figure 14(d) represents a speedup of about 2.5. In other words, SW, MTP and LPS have a better acceleration performance than 0/1KP. One reason is that 0/1KP has non-deterministic dependencies. And another reason is that given the same data distribution (divided by columns), 0/1KP requires more communications due to its dependency relationship between vertices.

To show how DPX10 scales with graph sizes, we keep the number of places unchanged (40 places on 20 nodes) and vary the size of vertices from 200 million to 2 billion. The result is shown in Figure 15. LPS spend the minimum time since nearly a half of its vertices are not computed. 0/1KP take a little longer since it needs more time to resolve the dependencies as we discussed above. From the four experiments, it can be observed that DPX10 provides a linear scalability with graph sizes.

5.3.3. Fault Tolerance Evaluation

A node failure might occur at an arbitrary point during the program execution. The vertices and other information on that node would be lost, but the remaining nodes still keep their portion of the DAG. DPX10 catches the *DeadPlaceException* and starts the recovery process.

Figure 16 shows three normal circumstances of node failures. The first one illustrates a scenario of a node failure before the computation start. As shown in Figure 16(a). At that time, node 3 hasn't been participating in the computation. Therefore, after recovery the DAG is re-constructed and the parallelism is not diminished. Whereas the second circumstance is about the failure that is occurred during the computation. As shown in Figure

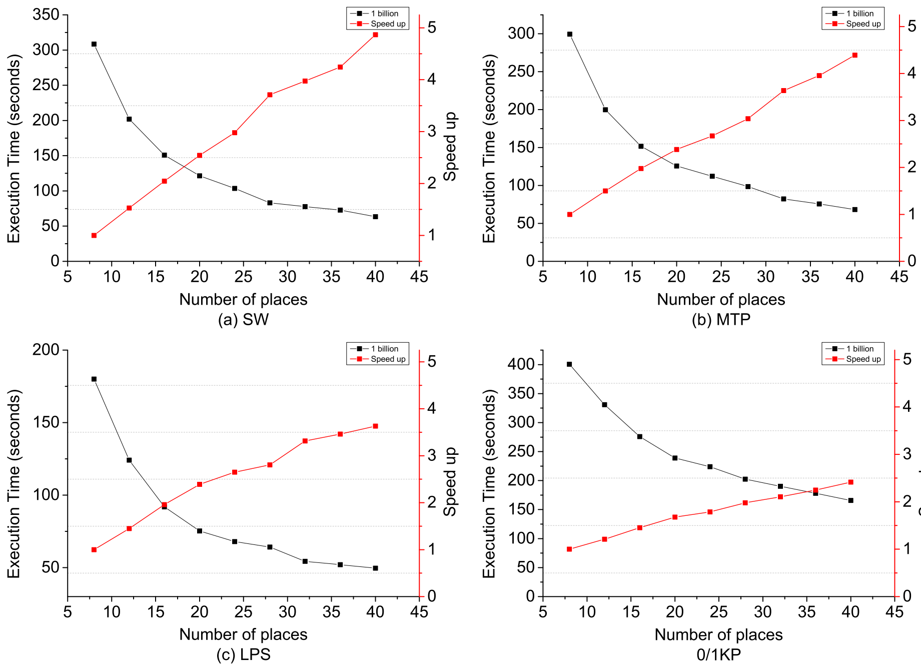


Figure 14. Execution time of four DP applications with 1 billion vertices on different number of places (up to 40 places on 20 nodes). Figures (a,b,c) show a speedup of about 4 for a 5 fold increase in nodes and Figure (d) represents a speedup of about 2.5.

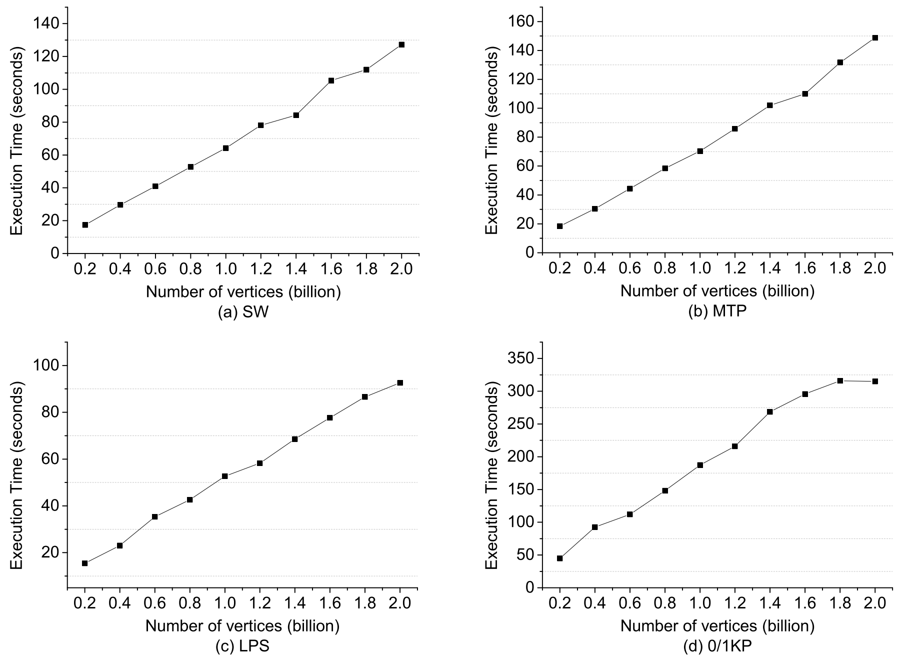


Figure 15. Execution time of four DP applications on 20 nodes (240 cores) with the number of vertices varying from 200 million to 2 billion.

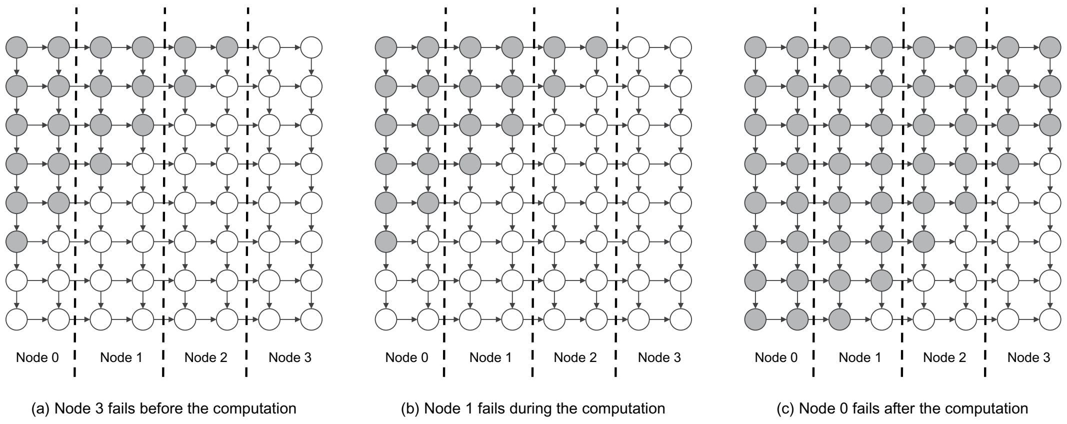

Node 0
Node 1
Node 2
Node 3
(a) Node 3 fails before the computation
(b) Node 1 fails during the computation
(c) Node 0 fails after the computation

Figure 16. Three different circumstances of node failures. Gray vertices are finished.

16(b), the node 1 fails in the middle of the computation. Almost a half of its vertices are finished and we need to re-compute those vertices. Worse, those finished vertices in node 1 are parents of vertices in node 2. In other words, the maximum parallelism cannot be achieved until those vertices in node 1 have been recovered. Figure 16(c) shows the third situation that the failure occurs after computations. Node 0 fails after it completes all its vertices. And there are no unfinished vertices in other nodes that still rely on the vertices in node 0. Hence the vertices in node 0 will be re-computed but no other nodes are infected.

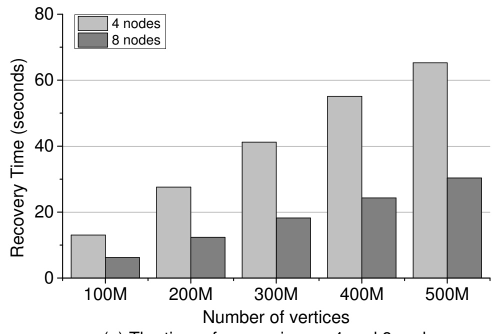
In those three circumstances, the second case does the most damage. So in this section, we try to simulate the second situation. We evaluate the price of fault tolerance by using the SW algorithm on 4 and 8 nodes with the number of vertices varying from 100 million to 500 million. The DAG is split by columns. The failure is triggered manually on node 1 in the middle of the execution. The program continues on the remaining nodes after the recovery. So a half of vertices are computed on 4 and 8 nodes, and more than half of them are computed on 3 and 7 nodes.

Figure 17(a) shows the time for recovering the distributed array. The time increases from 13 to 65 seconds on 4 nodes and from 6 to 30 seconds on 8 nodes, of which the result shows that the recovery time follows a good linear growth. On the other hand, the time for recovering on 8 nodes is half of it on 4 nodes since the recovery is processed in parallel, as discussed in Section 5.2.4.

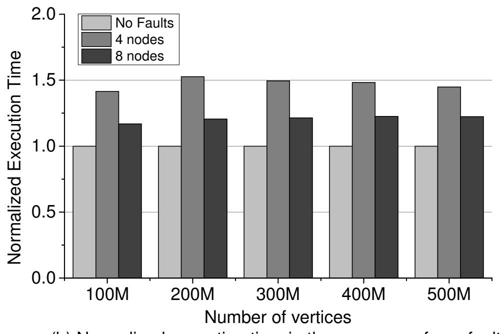
For one fault injection, Figure 17(b) presents the normalized execution time. It is apparent that the impact of one failure reduces with the increase in the number of computing nodes.

5.3.4. Straggler Strategy Evaluation

The straggler condition is very likely to happen at runtime, in particular in a heterogeneous environment. Straggler tasks can substantially slow down the whole program since the tasks in the DP matrix have a strong data dependency between them. Figure 18 is a regular DAG of DP algorithms. According to the number of computing vertices (tasks) and the number of activities, we can classify the whole computation into three computing domains: two non-saturated computing domains and one saturated computing domain. In the non-



(a) The time of recovering on 4 and 8 nodes.



(b) Normalized execution time in the presence of one fault.

Figure 17. The fault tolerance evaluation results with SW algorithm running on 4 and 8 nodes.

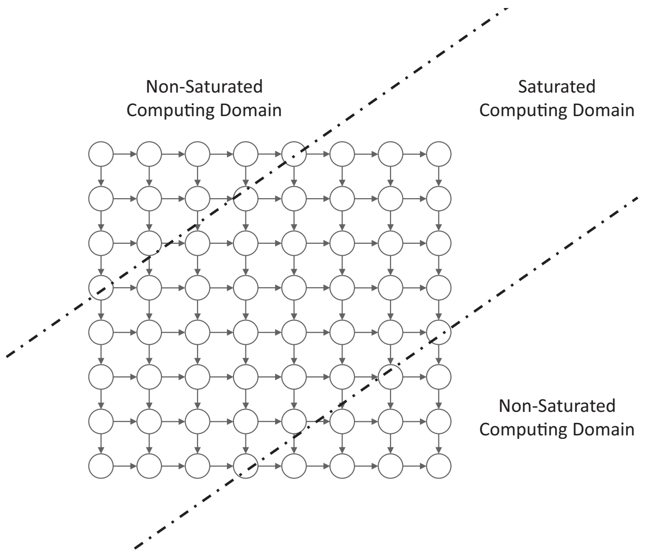
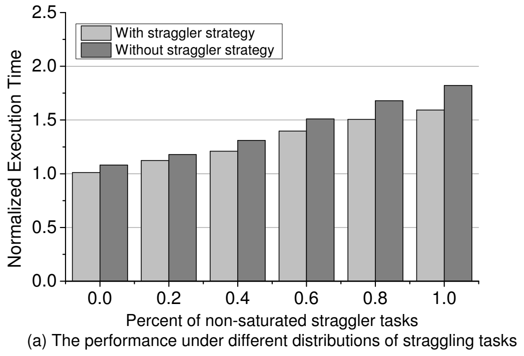


Figure 18. The computing distribution model for a regular DP algorithm.

saturated computing domain, its maximum parallelization degree is less than the number of computing activities. It implies that there must be some idle activities when the computation is going on. For the saturated computing domain, its maximum parallelization degree is greater than or equal to the number of computing activities. All activities should be busy, and no idle activities exist during the computation in this domain. Therefore, straggler tasks in saturated domains would cause less damage than in non-saturated domain since the delay of saturated straggler tasks is more likely to be hidden.

We use the Smith-Waterman algorithm with 100 million vertices on 5 computing nodes to evaluate our straggler strategy. To emulate a straggler task, a sleep method (10ms) is invoked before the real work starts. The number of straggler tasks is set to 1000. We vary the percent of non-saturated tasks in all straggler tasks to see the different impact of straggler condition happening in the saturated region and the non-saturated region. The



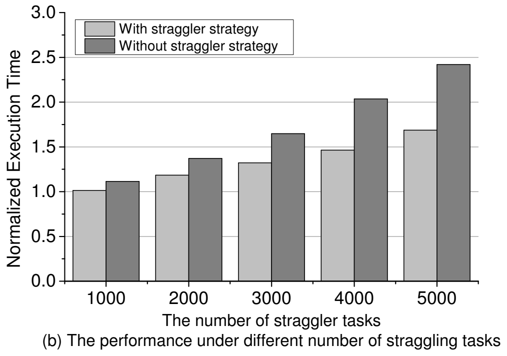


Figure 19. The compared performance results for DPX10 with straggler strategy to that without straggler strategy.

result is shown in Figure 19(a). The running time is normalized to the program without straggler tasks. As the percent of non-saturated region increases from 0 to 1, normalized time increases from 1.08 to 1.82, implying that the straggler tasks in the non-saturated domain have a larger impact on the performance than those in the saturated domain. The explanation is that, in the non-saturated domain, the number of computing tasks is less than the number of activities as we discussed above. More straggler tasks will make idle activities waiting for a longer time for computing vertices, whereas in saturated domains there are sufficient computing vertices. With the straggler strategy enabled, the normalized time increases from 1.01 to 1.59, i.e., there is about 14% performance improvement with straggler strategy.

Moreover we conduct another experiment with a different number of straggler tasks. The percent of non-saturated tasks is set to 0.2 and the number of straggler tasks increases from 1000 to 5000. The result is presented in Figure 19(b). As we can see, our straggler strategy can reduce the execution time (average 20%) in the case of stragglers, especially when the number of tasks is large.

6. Related Work on Accelerating DP Algorithms

In this section we review related work close to us from the following three aspects: 1) DP Parallelization; 2) Graph Processing Framework; 3) X10 and APGAS.

6.1. DP Parallelization

There are an abundant of literature work for DP parallelization. Zheng et al. [27] introduced parallel DP based on stage reconstruction and then applied it to solve the optimized operation of cascade reservoirs. Hamidouche et al. [28] proposed a parallel BSP (Bulk Synchronous Parallel) strategy to execute Smith-Waterman algorithm on multiple multicore and manycore platforms. The hardware like GPU and FPGA has also been used in work [29–31] to accelerate the DP algorithm that is designed for sequence alignment problems. All those work target at a particular application so the features of the problem can be utilized to accelerate the program. DPX10 aims at a kind of DP algorithms. The goal of DPX10 is not only the performance but also the simplicity and reliability.

Maleki et al. [32] proposes a new parallel approach for a class of DP algorithms called "linear-tropical dynamic programming (LTDP)". It breaks data-dependencies across stages and fixes up incorrect values later in the algorithm, which allows multiple stages to be computed in parallel despite dependencies among them. The drawback to this approach is it brings more work to users to write DP programs.

6.2. Graph Processing Framework

Hadoop [25] is an open source implementation of MapReduce [8]. It has been a popular platform for batch-oriented applications, such as information retrieval. The computation is specified by the map and the reduce function. And some recent systems add iteration capabilities to MapReduce. CGL-MapReduce is a new implementation of MapReduce that caches static data in RAM across MapReduce jobs [33]. HaLoop extends Hadoop with the ability of evaluating a convergence function on reducing outputs [34]. But neither CGL-MapReduce nor HaLoop provide fault tolerance across multiple iterations. Moreover, the data flow of these systems is limited to a bipartite graph, which cannot represent the DP algorithms.

Pregel [9] is a computational model for processing large graphs. Programs are expressed as a sequence of supersteps. Within each superstep the vertices compute in parallel, each executing the same user-defined function that expresses the logic of a given algorithm [9]. DPX10 has a similar idea as Pregel, "think like a vertex". But DPX10 is a tailored system for DP applications. Different from Pregel, it contains a DAG pattern library to further simplify the graph programming based on the observation that most of DP algorithms are of the same DAG structure except their data size. Moreover, the implementation of Pregel adopts the distributed memory model, whereas DPX10 takes the APGAS model, which is a hybrid model of the shared memory model and the distributed memory model.

There are also some general-purpose DAG engine like Dryad [35], DAGue [36] and CIEL [37]. They allow data flow to follow a more general directed acyclic graph. These systems target on a large kind of problems which may have various DAGs. So the programmer needs to explicitly express the algorithm as a DAG of tasks and have to handle the communications on their own. In contrast, DPX10 provides a simple interface to express DP algorithms and handles all parallel complexities automatically. In addition, eight commonly used DAG patterns are shipped with DPX10 for immediate use.

Several recent projects [38, 39] have proposed a task-based programming model. They mainly focus on the applications that consist of dependent tasks where each task normally runs for a long time. They are not suitable for computing-intensive DP problems, which consist of plenty of computing tasks but each of them has a relatively short execution time.

6.3. X10 and APGAS

PGAS model assumes a global memory address space that is logically partitioned and a portion of it is local to each process or thread [40]. The novelty of PGAS is that the portions of the shared memory space may have an affinity for a particular process, thereby exploiting locality of reference. The PGAS model is the basis of Unified Parallel C [41], UPC++ [42], Co-Array Fortran [43], Global Arrays [44], SHMEM [45], etc. APGAS model permits both local and remote asynchronous task creation [12]. Two programming languages that use this model are Chapel [15] and X10 [11].

There have been few X10 libraries or frameworks built on top of APGAS. ScaleGraph is an X10 library targeting billion scale graph analysis scenarios. Compared with non-PGAS alternatives, ScaleGraph defines concrete and simple abstractions for representing massive graphs [46]. Acacia [47] is a distributed graph database engine for scalable handling of large graph data. Acacia operates between the boundaries of private and public clouds. It will burst into the public cloud when the resources of the private cloud are insufficient to maintain its service-level agreements. ClusterSs is a StarSs [48] member designed to execute on clusters of SMPS. Tasks of ClusterSs are asynchronously created and assigned to available resources with the support of the APGAS runtime [49].

Since X10 and APGAS are new for the HPC community, we believe a lot of libraries or frameworks need to be developed to support the language to achieve its productivity goals [46].

7. Open Problems

Despite many recent efforts on parallelizing dynamic programming algorithms, there are a number of open problems remained to be explored in future. We elaborate some of them from the following aspects.

7.1. Generality and Simplicity

There have been a number of parallelization proposals [5, 7, 32, 50–52] on DP. Many of them are targeting a specific problem. For example, the Smith-Waterman (SW) algorithm, based on dynamic programming, is one of the most fundamental algorithms in bioinformatics. Some work [50, 51] implement it on a single general purpose microprocessor. They parallelized the algorithm with SIMD method at the instruction level. SparkSW [52] is a distributed implementation of SW algorithm based on Apache Spark [53]. Most of these studies only work for a single DP algorithm and lack generality and simplicity for supporting other DP algorithms parallelization. As we proposed in Section 5, EasyPDP and DPX10 concentrate on the distribution and parallelization of DP algorithms of the type $2D/0D$. So there are still more work need to be done to support the type of $2D/iD(i >= 1)$.

7.2. Exploring Emerging Hardware

Emerging hardware is available at different layers. For example, in the storage layer, we now have solid-state disk and nonvolatile RAM, which are much faster than Hard Disk (HD). In the computation layer, there are a set of accelerators such as GPU, AMD Accelerated Processing Unit (APU), and Field Programmable Gate Array (FPGA). Moreover, in the network layer, remote direct memory access is an efficient hardware tool for speeding network transfer. For a computing system, it is important to adopt this emerging hardware to improve the performance of applications. Currently, generous frameworks or systems for

DP applications are mostly designed for the CPU platform. More research efforts are required to efficiently utilize this emerging hardware at different layers for existing computing systems.

8. Conclusion

In this chapter, we have discussed the importance of dynamic programming techniques in various areas. The chapter reviewed the classic parallel programming models, i.e. shared memory and distributed memory along with a new model PGAS and some PGAS-based languages. A DAG data driven model was proposed for parallel programming of DP applications. Based on this model, two parallel computing systems (i.e., EasyPDP and DPX10) were introduced for multi-core and distributed computing systems, respectively. Some open problems are also presented for future exploration.

Acknowledgment

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 61602336, 11573019, and U1531111.

References

[1] J. U. Bowie, R. Luthy and D. Eisenberg, *Method to identify protein sequences that fold into a known three-dimensional structure*, Science **253** (1991) (5016), pp. 164–170.

[2] X. Huang and K. Chao, *A generalized global alignment algorithm*, Bioinformatics **19** (2003) (2), pp. 228–233.

[3] C. Ciressan, E. Sanchez, M. Rajman and J. Chappelier, *An FPGA-based coprocessor for the parsing of context-free grammars* (2000), pp. 236–245.

[4] M. Farach and M. Thorup, *Optimal evolutionary tree comparison by sparse dynamic programming* (1994), pp. 770–779.

[5] S. Tang, C. Yu, J. Sun, B.-S. Lee, T. Zhang, Z. Xu and H. Wu, *Easypdp: an efficient parallel dynamic programming runtime system for computational biology*, Parallel and Distributed Systems, IEEE Transactions on **23** (2012) (5), pp. 862–872.

[6] Z. Galil and K. Park, *Parallel Algorithms for Dynamic Programming Recurrences with More Than O(1) Dependency*, Journal of Parallel and Distributed Computing **21** (1994) (2), pp. 213–222.

[7] J. Du, C. Yu, J. Sun, C. Sun, S. Tang and Y. Yin, *EasyHPS: A Multilevel Hybrid Parallel System for Dynamic Programming*, in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* (IEEE, 2013), pp. 630–639.

[8] J. Dean and S. Ghemawat, *MapReduce: simplified data processing on large clusters*, Communications of the ACM **51** (2008) (1), pp. 107–113.

[9] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser and G. Czajkowski, *Pregel: a system for large-scale graph processing*, in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (ACM, 2010), pp. 135–146.

[10] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski and C. Kozyrakis, *Evaluating MapReduce for Multi-core and Multiprocessor Systems* (2007), pp. 13–24.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun and V. Sarkar, *X10: an object-oriented approach to non-uniform cluster computing*, Acm Sigplan Notices **40** (2005) (10), pp. 519–538.

[12] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky and O. Tardieu, *The asynchronous partitioned global address space model*, in *The First Workshop on Advances in Message Passing* (2010), pp. 1–8.

[13] R. Rabenseifner, G. Hager and G. Jost, *Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes*, in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on* (IEEE, 2009), pp. 427–436.

[14] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan and R. Thakur, *Hybrid parallel programming with MPI and unified parallel C*, in *Proceedings of the 7th ACM international conference on Computing frontiers* (ACM, 2010), pp. 177–186.

[15] D. Callahan, B. L. Chamberlain and H. P. Zima, *The cascade high productivity language*, in *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on* (IEEE, 2004), pp. 52–60.

[16] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands *et al.*, *Productivity and performance using partitioned global address space languages*, in *Proceedings of the 2007 international workshop on Parallel symbolic computation* (ACM, 2007), pp. 24–32.

[17] S. Mitra, N. Seifert, M. Zhang, Q. Shi and K. S. Kim, *Robust system design with built-in soft-error resilience*, IEEE Computer **38** (2005) (2), pp. 43–52

[18] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe and A. G. Nowatryk, *Fingerprinting: bounding soft-error-detection latency and bandwidth*, architectural support for programming languages and operating systems **39** (2004) (11), pp. 224–234.

[19] C. Wang, C. Yu, J. Sun and X. Meng, *DPX10: An Efficient X10 Framework for Dynamic Programming Applications*, in *Parallel Processing (ICPP), 2015 44th International Conference on* (2015), pp. 869–878.

[20] C. Wang, C. Yu, S. Tang, J. Xiao, J. Sun and X. Meng, *A general and fast distributed system for large-scale dynamic programming applications*, Parallel Computing **60** (2016), pp. 1 – 21,
http://www.sciencedirect.com/science/article/pii/S016781911630103X.

[21] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski and S. Matsuoka, *Design and modeling of a non-blocking checkpointing system*, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (IEEE Computer Society Press, 2012), p. 19.

[22] D. Cunningham, D. Grove, B. Herta, A. Iyengar, K. Kawachiya, H. Murata, V. Saraswat, M. Takeuchi and O. Tardieu, *Resilient X10: efficient failure-aware programming*, in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming* (ACM, 2014), pp. 67–80.

[23] J. A. Hartigan and M. A. Wong, *Algorithm AS 136: A K-Means clustering algorithm*, Journal of the Royal Statistical Society. Series C (Applied Statistics) **28** (1979) (1), pp. 100–108.

[24] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz and I. Stoica, *Improving MapReduce performance in heterogeneous environments*, Operating Systems Design and Implementation **8** (2008) (4), p. 7.

[25] K. Shvachko, H. Kuang, S. Radia and R. Chansler, *The Hadoop distributed file system*, in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (IEEE, 2010), pp. 1–10.

[26] X.-J. Yang, X.-K. Liao, K. Lu, Q.-F. Hu, J.-Q. Song and J.-S. Su, *The TianHe-1A supercomputer: its hardware and software*, Journal of Computer Science and Technology **26** (2011) (3), pp. 344–351.

[27] H. Zheng, Y. Mei, K. Duan and Y. Lin, *Parallel dynamic programming based on stage reconstruction and its application in reservoir operation*, in *Computer and Information Science (ICIS), 2014 IEEE/ACIS 13th International Conference on* (IEEE, 2014), pp. 327–336.

[28] K. Hamidouche, F. M. Mendonca, J. Falcou, A. C. M. A. de Melo and D. Etiemble, *Parallel Smith-Waterman Comparison on Multicore and Manycore Computing Platforms with BSP++*, International Journal of Parallel Programming **41** (2013) (1), pp. 111–136.

[29] J. Singh and I. Aruni, *Accelerating Smith-Waterman on heterogeneous cpu-gpu systems*, in *Bioinformatics and Biomedical Engineering,(iCBBE) 2011 5th International Conference on* (IEEE, 2011), pp. 1–4.

[30] M. Korpar and M. Šikić, *SW#–GPU-enabled exact alignments on genome scale*, Bioinformatics (2013), p. btt410.

[31] E. F. De Sandes, G. Miranda, A. De Melo, X. Martorell, E. Ayguade *et al.*, *CUDAlign 3.0: Parallel biological sequence comparison in large GPU clusters*, in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on* (IEEE, 2014), pp. 160–169.

[32] S. Maleki, M. Musuvathi and T. Mytkowicz, *Parallelizing dynamic programming through rank convergence*, in *ACM SIGPLAN Notices* (ACM, 2014), vol. 49, pp. 219–232.

[33] J. Ekanayake, S. Pallickara and G. Fox, *MapReduce for data intensive scientific analyses*, in *eScience, 2008. eScience'08. IEEE Fourth International Conference on* (IEEE, 2008), pp. 277–284.

[34] Y. Bu, B. Howe, M. Balazinska and M. D. Ernst, *HaLoop: efficient iterative data processing on large clusters*, Proceedings of the VLDB Endowment **3** (2010) (1-2), pp. 285–296.

[35] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly, *Dryad: distributed data-parallel programs from sequential building blocks*, European Conference on Computer Systems **41** (2007) (3), pp. 59–72.

[36] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier and J. Dongarra, *DAGuE: A generic distributed DAG engine for high performance computing*, Parallel Computing **38** (2012) (1), pp. 37–51.

[37] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy and S. Hand, *CIEL: A Universal Execution Engine for Distributed Data-Flow Computing.*, in *NSDI* (2011), vol. 11, pp. 9–9.

[38] E. H. Rubensson and E. Rudberg, *Chunks and Tasks: a programming model for parallelization of dynamic algorithms*, Parallel Computing (2013).

[39] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde and I. T. Foster, *Turbine: A distributed-memory dataflow engine for high performance many-task applications*, Fundamenta Informaticae **128** (2013) (3), pp. 337–366.

[40] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao and D. Chavarría-Miranda, *An evaluation of global address space languages: co-array fortran and unified parallel C*, in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (ACM, 2005), pp. 36–47.

[41] T. El-Ghazawi and L. Smith, *UPC: unified parallel C*, in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (ACM, 2006), p. 27.

[42] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan and K. Yelick, *UPC++: a PGAS extension for C++*, in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International* (IEEE, 2014), pp. 1105–1114.

[43] R. W. Numrich and J. Reid, *Co-Array Fortran for parallel programming*, in *ACM Sigplan Fortran Forum* (ACM, 1998), vol. 17, pp. 1–31.

[44] J. Nieplocha, R. J. Harrison and R. J. Littlefield, *Global arrays: a portable shared-memory programming model for distributed memory computers*, in *Proceedings of*

the 1994 ACM/IEEE conference on Supercomputing (IEEE Computer Society Press, 1994), pp. 340–349.

[45] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel and L. Smith, *Introducing OpenSHMEM: SHMEM for the PGAS community*, in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (ACM, 2010), p. 2.

[46] M. Dayarathna, C. Houngkaew and T. Suzumura, *Introducing ScaleGraph: an X10 library for billion scale graph analytics*, in *Proceedings of the 2012 ACM SIGPLAN X10 Workshop* (ACM, 2012), p. 6.

[47] M. Dayarathna and T. Suzumura, *Towards scalable distributed graph database engine for hybrid clouds*, in *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds* (IEEE Press, 2014), pp. 1–8.

[48] J. Labarta, *StarSS: A programming model for the multicore era*, in *PRACE WorkshopNew Languages & Future Technology Prototypes at the Leibniz Supercomputing Centre in Garching (Germany)* (2010).

[49] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi and J. Labarta, *ClusterSs: a task-based programming model for clusters*, in *Proceedings of the 20th international symposium on High performance distributed computing* (ACM, 2011), pp. 267–268.

[50] T. Rognes and E. Seeberg, *Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors*, Bioinformatics **16** (2000) (8), pp. 699–706.

[51] M. Farrar, *Striped Smith-Waterman speeds database searches six times over other SIMD implementations*, Bioinformatics **23** (2007) (2), pp. 156–161.

[52] G. Zhao, C. Ling and D. Sun, *SparkSW: scalable distributed computing system for large-scale biological sequence alignment*, in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on* (IEEE, 2015), pp. 845–852.

[53] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, *Spark: cluster computing with working sets* **10** (2010), pp. 10–10.