Parallel Computing 60 (2016) 1–21

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A general and fast distributed system for large-scale dynamic programming applications

Chen Wang[a], Ce Yu[a,*], Shanjiang Tang[a], Jian Xiao[b], Jizhou Sun[a], Xiangfei Meng[c]

[a] School of Computer Science and Technology, Tianjin University, Tianjin, China
[b] School of Computer Software, Tianjin University, Tianjin, China
[c] National Supercomputer Center in Tianjin, Tianjin, China

ARTICLE INFO

Article history:
Received 17 December 2015
Revised 24 September 2016
Accepted 29 September 2016
Available online 4 October 2016

Keywords:
Dynamic programming
Programming framework
X10
APGAS

ABSTRACT

Dynamic programming is an important technique widely used in many scientific applications. Due to the massive volume of applications' data in practice, parallel and distributed DP is a must. However, writing a parallel and distributed DP program is difficult and error-prone because of its intrinsically strong data dependency. In this paper, we present DPX10, a DAG-based distributed X10 framework aiming at simplifying the parallel programming for DP applications. DPX10 enables users to write highly efficient parallel DP programs without much effort. For DPX10 programming, users only need to do two things: 1) Instantiating a DAG pattern by indicating the dependency between vertices of the DAG; 2) Implementing the DP application's logic in the **compute** method of the vertices. DPX10 provides eight commonly used DAG patterns and a simple API to allow users to customize their own DAG patterns. All the tiresome work of DP parallelization including DAG distribution, tasks scheduling, and tasks communication are hidden from users and covered by DPX10. Moreover, DPX10 is fault-tolerant and has a mechanism to handle the problem of straggler tasks, which run much slower than other tasks due to unexpected resource contention. Finally, we use four DP applications with up to 2 billion vertices running on 240 cores to demonstrate the simplicity, efficiency, and scalability of our proposed framework.

1. Introduction

In many application domains such as economics and bioinformatics, dynamic programming is a practical and efficient solution. However, due to the huge volume of data, parallelizing DP in a distributed manner is necessary and crucial to keep the computation time at acceptable levels. But the intrinsically strong data dependency of DP makes it difficult for programmers to write a correct and efficient parallel and distributed DP program.

There have been a number of parallelization proposals [1–6] 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 [1,2] implement it on a single general purpose microprocessor. They parallelized the algorithm with SIMD method at the instruction level. SparkSW [3] is a distributed implementation of SW algorithm based on Apache

* Corresponding author.
E-mail addresses: wangvsa@tju.edu.cn, wangvsa@163.com (C. Wang), yuce@tju.edu.cn (C. Yu), tashj@tju.edu.cn (S. Tang), xiaojian@tju.edu.cn (J. Xiao), jzsun@tju.edu.cn (J. Sun), mengxf@nscc-tj.gov.cn (X. Meng).

http://dx.doi.org/10.1016/j.parco.2016.09.002
0167-8191/

Spark [3]. These studies only work for a single DP algorithm and lack generality and simplicity for supporting other DP algorithms parallelization.

On the other hand, to ease parallel programming for DP applications, some runtime systems and frameworks are proposed in recent years [5,6]. They can be classified according to two classic parallel computing models, i.e., shared memory model (favors the data communication but its scalability is a problem) [7] and distributed memory model (it has good scalability, but the communication cost is high) [8]. For example, EasyPDP [5] is a shared memory based runtime system for DP algorithms. It allows users to write parallel DP program easily and run them in a single machine consisting of multiple CPU cores. In contrast, EasyHPS [6] is a distributed runtime system for DP algorithms based on the distributed memory model. However, there is a lack of a general purpose system for DP algorithms that combines the shared memory model and the distributed memory model together for better data communication and scalability.

In this paper, we present DPX10, a DAG-based X10 [9] framework for DP applications. DPX10 is a vertex-centric and fine-grained 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) [10] model, which combines the shared memory model with distributed memory model. The vertices of DAG are distributed to all computing nodes and on each node, one worker is responsible for scheduling those vertices. Multi-threads are spawned by each worker to execute the vertices. The data is accessed through shared memory model if it is stored on a local node, otherwise, distributed memory model is used.

DPX10 can express the DP computations and hides the messy details of parallelization, data distribution and fault tolerance. Users can specify the computation in terms of a **DAG pattern** and a **compute** method. DPX10 coordinates the distributed execution of a set of data-parallel tasks arranged according to the data-flow DAG. Eight commonly used DAG patterns are provided in the DAG pattern library, which covers a wide range of DP problems. Moreover, several DP applications are utilized to demonstrate DPX10's simplicity, efficiency and scalability.

Specifically, we make the following major contributions:

1. Propose a straightforward and powerful abstraction to express DP computations, combined with an X10 implementation of this abstraction that achieves high performance on commodity clusters. To our knowledge, this is the first programming framework for DP problems based on APGAS model.
2. Propose a new recovery mechanism for distributed arrays instead of the periodicity snapshot method provided by X10.
3. Present the implementation of our framework, showing its simplicity, efficiency, reliability, scalability and the ability to solve straggler problems.

The rest of the paper is structured as follows. Section 2 briefly describes some background information. The model of computation is discussed in Section 3. Section 4 introduces DPX10 framework and the implementation of it, including the DAG pattern library, worker implementation, and fault-tolerance, etc. In Section 5 we present two applications to show the process of writing DP programs with DPX10. In Section 6 we report our experimental evaluation of the system. We conclude in Sections 7 and 8 with a discussion of the related literatures and future research directions.

2. Background

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. Each model has its 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 [10–14] 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) [14] 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. X10 and APGAS

X10 is a high-performance and high-productivity programming language developed by IBM Research. It aims to enable users to write highly efficient parallel and distributed programs easily and quickly.

X10 is based on the asynchronous partitioned global address Space (APGAS) [10] model, a variant of the PGAS model. APGAS model permits both local and remote asynchronous task creation. It introduces two key concepts: **places** and **asyn-**

chronous. A place is a collection of data and worker threads operating on the data. Places are typically realized as operating system processes, which is a coarse-grained parallelism. Asynchronous is fundamental to the language which is denoted by the keyword **async**. Statement **async S** is a fine-grained parallelism, which tells X10 to run S as a separate and concurrent **activity**, similar to threads in the operating system.

The latest major release of X10 is X10 2.5.3. It has been constructed via source-to-source compilation to either C++ or Java (termed as Native X10 or Managed X10) [15]. Since we are more interested in performance than portability, the current version of DPX10 only targets the Native X10.

An X10 program consists of at least one class definition with a main method. The number of places and the mapping from places to nodes can be specified by users at launch time. The execution starts with the main method at Place (0).

2.3. The Dynamic Programming (DP) algorithm

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 [16]: 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 $1 \leq i, j \leq n$,

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

where x_i, y_i are computed in constant time.

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

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

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

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

In this paper, we concentrate on the distribution and parallelization of DP algorithms of the type $2D/0D$, which are important DP algorithms for many applications, e.g., 0/1 knapsack problem, longest substring problem, and Smith–Waterman algorithm, etc. We leave the consideration of $2D/iD(i >= 1)$ to future work.

2.4. The vertex-centric framework

The vertex-centric frameworks (e.g., Pregel [17], GraphLab [18]) are platforms that iteratively execute a user-defined program over vertices of a graph. The vertex program is designed from the perspective of a vertex, receiving as input the vertex's data as well as data from adjacent vertices and incident edges. The vertex program is executed across vertices of the graph synchronously, or may also be performed asynchronously. Execution halts after either a specified number of iterations or all vertices have converged. The vertex-centric programming model is less expressive than conventional graph-omniscient algorithms but is easily scalable with more opportunity for parallelism [19].

Based on the granularity of the tasks, a graph processing program has two parallel scheduling models, i.e., fine-grained and coarse-grained. Fine-grained parallelism means individual tasks are relatively small regarding code size and execution time. The data is transferred among processors frequently in amounts of one or a few memory words. Coarse-grained is the opposite: nodes are infrequently communicated, after larger amounts of computation. The finer the granularity, the greater the potential for parallelism and hence speed-up, but the greater the overheads of synchronization and communication [20]. In order to attain the best parallel performance, the best balance between load and communication overhead needs to be found. If the granularity is too fine, the performance can suffer from the increased communication overhead. On the other side, if the granularity is too coarse, the performance can suffer from load imbalance.

3. DPX10 parallel programming model

DPX10 focuses on the DP algorithms of type $2D/0D$ as we discussed in Section 2.3. The computation 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 DAG to represent the DP matrix and the dependency relationship between cells.

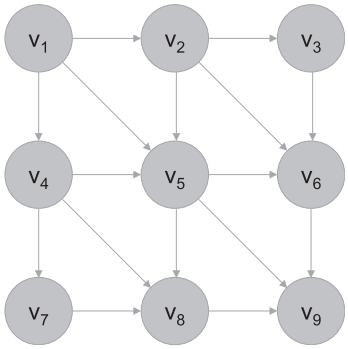

Fig. 1. An example DAG.

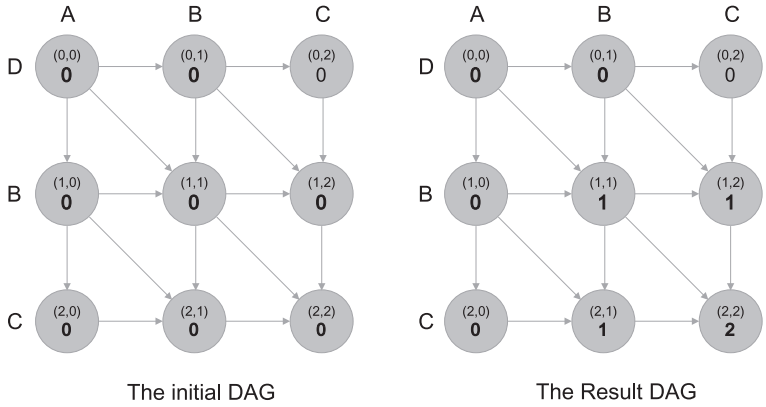

Fig. 2. An example DAG of longest common substring problem.

A directed graph 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 Fig. 1. A path in a directed graph can be described by a sequence of edges having the property that the ending vertex of each edge in the sequence is the same as the starting vertex of the next edge in the sequence; a path forms a cycle if the starting vertex of its first edge equals the ending vertex of its last edge. A DAG is a directed graph that has no cycles.

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.

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. The library will be discussed in Section 4.2.2.

A typical DPX10 computation consists of inputting 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. When the program terminates, the result of each vertex can be accessed by using the system APIs, provided in Section 4.1.

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.

Fig. 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

```
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;
}
```

Fig. 3. The DPX10App and Vertex API foundations.

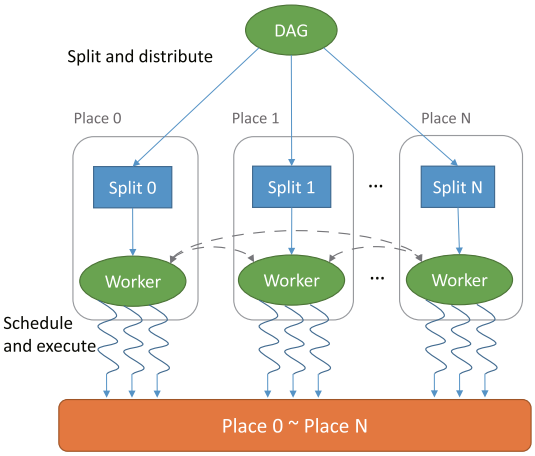

Fig. 4. Logical flow of DPX10's execution.

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*.

4. DPX10 framework

We start by introducing the interface of DPX10. Then we describe our system design and implementation.

4.1. The programming interface

Writing a DPX10 application involves implementing the predefined **DPX10App** interface (see Fig. 3). 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.

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.

4.2. 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.

4.2.1. Execution overview

Fig. 4 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];
}
```

Fig. 5. The DAG API foundations.

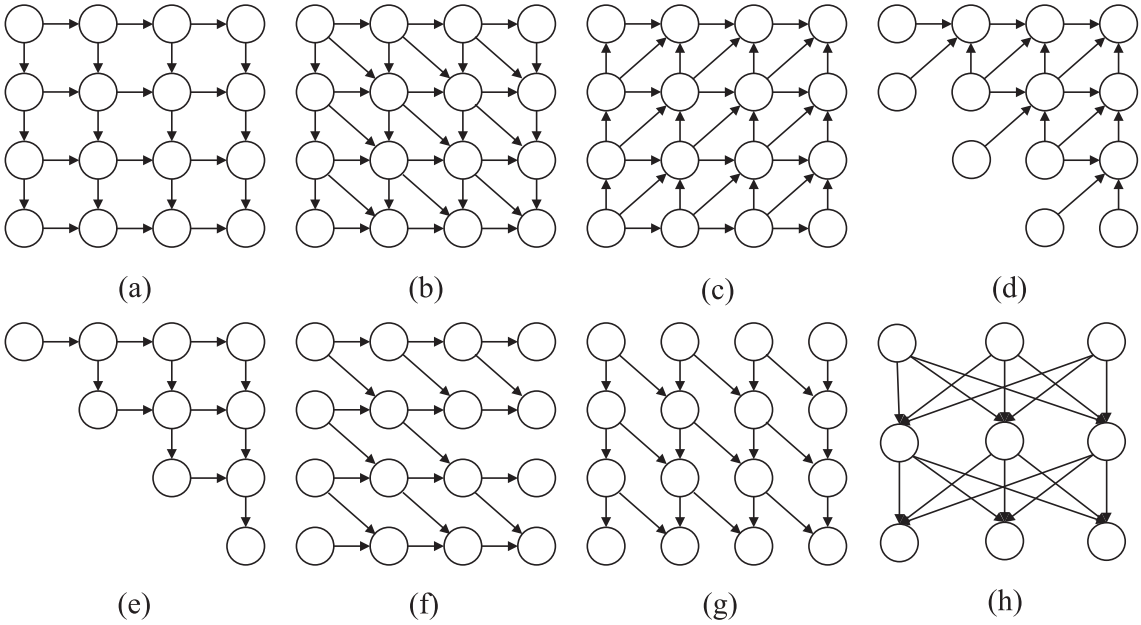

Fig. 6. Eight built-in DAG patterns.

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.

4.2.2. 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. All DAG patterns are subclasses of **DAG** class. Some important APIs of the DAG class are shown in Fig. 5. 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.

The DAG pattern library is a major component in DPX10. It provides built-in DAG patterns and exposes a simple API for users to customize their own DAG patterns. As described above, the optimal substructure of each DP problem can be represented as a DP formulation. Therefore, we can build a corresponding DAG for each DP problem. In the current implementation, the DAG pattern library contains eight built-in DAG patterns, as shown in Fig. 6. For example, the LCS algorithm we used in Section 3 is a classical DP algorithm. And its DAG is shown in Fig. 6(b).

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. Fig. 7 demon-

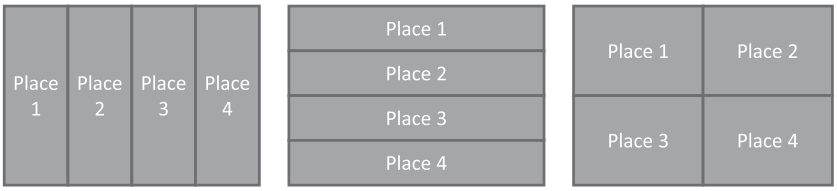

Fig. 7. Three type of distributions.

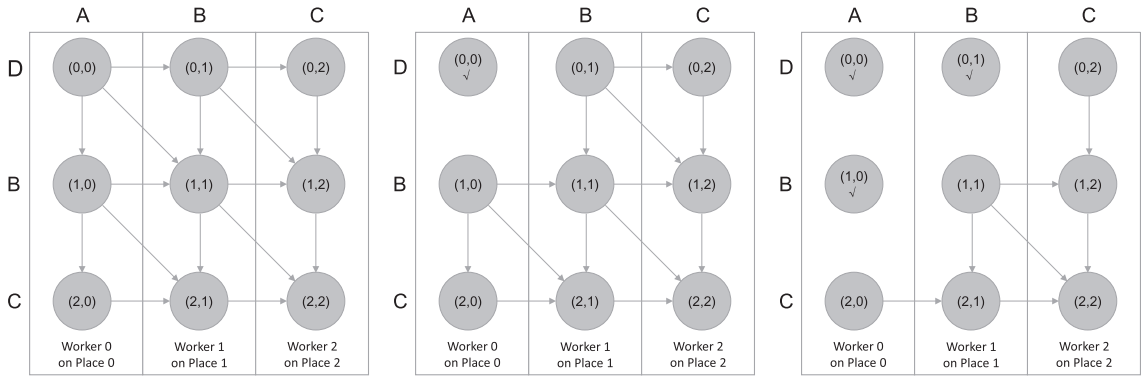

Fig. 8. An LCS example to show the DAG distribution and communications between workers. All vertices are divided by columns and distributed into three places. Vertices with a check marker below their coordinates are finished vertices.

strates 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. Fig. 8 shows a DAG consisting of nine vertices which are distributed (BLOCK_COLUMN) into three places. Vertices with a check marker below their coordinates are finished vertices. And those without a parent indicate an in-degree of zero. As shown in Fig. 8 (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 Fig. 8 (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 6), all the vertices below the diagonal are useless. Consequently, these vertices are marked as finished at the initialization phase.

4.2.3. 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 4.2.2 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 spec-

ified 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.

4.2.4. 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 - 21). Finally, we replace ODA with NDA (Line 23).

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 decreaseIndegree;
20         v;
21     end
22 end
23 // Replace oldArray with newArray
24 oldArray ← newArray;
```

Fig. 9 demonstrates this recovery process with an example containing a DAG matrix of 3×6 partitioned into 3 parts for three places. As shown in Fig. 9(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

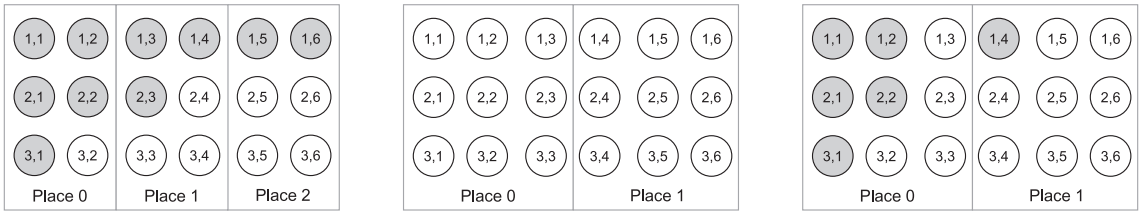

Fig. 9. An example of recovery process. Gray vertices denote completed vertices. When a failure occurs, a new DAG with the same size as the old one is constructed. All completed vertices except the inaccessible or remote ones are restored into the new DAG.

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 Fig. 9(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 Fig. 9(c). Notice 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.

4.2.5. 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} $(1 \le j \le m)$, 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. Case study

This section uses two DP applications to demonstrate the process of writing DP programs with DPX10.
From a user's perspective, it only takes three steps to implement a DP application with DPX10.

1. Choose a built-in DAG pattern or write a customized one.
2. Implement the **compute** and **appFinished** method by inheriting the DPX10App class.
3. Launch the DPX10 program.

The number of places and the mapping from places to nodes can be set as arguments or environment variables as regular X10 programs.

5.1. Smith–Waterman algorithm

The Smith–Waterman algorithm is a widely used dynamic programming algorithm in computational biology, with several important variants and improvements. It performs local sequence alignment that is for determining similar regions between two strings or nucleotide or protein sequences. For simplicity, we only take adjacent elements into account in the calculation. The scoring matrix H is built as follows:

$$H(i,0) = 0, 0 \leq i \leq m$$

$$H(0,j) = 0, 0 \leq j \leq m$$

$$H(i,j) = \max \begin{cases} 0 \\ H(i-1,j-1) + s(a,b) \\ \max\{H(i-1,j), H(i,j-1)\} + p \end{cases} \quad (1)$$

where:

- a, b are strings over the Alphabet
- m, n is the length of a and b
- $s(a, b)$ is the similarity function on the alphabet, $s(a, b) = +2$ if $a = b$(match), -1 if $a \neq b$(mismatch)
- $p = -1$ is the gap penalty
- $H(i, j)$ is the maximum Similarity-Score between a suffix of $a[1 \ldots i]$ and a suffix of $b[1 \ldots j]$

The DAG pattern is the same as the LCS algorithm, which is already provided by the DAG pattern library, as shown in Fig. 6(b). An implementation of the Smith–Waterman algorithm is shown in Fig. 10, omitting some irrelevant details.

The **SWApp** class inherits from **DPX10App** class. The value type of the vertex is **Int**, which is enough for storing the similarity score. The **compute** method implements the logic of the algorithm, each vertex calculates three values and returns the maximum one based on the Eq. (1). The dependent vertices are provided as the parameter **vertices**, for example, when computing (2, 2), **vertices** is a list of vertices (1, 1), (2, 1), (1, 2).

The backtracking is performed in **appFinished()** method. It is very straightforward as users can use the given **dag** like a normal 2D array.

5.2. 0/1 Knapsack problem

The Knapsack problem is about combinatorial optimization: Given a set of items, each with a mass and value, determines the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The most common Knapsack problem is the 0/1 Knapsack problem, which can be formulated as: Given n items $\langle z_1, \cdots, z_n \rangle$ where each item z_i has a value v_i and weight w_i. x_i is the copies of item z_i, which is zero or one. The goal is to maximize $\sum v_i$ subject to $\sum w_i x_i \leq W$, where W is the maximum weight that we can carry in the bag.

We use 0/1 Knapsack problem to show the process of implementing a new DAG pattern. The user extends from **DAG** class and then implements two methods: **getDependency** and **getAntiDependency**, as discussed in Section 4.1.

Assume $w_1, w_2, \cdots, w_n$ and W are strictly positive integers. Define $m(i, j)$ to be the maximum value that can be attained with a weight less than or equal to j using items up to i (first i items). Thus, $m(i, j)$ can be defined recursively as follows:

$$m(i,j) = \begin{cases} m(i-1,j) & w_i > j \\ \max\{m(i-1,j), m(i-1,j-w_i) + v_i\} & w_i \leq j \end{cases} \quad (2)$$

```
public class SWApp extends DPX10App[Int] {
    var str1:String, str2:String;
    static val MATCH_SCORE = 2n;
    static val DISMATCH_SCORE = -1n;
    static val GAP_PENALTY = -1n;

    public def compute(i:Int, j:Int, vertices:Rail[Vertex[Int]]):Int {
        if(i==0n || j==0n)
            return 0;
        else {
            var lefttop:Int = 0n, left:Int = 0n; top:Int = 0n;
            for(vertex in vertices) {
                if(vertex.i==i-1n && vertex.j==j-1n) {    // (i-1, j-1)
                    lefttop = vertex.getResult();
                    lefttop += str1.charAt(i)==str2.charAt(j) ?
                         MATCH_SCORE : DISMATCH_SCORE;
                }
                if(vertex.i==i-1n && vertex.j==j)         // (i-1, j)
                    top = vertex.getResult() + GAP_PENALTY;
                if(vertex.i==i && vertex.j==j-1n)         // (i, j-1)
                    left = vertex.getResult() + GAP_PENALTY;
            }
        }
        return max(lefttop, left, top);
    }

    // A backtracking method is performed
    public def appFinished(dag:Dag[Int]):void {
        var i:Int = str1.length() as Int - 1n;
        var j:Int = str2.length() as Int - 1n;
        while(true) {
            if(i==0n|| j==0n) break;
            val c1 = str1.charAt(i);
            val c2 = str2.charAt(j);
            if(c1==c2) Console.OUT.print(c1);
            else Console.OUT.print("-");
            val left = dag.getVertex(i-1n, j).getResult();
            val up = dag.getVertex(i, j-1n).getResult();
            val leftup = dag.getVertex(i-1n, j-1n).getResult();
            if(left >= up && left >= leftup) {
                i = i - 1n;
            } else if(up >= left && up >= leftup) {
                j = j - 1n;
            } else {
                i = i - 1n;
                j = j - 1n;
            }
        }
    }
}
```

Fig. 10. Smith–Waterman algorithm implemented in DPX10.

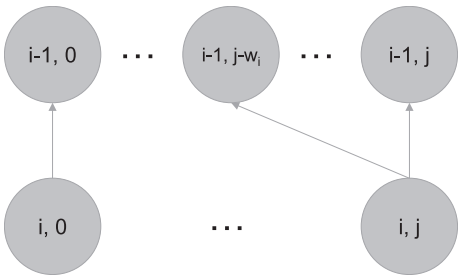

Fig. 11. The DAG pattern for 0/1 Knapsack problem.

```
public class KnapsackDag[T] extends Dag[T] {

    public def getDependency(i:Int, j:Int):Rail[VertexId] {
        if( i == 0n || j == 0n ) {
            return new Rail[VertexId](0);
        } else {
            if( Knapsack.weight(i-1) <= j )
                return [ new VertexId(i-1n, j),
                 new VertexId(i-1n, j-Knapsack.weight(i-1)) ];
            else
                return [ new VertexId(i-1n, j) ];
        }
    }

    public def getAntiDependency(i:Int, j:Int):Rail[VertexId] {
        if ( i == 0n ) {
            return [ new VertexId(i+1n, j) ];
        } else if( i == Knapsack.ITEM_NUM ) {
            if( j+Knapsack.weight(i-1) > Knapsack.CAPICITY )
                return new Rail[VertexId](0);
            else
                return [ new VertexId(i, j+Knapsack.weight(i-1)) ];
        } else {
            if( j+Knapsack.weight(i-1) > Knapsack.CAPICITY )
                return [ new VertexId(i+1n, j) ];
            else
                return [ new VertexId(i+1n, j),
                 new VertexId(i, j+Knapsack.weight(i-1)) ];
        }
    }
}
```

Fig. 12. The implementation of 0/1 Knapsack problem's DAG Pattern.

We can conclude the DAG pattern from the recursive formulation, as shown in Fig. 11. Its DAG pattern class is shown in Fig. 12. The **KnapsackDag** class inherits from **Dag** class. In **getDependency** method, each vertex specifies its dependencies. Vertices $(0, j)$ and $(i, 0)$ are initialized with zero and have no dependencies so we return an empty list. Based on the Eq. (2), two vertices $(i-1, j)$ and $(i-1, j-w_i)$ are returned if the bag can carry the ith item or one vertex $(i-1, j)$ is returned if the capacity is not enough for the ith item. As in **getAntiDependency** method, vertices $(i+1, j)$ and $(i+1, j+w_i)$ are returned if $0 < i < n$ and $j < W$. And another three boundary conditions are considered as following.

- return an empty list, if $i = 0$ or $i = n, j + w_i > W$
- return vertex $(i+1, j)$, if $0 < i < n, j + w_i > W$
- return vertex $(i+1, j+w_i)$, if $i = n, j + w_i <= W$

Due to space limitations, the compute method and appFinished method are skipped.

6. 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.

Table 1
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

- 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 DAG pattern of four algorithms are shown in Figs. 6(b), (a), (d) and 11 respectively.
The time for initializing the cluster, generating test graphs, and verifying results were not included in the measurements.

6.1. 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 1. 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 0/1KP is not provided by DPX10. So users need to implement it as we discussed in Section 5.2, which costs 45 extra lines. Even though, implementing a custom DAG is much easier since it doesn't involve any parallel programming.

6.2. Scalability

As an indication of how DPX10 scales with places, Fig. 13 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. Figs. 13(a)–(c) reveal a speedup of about 4 for a 5 fold increase in nodes and Fig. 13(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 Fig. 14. LPS spend the minimum time since nearly a half of its vertices are not computed as its DAG shows (Fig. 6(d)). 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.

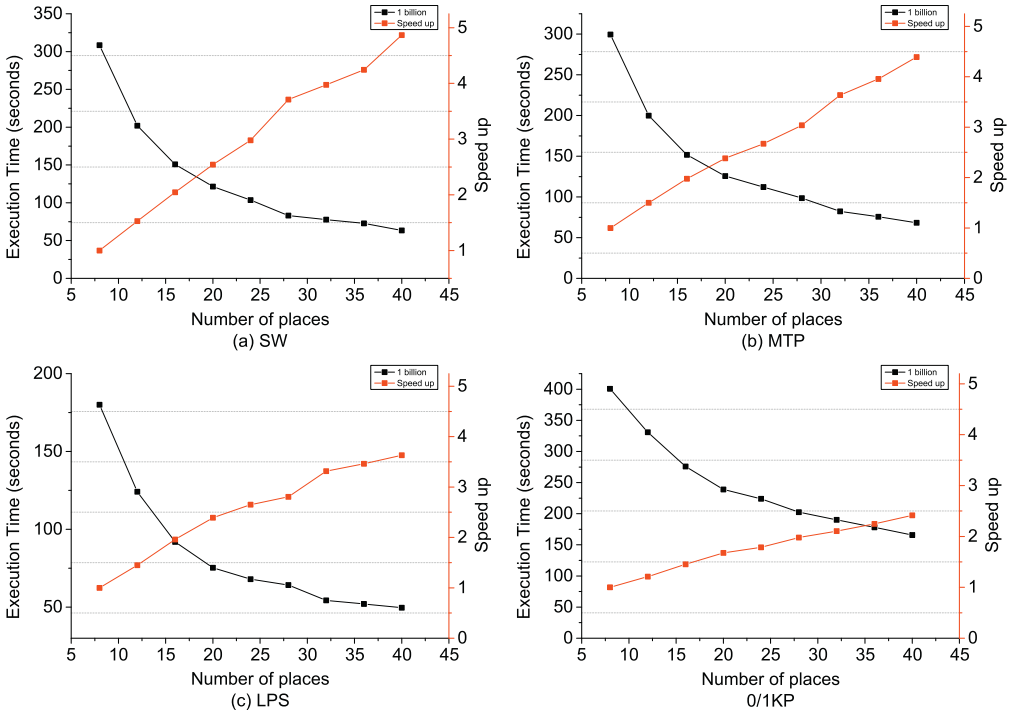

Fig. 13. Execution time of four DP applications with 1 billion vertices on different number of places (up to 40 places on 20 nodes). Figures (a,b,c) show a speedup of about 4 for a 5 fold increase in nodes and Figure (d) represents a speedup of about 2.5.

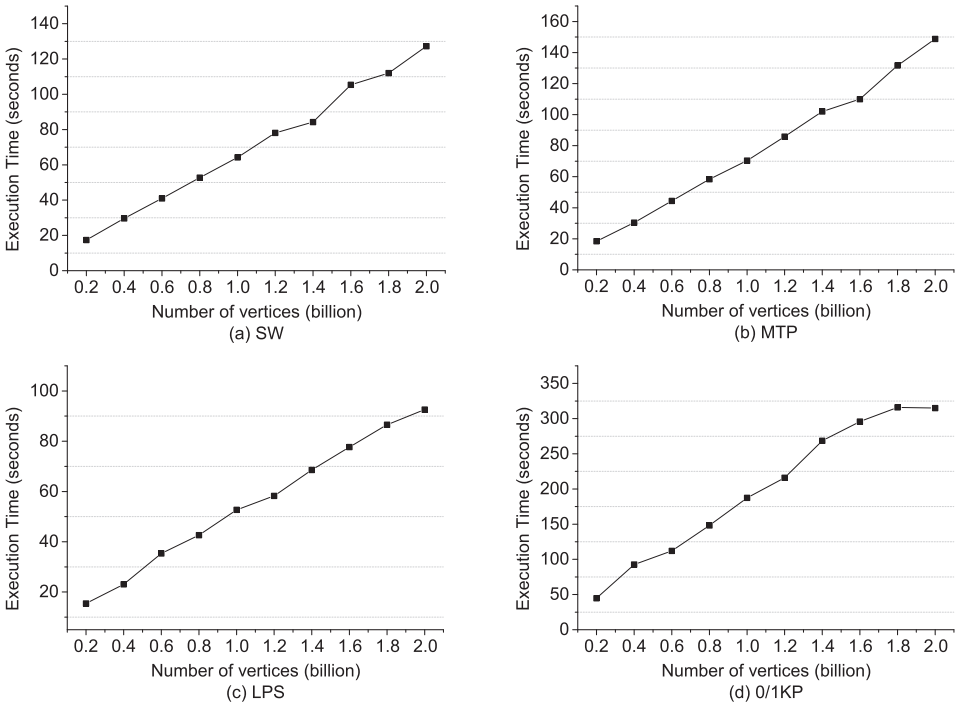

Fig. 14. Execution time of four DP applications on 20 nodes (240 cores) with the number of vertices varying from 200 million to 2 billion.

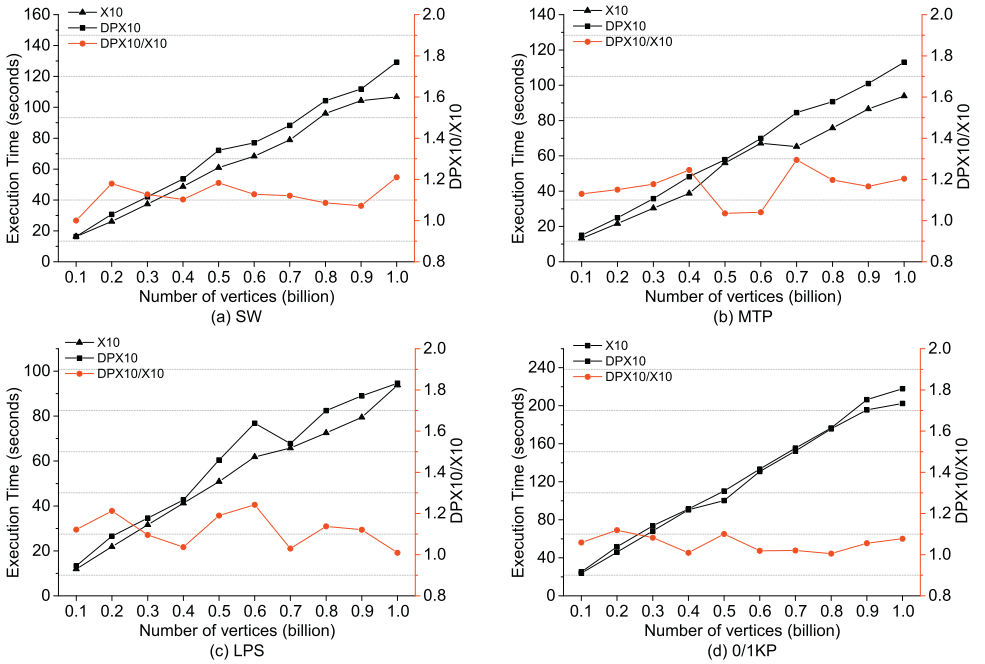

Fig. 15. Execution time of four DP applications implemented by DPX10 and X10 on 10 nodes. The compared DPX10/X10 rate is between 1.0 to 1.2 in all four applications.

6.3. Overhead

To provide an easy-to-use interface and automatically handle the parallel complexity, a little sacrifice of the performance is expected. But the balance between performance and simplicity must be maintained properly. The overhead of DPX10 mainly attributes to DAG operations, worker management, fault tolerance mechanism, etc.

To evaluate DPX10's overhead, we implemented the same four applications with native X10 and compared them with DPX10's implementation. For the sake of simplicity and fairness, the cache list was not used, and other configurations were set to the same.

We ran the programs on 10 nodes varying in graph sizes from 100 million to 1 billion. Fig. 15 shows that the native X10 version slightly outperforms DPX10's implementation. Moreover, the DPX10/X10 rate is about 1.0 to 1.2 in all four applications, which indicates that the overhead of DPX10 is acceptable.

6.4. Performance impact of varying cache sizes and block sizes

Different cache sizes. In the process of DPX10 computations, workers need to communicate with others to get results of dependent vertices. To alleviate the costs of frequent communications, in DPX10, we adopt a fixed-length FIFO cache mechanism as discussed in Section 4.2.3. It is based on the observation that adjacent vertices may rely on the same vertices, implying that the network communication can be reduced for adjacent vertices on the same machine by caching those same vertices. Users can set the length of the cache list. We use the SW algorithm with 1 billion vertices running on 20 nodes to evaluate the performance impact of different cache sizes. Fig. 16(a) shows the result. When cache is disabled (size of zero), we get the worst performance. The minimum time is achieved by setting the cache size to 100. After that, the execution time grows gradually since workers have to traverse the cache list every time they compute a vertex.

Different block sizes. Vertices of a DP DAG can be split into blocks. And each block has the same dependency relationship as their vertices. Instead of scheduling each vertex, workers treat blocks as scheduling units. The latter approach is usually called a block-based or a coarse-grained parallelization method, whereas the former one is a fine-grained approach. We conducted an experiment for SW algorithm with different dimensions of blocks. The program contained 1 billion vertices and ran on 20 nodes. The result is shown in Fig. 16(b), the best performance is achieved by setting the block size to 100×100. It shows us that either too large or too small block sizes make a poor performance. There tends to be a medium value that produces the optimal performance. The reason is that the block-based method can greatly reduce the time of communications but result in less parallelism.

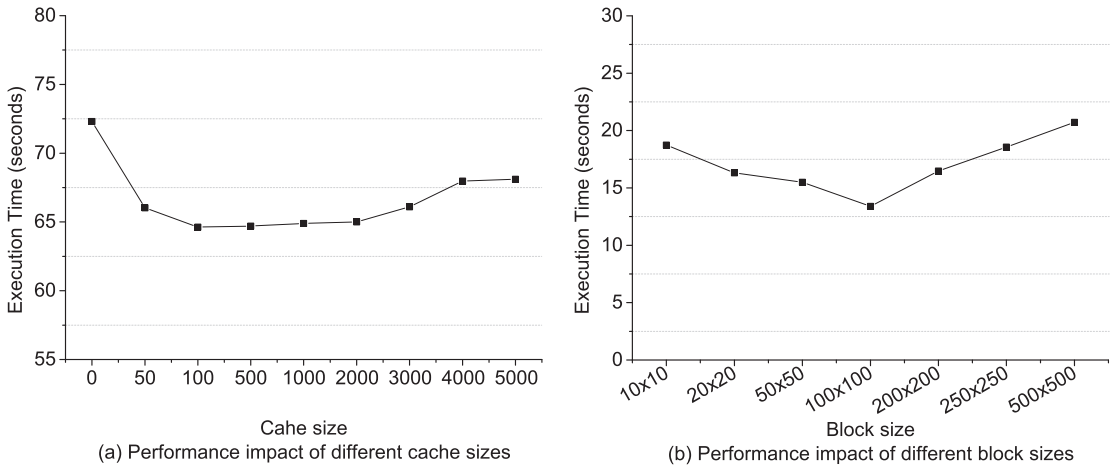

Fig. 16. Performance impact of different cache sizes and block sizes.

Table 2
Execution time of DPX10 and Z-align.

Number of vertices	DPX10 fine-grained	DPX10 block-based	Z-align
200M	17.355	4.746	2.091
400M	29.680	5.594	3.288
600M	40.952	8.615	4.41
800M	52.751	10.602	5.603
1G	64.264	13.401	6.836

6.5. Compare to a native MPI-based approach

In this section, we compare the SW algorithm written with DPX10 to Z-align [27], which is a specifically optimized SW program based on MPI. Although there are other work [28–31] on accelerating SW algorithm, they are either not pairwise algorithms or designed for specific devices like FPGA, GPU. Our SW program like Z-align, concentrates on pairwise (seq $\times$ seq) sequence alignment problems.

Table 2 shows the comparison between DPX10 Fine-Grained, DPX10 Block-based and Z-align running on 20 computing nodes. For the Block-based DPX10 implementation, the cache size was set to 100 and the block size was set to 100 $\times$ 100. As shown in the table, both Z-align and DPX10 follow a good scalability. However, Z-align is twice as fast as block-based DPX10 and eight times faster than fine-grained DPX10, which is an acceptable result since Z-align is an application-specific algorithm and DPX10 is a general framework that introduces some overhead due to its extra methods calling, straggler mechanism, fault tolerance, and etc. As we discussed in Section 6.3, we have to compromise some performance to be a general and easy-to-use framework.

6.6. 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.

Fig. 17 shows three normal circumstances of node failures. The first one illustrates a scenario of a node failure before the computation start. As shown in Fig. 17(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 Fig. 17(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. Fig. 17(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

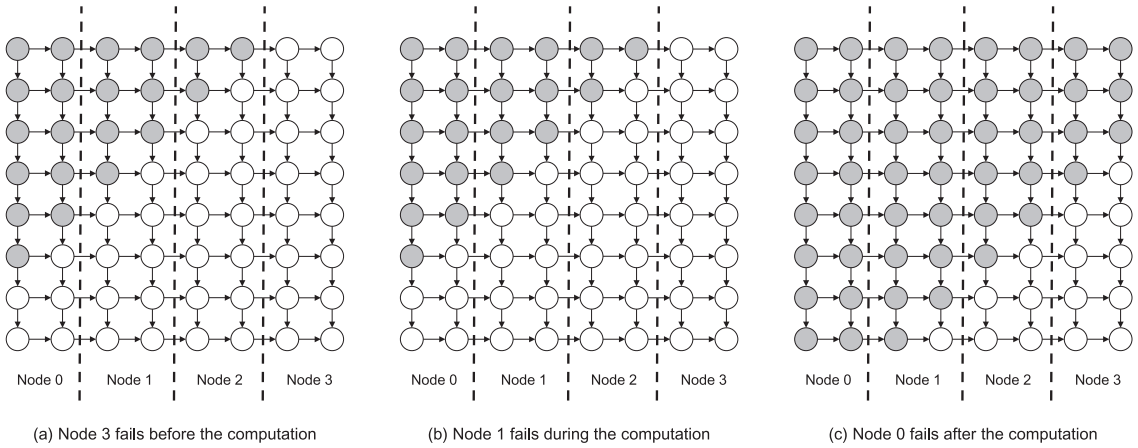

Fig. 17. Three different circumstances of node failures. Gray vertices are finished.

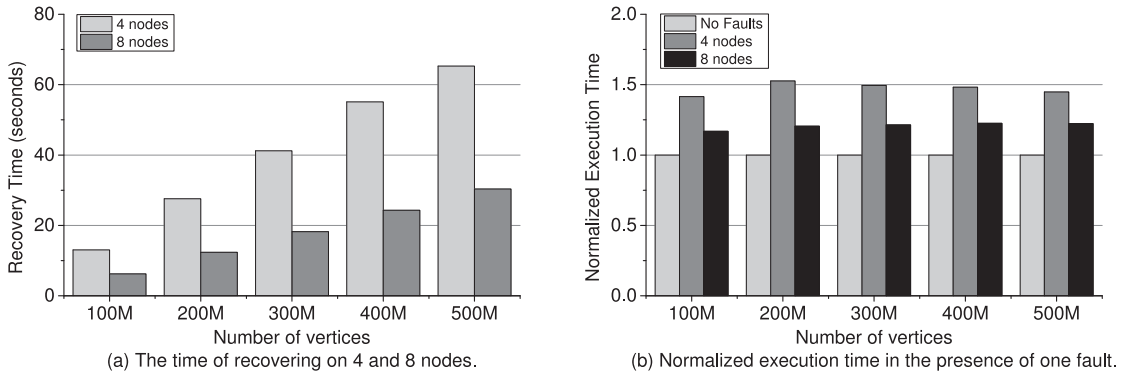

Fig. 18. The fault tolerance evaluation results with SW algorithm running on 4 and 8 nodes.

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.

Fig. 18 (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 4.2.4.

For one fault injection, Fig. 18(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.

6.7. 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. Fig. 20 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-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 result is shown in Fig. 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,

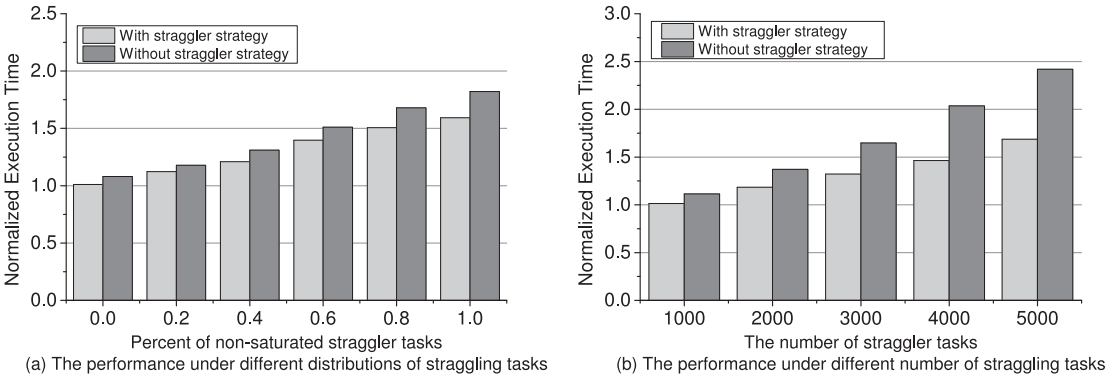

Fig. 19. The compared performance results for DPX10 with straggler strategy to that without straggler strategy.

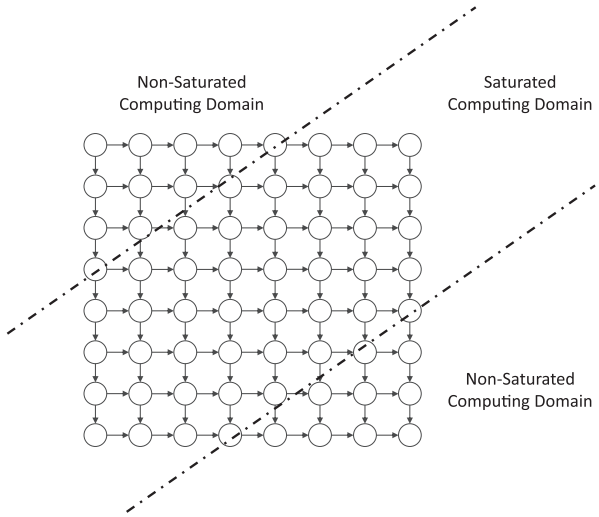

Fig. 20. The computing distribution model for a regular DP algorithm.

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 Fig. 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.

7. Related work

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

7.1. Graph processing framework

Hadoop [25] is an open source implementation of MapReduce [32]. 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 [17] 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 [17]. 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.

7.2. DP parallelization

There are an abundant of literature work for DP parallelization. Zheng et al. [40] introduced parallel DP based on stage reconstruction and then applied it to solve the optimized operation of cascade reservoirs. Hamidouche et al. [41] proposed a parallel BSP (Bulk Synchronous Parallel) strategy to execute Smith–Waterman algorithm on multiple multicore and many-core platforms. The hardware like GPU and FPGA has also been used in work [28,29,42] 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. [4] 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. The closest match to DPX10 is EasyPDP [5]. It's a parallel dynamic programming runtime system designed for computational biology. The biggest limitation is that EasyPDP can only run on a single node. Moreover, only one thread is used to schedule tasks, which can be a bottleneck when it comes to a lot of tasks. To address this issue, we distribute vertices among all places, and each place has a worker that is responsible for scheduling the local vertices.

7.3. 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 [43]. 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 [44], UPC++ [45], Co-Array Fortran [46], Global Arrays [47], SHMEM [48], etc. APGAS model permits both local and remote asynchronous task creation [10]. Two programming languages that use this model are Chapel [13] and X10 [9].

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 [49]. Acacia [50] 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 [51] 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 [52].

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 [49].

8. Conclusion and future work

This paper proposes DPX10, a simple and powerful abstraction for DP applications as well as an X10 implementation of this abstraction. DPX10 uses both shared memory model and distributed memory model to achieve high utilization of the distributed hardware environment. DPX10 lets developers easily create distributed DP applications without requiring them to master any concurrency techniques beyond being able to choose or draw a DAG pattern of their algorithms. The details of DP parallelization include vertices distribution, tasks scheduling and tasks communication are hidden from users and taken care of by the framework. Two representative applications are given to describe how to write a DP program with DPX10. Moreover, DPX10 provides a new recovery method for the distributed DAG which is more efficient than the periodical snapshot mechanism provided by X10.

We have compared DP programs written with DPX10 and with X10 directly to show the simplicity of DPX10. We have demonstrated excellent scalability and high efficiency for four DP algorithms with the number of vertices varying from 200 million to 2 billion. We've also evaluated the overhead of DPX10 and the performance of the new recovery method for distributed arrays.

Currently, the entire computation state resides in RAM. We are working on spilling some data to local disk to handle larger scale of DP problems. Some sophisticated scheduling and cache techniques are considered to be developed to improve efficiency and to support more scenarios [53,54].

Finally, the DPX10 source code is publicly available for downloading at http://github.com/wangvsa/DPX10/.

Acknowledgment

This work was supported by National Natural Science Foundation of China (No.61303021).

References

[1] T. Rognes, E. Seeberg, Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors, Bioinformatics 16 (8) (2000) 699–706.
[2] M. Farrar, Striped Smith–Waterman speeds database searches six times over other SIMD implementations, Bioinformatics 23 (2) (2007) 156–161.
[3] G. Zhao, C. Ling, 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.
[4] S. Maleki, M. Musuvathi, T. Mytkowicz, Parallelizing dynamic programming through rank convergence, in: ACM SIGPLAN Notices, 49, ACM, 2014, pp. 219–232.
[5] S. Tang, C. Yu, J. Sun, B.-S. Lee, T. Zhang, Z. Xu, H. Wu, Easypdp: an efficient parallel dynamic programming runtime system for computational biology, Parallel Distrib. Syst., IEEE Trans. 23 (5) (2012) 862–872.
[6] J. Du, C. Yu, J. Sun, C. Sun, S. Tang, 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.
[7] A. Itzkovitz, A. Schuster, Shared memory model.
[8] B. Murdock, Learning in a distributed memory model, in: Current Issues in Cognitive Processes: The Tulane Flowerree Symposium on Cognition, 2014, pp. 69–106.
[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, V. Sarkar, X10: an object-oriented approach to non-uniform cluster computing, Acm Sigplan Notices 40 (10) (2005) 519–538.
[10] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, O. Tardieu, The asynchronous partitioned global address space model, in: The First Workshop on Advances in Message Passing, 2010, pp. 1–8.
[11] R. Rabenseifner, G. Hager, 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.
[12] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, 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.
[13] D. Callahan, B.L. Chamberlain, 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.
[14] 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.
[15] D. Grove, O. Tardieu, D. Cunningham, B. Herta, I. Peshansky, V. Saraswat, A performance model for X10 applications: what's going on under the hood? in: Proceedings of the 2011 ACM SIGPLAN X10 Workshop, ACM, 2011, p. 1.
[16] Z. Galil, K. Park, Parallel algorithms for dynamic programming recurrences with more than o(1) dependency, J. Parallel Distrib. Comput. 21 (2) (1994) 213–222.
[17] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, 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.
[18] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed GraphLab: a framework for machine learning and data mining in the cloud, Proc. VLDB Endowment 5 (8) (2012) 716–727.
[19] R.R. McCune, T. Weninger, G. Madey, Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing.
[20] S. Spacey, W. Luk, P.H. Kelly, D. Kuhn, Improving communication latency with the write-only architecture, J. Parallel Distrib. Comput. 72 (12) (2012) 1617–1627.
[21] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B.R. de Supinski, 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, 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, M.A. Wong, Algorithm AS 136: A K-Means clustering algorithm, J. R. Stat. Soc. Series C (Applied Statistics) 28 (1) (1979) 100–108.
[24] M. Zaharia, A. Konwinski, A.D. Joseph, R.H. Katz, I. Stoica, Improving MapReduce performance in heterogeneous environments, Oper. Syst. Design Implementation 8 (4) (2008) 7.
[25] K. Shvachko, H. Kuang, S. Radia, 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, J.-S. Su, The TianHe-1A supercomputer: its hardware and software, J. Comput. Sci. Technol. 26 (3) (2011) 344–351.
[27] A. Boukerche, R.B. Batista, A.C.M.A. De Melo, Exact pairwise alignment of megabase genome biological sequences using a novel z-align parallel strategy, in: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, IEEE, 2009, pp. 1–8.
[28] 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.
[29] J. Singh, 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] Y. Yamaguchi, H.K. Tsoi, W. Luk, FPGA-based Smith–Waterman algorithm: analysis and novel design (2011).
[31] J. Allred, J. Coyne, W. Lynch, V. Natoli, J. Grecco, J. Morrissette, Smith–Waterman implementation on a FSB-FPGA module using the intel accelerator abstraction layer(2009).
[32] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

[33] J. Ekanayake, S. Pallickara, 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, M.D. Ernst, HaLoop: efficient iterative data processing on large clusters, Proc. VLDB Endowment 3 (1-2) (2010) 285–296.
[35] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, Eur. Conf. Comput. Syst. 41 (3) (2007) 59–72.
[36] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, DAGuE: A generic distributed DAG engine for high performance computing, Parallel Comput. 38 (1) (2012) 37–51.
[37] D.G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, S. Hand, CIEL: A universal execution engine for distributed data-flow computing., NSDI, 11, 2011. p. 9–9.
[38] E.H. Rubensson, E. Rudberg, Chunks and Tasks: a programming model for parallelization of dynamic algorithms, Parallel Comput. (2013).
[39] J.M. Wozniak, T.G. Armstrong, K. Maheshwari, E.L. Lusk, D.S. Katz, M. Wilde, I.T. Foster, Turbine: a distributed-memory dataflow engine for high performance many-task applications, Fundamenta Informaticae 128 (3) (2013) 337–366.
[40] H. Zheng, Y. Mei, K. Duan, 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.
[41] K. Hamidouche, F.M. Mendonca, J. Falcou, A.C.M.A. de Melo, D. Etiemble, Parallel smith-waterman comparison on multicore and manycore computing platforms with BSP++, Int. J. Parallel Program. 41 (1) (2013) 111–136.
[42] M. Korpar, M. Šikić, SW#–GPU-enabled exact alignments on genome scale, Bioinformatics (2013) btt410.
[43] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, 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.
[44] T. El-Ghazawi, L. Smith, UPC: unified parallel c, in: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ACM, 2006, p. 27.
[45] Y. Zheng, A. Kamil, M.B. Driscoll, H. Shan, K. Yelick, UPC++: a PGAS extension for c++, in: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IEEE, 2014, pp. 1105–1114.
[46] R.W. Numrich, J. Reid, Co-Array Fortran for parallel programming, in: ACM Sigplan Fortran Forum, 17, ACM, 1998, pp. 1–31.
[47] J. Nieplocha, R.J. Harrison, 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.
[48] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, 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.
[49] M. Dayarathna, C. Houngkaew, 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.
[50] M. Dayarathna, 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.
[51] J. Labarta, StarSS: a programming model for the multicore era, in: PRACE Workshop New Languages & Future Technology Prototypes at the Leibniz Supercomputing Centre in Garching (Germany), 2010.
[52] E. Tejedor, M. Farreras, D. Grove, R.M. Badia, G. Almasi, 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.
[53] Y. Guo, J. Zhao, V. Cave, V. Sarkar, SLAW: a scalable locality-aware adaptive work-stealing scheduler, in: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, IEEE, 2010, pp. 1–12.
[54] O. Tardieu, H. Wang, H. Lin, A work-stealing scheduler for X10's task parallelism with suspension, ACM SIGPLAN Notices 47 (8) (2012) 267–276.