2015 44th International Conference on Parallel Processing

DPX10: An Efficient X10 Framework for Dynamic Programming Applications

Chen Wang, Ce Yu *, Jizhou Sun
School of Computer Science and Technology
Tianjin University
Tianjin, China
Email: {wangvsa,yuce,jzsun}@tju.edu.cn

Xiangfei Meng
College of Computer and Control Engineering, Nankai University
National Supercomputer Center in Tianjin
Tianjin, China
Email: mengxf@nscc-tj.gov.cn

***Abstract*—X10 language and Asynchronous Partitioned Global Address Space (APGAS) model is an emerging mechanism for programming high-performance computers and commodity clusters. However, little work exists on distributed programming framework for dynamic programming (DP) problems based on X10 and APGAS model. In this paper we present DPX10, an efficient distributed X10 framework for DP applications. DPX10 enables developers to write highly efficient DP programs without much effort. A DPX10 program is specified by a directed acyclic graph (DAG) pattern and a compute method for the vertices. DPX10 provides eight commonly used DAG patterns and a simple API to create custom patterns. The system handles all the tiresome work of implementing parallelization including DAG distribution, vertices scheduling, and vertices communication. Moreover, a new recovery method for distributed arrays is developed to provide transparent fault tolerance. We describe the design of the framework and use four DP applications with up to a billion vertices on 120 cores to demonstrate its simplicity, efficiency, and scalability.**

***Keywords*-X10; APGAS; dynamic programming; programming framework**

I. Introduction

In the advent of multi-core and multi-node hardware technologies, next-generation architectures are increasing not only in size but also in complexity. Parallel languages and programming models need to provide simple means for developing applications while not hindering the performance. In the last years, the research community has initiated different efforts to create a suitable and robust programming model for such architectures [1].

X10 is a modern object-oriented programming language in the PGAS family driven by the motto "Performance and Productivity at Scale". It extends the PGAS model with asynchrony (yielding the APGAS programming model). X10 and APGAS are quickly emerging to be an important mechanism for programming high-performance computers and commodity clusters.

For many computing expensive applications, dynamic programming (DP) is a practical and efficient solution. But the performance is limited due to the burgeoning volume of data, which makes parallelism necessary and crucial to keep the computation time at acceptable levels. However, the intrinsically strong data dependency of dynamic programming makes it difficult for programmers to write a correct and efficient distributed program.

Although there has been a great deal of parallel programming models or frameworks, little work exists aimed at DP problems [2]. And most of these frameworks have been developed using C/C++/Java programming languages following programming models such as MPI, MapReduce, and Bulk Synchronous Message Passing [3].

In this paper we present DPX10, an X10 framework that 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 developed in advance, which can cover a wide range of DP problems. Moreover, several DP applications are utilized to demonstrate DPX10's simplicity, efficiency and scalability.

Specifically, we provide the following major contributions:

1) We develop a simple 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) The framework handles all parallel details like task scheduling, data distribution and fault tolerance, which makes it easy for programmers to write a distributed DP application. Some simple DP algorithms (e.g., Smith-Waterman algorithm, 0/1 Knapsack problem) can be written with even fewer lines of code than their serial version.
3) DPX10 provides transparent fault tolerance based on our new recovery mechanism for distributed arrays instead of the periodicity snapshot method provided by X10.

The rest of the paper is structured as follows. Section II briefly describes the X10 language and APGAS model. Section III introduces the dynamic programming problems

* Ce Yu (yuce@tju.edu.cn): the corresponding author.

0190-3918/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPP.2015.96

and its classification. The model of the computation is discussed in section IV. Section V presents some important APIs in DPX10. Section VI discusses implementation issues, including the DAG pattern library, worker implementation, and fault tolerance. In section VII we present two demo applications to show the process of writing DP programs with DPX10. And in Section VIII we report our experimental evaluation of the system. We conclude in Section IX and X with a discussion of the related literature and of future research directions.

II. X10 And APGAS

In this section, we briefly describe the APGAS model and the X10 language constructs which have been used to develop DPX10. More information is available from X10 language specification [4] and from X10 web site **http://x10-lang.org**.

X10 is a high-performance, high-productivity programming language developed by IBM Research in collaboration with academic partners. It has been developed from the beginning with the motivation of supporting hundreds of thousands of application programmers and scientists with providing ease of writing HPC code [5]. Previous programming models use two separate levels of abstraction for shared-memory thread-level parallelism (e.g., pthreads, Java threads, OpenMP) and distributed-memory communication (e.g., JMS, RMI, MPI) which results in considerable complexity when trying to create programs that follow both the approaches [6].

APGAS model is used by X10 to address this problem. APGAS model introduces two key concepts: **places** and **asynchronous**. A place is a collection of data and worker threads operating on the data. Places are typically realized as operating system processes. So the number of places is usually equal to the number of processors. Asynchronous is fundamental to the language which denoted by the keyword **async**. The statement **async S** tells X10 to run S as a separate and concurrent **activity** which is similar to threads in the operating system.

The latest major release of X10 is X10 2.5.1. It has been constructed via source-to-source compilation to either C++ or Java (termed as Native X10 or Managed X10) [7]. Since we are more interested in performance rather than portability, the current version of DPX10 has been developed targeting 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 the user at launch time. The execution starts with the main method at Place (0).

III. The DP Problem

Dynamic programming algorithm is an important algorithm design technique in many scientific applications. DP problems are 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.

DP problems can be classified in terms of the matrix dimension and the dependency relationship of each cell on the matrix [8]: 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. DPX10 can also express the type of $2D/iD(i >= 1)$, nonetheless, the performance is less than satisfactory. We will address that in the future work.

IV. Programming Model

A DAG is defined to describe the DP algorithms. It 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.

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.

Many DP applications consisting of a set of data or tasks with the data dependencies are modeled as a DAG. Moreover, there are often some applications whose DAG diagrams are almost the same except for their sizes. In view of the reuse concept, we could make those frequently used DAGs as DAG patterns and establish a DAG pattern library to classify and store them. The library will be discussed in section VI-B.

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.

Within execution phase the vertices with zero-indegree compute in parallel, each executing the same user-defined **compute()** method that expresses the logic of a given

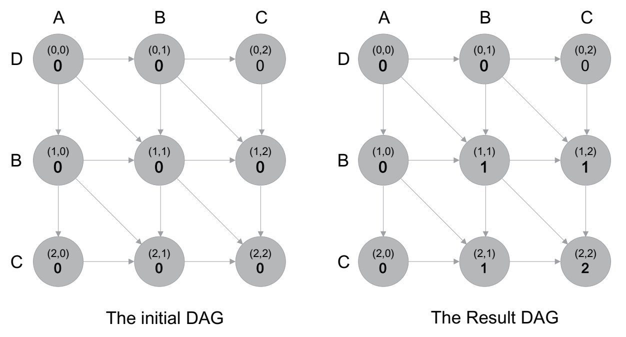

Figure 1. An example DAG of longest common substring problem.

```
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 2. The DPX10App and Vertex API foundations.

algorithm. And the indegree of other vertices will be decremented when their dependent vertices complete. When the program terminates the result of each vertex can be accessed using the system APIs, see section V.

Figure 1 illustrates these concepts using a simple example: given two strings *ABC* and *DBC*, find the longest common substring (LCS) of them. At initial phase the DAG is constructed and nine vertices are initialized with the value zero. The computation starts from the zero-indegree 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 using backtracking method to get the substring *BC*.

V. The DPX10 API

This section discusses the most important aspects of DPX10's API, omitting relatively mechanical issues.

Writing a DPX10 application involves implementing the predefined **DPX10App** interface (see Figure 2). Its template argument defines the value type associated with vertices. Each vertex has an associated computing result of the specified type. Limiting the graph state managed by the framework to a single value per vertex simplifies the main computation, distribution and fault tolerance.

The **compute()** method should be implemented by the user. It will be executed on each vertex at runtime. Parameter **(i, j)** pair is the unique identifier indicates which vertex is computing. Vertices communication is hidden from users, the dependency is resolved automatically by DPX10 and passed as the parameter **vertices**. The user can inspect the value associated with these vertices via **getResult()** method in **Vertex** class.

```
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 3. The DAG API foundations.

When the program terminates, the **appFinished()** method will be invoked where the final result should be processed. The argument **dag** can be used to access the result of each vertex.

A. DAG Pattern

The DAG pattern represents a series of DP algorithms which except the size, have the same DAG structure. All DAG patterns are a subclass of **DAG** class. Some important APIs of the DAG class are shown in Figure 3. 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 the DPX10 runtime to automatically resolve the dependencies. They need to be implemented by the user when creating a custom DAG pattern. The **getDependency()** method returns a list of identifiers represent the 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 be decremented when vertex **(i, j)** is finished.

VI. Implementation

DPX10 has been designed from ground-up with the aim of computing intensive DP problems. Its goal is to support efficient execution on multiple nodes and multiple cores without burdening the programmer with concurrency management. DPX10 consists of a DAG pattern library to represent a DP algorithm, some useful APIs and the runtime that handles distribution, scheduling and fault recovery. DPX10 does not depend on any third libraries.

A. Execution Overview

Figure 4 shows the overall flow of a DPX10 operation in our implementation. 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 exams vertices on each place and

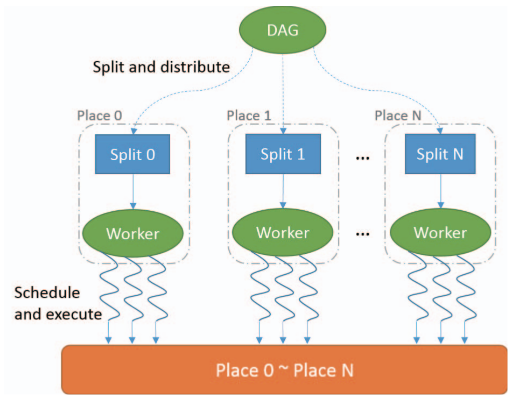

Figure 4. Logical flow of DPX10's execution.

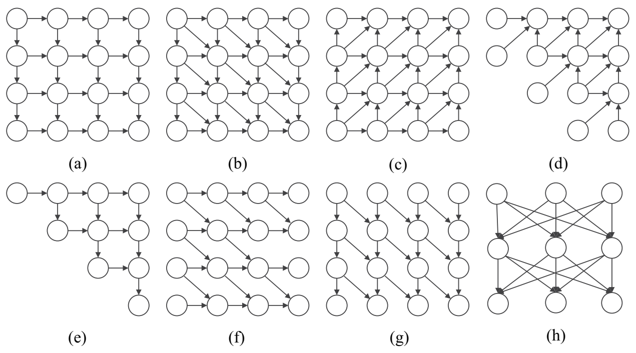

Figure 5. Eight built-in DAG patterns.

inserts those with zero-indegree into a local ready list to wait for scheduling.

2) DPX10 starts up one worker on each place. Each worker is responsible for scheduling local vertices and executing the user's **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.

B. DAG Pattern Library

The DAG pattern library is a major component in DPX10. It provides the built-in DAG patterns and exposes a simple API for users to create custom 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 Figure 5.

For example, the LCS algorithm we used in section IV is a classical DP algorithm. It's 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 m is the length of string x, n is the length of string y and $0 \leq i \leq m, 0 \leq j \leq n$; $F[i,0], F[0,j]$ have been initialized with zero.

Correspondingly, the DAG of LCS algorithm can be established based on the above formulation, as shown in Figure 5 (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-indegree are schedulable. In addition, a finish flag is kept for each vertex to identify its status and to help recover the result after a failure happens.

All vertices are stored in a distributed array (*DistArray* class) provided by X10. How to distribute them among the places can be flexibly defined by using a *Dist* structure. See the X10 document [4] for the details. By default vertices are spliced and distributed along with column.

C. The Worker Implementation

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

The scheduling strategy can be specified by the user. By default, we use a local scheduling strategy which execute the vertex on the local place. We also provided another two methods: random scheduling and minimum communication scheduling. The latter one calculates the total cost of communication for executing them in each place and choose the minimum one. This strategy introduces some extra overhead and should be used in appropriate scenarios. More scheduling methods will be developed in our future work.

When a vertex is going to be computed, the worker will spawn a new activity which is parallel with the current one. In this activity the worker first retrieves the dependent vertices using the **getDependency()** method that we discussed early, then pass them along with the identifier of the current vertex to the user-defined method **compute()**. So the user can implement the logic of the algorithm without considering dependencies and communications. After the compute method returned, the worker sets the result of the computing vertex and decrements the indegree of vertices which are relied on the current one. If the indegree goes to zero, the vertex will be inserted to the ready list on its local place. Finally the worker marks the vertex as finished and increases the *finished vertices counter*.

The dependency vertices that 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 using a static array and its size can be specified by the user. 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 in a short period.

D. Fault Tolerance

Fault tolerance is important because hardware and software faults are ubiquitous [9]. 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 use 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 [10]. 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 the user usually need all data to compute the final result. The second method is often adopted in iterative computations that in each step the master dispatches the tasks to the worker. The master maintains the computation status and the intermediate results. Once a worker node fails, the master can dispatch the tasks to the remaining workers. But this method is not fit for DPX10 too. The reason is that every worker in DPX10 holds a partition of the DAG and is responsible for scheduling the local vertices. The last method uses the 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* [10]. However the periodic snapshot mechanism is infeasible because a large volume of intermediate results may be produced in the progress of computing. Therefore, we adopt a new method.

Once a *DeadPlaceException* raises, the program will be paused and enter the recovery mode. DPX10 will create a new distributed array among the remaining places and restore the result of the finished vertices from the alive places. By default the result of remote vertices will be discarded since it may take less time to recompute them rather than copy them across the network. The user can change this behavior if the computation is more time-consuming than the communication. All unfinished vertices in the new array will be initialized (reset the indegree). After the recovery, the old distributed array will be replaced by the new one and the program will continue.

For instance, 12 vertices (3 rows and 4 columns) that were divided by the row and distributed into 3 places (1,2,3), as shown in Figure 6 (a). The vertices with gray background indicate that they were completed. At some point place 3 fails, then DPX10 reassigns all vertices to place 1 and place 2 as shown in Figure 6 (b). The result of vertices (1,1), (1,2)

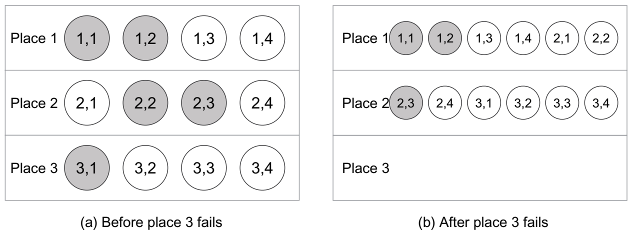

Figure 6. An example of recovery process.

and (2,3) are reserved. Notice that the result of vertex (2,2) is dropped because it was stored on the remote place (place 2).

After reassigns and recovers all the vertices, the system initializes the unfinished vertices and reset the indegree of them. Finally, DPX10 exits the recovery mode and continues the program.

The recovery process is executed in parallel on all alive places. Unfortunately, there is a limitation in the current version of Resilient X10 that its execution will be aborted if Place 0 is dead [10]. This is an issue needs to be addressed in their future work.

E. Refinements

Although the basic functionality provided by default is sufficient for some needs, we have provided a few useful extensions.

- **Distribution of DAG**: the user can define the partition and distribution of the DAG using a *Dist* structure to realize a better locality.
- **Initialization of DAG**: the default initialization method can be override to initialize the vertices on demand such as set the unneeded vertices as finished.
- **Scheduling strategy**: three scheduling methods are provided: local scheduling, random scheduling, and minimum communication scheduling.
- **Cache size**: the size of the cache list on each worker can be specified to achieve maximum benefit.
- **Restore manner**: by default the result of the finished vertices on the remote places will be abandoned during recovery. But the user can tell DPX10 to restore them if the computation is more time consuming than data transfer.

VII. Applications

This section uses two simplified DP applications to demonstrate the process of writing DP programs with D-PX10.

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 custom one.
2) Inherit from the **DPX10App** class, implement the **compute()** and **appFinished()** method.

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 normal X10 programs.

A. 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:

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

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

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

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 $a[1 \dots i]$ and a suffix of $b[1 \dots j]$

The DAG pattern is the same as the LCS algorithm, which is already provided by the DAG pattern library, as shown in Figure 5 (b). An implementation of Smith-Waterman algorithm is shown in Figure 7, 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 return the maximum one based on the Equation (1). The dependent vertices are provided as parameter **vertices**. For example, when computing $(2,2)$ the **vertices** is a list of vertices $(1,1), (2,1), (1,2)$.

The result processing code is omitted which may be a backtracking method to print out the best match.

```
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) {
                    lefttop = vertex.getResult();
                    lefttop += str1.charAt(i)==str2.charAt(j) ?
                        MATCH_SCORE : DISMATCH_SCORE;
                }
                if(vertex.i==i-1n && vertex.j==j)
                    top = vertex.getResult() + GAP_PENALTY;
                if(vertex.i==i && vertex.j==j-1n)
                    left = vertex.getResult() + GAP_PENALTY;
            }
        }
        return max(lefttop, left, top);
    }

    public def appFinished(dag:Dag[Int]):void {}
}
```

Figure 7. Smith-Waterman algorithm implemented in DPX10.

B. 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 $\sum w_i x_i \le W$, 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 V.

Assume $w_1, w_2, \cdots, w_n, W$ are strictly positive integers. Define $m(i,j)$ to be the maximum value that can be attained with 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:

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

We can conclude the DAG pattern from the recursive formulation, as shown in Figure 8. Its DAG pattern class is shown in Figure 9. The **KnapsackDag** class inherits from **Dag** class. In **getDependency()** method, each vertex specify 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 Equation (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.

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

VIII. Experiments

In this section, we evaluate the performance of DPX10 running a variety of DP applications on Tianhe-1A [11]. Each computing node of Tianhe-1A system is a multi-core SMP server which has dual 2.93Ghz Intel Xeon 5670

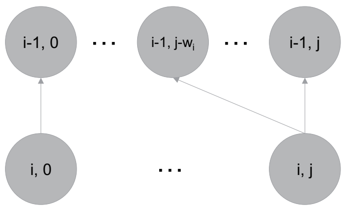

Figure 8. The DAG pattern for 0/1 Knapsack problem.

```
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)) ];
        }
    }
}
```

Figure 9. The implementation of 0/1 Knapsack problem's DAG Pattern.

six-core processors (total 12 cores per node/24 hardware threads). Each node has 24GB memory and 120GB SSD and are connected with Infiniband QDR. The Kylin Linux system is deployed. We used latest X10 release version, X10 2.5.1. The X10 distribution was built to use Socket runtime.

Two important environment variables need 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 our experiments. And *X10_NPLACES* was twice the number of computing nodes used in the experiment.

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

The Smith-Waterman algorithm and Knapsack problem were already discussed. The recursive formulation of another two applications are 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 DAG pattern of four algorithms are shown in Figure 5 (b),(a),(d) and Figure 8, respectively.

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

A. Scalability

As an indication of how DPX10 scales with computing nodes, Figure 10 shows the runtime for SWLAG, MTP, LPS, and 0/1KP with 300 million vertices. The number of computing nodes is increased from 2 to 12.

As you can see, the time goes down quickly at first and then reaches a plateau as the number of nodes increases. The increase of the nodes can reduce the time for executing the non-dependent vertices but can also increase the time for data transmission. Because of the strong data dependency, the speedup curves are not ideal. Figure 10 (a) to Figure 10 (c) reveal a speedup of about 4 for a 6 fold increase in nodes and Figure 10 (d) represents a speedup of about 3. In other words, SWLAG, MTP and LPS have a better acceleration performance than 0/1KP. One reason is 0/1KP has nondeterministic dependencies. And another reason is that given the same data distribution (divided by row), 0/1KP requires more communications due to its dependency relationship between vertices.

To show how DPX10 scales with graph sizes, we kept the number of nodes unchanged (10 nodes) and varied the size of vertices from 100 million to a billion. The result is shown in Figure 11. 0/1KP takes 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 linear scalability with the graph size.

B. Overhead

To provide an easy use of interface and automatically handle the parallel complexity, a little sacrifice of the performance is inevitable. But the balance between performance

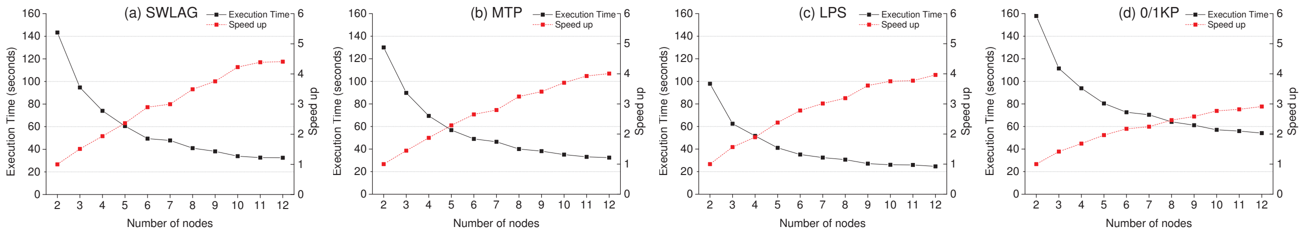

Figure 10. Execution time of four DP applications with 300 million vertices on different number of nodes (up to 144 cores/12 nodes). Figures (a,b,c) show a speedup of about 4 for a 6 fold increase in nodes and Figure (d) represents a speedup of about 3.

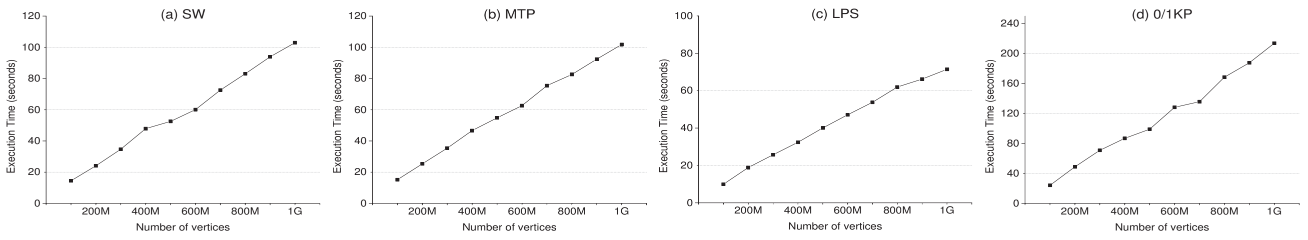

Figure 11. Execution time of four DP applications on 10 nodes (120 cores) with the number of vertices varying from 100 million to 1 billion.

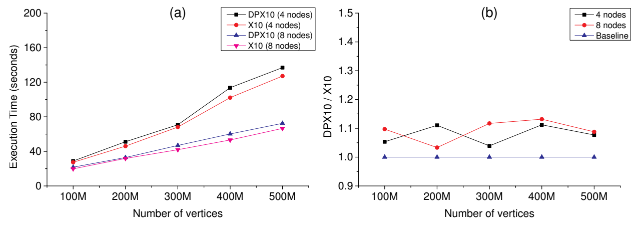

Figure 12. (a) Execution time of SWLAG algorithm implemented by DPX10 and X10 on 4 and 8 nodes; (b) The compared DPX10/X10 rate results on 4 and 8 nodes.

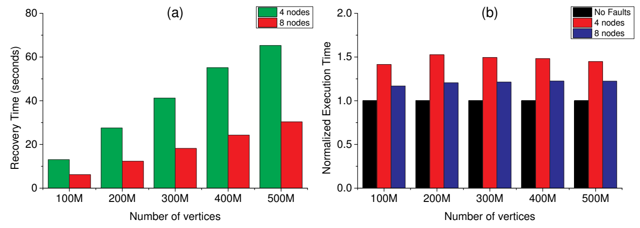

Figure 13. (a) The time of recovering on 4 and 8 nodes; (b) Normalized execution time in the presence of one fault on 4 and 8 nodes.

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 SWLAG algorithm with native X10 and compared it 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 two programs on 4 and 8 nodes varying in size from 100 million to 500 billion vertices. Figure 12 (a) shows that the X10 version slightly out performs DPX10's implementation. And the DPX10/X10 rate is about 1.02 to 1.12 as shown in Figure 12 (b), which indicates that the overhead of DPX10 is negligible.

C. Recovery

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 *Dead-PlaceException* and starts the recovery process.

In this section, we evaluate the expense of fault tolerance using SWLAG algorithm on 4 and 8 nodes with the number of vertices varying from 100 million to 500 million. The failure was triggered manually in the middle of the execution. The program continued on the remaining nodes after the recovery. So half of the vertices were computed on 4 and 8 nodes, and more than half of them were computed on 3 and 7 nodes.

Figure 13 (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 VI-D.

For one fault injection, Figure 13 (b) presents the normalized execution time. It is apparent that the impact of one failure reduces with the increase of the number of computing nodes.

IX. Related Work

In this section we discuss existing related projects. Most of them are designed on top of MPI, MapReduce, etc. Existing work on programming framework for X10 and APGAS model is scarce.

MapReduce [12]–in particular its open source implementation, Hadoop [13] is a popular platform for batch-oriented jobs, namely large-scale text mining for information retrieval [14]. 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 [15]. HaLoop extends Hadoop with the ability to evaluate a convergence function on reduce outputs [16]. But neither CGL-MapReduce nor HaLoop provides fault tolerance across multiple iterations. Moreover, the data flow of these systems is limited to a bipartite graph, which can not 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 express the logic of a given algorithm [17]. DPX10 has a similar idea as Pregel, “think like a vertex”. But other aspects like programming model, worker implementation, fault tolerance, etc. are totally different. Most of all, DPX10 and Pregel are targeting on different problems.

There are also some general-purpose DAG engine like Dyrad [18], DAGue [19] and CIEL [14]. 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 need 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 [20], [21] have proposed a task-based programming model. They mainly focus on the coarse-grain applications where each task normally spends a long time. Therefore, they are not suitable for computing-intensive DP problems, which consist of a great amount of computing tasks but each of them has a relatively short execution time.

The closest match to DPX10 is EasyPDP [2]. It is a parallel dynamic programming runtime system designed for computational biology. The most 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.

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 to non-PGAS alternatives, ScaleGraph defines concrete, simple abstractions for representing massive graphs [3]. Acacia [22] 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 [23] member designed to execute on clusters of SMPS. The tasks of ClusterSs are asynchronously created and assigned to the available resources with the support of the APGAS runtime [1].

Since X10 and APGAS is new for HPC community, we believe a lot of libraries or frameworks need to be developed to support the language to achieve it’s productivity goals [3].

X. Conclusion And Future Work

X10 language and APGAS programming model are growing in popularity and importance as they provide high productivity and allow for the best utilization of hardware features. Still, little work addresses dynamic programming models for X10 and APGAS.

The major contributions of this paper include a simple and powerful abstraction for DP applications and an X10 implementation of this abstraction. 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. Two demo applications are given to describe how to write a DP program with DPX10. DPX10 provides a new recovery method for the distributed DAG which is more efficient than the periodical snapshot mechanism provided by X10.

We have demonstrated excellent scalability and high efficiency for four DP algorithms with the number of vertices varying from 100 million to a billion. We also evaluate 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 enable computations on large scale of DP problems. Some sophisticated scheduling and cache techniques are considered to be developed to improve efficiency and to support more scenarios [24], [25]. Planned and ongoing work of DPX10 also includes developing more DAG patterns and implementing new demo applications.

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] 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.

[2] 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*, vol. 23, no. 5, pp. 862–872, 2012.

[3] 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.

[4] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, "X10 language specification," 2011.

[5] 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*, vol. 40, no. 10, pp. 519–538, 2005.

[6] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar, "May-happen-in-parallel analysis of x10 programs," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007, pp. 183–193.

[7] D. Grove, O. Tardieu, D. Cunningham, B. Herta, I. Peshansky, and 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.

[8] Z. Galil and K. Park, "Parallel algorithms for dynamic programming recurrences with more than o(1) dependency," *Journal of Parallel and Distributed Computing*, vol. 21, no. 2, pp. 213–222, 1994.

[9] 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.

[10] 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.

[11] 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*, vol. 26, no. 3, pp. 344–351, 2011.

[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] 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.

[14] 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*, vol. 11, 2011, pp. 9–9.

[15] 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.

[16] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[17] 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.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[19] 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*, vol. 38, no. 1, pp. 37–51, 2012.

[20] E. H. Rubensson and E. Rudberg, "Chunks and tasks: a programming model for parallelization of dynamic algorithms," *Parallel Computing*, 2013.

[21] 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*, vol. 128, no. 3, pp. 337–366, 2013.

[22] 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.

[23] 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.

[24] Y. Guo, J. Zhao, V. Cave, and 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.

[25] O. Tardieu, H. Wang, and H. Lin, "A work-stealing scheduler for x10's task parallelism with suspension," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 267–276, 2012.