Using the Cowichan Problems to Investigate the Programmability of X10 Programming System

Jeeva Paudel
Department of Computing Science
University of Alberta, Edmonton, Canada

José Nelson Amaral
Department of Computing Science
University of Alberta, Edmonton, Canada

ABSTRACT

In today's era of multicores and clustered architectures, high performance and high productivity are central concerns in the design of parallel programming languages that aim to solve large computational problems. X10 is a language based on state-of-the-art object-oriented programming ideas and claims to take advantage of their proven flexibility and ease-of-use to solve a wide spectrum of programming problems. The Cowichan problems are a set of computational problems that were designed to stress parallel programming environments and to assess their programmability. This paper uses Cowichan problems to assess the flexibility of X10.

1. INTRODUCTION

The decades-long shift in hardware trend from uniprocessors to multi-core processors for high performance has led to the design and implementation of several parallel programming systems. These systems primarily focus on enabling programmers to easily write parallel applications and quickly achieve high performance. Unfortunately, these systems lie at two extreme ends. Some of these systems support only specialized programming models, such as a shared memory model, and allow programmers to achieve high performance for only a selected class of applications. Clearly, such a performance is not portable across all classes of applications. Other systems provide specialized low-level language constructs to yield high performance over a wider range of applications, but incur huge overhead in the form of drastically increased software development time and effort.

X10 [2] is a recent addition to the family of parallel programming languages that aims to boost both productivity and performance on modern multi-core and clustered architectures. Although there have been some studies [4] assessing the productivity of the X10 programming system, we are not aware of any comprehensive assessment of X10's strengths and weaknesses using a set of parallel programming tasks designed to include diverse programming constructs. This paper reports on our study of X10's programmability using a suite of applications called the *Cowichan* problems [8, 10]. The specific contributions resulting from this work are:

1. an X10 implementation of the Cowichan problems,
2. a discussion of strategies used for their parallelization,
3. performance assessments based on different metrics including number of places, cores, threads, and problem sizes,
4. identification of strengths and weakness of the language, and
5. an experience report from using the X10 programming environment.

The rest of the paper is organized as follows: Section 2 provides a brief overview of the important language constructs and concepts in X10. Section 3 introduces the Cowichan problems. Section 4 details the experimental set up used for compilation and execution of the programs. Then, Section 5 describes each of these problems, and reports on the language's expressive capabilities along with performance analyses. Section 6 presents practical observations based on the programming experience, Section 7 compares and contrasts important related work and finally, Section 8 summarizes our findings.

2. OVERVIEW OF X10

X10 is a modern object-oriented language designed for high-performance, and high-productivity, programming of parallel and multi-core computer systems.

X10 is designed on top of the serial subset of Java, but replaces Java's lower level thread-based concurrency model with more flexible higher-level concurrency constructs such as `async`, `atomic` and `finish`. The full language reference manual is available at [7], what follows here is merely a brief overview of the key features of X10 necessary to understand the rest of the paper.

2.1 async <*stmt*>

The statement `async` <*stmt*> causes the parent activity to create a new child activity to execute <*stmt*> and return immediately, *i.e.* the parent activity can proceed immediately to its next statement.

2.2 finish <*stmt*>

The statement `finish` <*stmt*> causes the parent activity to execute <*stmt*> and then wait until all the sub-activities created within <*stmt*> have terminated globally. A statement being executed by an activity terminates locally when the activity completes all the computation related to that statement. Similarly, a statement terminates globally when itself, and any other activities that it may have spawned terminate globally.

2.3 atomic <*stmt*>

The statement `atomic` <*stmt*> causes <*stmt*> to be executed atomically, *i.e.* its execution occurs in an indivisible step during which <*stmt*> executes and terminates locally while all other concurrent activities in the same place are suspended. Compared to user-managed locking, the X10 user only needs to specify that a collection of statements should execute atomically and leaves the responsibility of lock management and alternative mechanisms for enforcing atomicity to the language implementation [7].

2.4 Places

A place is a repository of non-migrating mutable data objects and the activities that operate on the data. X10 uses places to induce the concept of locality. Every X10 activity runs in a place. The activities running in a place may access data items located at that place with the efficiency of on-chip access. Accesses to remote data may take orders of magnitude longer because they can only be accessed by dynamically spawning activities at places where the data reside.

Places are virtual – the mapping of places to physical locations is performed by a deployment step that is separate from the X10 program. Although objects and activities do not migrate across places in an X10 program, an X10 deployment may migrate places across physical locations based on affinity and load balance considerations.

2.5 Partitioned Global Address Space

X10 belongs to the family of partitioned global address space (PGAS) languages and supports the notion of private and shared memory. These languages deliver to the programmer the abstraction of a single address space even when the computation is running on a machine with distributed address spaces. For any data residing at its local memory, the processor is said to have affinity to the data. Each processor may operate directly only on the data that resides on its own address space and must use some indirect mechanism to access or update data at other processors.

3. THE COWICHAN PROBLEMS

The Cowichan problems form a standard test suite that is conceived specifically to determine the usability of programming systems for writing efficient parallel programs. Accordingly, there exist different implementations of the Cowichan problems in languages such as Orca [9] and UPC [6] to determine whether the language is a useful and an effective system for developing parallel programming solutions to computationally complex problems.

There are two different suites of Cowichan problems. The evaluation presented in this paper uses the original suite of six medium-sized realistic applications. Although these problems are more complex and take more time to implement than the "toy" problems in the other suite, such as prime number generation, they provide better insights into the programming capability, ease and efficiency of the language being evaluated. As a combined suite, these problems utilize a large variety of operations and techniques common in parallel programming as well as a number of memory reference patterns.

4. EXPERIMENTAL SETUP

This section describes the platform used for the performance assessment runs. The description also includes the compilation and execution environments used for evaluation.

4.1 Platform

The performance experiments were carried out in a multicore cluster environment consisting of a blade server with 16 nodes, each featuring two Quad-Core AMD Opteron processors (model 2350) clocked at 2 GHz, with 8 GB of RAM and 20 GB of swap space. The CentOS GNU/Linux version 5.2 operating system was used. Only 14 nodes were used for the assessment because one of the nodes in the server acts as a management node and another node is reserved for development activities.

The test programs were run in the cluster nodes in an environment where they did not have to compete for resources with other applications, excluding the system applications such as secure shell (ssh) and the system logger.

4.2 X10 compiler and its backend

The performance evaluation of X10 used the C++ backend of the X10 compiler. The Java backend was not used because it does not perform optimizations [11]. When using the C++ backend, it is important to use the right set of flags. The default compiler/build options are set to minimize the compile time, not to maximize the performance because the default set of options result in compiling with no optimizations. Therefore, it is necessary to pass the following two options to the X10C++ compiler for peak performance:

- -O to enable optimization
- -NO_CHECKS to disable array/rail bounds checking, null pointer checking, and place checking

The NO_CHECKS option was passed to the compiler during performance assessment only after the test programs compiled successfully in our platform for experimentation.

4.3 X10 runtime

X10RT is a library that an X10 program uses to communicate between *places*. There are three implementations of X10RT available for use, namely: *standalone*, *mpi*, and *pgas*. The *standalone* runtime is a trivial implementation that only supports single process (single place) execution of X10 programs. The *mpi* runtime is implemented on top of MPI2 and supports multiple places. The *pgas* runtime is implemented using the common PGAS runtime and also supports multiple places. All experiments use the default runtime chosen by X10: *pgas_sockets*.

4.4 Places and Processors

To run a program in multiple places on Linux, we first need to run `bin/manager`, and then invoke `bin/launcher -t` $< nplaces >$ `./` $< class\_name >$ to specify the desired number of places. In order to chose the nodes of interest in the cluster, and the number of processors to be used in each node, we used the *pbs_resources* available in our linux cluster.

5. THE COWICHAN PROBLEMS IN X10

This section describes the four problems used for the evaluation of X10 in this paper. This description is written after the presentation by Wilson *et al.* [8].

5.1 Turing Ring

The Turing Ring model describes a spatial system in which predators/preys interact within locations and migrate between locations. The predator/prey system evolves in a ring-shaped world. The system can be visualized as a model for a series of locations around a lake in which foxes and rabbits live. The following coupled differential equations model the interactions between predators and preys and their migration to neighboring locations.

$$\frac{dX_i}{dt} = X_i(r_x + c_{xx}X_i + c_{xy}Y_i) + \mu_x(X_{i+1} + X_{i-1} - 2X_i)$$
$$\frac{dY_i}{dt} = Y_i(r_y + c_{yx}X_i + c_{yy}Y_i) + \mu_y(Y_{i+1} + Y_{i-1} - 2Y_i) \quad (1)$$

where $r_{x,y}$ are birth rates, $C_{a,b}$ represent local interactions, $\mu_{x,y}$ are migration rates between neighboring locations, and the predator and prey populations in location i are represented by X_i, Y_i, respectively.

5.1.1 Sequential Solution

The simulation of the system's evolution discretizes the equations and updates the cell populations at small steps. The sequential solution uses the fourth-order Runge-Kutta method for solving these differential equations.

The sequential implementation of the Turing Ring problem is straightforward. The predator and prey populations are stored in different arrays. Then, the goal is to update the populations of each location, represented by an array index, using the initialization parameters, such as initial populations, migration, death and birth rates. Next, the algorithm iterates through the following steps until the population stabilizes:

1. receive migration information from neighboring locations from the previous iteration.
2. update populations of the ring using birth and death rates.
3. update population to account for migration, and determine which animals move to the neighboring locations.
4. send migration information to the neighboring nodes.

5.1.2 Parallelization Strategy

To parallelize the Turing Ring, the problem is decomposed in such a way that each processor receives the same number of locations in the ring and the locations assigned to a processor are contiguous. Since animals move only between adjacent locations, each processing node only needs to exchange information regarding its location with two other neighboring nodes, as shown by the connecting arrows in the Figure 1. Figure is adopted from the description of the parallelized Orca implementation of Turing Ring [9].

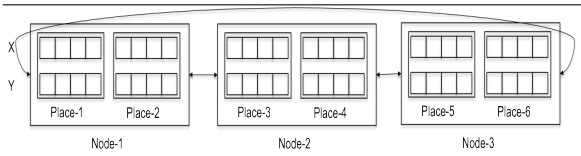

Figure 1: Parallelization of Turing Ring problem.

As shown in Figure 1, each place holds its part of the ring representing predator and prey population in two arrays. On each iteration, each place performs the four steps listed above for the sequential algorithm, and then performs an additional synchronization step to wait for the other places to finish the current iteration. The `finish` statement is used for the synchronization and a `copyto` operation is used to communicate the number of predator and preys that move to the adjacent location.

The Turing Ring problem is intended to assess the flexibility of a programming environment in managing data distribution and dynamically changing load. In the X10 solution, population data of predator and prey in the Turing Ring are stored in distributed arrays and distributed using block distribution. The information regarding dynamically changing loads is exchanged between different processors using array copies.

After data distribution, each processor is responsible for updating the population in its part of the ring, which corresponds to steps 2, 3 and 4 of the above algorithm. In the parallel solution these steps can be run asynchronously because they only need data that are local to each place. However, the nodes must synchronize before the execution of step 1, because they need data from the neighboring locations, which may be assigned to remote places. A code snippet of the parallelized version of Runge-Kutta method that each processor runs to update its population is shown in Listing 1. In the listing, $yk1$ represents the slope at the beginning of the interval for the prey population. Using $yk1$, the slope at the mid-point of the interval, i.e., $yk2$ is determined. Similarly, other slopes $yk3$, and $yk4$ are also determined to update the prey population y at the end of the interval. $xk1$ represents the slope at the beginning of the interval for the predator population.

The X10's parallel array `scan` operator is specially useful in updating the predator and prey population of locations in the ring. Given a distributed array of locations, for each place, the `scan` operator can obtain the coefficient values for the Runge-Kutta method for the neighboring locations, which may be assigned to different places.

```
finish async ateach(p in y.dist) {
    for (i in y.region) {
        // k1 coefficient for prey
        yk1(i) = step * evaluate(1,x,y,i);
        yt1(i) = y(i) + 0.5 * yk1(i);
        // k1 coefficient for predator
        xk1(i) = step * evaluate(2, x, y, i);
        xt1(i) = x(i) + 0.5 * xk1(i);
    }
    ...
    ...
}
```

Listing 1: Code segment for Parallel Turing Ring Problem

5.1.3 Data Collection for Performance Analysis

For performance evaluation, both sequential and parallel versions of the Turing Ring problem were implemented. The sequential version does not use any of X10's language constructs for concurrency or data distribution. Further, the sequential implementation is executed in a single place using a single processor. The sequential program was run for different number of locations ranging from 1000 to 8000.

The parallel version was implemented using X10's language constructs for concurrency and data distribution to achieve efficient parallelism. The parallel version was also run with different number of locations ranging from 1000 to 8000. For each given number of location, the executions were performed over different number of worker processes, ranging from 1 to 1024[1].

In current implementations of X10, once a program begins executing, each place in the program is bound to a particular process on a particular node, and this binding is not changed for the lifetime of the program's execution. As a result, increasing the number of places leads to the use of an increased number of processors until they are available. To allow binding of the places to separate processors, we increased the number of processors to be used by the same amount by which the number of places were incremented. When the available number of processors is exhausted, multiple places are bound to a single processor.

Each execution instance chosen for different number of places, processors and locations were repeated for 500 times to determine their arithmetic mean and a 95% confidence interval. The timing measurements were obtained using the system wall clock. For all the measurements, the initial distribution of population over the locations in a ring was done randomly, and the birth, death and migration values that yield stable behavior were used. The parameters for different runs, such as number of places, processors, and locations were passed as command line arguments to the X10 runtime, compiler, program and the batch job processor (pbs_resources in our case).

[1]The initial number of worker processes in a place was controlled by setting the environment variable X10_NTHREADS. This variable was set to 4 except when lesser number of processes were required for testing. Our test platform supports one-to-one mapping of up to 112 places (= 14nodes × 2 processors × 4 cores), which is 448 processes. To test with higher number of processes, the number of places was increased, which meant that multiple places needed to be mapped to a single core.

5.1.4 Performance Analysis

Figures 2 and 3 respectively show the execution time for different problem sizes for the sequential and parallel solutions to the Turing Ring problem. The graphs also show the lower and upper limits for 95% confidence interval for each reported execution time. For the sequential solution, the execution time is directly proportional to the number of locations that need to be processed. This is expected because the amount of calculation to be done depends on the number of locations in the ring.

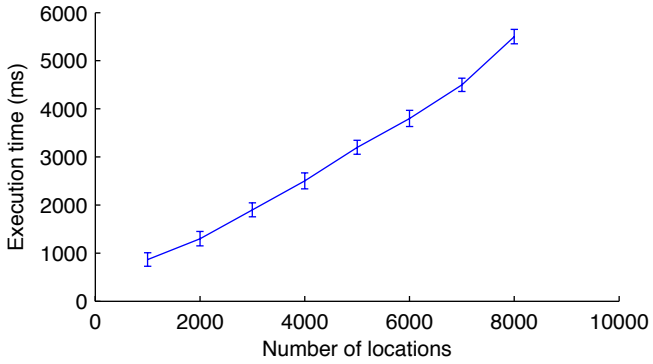

Figure 2: Execution time of sequential Turing Ring solver.

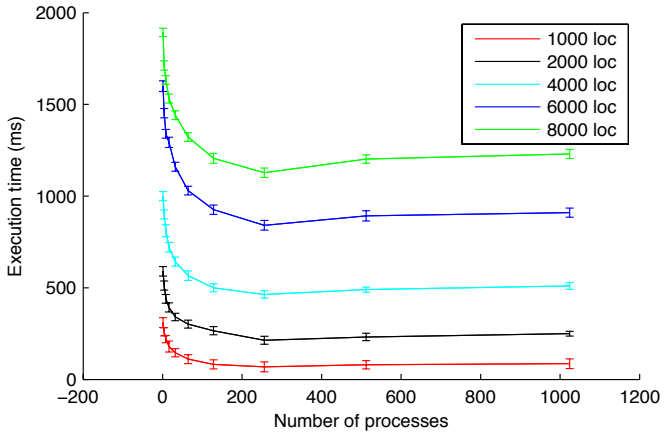

Figure 3: Execution time for parallel Turing Ring solver.

The graphs shown in Figure 3 indicate that the execution time for the parallel version of the Turing Ring is lower than that for the corresponding sequential version. For example, using 112 cores and 112 places (with 4 worker threads per place, i.e., a total of 448 threads), the speedup with 1000 locations is around 4. This speedup is, in fact, close to the value that should be expected based on Amdahl's law [5]. The execution time for sequential and parallel components were measured using the system wall clocks. The maximum time taken to execute a top-level async expression at a place was used to determine the time-requirement of the parallel-component. Similarly, the time taken for various parameter initializations for the Turing Ring and the final data collection and display were used to calculate the time requirement for the sequential components.

The reduction in execution time of the parallel implementation stems from the use of asynchronous activities to find the Runge-Kutta coefficients for predator and prey populations for each locations. The measured speedup for the calculation of such coefficients, using the sequential version and the parallel version with 112 places and 1000 locations, is 3.6. Therefore, the maximum speedup for the entire Turing Ring application when using 112 processors is $1 + 3.6 = 4.6$, which is close to our observed speed up of around 4 times. Similarly, for 112 places the effective speed up obtained from parallelization are respectively $1250/280 = 4.64$, and $2500/510 = 4.9$ for 2000, and 4000 locations.

Another notable trend in parallel performance is that the peak performance results when the initial number of processes is 448. Beyond that, the performance decreases because additional places are needed to support increased number of processes, which means that multiple places will be assigned to a single processor resulting in sequential consistency between activities of such places.

Speedup is also limited by the time spent managing communication between places. The communication is required to share the migration information across neighboring locations that reside in separate places. As the number of places increases, the amount of information that needs to be shared across processors also increases. Adding processors thus adds to the communication time, which in turn reduces the speedup in execution performance.

Another limitation on speed is the increased number of synchronization points. When the ring is distributed among several processors, the processes must synchronize at the end of each iteration of population update so that they can use the updated values of the population to carry on further computation. The top-level *finish* construct in the runge4 method enforces such synchronization.

5.2 Active Chart Parsing

The problem of *Active Chart Parsing* involves generating all possible derivations of a sentence by using the rules in a given ambiguous grammar. This problem becomes very compute intensive for large grammars and sentences and hence, lends to the test of task-parallelism capabilities of X10.

5.2.1 Sequential Solution

The sequential algorithm starts by inserting nodes before each word and after the last word in a given sentence. Then, it creates one edge between two adjacent nodes for each category to which the word between the nodes may belong. Next, it uses the production rules in the specified grammar to create edges of new categories from the existing edges. For example, if there exists an edge of category X between nodes n_1 and n_2, and the grammar contains a production Y $\rightarrow$ X, then the algorithm can insert a new edge of category Y between n_1 and n_2. Similarly, for an edge of category X between n_1 and n_2 followed by an edge of category Y between n_2 and n_3, the algorithm can insert a new edge of category X between n_1 and n_3 if the grammar contains the rule Z $\rightarrow$ X Y.
A sample chart, adopted from the original description of Active Chart Parser by Wilson *et al.* [8], is shown in Figure 4, where *NP* means Noun Phrase, and *VP* means VerbPhrase.

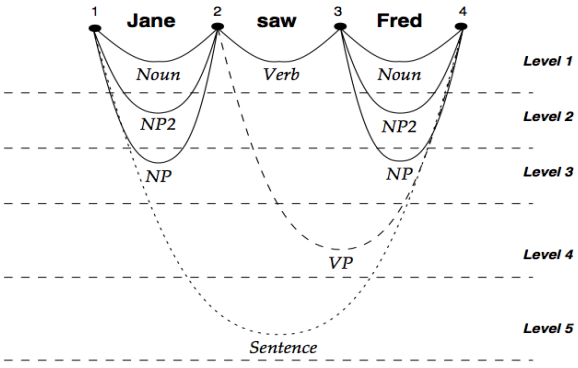

Figure 4: A sample chart for ambiguous grammar.

This algorithm terminates when no more unique edges can be added to the chart: the final chart contains all possible derivations of the input sentence. The implementation of this algorithm requires the following data structures:

- A grammar to store the production rules from the left hand side category to the right hand side categories.
- An edge in a chart to store the source and destination nodes, and the name of the terminal or non-terminal that the edge represents.
- A chart to store the sets of edges mapped to the words.

All these data structures can be directly represented in X10, and the related snippets of code are shown in Figure 5.

```
class Rule {
  var left: Category;
  var right: Array[Category];
}
```

```
class Edge {
  var rule: Rule;
  var origin: int;
  var bases: Set[Edge];
}
```

```
class Grammar {
  val name: String;
  val rules: Map[Category, Set[Rule]];
}
```

```
class Chart {
    var edgeSets: SortedMap[Integer, Set[Edge]];
}
```

Figure 5: Data structures used in Active Chart Parsing.

5.2.2 Parallelization Strategy

The parallelization of Active Chart Parsing assigns one process to every word in the given sentence. Each process then creates its own set of edges containing edge for each possible category of the given word. This edge information must be updated by both the source and destination nodes, and hence, the algorithm needs a mechanism to share this information.

The parallel Active Chart Parsing algorithm, based on Orca implementation of the problem [9], is shown in algorithm 1.

Algorithm 1 Parallel Active Chart Parsing

Require: W: the word assigned to this processor
n: the node number
PE: set of pending edges
EC: collection of edges
ADD all categories to which W may belong to PE[n] and PE[n+1]
while PE[n] is not empty **do**
  $categ \Leftarrow retrieve\_categ(PE[n])$
  if *categ* is a sentence **then**
    print(*categ*)
  else
    add *categ* to local incoming/outgoing edges
    add new edges from the existing ones using grammar rules
  end if
end while

This problem is intended to test X10's support for fine-grained parallelism because generating new edges is cheap compared to the typical cost of interprocess communication [8]. Although fine-grained parallelism is easily achieved using `async` expressions, they incur significant overhead because of the underlying fork-join based implementation strategy. Also, as the neighboring processes must communicate for every incoming or outgoing edge added to its local chart, the communication overhead is also high.

5.2.3 Performance Analysis

Table 1: Performance of Active Chart Parser when assigning each word to a different processor.

Words	Processors	Execution Time(s)	
		Sequential	Parallel
3	3	6	6
8	8	10	12
16	16	128	144
29	29	159	168
41	41	252	276

Figure Table 1 shows the performance of sequential and parallel Active Chart Parser. As each word is assigned to a different processor, the number of processors used is the same as the number of words in a sentence. The parallel adaptation of Active Chart Parser is slower than its sequential counterpart. There are a number of factors contributing to this behavior. First, the parallel version suffers from communication overhead because different words are assigned to different processes and inter-process communication is need to share edge information in the parallel version. In the sequential version, this is not required because all words reside in a single place. Second, the number of edge updates for a word depends on the number of categories it may belong to and the number of production rules for each of those categories. The load-imbalance across processors is also quite high because each word is assigned a single processor.

One way to improve the performance of the parallel Active Chart Parser is to assign multiple words to a single processor. The improvement results from a reduced inter-node communication that would otherwise be needed when mapping each single word to a different processor. The improved performance resulting from distribution of multiple words to a single processor substantiates our earlier claim that communication overhead reduces the overall speed up achieved from the parallelization effort.

Table 2: Performance of Active Chart Parser when assigning each word to a different processor.

Words	Processors	Execution Time(s)	
		Sequential	Parallel
3	2	6	6
8	4	10	11
16	8	128	121
29	16	159	149
41	24	252	236

Table 2 shows a performance report for such a mapping between words and processors. In this case, the performance of the parallel Active Chart Parser is better than that of its sequential counterpart.

At the time of implementation of these problems, we are not aware of any runtime optimization mechanism in X10 that automatically overlaps communication with computation. On that note, runtime optimizations that schedule communication messages to overlap with computation would be highly desirable in X10.

5.3 Matrix Chain Multiplication

Matrix Chain multiplication is an optimization problem that involves finding the most efficient way of multiplying matrices together given a sequence of matrices. The goal is to decide the optimal order of matrix multiplication rather than carrying out the multiplications themselves.

For example, given a chain of matrices A, B, C, and D, there are several different orders in which they could be multiplied, such as: $(ABC)D = (AB)(CD) = A(BCD) = A(BC)D = \ldots$ and so on. Although any of these legal combination of parentheses gives a correct result, the number of multiplication operations will be different for different parenthesizations. Let A be 5×4, B be 4×6 and C be 6×2, then:

$$\text{cost of [(AB) C]} = (5 \times 4 \times 6) + (5 \times 6 \times 2) = 180$$
$$\text{cost of [A (BC)]} = (4 \times 6 \times 2) + (5 \times 4 \times 2) = 88$$

Clearly, the second order of multiplication is cheaper than the first order here.

5.3.1 Sequential Solution

The sequential solution uses a dynamic approach to solve the matrix chain multiplication problem. This approach involves breaking the problem into subproblems, whose solution can be combined to solve the entire problem. The subproblems are: where to split the chain (say k), and how to parenthesize the subchains $A_{1\ldots k}$ and $A_{k+1\ldots n}$. To determine the best value of k, the algorithm simply consider all possible choices of k, and picks the best out of them. Then it builds up the solution by computing the subchains of length $2, 3, \ldots, n$ (the number of matrices).

The sequential algorithm simply computes the minimum cost of multiplication for different subsequences of the given matrix chain. For each l from 2 to n, the number of matrices (or the length of matrix chain), the algorithm computes the minimum cost for each subsequence of length l.

5.3.2 *Parallelization Strategy*

The parallel solution distributes the matrix chain into different places and performs the cost comparison for each subsequence in parallel. While updating the minimum cost of multiplication order found so far, they had to be done atomically to ensure that no two places update this value in an incoherent manner. This problem is intended to test how easily embarrassingly parallel applications can be managed and how easily data can be distributed across different machines.

The parallel implementation of this problem is shown in Listing 2. The matrix chain was distributed among processors using distributed arrays. Concurrency constructs, such as `ateach`, were used to perform local computations at each machine to determine optimal ordering of matrices.

```
finish async ateach((p) in dim.dist) {
 for (len(x) in dim.dist.region){
  var l: Int = x;
  for (var i: int = p*mSize; i < (p+1)*mSize; i++){
   var j: int = i+l-1;
   cost(i,j) = Int.MAX_VALUE;
   for (var k: int = i; k <= j-1; k++) {
    var thisCost: int = cost(i,k) + cost(k+1,j)
                        + dim(i-1) * dim(k) * dim(j);
    if (thisCost < cost(i,j)) {
     atomic {
      cost(i,j) = thisCost;
      s(i,j) = k;
     }
    }
   }
  }
 }
}
```

Listing 2: Code segment for parallel Matrix Chain solver

In the above listing, $s[i, j]$ holds the value of k providing the optimal splitting point. For example, $s[i, j] = k$ indicates that the optimal way of multiplying the subchain $A_{i...j}$ is to first multiply the subchain $A_{i...k}$, followed by $A_{k...j}$, and then finally to multiply these two together.

5.3.3 *Performance Analysis*

The execution times for sequential computation of matrix multiplication costs for different chain lengths are shown in Figure 6. As the matrix chain length increases, the execution time required for the sequential version of the matrix-chain ordering problem increases as well. This increase is expected because the cost of each possible combination of matrices in a given matrix length needs to be tested to find the combination with the least cost. This computation increases as the length of the matrix chain increases.

Figure 7 shows that increasing the number of processors decreases the execution time, but the speed up is sublinear and the solution does not scale. In fact, the maximum speed up obtained from parallelized implementation is around two times. This is because the parallelism available in matrix chain problem is limited by three major factors:

1. While each place looks for the optimal cost of ordering within the subchain assigned to itself, the algorithm must look for subchain division points that might comprise matrices belonging to separate places as well. This requires that the program creates several asynchronous activities to access the remote matrices.

2. Although different processors may carry out cost computation in parallel, the updates to the cost table to replace current values with a lower cost value must occur atomically. As a result, any of the activities running in parallel cannot explore other possible subchains within their portion of matrix chain without updating the cost table with the currently computed value when it is necessary.

3. Each of the activities dealing with their portion of matrix chain must also wait for their neighboring activities to finish because they will need the value computed by the preceding activity to compute the current cost of subchain multiplication.

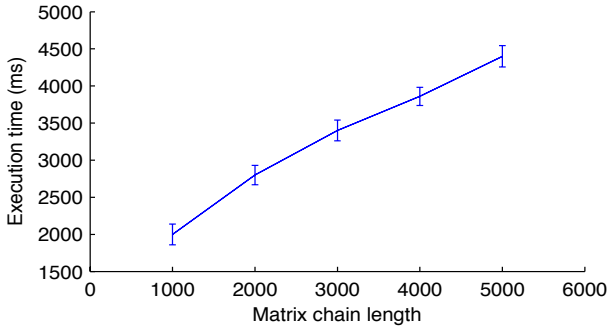

Figure 6: Execution time for sequential Matrix Chain solver.

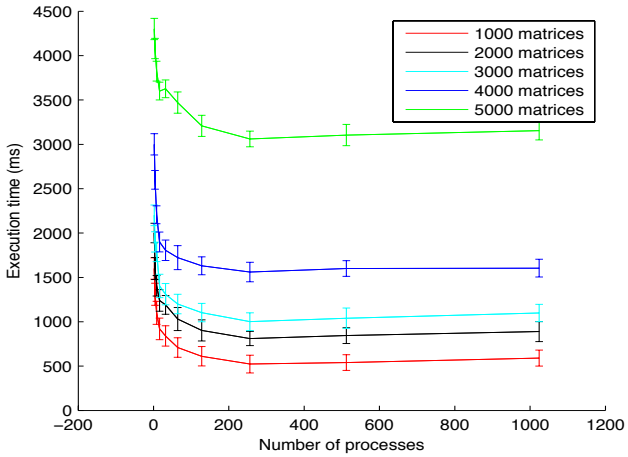

Figure 7: Execution time for parallel Matrix Chain solver.

With these limitations to parallelism, the parallel implementation shows only small performance improvement even with 112 places. As seen from Figure 7, the execution speed drops off after this point. As the number of places is increased beyond 112, the performance starts to drop because now there are no more free processors available to execute activities in the places; they must wait until activities residing in other places and assigned to the current processor finish execution.

One possibility for further improving the level of parallelism in the implementation of matrix chain problem shown in Listing 2 would be to parallelize the inner *for*-loop that computes the cost of multiplying two matrices. However, the disadvantage of such an approach is the increased number of synchronization points, because the cost of multiplying current two matrices depends on the cost of multiplying matrices before this matrix in a given subsequence.

5.4 Skyline Matrix

A Skyline Matrix is an $N \times N$ matrix whose values cluster around its diagonal. In other words, there exist constants r_i and c_i such that for $1 \leq r_i \leq i$ and $1 \leq c_j \leq j$, each row i has non-zero values from r_i to i, and each column j has non-zero values from c_j to j.

Given an $N \times N$ Skyline Matrix A, and an N-vector b, the problem is to find an N-element vector x such that $Ax = b$.

5.4.1 Sequential Solution

The sequential version uses the popular Gaussian Elimination method to obtain an LU decomposition of the given matrix to solve the equation $Ax = b$. This approach to solution does not derive much benefit from the Skyline Matrix in terms of its representation and the underlying computations.

5.4.2 Parallelization Strategy

The Gaussian Elimination approach to solving the Skyline Matrix problem does not capitalize on the special property of skyline matrices. To capitalize on the special structure of Skyline Matrix data and to avoid all operations and calculations that would otherwise involve using the zeroes excluded from the storage, the parallel algorithm uses Doolittle's method [3] for LU decomposition.

The parallelization of the LU-decomposition distributes different portions of L and U matrices among different processors. It maintains the separation of matrix at the diagonal and distributes the matrix data by row as well as by column as shown in Figure 8. With this distribution, each machine stores a sub-diagonal row and its corresponding supra-diagonal column of the given matrix A. Such a data distribution is adopted from the implementation of the skyline matrix solver in Orca [9].

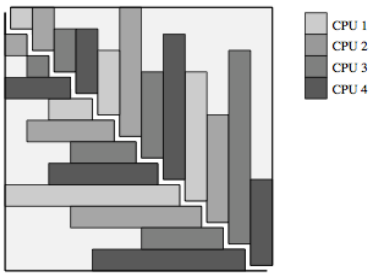

Figure 8: Distribution of Skyline Matrix data.

This data layout allows programmers to make assumptions about inherent balancing of workload among different processors. In other words, given a large enough input of matrices, all processors will remain busy during most of the LU-decomposition phase.

With the progress of the algorithm, the matrix A must transform into L and U. However, in Doolittle's method of LU decomposition, only local matrix values stored in a machine are not sufficient to transform an element of A into a lower-triangular or an upper-triangular matrix element. For example, in a machine storing the ith sub-diagonal row and supra-diagonal column of A, the transformation of an element a_{ij} of A into an element l_{ij} of L requires the column U_j and a prefix of row L_i comprised of all elements to the left of l_{ij}. Therefore, after a machine computes the elements of row L_i and column U_j, they must be copied to other machines as well.

The Skyline Matrix problem is intended to test the capability of a language for representing the rows and columns of a matrix that have varying ranges, and for distributing them across different machines.

X10's ability to support different range of regions and its support for distributed arrays proved efficient in such a data distribution. The copying of arrays between different nodes uses the `copyTo` method. Using this method for large matrices caused a runtime error because of small buffer size. The temporary workaround was to increase the X10RT_DATABUFFERSIZE in the x10rt. However, it would be desirable to be able to change this value through command line as an option to the X10 runtime.

5.4.3 Performance Evaluation

The performance evaluation of the sequential and parallel skyline matrix solvers uses matrices of sizes 1000, 2000, 3000, and 4000 (and 5000 for the sequential solution). The matrices were randomly filled and about 50% of their elements are non-zero.

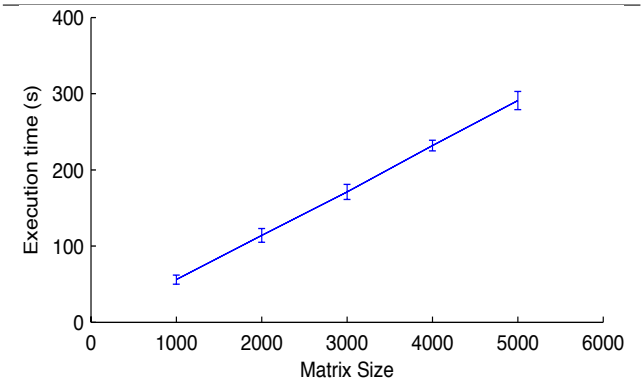

Figure 9: Execution time for sequential Skyline Matrix solver.

For the sequential algorithm, the execution time increases linearly as we increase the matrix size. This is expected behavior because for larger matrix size, there are increased computations involving conversion of an element a_{ij} in the matrix A into lower/upper triangular matrix element.

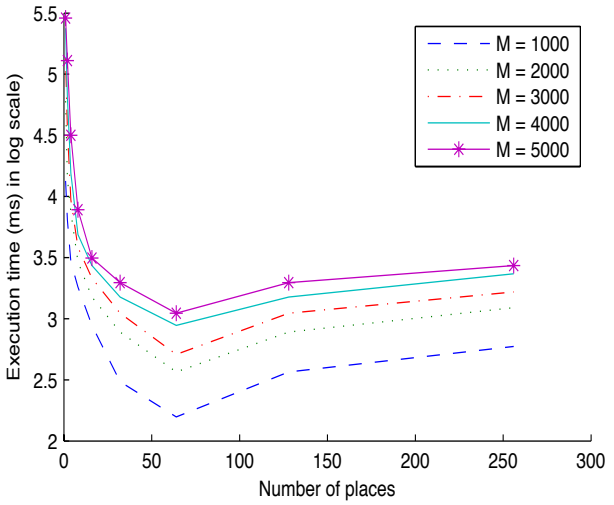

Figure 10: Execution time for parallel Skyline Matrix solver.

Figure 10 shows the performance of parallelized Skyline Matrix solver, where M represents the matrix size. The decrease in execution time is not linear with the increase in the number of processors. As explained earlier, the speed up obtained by division of task among processors is diminished by the cost of communication and copying of arrays between them. The maximum speedup over the sequential algorithm, for a matrix of 5000×5000 is $280/exp(3.2) = 11.4$. This speed up is obtained with 112 places, beyond which the performance drops because further increase in the number of places means a single processor must execute activities residing in multiple places in a sequential manner.

6. REFLECTIONS ON THE EXPERIENCE

This section presents several practical observations that are derived from our experience of coding the Cowichan problems in X10. These observations include a pleasant surprise about the prediction of performance based on small runs, the structure of parallelism in the Cowichan problems studies, issues with language constructs and their implementation and limitations in the programming environment.

6.1 Performance Testing and Prediction

During the performance experimentation of different parallelization strategies in X10 for the Cowichan problems we found out that preliminary results obtained with a small number of places (4 or 8) were very good predictors for the potential of a solution to deliver reasonable performance for larger configurations. In other words, expensive communication patterns, excessive data transfers, and excessive creation of remote references to data, which hinder performance, could be easily identified with a few small runs of the parallel program.

6.2 Structure of Efficient Parallelism

Three of the parallelization strategies — Turing Ring, Active Chart Parsing, and Matrix Chain — have a similar control structure. The root activity (corresponding to the main method) has a top-level *finish* under which it uses *async*s to create a top-level asynchronous activities at each place that runs for a long time. The level of parallelism can be further improved by parallelizing the *for*-loops inside the outer level *finish*. However, this parallelism can come at the cost of increased barrier synchronization, if there are intra-loop dependencies, such as the one in the case of matrix chain ordering problem.

6.3 Language Issue

6.3.1 Fine-grained asyncs

We found that X10's runtime performance for fine-grained parallelism implemented using fine-grained asyncs was poor. This was evident in the active chart parser solution.

6.3.2 Buffer Size

The maximum message size that X10 runtime imposes can be insufficient, specially when running applications with large array copy code.

6.3.3 FileChannel

Input/Output in X10 is currently constrained because it uses just a single thread to perform input/output while other threads are busy waiting and constantly interrupting the input/output thread in an attempt to steal work. It would be useful to have support for FileChannel in X10 and allow concurrent accesses to a file by multiple threads.

6.4 Programming Environment

The error messages generated by the X10 compiler can be cryptic in some cases, making it difficult to debug programs.

```
1 public class Sample {
  ...
3 val initImg: Array[MyArray]{rank==1};
  public def this(...) {
5 ...
  initImg = new Array[RemoteArray](atlasImg.region(),
7       (r:Point) => {(new RemoteArray(srcImg(r)))});
  }
9 ...
  }
```

Listing 3: An incorrect program.

For example, an attempt to compile the code shown in Listing 3 produces the following error:

Constructor this(reg: x10.array.-Region, init: (a1:x10.array.Point-self.rank==reg.rank)=>x10.array.RemoteArray): x10.array.Array-[x10.array.RemoteArray]-self.region==reg cannot be invoked with arguments (x10.array.-Regionself==atlasImg.region, <anonymous class>).

This error message is inadequate because it fails to specify the location of the problem, which lies with the closure type in the given example.

Therefore, more intuitive error messages, and proper debugging and testing tools would be highly desirable to considerably improve programmer productivity.

7. RELATED WORK

There have been previous studies on assessing the programmability, productivity and performance of different parallel programming systems [1, 4, 9].

Kemal *et al.* [4] performed a productivity study comparing the productivity of three parallel programming languages: C+MPI, UPC, and X10. The study focused on determining how easily and quickly could programmers learn this language and use it for application development. Only one candidate problem, Smith-Waterman's local sequence matching algorithm, was used in their study.

This study is different because it focuses on using X10 to develop a fairly diverse set of programs. Additionally, our aim is to determine how easily and how well X10 could meet the programmability requirements posed by the problems in terms of data distribution, dynamic parallelism, expressiveness, and memory models.

Wilson *et al.* [9] assessed the usability of Orca as a parallel programming system. There are significant differences between Orca and X10 that affects the solutions to the Cowichan problems. For instance, in Orca, arrays do not require boundaries as part of type definition. In X10, array boundaries must form part of type declaration, but there are several powerful operators on arrays that allow programmers to select sub-regions of an array. X10 also supports multi-dimensional and distributed arrays making it easier to define how to split arrays and distribute across different nodes. Orca lacks such flexibility in terms of arrays.

Similarly, communication between running processes in Orca occurs through shared objects, which are replicated in each processes' local memory. X10, however, relies on the PGAS programming model and uses messages to communicate between multiple processors. Also, in X10, the shared values need not be replicated in local memories.

In Orca, there are no global variables, parameters should be used instead. Similarly, only immutable variables can be global in X10. Both allow dynamic creation of new independent processes on either the same or on other CPUs through `fork` and `async` statements.

Another study of the programmability of a parallel programming language is by Mehta *et al.* [6] and is based on Unified Parallel C (UPC). UPC is a parallel extension to the procedural C programming language, whereas X10 is built on top of serial subset of Java, an object oriented language. Both X10 and UPC support partitioned global address space, but X10 proves to be richer through its support for asynchronous activities. Although both UPC and X10 allow data parallelism and stripped allocation of arrays, X10 provides more flexible distribution of data than UPC.

Overall, compared to Orca and UPC, X10 provides a more flexible memory model through asynchronous partitioned global address space, and data distribution through various kinds of array distribution constructs.

8. CONCLUDING REMARKS

This paper reports on our investigation of the programmability of X10 using the Cowichan problems. X10 contains a rich set of language constructs and runtime support for implementing a wide variety of applications running on shared-memory or distributed-memory hardware.

The language's strength lies in its flexible treatment of concurrency through lightweight activities, distribution through distributable arrays, and locality through a PGAS memory model, within an integrated type system.

The downside of the language is that its runtime performs poorly in its handling of fine-grained asynchronous activities. While the language constructs in X10 are expressive enough to support various levels and kinds of parallelism, there is a lack of matching error reporting, debugging and testing framework to support application development in an X10 environment.

References

[1] J. Anvik, J. Schaeffer, D. Szafron, and K. Tan. Asserting the Utility of CO_2P_3S Using the Cowichan Problem Set. *Journal of Parallel and Distributed Computing*, 65:1542–1557, December 2005.

[2] 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. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[3] J. T. Feo. *Comparative Study of Parallel Programming Languages: The Salishan Problems*. Elsevier Science Inc., New York, NY, USA, 1992.

[4] E. Kemal, S. Vivek, and E.-G. Tarek. An Experiment in Measuring the Productivity of Three Parallel Programming Languages. In *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing*, pages 30–37, Austin, USA, 2006. IEEE Computer Society.

[5] S. Krishnaprasad. Uses and Abuses of Amdahl's Law. *Journal of Computing Sciences in Colleges*, 17(2), 2001.

[6] P. Mehta. UPC Cowichan Report. Technical report, University of Alberta, 2005.

[7] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification. http://x10.codehaus.org/x10/documentation, Last access: 22 March, 2011.

[8] G. Wilson. Assesing the Usability of Parallel Programming Systems: The Cowichan Problems. In *IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 183–193, 1993.

[9] G. Wilson and H. E. Bal. Using the Cowichan Problems to Assess the Usability of Orca. *IEEE Parallel and Distributed Technology: Systems and Technology*, 4:36–44, September 1996.

[10] G. Wilson and R. Irvin. Assessing and Comparing the Usability of Parallel Programming Systems, 1995.

[11] X10-Team. Performance Tuning an X10 Application. http://x10.codehaus.org/, Last access: 24 March, 2011.