Introduction to the X10 Implementation of NPB MG

Tong Wen
tongwen@us.ibm.com
IBM T. J. Watson Research Center

Dec 04, 2006

1 Introduction

X10 is a new Partitioned Global Address Space (PGAS) language being developed at IBM as part of the DARPA HPCS project. (Cray's Chapel and Sun's Fortress are the two other languages being developed in the DARPA project.) X10 is based on Java with extensions for large-scale and heterogeneous parallel programming. The fundamental distinction between X10 and other PGAS languages is that its model of parallelism is not Single Program Multiple Data (SPMD). X10 supports fine-grained parallelism (both data and task parallelism) so as to face the challenge posed by the explosion of hardware parallelism in the emerging computer architectures. The unit of computation in X10 is an *activity* (light-weighted threads) at a *place* – a place may be considered as a virtual shared memory multiprocessor (SMP). The number of activities in each place varies dynamically. Activities can spawn other activities locally or remotely. In this report, we explore the ways to fully express in X10 the logical parallelism in numerical algorithms through examples found in computational kernels such as Conjugate Gradient (CG), LU factorization (LU), and Multigrid (MG). Issues regarding how X10 handle the hierarchical and heterogeneous nature of the emerging large-scale computer platforms will be addressed in later studies.

The NPB MG benchmark (1) uses a V-cycle Multigrid algorithm to solve Poisson's equation on a rectangular domain with periodic boundary condition. It involves applying a set of stencil operations sequentially on grids at each level of refinement. In the V cycle, the computation starts from the top (finest) refinement level, going down level by level toward the bottom then back up to the top. The stencil operations are restriction, prolongation, evaluation of residual, and point relaxation. They are implemented in class `MGOP`. The overall implementation includes classes `MGDriver`, `MGOP`, `LevelData`, and `Util`. In `MGDriver`, the test problems are set up and the solver defined in `MGOP` is called. The distributed array is implemented in `LevelData`. Constants and commonly used methods are defined in `Util`.

The benchmark was first implemented in the usual way as programming a SPMD model, then more parallelism that can be expressed explicitly in X10 is added, which by comparison is hard for SPMD languages to achieve. We use abstract performance metrics to quantify the parallelism we add. Here, we compute the length of critical execution path and the ideal speed up (the ratio of the total computation cost (communication cost not included) over the length of critical path).

2 Data Structures

In X10, programmers have full control over the distribution of data structures across places. A *distribution* in X10 is a map from a *region*, a set of *point*s (arrays of integers), to a set of places, and a multidimensional array can be defined on a distribution where the points serve as its indexes. X10 supports array abstractions similar to Titanium (3). Currently, the number of places is determined at execution time. (We are considering permitting places to be created dynamically in a later version.) There are two ways to construct distributed arrays in X10. The first one uses the union of disjoint distributions. This approach is used to build the distributed sparse matrix in the CG benchmark. The other is the general encapsulation approach (array of

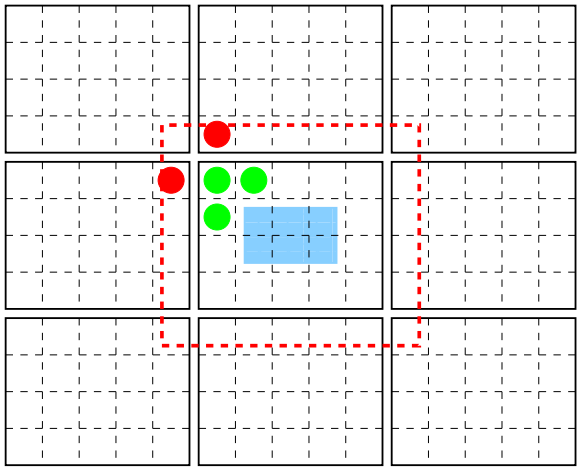

Figure 1: In this figure, a 2D rectangular grid is partitioned into 3 by 3 subgrids. To apply a 5-point stencil at the boundary of each subgrid, values from its neighbors are needed. To cache these values, a one-deep layer of ghost cells is added to each subgrid. One observation is that the application of this stencil in the inner region of each subgrid, for example, the blue-colored region of the middle one, is independent of the values on other subgrids.

arrays): a local array (or other data structure) is encapsulated in an X10 *value* class, (Instances of value classes can be referenced and operated upon from any place.) then an array of this wrapper class is built on a distribution. This approach is used to build the distributed vectors in the CG benchmark and the distributed arrays in the MG benchmark.

As indicated in Figure 1, a one-deep layer of ghost values is added to each sub-array to cache the values from neighboring ones. Thus, the distributed array is implemented using the encapsulation approach rather than using the union-of-distribution one, because the resulting sub-arrays are overlapping with each other. The ghost values are updated by the `exchange` method of class `LevelData` where the physical periodic boundary condition is also applied.

In the following segments of X10 code, a wrapper class and an array of it are defined. For each place, there is an array element containing a 2D array assigned to it. All the 2D arrays have the same region $[0:255, 0:255]$ but different distributions. The distributed array in MG is implemented in class `LevelData`.

```
value Wrapper{
      double [.] m_array;
      Wrapper(final double [.] a_array){
          m_array=a_array;
      }
      ...
}
...
Wrapper value [.] m_u=new Wrapper [place.MAX_PLACES];
...
  Region R=[0:255,0:255];
  for (point [i]:dist.factory.unique()){
      final dist D=R->place.factory.place(i);
      m_u[i]=new Wrapper(new double [D]);
  }
...
```

```
finish updateGhost();
finish ateach(point [i]: AllPLACES){
    for (point p:localRegion(i))
           stencilOp(p);
}
```

Figure 2: This pseudocode shows the bulk-synchronous implementation of a stencil operation.

```
finish {
    //Inner region
    ateach(point [i]:ALLPLACES){
         for (point p: InnerLocalRegion(i))
                stencilOp(p);
    }

    // Ghost layer
    finish updateGhost();

    //Boundary layer
    ateach(point [i]:ALLPLACES){
        for (point p: LocalRegion(i)-InnerLocalRegion(i))
                stencilOp(p);
    }
}
```

Figure 3: This pseudocode shows how X10 can overlap communication with computation easily in the implementation of a stencil operation.

3 Stencil Operations

The usual way to implement a stencil operation is bulk-synchronous as shown in Figure 2. The ghost values are updated first by copying the corresponding values from neighboring subarrays. Then the stencil operation is performed on local subarrays in parallel. From Figure 1, we can see that the application of a stencil operation in the inner region of a subgrid can be performed in parallel with the update of ghost cells. Unlike in SPMD languages, this parallelization can be easily expressed in X10 as shown in Figure 3. The stencil operations are implemented in class `MGOP`.

4 Expressing the Parallelism

We use the length of critical execution path to measure the extra level of parallelism that can be expressed in X10 by comparing with the bulk-synchronous implementation. The total cost of computation and the critical path length are in abstract performance metric (supported by X10). The ratio of these two quantities defines the ideal speedup. In this section, we first measure the bulk-synchronous implementation where the parallelism is only from partitioning the computation across places. We then improve the implementation by overlapping operations and see how much the ideal speedup can be increased. Lastly, the fine-grained parallelism is added by replacing the `for` loop as shown in Figure 3 with `foreach` loop.

The first set of ideal speedups are shown in Figure 4. Here, the update of ghost values (communication) and the local stencil operations (computation) are first overlapped. In the update of ghost values, the ghost regions at the two faces of each 3D subgrid along one dimension can be updated simultaneously. This is achieved easily be replacing a `for` loop with a `foreach` loop. The fine-grained parallelism is finally added by replacing any unordered `for` loop with `foreach` loop. We can see that as more parallelism is added the larger the ideal speedup.

Abstract performance metric	
FP_PERCLOCK	4
LATENCY	375 (cycles)
BANDWIDTH	5.3 (bytes/cycle)
Computation Cost (Class S)	3,272,372(cycles)

Ideal Speedup				
Number of places	**1**	**2**	**4**	**8**
Bulk-synchronous	1	1.74	3.17	5.46
Overlap comm & comp	1.19	2.22	4.08	7.05
Overlap comm	1.19	2.35	4.51	8.33
Replace `for` with `foreach`	5382.2	27.6	32.8	37.3

Figure 4: The ideal speedups are computed for four cases. For each case, different number of places are used to run the benchmark (Class S). The abstract performance metrics are defined in the top table. The computation and communication costs are inserted in the code manually. The runtime computes the critical path length. The base case is the bulk-synchronous implementation (SPMD style). Then parallelism is added gradually. First, the communication and computation are overlapped in each stencil operation. Secondly, communication is overlapped in the procedure of updating ghost values. Lastly, each unordered `for` loop in every stencil operation is replaced with a `foreach` loop.

Abstract performance metric	
FP_PERCLOCK	4
LATENCY	37.5 (cycles)
BANDWIDTH	5.3 (bytes/cycle)
Computation Cost (Class S)	3,272,372(cycles)

Ideal speedup				
Number of places	**1**	**2**	**4**	**8**
Bulk-synchronous	1	1.85	3.58	6.83
Overlap comm & comp	1.19	2.38	4.67	9.07
Overlap comm	1.19	2.43	4.83	9.55
Replace `for` with `foreach`	5382.2	53.0	77.3	112.3

Figure 5: The setting is the same as in Figure 4 except that the communication latency is reduced by a factor of 10. We can see that communication latency is the performance bottleneck of this kind of application

In the last case shown in Figure 4, the ideal speedup decreases dramatically from using 1 place to 2 places. The cause is the high communication-computation ratio, that is, the cost to partition computation across computing nodes (places) is relatively high. In Figure 5, the communication latency is reduced by a factor of 10. We can see that communication latency is the performance bottleneck of this kind of applications.

5 Conclusion

From this experiment, we can see that fine-grained parallelism can be easily expressed in X10 in contrast to existing SPMD parallel programming languages. That gives X10 an advantage when programming large-scale scientific applications on the emerging computer platforms where more hardware parallelism is available.

Acknowledgments

X10 is being supported in part by DARPA under contract No. NBCH30390004.

References

[1] D. H. Bailey, et. al. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63-73, Fall 1991.

[2] P. Charles, et. al. X10: an object-oriented approach to non-uniform cluster computing. *The Proceedings of OOPSLA '05*. Fall 2005.

[3] P. Hilfinger (ed.) et. al. Titanium Language Reference Manual, Version 2.19. Technical Report UCB/EECS-2005-15, UC Berkeley, 2005.