Inferring Arbitrary Distributions for Data and Computation

Soham Sundar Chakraborty *
Tata Research Development and Design Centre,
54-B, Hadapsar Industrial Estate,
Pune, MH, India, 411013
sohamsundar.chakraborty@tcs.com

V. Krishna Nandivada
IBM Research India
Embassy Golf Links Business Park,
Bangalore, KA, India, 560071
nvkrishna@in.ibm.com

Abstract

In the era of mult-core systems, one of the key requirements of achieving better utilization of multiple available cores is that of parallelization of code across multiple distributed nodes; this involves (re)distribution of both data and computation. Such a transformation can be a fairly tedious activity considering the possible dependencies (data, control) and interference between different segments of the code. Further, to keep the data accesses local, computation distribution requires appropriate data distribution and vice versa. And this inter-dependence between distribution of data and computation makes the problem challenging. Another important challenge in this context is that the desired distribution may not be one of the well-known distributions (such as blocked, cyclic etc), and thus reasoning about it can be non-trivial. We present a refactoring framework that can help an application developer to incrementally distribute programs in the context of distributed memory multi-core systems. Given a loop and an array accessed therein, the goal of our framework is to distribute the array based on a specified distribution for the loop (or vice versa) such that the number of remote accessed are reduced. Our framework goes beyond the well-known distributions, and can handle any arbitrary distributions. In our initial investigation, we have used our transformations on varied parallel benchmark programs and have been able to show its applicability along the expected lines.

Categories and Subject Descriptors:
D.3.2 [**Language Classification**] Concurrent, distributed, and parallel languages D.1.3 [**Concurrent Programming**] Distributed programming

General Terms:
Languages, Performance

Keywords:
User specified Distributions, Automatic redistribution

* Work done at IBM IRL.

Onward! 2010, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

1. Introduction

Improving the utilization of the available multiple cores is assuming an important role in the era of multi-core systems. The impact can be observed in the context of both legacy and new applications. While the development of new applications needs to take into consideration the multiple available cores, the legacy applications have to be either retargeted or have their runtime (for instance, operating system or hypervisor) modified to be able to take advantage of the multiple cores. The improvements to the runtime are especially exciting to those legacy applications where the source code is not available, or where the runtime parameters greatly influence the performance gains, which are otherwise not known at the source code level. One drawback of this approach is that the operating system or hypervisor may not be able to utilize the structure of the program and the expertise of the programmer who might be able to assist with the task. Thus, rewriting (or porting) existing application to suit the needs of the new multi-core systems is gaining interest. The challenge is compounded in the context of distributed memory multi-core systems; because of the issues arising out of locality of data, parallelization of computation across distributed cores impacts the distribution of data, and vice versa. Thus, such a transformation can be a fairly tedious job, considering the possible dependencies (data, control), and interference between different segments of the code. A tool to selectively refactor parts of the code (legacy or new) to distribute computation and/or data, would be a great help in this direction.

A refactoring tool can be seen as an aid to incremental programming [9], and can be part of an incremental programming environment [13]. Popular programming environments like Eclipse provide a popular platform to incrementally improve programs by applying different refactorings. In this paper, we propose a new refactoring to incrementally distribute existing applications and thereby port applications to multi-core systems; we accomplish these with some minor

guidance from the application developer. Another important use of such an approach is to incrementally develop/tune existing applications, wherein the programmer starts from the existing application (legacy or otherwise) and uses the refactoring tool to improve the performance in a trial-and-error method.

In this paper, we present techniques to incrementally distribute programs written in language frameworks, such as UPC [11] and X10 [12], that allow *distribution* of data and code. A distribution is defined as a map from loop or array indices to computing units (cores / *places* in the context of X10). Given an application, the programmer may decide to distribute any particular parallel loop. To ensure that the array accesses in the distributed loop are held in sync with the distribution, it is expected that the arrays would also be distributed (otherwise, the application may have to pay the penalty of remote accesses of the arrays). Which in turn may require further appropriate transformations to other loops accessing the array, and so on.

To illustrate the challenge, we show a snippet of code from the FluidAnimate benchmark from the Parsec 2.1 suite [5] in Figure 1; we have ported the example to X10 for the sake of presentation, and inlined a method. The `foreach` loop creates a parallel loop for each value of `i` (varying from `1` to `threadnum`), and each iteration uses the smoothed particle hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. Now, say the `foreach` loop is to be distributed using a blocked distribution (with blocking factor `K`) then the arrays `grids`, `cells` and `cnumPars` should also be distributed (incidentally using the same blocking distribution, with a blocking factor of `K`) to avoid remote accesses. Additionally, the array declaration, initialization, and any other accesses should also be suitably modified. The problem of inferring suitable distributions for loops/arrays becomes further challenging in the context of languages like X10, where the programmer is allowed to specify arbitrary distributions. In this paper, we address this challenge by generating a new distribution for a loop(array) based on the programmer specified distribution for the array (loop) and the program syntax such that the number of remote accesses are minimized.

Contributions

- We present an unified algorithm that can distribute a loop based on any arbitrary distribution of an array accessed there in or vice versa. Our approach distributes data and computation in an interleaved fashion - distribution of data may result in distributing computation where the data is accessed, which in turn may require distribution of other data accessed in the computation and so on.
- We present the loop distribution techniques in the context of a refactoring tool that can help in incremental distribution of applications.

```
int main(...){
final Cell [] cells = ...
final int [] cnumPars = ...
final int framenum = ...;
foreach([i] [1:threadnum]) {
 for(int k = 0; k < framenum; ++k) {
 ...
  for(int z=grids[i].sz;z<grids[i].ez;++z)
   for(int y=grids[i].sy;y<grids[i].ey;++y)
    for(int x=grids[i].sx;x<grids[i].ex;++x)
    { int index = (z*ny + y)*nx + x;
      Cell cell = cells[index];
      int np = cnumPars[index];
      for(int j = 0; j < np; ++j) {
       cell.density[j] = ...;
       cell.a[j] = ...; } } ... } } }
```

Figure 1. Snippet of FluidAnimate – a benchmark from Parsec 2.1 [5].

- We have applied our techniques on a varied set of benchmarks and have found our techniques to be useful.

In this paper, we use X10v1.4 as the basis language for discussing the techniques on distributions. However, they can be applied to other language frameworks as well.

1.1 Related Work

Traditional automatic parallelization techniques have been well studied in research [7, 15, 18]. There is also work in the area of refactoring for parallelism [2, 10, 19, 24],which rely on the programmer to specify what loops to transform. Dig et al [10] introduce concurrent libraries via a refactoring mechanism to ensure thread-safety of data types. Markstrum et al [26] present a refactoring tool based approach for incremental parallelization of code in the context of task parallel languages like X10. In this paper, we present a framework to incrementally parallelize programs (by distributing loops and arrays) in the context of distributed memory multi-core systems. Automatic data and code distribution has been a well researched area and the hardness of the problem is well documented [3, 8]. Mace [25] has studied the optimal data storage problem as shapes problem and has proved it to be NP-complete. Most of the prior work deals with finding an ideal distribution for an array or a loop for efficient execution, and thus ensuring a better utilization of resources. Koelbel et al [20] present a functional language called BLAZE that lets the programmer specify data partitions and the compiler automatically does the process partitioning. Rogers and Pingali [27] partition the given set of tasks in a sequential program based on the programmer specified data-partition, to enhance locality of reference. Prior work in distribution of data in the context of HPF [4, 14] dealt with inferring predefined data distribution (such as blocked, blocked cyclic, cyclic and so on) for a given a program text. There has been prior work on inferring efficient data distribution from a

given distribution of the computation [21, 22], and inferring efficient distribution for a computation from a given distribution for an array [23].

In contrast to these prior works we chiefly differ in the following three ways: (a) We identify that distribution of data and computation are interrelated and present an unified approach to re-distribute both computation and data. (b) Unlike prior work where programmer has no control on the distribution process, we provide the control to the programmer to decide the target computation or array to be distributed. (c) Our approach handles arbitrary distributions of data and computation, even in contexts like X10 that allow specification of arbitrary distributions.

Organization: We first present a brief introduction to X10 language in Section 2. Section 3 presents our techniques to redistribute loops and arrays. We discuss some case studies in Section 4. Finally we discuss some future directions in Section 5 and conclude in 6.

2. X10 Background

In this section we present a brief background to some of the relevant features of X10 for this paper. In this paper, we confine ourselves to simplified X10 programs that have only simple expressions (similar to the expressions in three-address-codes) and every statement has an associated label with it. Details about standard X10 v0.41 can be found in the X10 reference manual [12].

`async (p) S` creates a new child task/activity to execute `S`, at place `p` and this new activity runs in parallel with the parent activity. Notions of activities and places become clear through the association of activities to threads of execution and places to processors in the program. Any dereference of an object, created at a place p, is considered local if it is dereferenced in an activity running at place p. The indexical constant `here` evaluates to the place at which the current activity is running. Each object has a final field named `location` that points to the place where the object was created. X10 restricts accesses to remote memory and a runtime exception is thrown if an activity accesses remote data; note that X10 disallows migration of objects and activities.

`future (p) expr` creates an activity to evaluate the expression `expr` at the place `p` in an asynchronous way and returns a handler to the activity. The return value of `expr` is received by invoking the `force()` method on the handler. A `future` is different from an `async` in terms of returning a value; `force` waits for the value to become available before returning. Unlike an `async` the `future` construct is used to evaluate an expression.

`finish S` is a structured barrier statement, wherein `S` is executed with a surrounding barrier such that all activities created inside `S` have to terminate (including transitively spawned activities) for the barrier to terminate.

The statement `foreach (point p : R) S` creates a parallel loop iterating over all the points in region `R`, by launching each iteration as a separate activity executing `S`. A *point* is an element of an n-dimensional Cartesian space ($n \geq 1$) with integer-valued coordinates and a region is a set of points. A region can be used to specify an array element index space or loop iteration space.

`ateach (point p: D) S` is a parallel loop distributed over the distribution `D`. A *distribution* is a map from a region (defined by a set of points) to a set of places; both arrays and loops may be distributed. A loop L distributed over a distribution D_L iterates over all points in the domain of the D_L; each iteration of the loop (say, corresponing to a point i) is run at a place determined by $D_L(i)$. Similarly, for an array A distributed over a distribution D_A the location of the array element j is determined by $D_A(j)$. If $A[j]$ is accessed in L (say in the iteration corresponding to point i), then it will result in a remote access unless $D_A(j) = D_L(i)$. For efficient layout of data, X10 defines several standard distributions (such as blocked, cyclic, blocked-cyclic and so on) and also allows a new distribution to be defined as a piece of X10 code in the programmer's code. The signature of an X10 function representing a distribution is given by the function `place dist (point p)`.

3. Code and Data Distribution

Ideally, in the context of distributed memory multi-core systems, the data and the computation should be distributed in sync, such that the data is maximally accessed locally, thereby avoiding the *cost* of data communication across different cores (or *places* in the X10). Depending on the language framework, accessing remote data may be disallowed (X10 explicit syntax) or involve remote communication (X10 implicit syntax, UPC).

In this paper we present techniques, wherein given a loop L and an array A accessed therein, the following two *key challenges* can be answered:
(i) Given a distribution D_A for the array A, derive an *efficient* distribution D_L for L.
(ii) Given a distribution D_L for the loop L, derive an *efficient* distribution D_A for A.
The efficiency of a distribution is measured by the number of resulting remote accesses (fewer the better). Note that the two key challenges are interlinked: redistributing a given loop may require redistribution of the arrays accessed therein, and redistributing a given array may require redistribution of the loops where it is accessed.

Given a distributed loop L, each iteration of it may be running at a different place. An array A accessed in L, may be accessed at multiple program points within it, and at different program points different array elements may be accessed (Say $j_1, j_2, \cdots, j_n$). If the loop has to be distributed over a distribution D_L then for each of the accesses of A in every iteration (with index i) to be local, the array A must be distributed in using a distribution D_A, such that $D_A(j_1) = D_A(j_2) = \cdots = D_A(j_n) = D_L(i)$. In general, it

Arrays	:	Set of arrays
L	:	Set of labels
S-Exprs	:	Set of statement-expressions
Loops$\subseteq$ L	:	Set of labels of the loops
vExprs	=	S-Exprs $\cup \{$void$\}$
F	:	Loops $\times$ Arrays $\rightarrow P($vExprs$)$

Figure 2. Helper sets and maps.

is not possible to always guarantee that we can generate such a distribution for the array, from the given distribution of a loop (or vice versa). Considering the similarities between the two key challenges, we present a simple unified approach based on data flow analysis, backward slicing, and reverse execution [6] to generate a target distribution so as to reduce the number of remote accesses.

3.1 Definitions

We first present a few definitions in Figure 2. Arrays is the set of array variables from the program, and L is the set of labels in the program[1]. S-Exprs is the set of all possible X10 *statement-expressions*. A statement-expression is defined as a sequence of zero or more statements, followed by an expression, called *last-expression*; the last-expression may be optionally guarded by a predicate. For any loop L and an array A accessed in the loop the map $\texttt{F}(L, A)$ returns the set of all the statement-expressions using which the different elements of A may be accessed in L. Say, V_f is the set of zero or more free variables that L may access (use or define); these are defined in the outside environment of the loop. One or more of the variables from the set V_f may also be accessed in $\texttt{F}(L, A)$. Note the set V_f does not include the loop index variable(s). For any iteration of the loop L and an element $s_e \in \texttt{F}(L, A)$, s_e/V_f (evaluation of s_e in the presence of the environment V_f) returns the indices of the array slots accessed in that iteration, or not return any value (if no array slot is accessed in that iteration).

Figure 3 presents the pseudo-code for the map F. The algorithm takes as input a loop L and an array A, and returns a set of statement-expressions using which the array may be accessed in the loop. For each access of the array A and its aliases (computed using [1]) in the loop L, we first identify the variable e using which the array element is accessed[2] For each of the expression, we compute the corresponding statement-expression, by computing a backward slice [28]; we add this statement-expression to the F map return value. The function Slice(L', L, e) returns a statement-expression corresponding to a backward slice bounded within the region of the loop body of L, starting from node L upto the

[1] As discussed in Section 2 we use a simplified syntax wherein every statement has an unique label associated with it

[2] Internally our input program is represented in three-address format. Thus every array element is accessed using a variable only.

```
Vector ⟨Expr⟩ Function F(L, A, success)
Input: Loop L, Array A
Output: boolean success
begin
    Vector⟨Expr⟩ Fk = new Vector⟨Expr⟩();
    success = true;
    foreach index variable e used to access the array
    A and its may aliases do
        L' = label of the statement where A is accessed
        using e;
        Block=Slice(L', L, e);
        if Block contains L' or has side-effects then
            success = false;
            break ;
        Fk.add(Block);
    return F_k;
end
```

Figure 3. Algorithm to evaluate the map F

statement L', and following the data and control dependence edges induced by e; e depends on this statement-expression.

Figure 4(a) shows an illustrative program, that evaluates the randomness of the builin random number generator. It first generates a large number (10000) of random numbers and stores them in the array A. It creates a two element array sum (initialized to zero). In the first loop, it sums up all the even indexed elements of A into sum[0], and the odd ones into sum[1], and uses these sum values to compute the difference. For a large enough sample space, this difference should approach zero. In the second loop, we copy the generated random numbers into an array of half the size, by summing up two numbers at a time, and do some computation (not shown) on the summed up elements. Figure 4(b) shows the F map.

We use a new program representation, namely a LAG (loop array graph), that helps in efficient code and data distribution. A LAG is a weighted undirectional bipartatite graph $G = (\texttt{Arrays}, \texttt{Loops}, E)$. An edge $(A_i, L_k) \in E$, iff $A_i \in$ Arrays, $L_k \in$ Loops, and A_i is accessed in L_k. Figure 4(c) shows the LAG for the code in Figure 4(a). The weights on the edges are indicative of the frequencies of accesses of the array in each loop it is accessed. We use some conservative static estimates to derive at these frequencies.

3.2 Distribution Algorithm

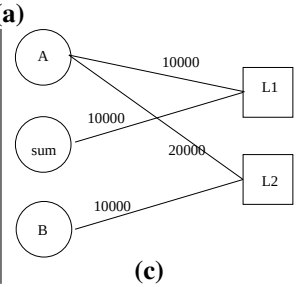
Considering the interdependencies and similarities between loop and array distributions, we develop an unified approach to answer the key challenges presented at the beginning of the section 3. To distribute a given node n (loop or array) in a LAG G, using a distribution D, we invoke the function redistribute (n, D, G) shown in Figure 5; Initially, all the edges in G are unmarked. For each unmarked edge (n, n'), we first mark the edge. We then invoke GenTD-LA,

```
Region R = [1:10000], R1 = [1:10000];
double []A=new double[R1]
          (point [i]){return random()};
double []sum = new double[[1:2]];
L1: finish foreach (point [i]: R){
  if(i%2==0){sum[0] += A[i];}
  else {sum[1] += A[i];} }
double diff = sum[0] - sum[1];
Region R2 = [1..5000]
double []B == new double[R2];
L2: finish foreach (point [j]: R2){
  B[j] = A[j] + A[j+5000]; ... }
```

(a)

```
F(L1,A)   = if(i%2==0) i;
            if(i%2==1) i;
F(L1,sum) = if(i%2==0) 0;
            if(i%2==1) 1;
F(L2,A)   = j;
            j+5000;
F(L2,B)   = j;
```

(b)



(c)

Figure 4. Example program, the F map, and the LAG.

Function redistribute(Node n, Distribution D, LAG G)
// (i) Distributes n using D
// (ii) Distributes any other nodes that might get impacted because of (i)
begin
  foreach *edge* $(n, n') \in G$ **do**
    if *edge* (n, n') *is already marked* **then** continue;
    mark the edge (n, n') in G;
    if n *is an Array* **then** GenTD-LA($n, n', D, success$);
    else GenTD-AL($n', n, D, success$);
    if $success$ **then** redistribute($n', n'.dist, G$);
  end
end

Figure 5. Distribute loops and arrays

or GenTD-AL depending on if the node n' is an array or a loop. The function GenTD-LA generates the target distribution for a loop based on the distribution of an array. Similarly, the function GenTD-AL generates the target distribution for an array based on the distribution of a loop. For each node n' whose distribution gets modified, we invoke the function redistribute to further distribute all the elements that get affected by n'. The input LAG may have cycles; we use the edge markings to avoid infinite-recursion.

Function GenTD-LA(A, L, D, $success$)
Input: Array A, Loop L, Distribution D
Output: $boolean$ $success$
begin
  Set $cFunc$ = getDist (L);
  if $cFunc$ == *null* **then** $cFunc$ = GenEmpty();
  foreach $e \in F(L, A, success)$ **do**
    if $\neg success$ **then** break;
    addCode($cFunc$, e, D, $success$);
    if $\neg success$ **then** break;
  if $success$ **then** setDist($L, cFunc$);
end

Figure 6. Generate Loop Distribution Function

The functions GenTD-LA (Figure 6), and GenTD-AL (Figure 7) use four helper functions: GenEmpty creates and returns an empty set; getDist(n) returns the current distribution of the node n as a set of statement-expressions. setDist(n, S) associates the set S as the code listing for the node n. And addCode ($cFunc$, e, D, $success$) adds the input statement-expression e to the code-listing $cFunc$, if the set of free variables in e are bound to only constant values. Further, the last-expression le is replaced by $D(le)$. The output variable $success$ is set if addCode updates $cFunc$.

The function GenTD-LA includes all the statement-expressions that are used to access the input array, as part of the distribution of the array. These statement-expressions are generated from the enumeration of the map F. The function GenTD-AL is a bit more involved: for each array element access we generate a statement-expression that corresponds to the index of the loop in which the element is accessed; we do this by generating the reverse execution code [6] for the statement-expression of the array index.

Function GenTD-AL(A, L, D, $success$)
Input: Array A, Loop L, Distribution D
Output: $boolean$ $success$
begin
  Set $cFunc$ = getDist (A);
  if $cFunc$ == *null* **then** $cFunc$ = GenEmpty();
  foreach $e \in F(L, A, success)$ **do**
    if $\neg success$ **then** break;
    $e' = \texttt{rev}(e, success)$;
    if $\neg success$ **then** break;
    addCode($cFunc$, e', D, $success$);
    if $\neg success$ **then** break;
  if $success$ **then** setDist($A, cFunc$);
end

Figure 7. Generate Array Distribution Function

```
   Function emit-dist-map(N)
   Input: Node N
1  begin
2      cFunc = getDist (N);
3      List CF' = Sort the elements of cFunc in
       decreasing order of priority;
4      boolean all-paths-covered = false;
5      foreach statement-expression e in CF' do
6          emit-code-element(e, all-paths-covered);
7          if all-paths-covered then break;
8      if ¬all-paths-covered: then
9          emit-code(" return 0;");
10 end
```

Figure 8. Emit Distribution Map

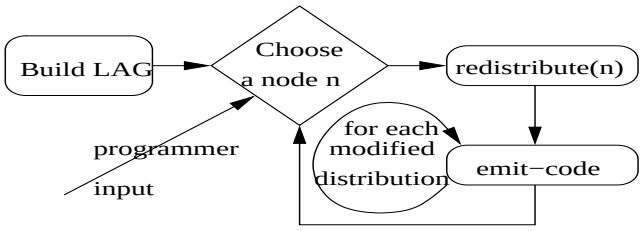


Figure 9. Overall Block Diagram

3.3 Code Generation

The overall block diagram of our framework is shown in Figure 9. After the `redistribute` function returns, we invoke the function `emit-dist-map` (Figure 8) for each node in the `LAG` whose distribution is modified, to emit the code for the new distribution. We sort the set of statement-expressions in the decreasing order of their frequencies and store in CF'; the frequency of a statement-expression is given by the frequency of the corresponding last-expression. We now construct the body of the distribution map, by emitting the code as given by the elements of CF'. We use an auxiliary functions `emit-code`(e) to emit code; it replaces the last expression $D(le)$ in e, by `return` $D(le)$. If we emit an unconditional `return` statement then the output variable *all-paths-covered* is set and then we stop emitting further code; as all the code following an unconditional `return` statement would become unreachable. A loop L may not access all the elements of an array A. Hence the distribution function of the loop D_L can be used to distribute the only those array elements which are accessed in L, and not others. The algorithm emits an unconditional `return 0` statement to handle any eventuality arising because of this, and guarantees that every element/iteration of the array/loop is distributed using this generated distribution. We emit this unconditional `return` statement, only if we did not encounter an unconditional `return` statement in line 7.

After generating the appropriate distributions for loops and arrays in the programs, relevant portions of the program need to be transformed: (a) For each loop that is redistributed, (i) we modify the loop header to distribute over the new distribution, (ii) for all the scalar variables that are created outside the loop and are accessed inside, we replace the access $v.f$ by `(`$v.location$ `== here)?`$v.f$`: future (`v`) `$v.f$`.force();` (iii) similarly, for all the scalar variables that are created outside the loop and are dereferenced to invoke a function, we replace the invocation $v.foo((a_1, \cdots)$ by $\begin{cases}(v.location == here)?v.foo(a_1, \cdots) : \\ \{\texttt{final}\ T_a\ fa_1 = a_1; \cdots; \\ \texttt{future}(v)\ v.foo(fa_1, \cdots).\texttt{force}()\}\end{cases}$
The arguments to the remote function calls are passed via `final` variables (a syntactic requirement of X10 – only final variables are visible across multiple activities). (b) For each array A that is redistributed using a distribution D_A, we (i) change the declaration of the array, and (b) replace the array accesses $A[j]$ by: $(D_A(j) ==$ `here`$)?A[j] :$ `future` $(D_A(j))A[j]$.`force`();

3.4 Discussion

- It can be seen that, our translation may litter the code with many ternary operator (?:) accesses, to do place-locality checks; which can be fairly expensive. We propose to use the place locality analysis of Agarwal et al [1], to eliminate most of the place checks.
- In Figure 5, it may be noted that we do not specify the order of traversal of the edges of the `LAG`. The order of traversal has an impact on the resulting distributions. Consider two loops L_1 and L_2 accessing two arrays A_1 and A_2. Say the `redistribute` function is invoked to redistribute L_1. Based on the order of the edges chosen two possible alternatives exist. The algorithm may use the distribution of L_1 to distrbute A_1, then use the resulting distribution of A_1 to distribute L_2, and finally use the distribution of L_2 to distribute A_2. Alternatively, the algorithm may use the distribution of L_1 to distrbute A_2, then use the resulting distribution of A_2 to distribute L_2, and finally use the distribution of L_2 to distribute A_1. Thus the resulting distribution function for A_1, A_2 and L_2 can be different. Identifying the optimal order of edge traversal for a given `LAG` is a nontrivial problem, and is left as a future work.
- The generation of the `F` maps forms the backbone of our translation scheme, which in turn depends on alias analysis. Thus, the efficiency of our translation, depends on precision of the underlying alias analysis.
- The complexity of our proposed technique is $O(n^3)$, where n is the number of statements in the program. We pre-compute the F and `rev` maps and reuse these maps in the functions `GenTD-LA` and `GenTD-AL` each of which has a complexity of $O(n)$. Since, there can $O(n^2)$ number of edges in the `LAG`, the complexity of the

```
dist D = cyclic (R, 2); // dist for L1
// distribution for the array A
Place distA (point i) {
  if (i%2 == 0) return D(i);
  if (i%2 == 1) return D(i);
  return 0; // redundant
}
// distribution for the array sum
Place distSum (point i) { return 0; }
// distribution for the array B
Place distB (point i) { return distA(i); }
// distribution for the Loop L2
Place distL2 (point i){ return distA(i); }
```

Figure 10. Generated dists for the program in Figure 4(a)

`redistribute` function is $O(n^3)$. Finally, the cost of invoking the `emit-dist-map` function for each modified node is again $O(n^3)$. Thus the overall complexity of our technique is $O(n^3)$.

Illustration

Consider the example code shown in Figure 4(a). Say the programmer identifies the loop `L1`, and wants it to be distributed using a cyclic distribution over two places. Since `L1` is accessing the arrays `A` and `sum`, these arrays need to be distributed. It can be seen that the array `A` is also accessed in loop `L2`. Once the array `A` is distributed in the previous step, we have to distribute the loop `L2`; which in turn would lead to redistribution of the array `B`. Figure 10 shows the generated distributions. It can be seen that the distribution of `L2`, `B` are same as the distribution of `A`. We can avoid emitting such distributions and reuse emitted distributions. A weakness of our framework can be seen in the distribution of `sum`. The function `GenTD-AL` fails (the function `rev` fails), and thus all the elements of `sum` are mapped to place 0; a smarter analysis could have done better. In the generated distribution for the array `A`, the last return statement is redundant – our code generation can be tuned to this effect.

4. Case Studies

In this section, we discuss our experience in applying the techniques presented in this paper onto real world benchmarks. We test the applicability of our techniques on benchmarks spanning three different benchmark suites – Eight benchmarks from Java Grande Forum [17] (JGF), Six benchmarks from HPC Challenge [16] (HPCC), and Ten benchmarks from Parsec [5].

Moldyn is part of the JGF benchmark suite. Figure 11(a) shows the parallel version of a part of the Moldyn benchmark. The goal is to distribute the first loop using a given distribution `D`. Figure 11(b) shows the transformation by our framework. Our refactoring framework first distributes the loop, then it distributes the array `P`, and then distributes the second and the third loop (all using the distribution `D`). It also transforms the code in the function `allreduce`; which involves distributing the loop therein, and the modification of the access to the variable `t.vir`. We have identified similar opportunities in two other benchmarks in the suite (crypt and montecarlo).

RandomAccess is part of the HPCC suite. Figure 12(a) shows the parallel version of the kernel of the RandomAccess program, and Figure 12(b) shows the transformation by our framework; the goal is to distribute the loop using a programmer-specified distribution `D` over the region `R`. The loop gets distributed, and so does the array `ranStarts` (using `D`). However, the loop also accesses the array `table`. There is no direct co-relation between the loop iterations and the accesses. And our tool does not change the distribution of `table`; as the function `GenTD-AL` returns with the variable $success$ set to $false$. Thus, we don't change the distribution of `table`, and introduce remote accesses. We have identified similar opportunities in FT benchmark of the suite.

BlackScholes is a part of the Parsec benchmark suite; it employs Black-Scholes Partial Differential Equation (PDE) to do option pricing. Figure 13(a) shows the parallel version of the snippet of the benchmark ported to X10. The parallel loop iterates over a region `[1:numOption]` with a stride of `NCO`. The goal is to distribute the array `prices` using a blocked distribution (`D`) over the region `[1:numOption]` with a blocking factor of `NCO`. Our algorithm distributes the array (say using `D`) and identifies the outermost loop has to be distributed using a distribution:

```
place dist (point i)
{ for (k=0;k<NCO;++k) return D(i+k);}.
```

(This code can be simplified to `return D(i+0)`.) Our algorithm further identifies that the same distribution is applicable to the arrays: `sptprice`, `strike`, `rate`, `volatility`, `otime`, and `otype`. Our algorithm also identifies that the array `data` has to be distributed, but the function `GenTD-AL` *fails*. Thus, our algorithm allocates all the elements of `data` at place 0. Note that, the appropriate distribution for the array `data` is actually identical to the distribution specified for the array `prices`. Similar to the BlackScholes benchmark, we have identified similar opportunities in all (ten) of the benchmarks from this suite.

Overall, we have observed that our redistribution algorithm is applicable in many different benchmark programs. We have also observed that there may be cases where our generated distributions sometimes are not concise and needs programmer help for better representation.

Performance gains: We have compared two versions of the benchmarks (i) benchmarks with only the parallel loops distributed, (ii) benchmarks where the parallel loops are distributed, along with other arrays and loops as suggested by our framework. We obtained the execution time numbers and observed significant improvement in performance (upto 100 x, for montecarlo). Due to the lack of space, we avoid presenting the details of these numbers, especially because

```
final region R=[1:NTHREADS];
void run() {
 finish foreach(point [j]:R)
         P[j].initialise(...);
 ...
 finish foreach(point [j]:R)
         P[j].runiters(...);
 ...
 finish foreach (point [j]:R){
   md myNode = P[j];
 ... } }
void allreduce() {
 finish foreach (point [j]: R) {
   for(point [k]: [0:(mdsize-1)]){
     ...
     P[j].one[k].zforce = ... }
   P[j].vir = t.vir; } }
```

(a)

```
final dist D=...(R)...;
void run() {
 finish  ateach (point [j]: D)
          P[j].initialise(...);
 ...
 finish  ateach (point [j]: D)
          P[j].runiters(...);
 ...
 finish  ateach (point [j]: D){
    md myNode = P[j];
   ... } }
void allreduce() {
 finish ateach (point [j]: D){
   for(point [k]: [0:(mdsize-1)]){
     ...
     P[j].one[k].zforce = ... }
   P[j].vir= future(t){ t.vir} .force(); } }
```

(b)

Figure 11. (a) Parallel Moldyn, (b) Distributed Moldyn. Changes shown in **bold**.

```
final region R = [0:maxPlaces-1];
final int[.] ranStarts=new int[R];
for (point p: R) ranStarts[p]=...;

...
finish foreach (point p : R) {
 int ran=nextRandom(ranStarts[p]);
 for (int count=1; count<=nUpdates;
                   count++) {
   final int j=f(ran);
   final int k=smallTable[g(ran)];
   final point q = Pt(j);
   atomic { table[q]=table[q] ^ k; }
   ran=nextRandom(ran);
 } }
```

(a)

```
final region R=[0:maxPlaces-1];
final dist D=... (R) ... ;
final int[.] ranStarts=new int[ D];
finish  ateach (point p: D)  ranStarts[p]=...;
...
finish  ateach (point p : D)  {
 int ran=nextRandom(ranStarts[p]);
 for (int count=1; count<=nUpdates;
                   count++) {
   final int j=f(ran);
   final int k=smallTable[g(ran)];
   final point q=Pt(j);
    async(table[q])
      { atomic{table[q]= table[q]^ k;}}
   ran=nextRandom(ran); } }
```

(b)

Figure 12. (a) Parallel RandomAccess, (b) Distributed RandomAccess. Changes shown in **bold**.

the importance of data locality in distributed threads is well known. These execution time behaviors were studied only for the benchmarks from JGF and HPCC. The Parsec benchmarks (written in C/C++ which do not support distributions) need to be ported fully to a language like X10 to study the gains; these are large benchmarks and porting them to X10 remains an interesting future work.

5. Future work

Arbitrary distributions: While X10 language description [12] discusses the idea of programmer-specified distributions, an efficient implementation of the language runtime that can support such arbitrary distributions is an open challenge.

Prototype : Implementing the framework in Eclipse type of environment is an involved exercise in itself and is left as future work. Further, making such an implementation applicable to multiple languages is another open challenge.

Optimizations: Our generated code is oblivious to the underlying distribution and thus may loose opportunities for generating efficient distributions. Identifying newer refactoring patterns and possible optimizations in the generated code would be an interesting area to explore.

```
region R=region (1:numOption,NCO);
finish foreach(point [i]: R){
 ...
 BlkSchlsEqEuroNoDiv(i,NCO,sptprice[i],
   strike[i], rate[i], volatility[i],
   otime[i], otype[i], 0);
 for (k=0; k<NCO; k++){
   prices[i+k] = price[k]; }
 ...
 for (k=0; k<NCO; k++) {
   priceDelta = data[i+k].DGrefval
                  - price[k]; }..}
```

Figure 13. Example snippet ported from BlackScholes

6. Conclusion

In this paper, we discuss our preliminary results on distributing arrays and loops using arbitrary programmer specified distributions. We feel that lack of automatic techniques to infer arbitrary distribution of data and computation had been one of the main reasons for programmers to restrict themselves pre-defined distributions. Our paper bridges and important gap in this context and we expect that there will be increased interest in the uses of arbitrary distributions by the application developers.

References

[1] S. Agarwal, R. Barik, V.K. Nandivada, R.K. Shyamasundar, and P. Varma. Static detection of place locality and elimination of runtime checks. In *Proceedings of the APLAS*, pages 53–74. LNCS, 2008.

[2] J. R. Allen and K. Kennedy. PFC: A program to convert fortran to parallel form. In *Proceedings of the Supercomputers: Design and Applications*, pages 186–203, August 1984.

[3] C. Ancourt and F. Irigoin. Automatic code distribution. In *Workshop on Compilers for Parallel Computers*, 1992.

[4] E. Ayguade, J. Garcia, M. Girons, M. L. Grande, and J. Labarta. DDT: A research tool for automatic data distribution in HPF. In *Scientific Programming*, 1995.

[5] C. Bienia, S. Kumar, J. Pal Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of PACT*, Oct 2008.

[6] B. Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Not.*, 34(4):61–69, 1999.

[7] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic generation of nested, fork-join parallelism. *Journal of Supercomputing*, pages 71–88, 1989.

[8] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Array distribution in data-parallel programs. In *Proceedings of LCPC*, pages 76–91. Springer-Verlag, 1994.

[9] W.R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings of OOPSLA*, pages 433–444, 1989.

[10] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the ICSE*, pages 397–407, 2009.

[11] Tarek El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC Language Specification v1.1.1, October 2003.

[12] V. Saraswat et al. Report on the experimental language X10, x10.sourceforge.net/docs/x10-101.pdf, 2006.

[13] P. H. Feiler and R. Medina-Mora. An incremental programming environment. *IEEE Trans. Software Engineering*, 7(5):472–482, September 1981.

[14] M. Gupta and P. Banerjee. PARADIGM: A compiler for automatic data distribution on multicomputers. In *In International Conference on Supercomputing*, pages 87–96, 1993.

[15] R. Gupta, S. Pande, K. Psarris, and V. Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25:13–14, 1999.

[16] HPC challenge benchmark. http://icl.cs.utk.edu/hpcc/.

[17] The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[18] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[19] K. Kennedy, Kathryn S. McKinley, and C-W. Tseng. Analysis and transformation in the ParaScope editor. In *Proceedings of the ICS*, pages 433–447. ACM, 1991.

[20] C. Koelbel, P. Mehrotra, and J. van Rosendale. Semi-automatic process partitioning for parallel computation. *Int. J. Parallel Program.*, 16(5):365–382, 1987.

[21] P Lee. Efficient algorithms for data distribution on distributed memory parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 8:825–839, 1997.

[22] P. Lee and W-Y Chen. Generating global name-space communication sets for array assignment statements. Technical report, Institute of Information Science, Academia Sinica.

[23] P. Lee and Z. M. Kedem. Automatic data and computation decomposition on distributed memory parallel computers. *ACM Trans. Programming Languages and Systems*, 24, 2002.

[24] S-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. SUIF explorer: an interactive and interprocedural parallelizer. In *PPoPP*, pages 37–48. ACM, 1999.

[25] M. E. Mace. *Memory storage patterns in parallel processing*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[26] S. Markstrum, R. M. Fuhrer, and T. D. Millstein. Towards concurrency refactoring for x10. In *PPOPP, poster*, pages 303–304, 2009.

[27] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the PLDI*, pages 69–80. ACM, Jun 1987.

[28] Mark Weiser. Program slicing. *IEEE Trans. Soft. Engg.*, 10(4):352–357, 1984.