Efficient, Portable Implementation Of Asynchronous Multi-place Programs

Ganesh Bikshandi
IBM STG, India
gbikshan@in.ibm.com

Jose G. Castanos
IBM T.J.Watson Research Center
castanos@us.ibm.com

Sreedhar B. Kodali
IBM STG, India
srkodali@in.ibm.com

V. Krishna Nandivada
IBM India Research Lab
nvkrishna@in.ibm.com

Igor Peshansky
IBM T.J.Watson Research Center
igorp@us.ibm.com

Vijay A. Saraswat
IBM T.J.Watson Research Center
vsaraswa@us.ibm.com

Sayantan Sur
IBM T.J.Watson Research Center
surs@us.ibm.com

Pradeep Varma
IBM India Research Lab
pvarma@in.ibm.com

Tong Wen
Interactive Supercomputing Inc.
tong.wen@gmail.com

Abstract

The X10 programming language is organized around the notion of places (an encapsulation of data and activities operating on the data), partitioned global address space (PGAS), and asynchronous computation and communication.

This paper introduces an expressive subset of X10, FLAT X10, designed to permit efficient execution across multiple single-threaded places with a simple runtime and without compromising on the productivity of X10. We present the design, implementation and evaluation of a compiler and runtime system for FLAT X10. The FLAT X10 compiler translates programs into C++ SPMD programs communicating using an active messaging infrastructure. It uses novel techniques to transform explicitly parallel programs into SPMD programs. The runtime system is based on IBM's LAPI (Low-level API) and is easily portable to other libraries such as GASNet and ARMCI.

Our implementation realizes performance comparable to hand-written MPI programs for well-known HPC benchmarks such as Random Access, Stream, and FFT, on a Federation-based cluster of Power5 SMPs (with hundreds of processors) and the Blue Gene (with thousands of processors). Submissions based on the work presented in this paper were co-winners of the 2007 and 2008 HPC Challenge Type II Awards.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Design, Languages

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.

Keywords X10, SPMD, compiler, runtime, PGAS, APGAS, asynchrony, HPC, HPC Challenge, Random Access, FFT, Stream

1. Introduction

The past several years have seen an explosion of mainstream architectural innovation — multi-cores, symmetric multiprocessors, clusters, and accelerators (such as the Cell processor, GPGPUs) — that now requires application programmers to confront varied concurrency and distribution issues. This raises the fundamental question: what programming model can application programmers productively use for such diverse machines and systems?

The X10 programming language [18] was designed to address the challenges of "productivity with performance" on these diverse architectures. Designed on a modern sequential object-oriented base with an advanced type system, X10 is an explicitly concurrent language that introduces a few core constructs for communication and distribution. These constructs are language-independent and form the basis of the *Asynchronous Partitioned Global Address Space* (APGAS) model. The APGAS model organizes computation into a collection of logical *places*. A place encapsulates data and one or more (asynchronously executing) *activities* that operate on the data. Places capture the idea of *locality* (data in the same place is "close", data in a different place is "far") and *heterogeneity* (one place may be targeted for a collection of tightly integrated cores, another may be targeted for a GPGPU). As in the *Partitioned Global Address Space (PGAS)* model (see languages such as UPC[5], Co-Array Fortran [15], and Titanium [12])) data in all places resides in a global address space; thus a field of an object can point to an object in a different place. Operations permit the allocation of data in multiple places as part of a single global data-structure (for example, a distributed array). In principle, an activity may perform any operation – read, write, call procedures, spawn other activities locally (at the same place), or remotely (at other places) etc. Constructs are provided for detecting termination and quiescence of activities, and for atomic execution.

With such flexibility comes an implementation challenge. Unlike other PGAS languages such as UPC, Co-array Fortran and Titanium that are organized around the notion of SPMD computa-

tion (when a computation is initiated, a thread is started in each place p0, p1, p2, ...), X10 is organized around *active messaging* [20]: computation is initiated with a single activity executing at place p0. Although, there has been research on how to compile and run SPMD style input programs, we are not aware of any work that tries to compile active messaging code onto multiple places and run it efficiently.

This paper introduces a subset of X10, FLAT X10, designed to be rich enough to express various HPC programs and yet retain a simple performance model. The programs are comparable to SPMD programs in languages such as UPC, but permit simple ways of overlapping computation with communication using the asynchronous features of FLAT X10. We also present a simple runtime and syntax-directed compilation strategy for FLAT X10, and evaluate its performance on several kernel programs. [1]

The compiler translates FLAT X10 programs to C++ programs. This strategy allows us to reuse powerful general purpose optimizations already built into C++ compilers for various architectures (e.g. x86, PPC).[2] The (C++) runtime is implemented on top of IBM's LAPI [13], a low-level active-messaging library that is the lowest programmable layer on the IBM Federation switch. LAPI permits efficient communication between multiple processes running on a cluster. We have also ported the relevant portions of LAPI on top of the Blue Gene Deep Computing Messaging Framework (DCMF) to get an implementation of FLAT X10 on the Blue Gene.

As evidence for the interestingness of FLAT X10 and the implementation in this paper, we remark that the FLAT X10 submissions (based on the work presented in this paper) were adjudged co-winners of the HPC Challenge Type II Awards for 2007, and also for 2008. Additionally, we show that several FLAT X10 programs achieve performance comparable to or better than corresponding MPI programs.

Our work makes the following contributions:

1. We identify a subset of X10, FLAT X10, which is rich enough to express many (SPMD) HPC programs of interest, while permitting the programmer to overlap computation with communication.
2. We show how these parallel programs with explicit synchronization primitives and asynchronous, fine-grained tasks may be compiled into multi-process SPMD programs that use an active-messaging library (LAPI).

   We show how to implement the FLAT X10 rooted exception model (which specifies how exceptions are propagated from activities to an enclosing termination detection construct) using the LAPI fence mechanism.

   We show that for single-threaded processes, FLAT X10's atomic may be implemented as a no-op.

   We also present efficient implementations of distributed arrays.

Additionally, we re-use several well-known techniques such as scalar privatization (this replicates side-effect free computation across places thus replacing communication with computation).

1.1 Rest of this paper

In the next section we motivate and introduce FLAT X10 and describe how its restrictions enable a particularly simple organization of the run-time. Section 3 describes the basic compilation scheme. Section 4 describes the organization of the runtime in more detail.

[1] Section 7 briefly discusses work completed after the submission of this paper which extends the ideas in this paper to an implementation of full X10.

[2] We used C++ rather than C so that we could take advantage of the C++ object model in implementing the X10 object model.

Section 5 describes experimental results obtained on a cluster of Power5 SMPs and on the Blue Gene. Finally we conclude with a discussion of related work and future work.

2. X10 and FLAT X10

Restricting our attention to concurrency control constructs, the X10 (v1.5) programming language can be thought of as extending the sequential subset of Java programming language (v1.4) with the following:

```
S   ::=  Statement
         finish S
         when (cond) S
         clock c = new clock();
         async (p)[clocked(c1,...,cn)]S
         c.resume();
         next;
         seqS
E   ::=  Expression
         new T[D](point p){E;};
         seqE
```

Here, for a syntactic category X, we use seqX to name a non-terminal whose productions specify the sequential constructs for X. For instance, in X10, the productions for seqS include conditionals (i.e. we have the production *seq(S)*::= if (e) S), (scalar and array) assignments, local variable declarations, (static and instance) method invocations, inner class declarations, try/catch/finally statements, throw statements, loops etc. The productions for seqE includes arithmetic operations, constructor invocations, etc

In brief, finish S executes the statement S and waits for all asyncs spawned (recursively) during its execution to terminate. A when statement suspends until such time as (if ever) the condition cond is true, and then it evaluates the body S. The successful evaluation of cond and the execution of S are performed in a single atomic step. Execution of the async statement creates a new activity at the place p, clocked on the clocks $c_1,\ldots,c_n$ (if the clause is present). This activity executes S asynchronously with the spawning activity. S is permitted to access final variables in its lexical environment.

A clock in X10 is a dynamically created barrier. Clocks are designed to permit determinate operations. Only activities registered on a clock may participate in operations on the clock. An activity *A* is automatically registered on a clock when it creates the clock. *A* can register a new activity *B* it is creating on some subset of clocks that *A* is registered on (using the clocked clause in async). The statement c.resume() signals to the clock c that the activity has completed its work in the current phase of the clock. The statement next causes the current activity to suspend until all activities registered on the clock have completed their work in the current phase of the clock. (Thus X10 supports *dynamic, split-phase* barriers.)

X10 provides a mapping (called *distribution*) from a collection of indices(called *points*) to places. The global array creation expression specifies that an array of type T is created at one or more places (as specified by the distribution D), with the element at index p initialized with the result of executing e.

X10 also provides some constructs as syntactic sugar over the basic constructs discussed above. For instance, atomic S abbreviates when(true) S. Similarly, ateach(point p: D) S abbreviates for(point p: D.region) async (D.dist(p)) S.

As the BNF productions indicate, these control constructs can be arbitrarily nested, subject to static semantic conditions. These conditions are motivated and defined explicitly in [18].

```
M ::= finish Z                                  | Z ::= ateach P                    | P ::= async (p)
      T[.] a = new T[u] (point p) {e}           |       clock c = new clock()       |        [clocked (c1,...,cn)] P
      Z                                         |       final T a = e               |       atomic P
      seqM                                      |       P                           |       next;
                                                |       seqZ                        |       seqP
```

Intuition: `M` ("Main") statements may be executed only from the "top level" activity. An array initializer is considered a top-level statement because it contains an implicit `finish`. `Z` ("place Zero") statements may be executed only at place `p0`. `P` ("any Place") statements may be executed at any place.

Figure 1. FLAT X10 Grammar.

2.1 FLAT X10 specification

FLAT X10 is obtained from `X10` by using the following intuitions. A key source of power in `X10` is the (unrestricted) `finish` statement. `finish S` requires a full distributed termination detection algorithm (e.g. see [3] as an entry point to the very rich literature on this topic), since `S` may spawn nested activities of arbitrary length, scattered across an arbitrary subset of places. However, most SPMD programs require a single global barrier. This can be obtained easily by restricting `finish` to occur only at "top level" statements, thus requiring `finish` statements to be *flat*.

Another source of expressiveness is arrays distributed over some subset of places in some programmer-specified fashion (e.g. block-cyclic). Such arrays are not hard to implement, at least conceptually. For convenience we choose to restrict FLAT X10 to only have arrays that are uniquely distributed across all places. (Arrays with user-defined distributions over all places can be built up from such arrays.)

We distinguish between two kinds of `asyncs`. The `ateach` statement collectively spawns (one or more) `asyncs` at each place in the underlying distribution. Since we are mostly interested in representing SPMD computations, we permit only activities at place 0 to execute `ateach` statements; further we require that the underlying distribution be unique. No additional restrictions are placed on the bodies of `ateach` statements.

Going beyond SPMD computations, we permit any activity to launch an activity at any other place – such activities are the primary vehicle for supporting communication overlapping with computation. For simplicity, we require that such `asyncs` be *flat*(i.e. they do not in turn spawn further asyncs) non-blocking and not throw any exceptions.[3] The requirement that the `async` not throw exceptions permits the implementation to reuse the LAPI completion notification mechanism (cf. `LAPI_FENCE`) to determine (at the source) that the `async` has terminated.

These restrictions are formalized in Figure 1. We further classify statements into three categories: *Main* statements (`M`), *Place Zero* statements (`Z`) and *Any Place* statements (`P`). `M` statements may be executed only from (methods called from within) the body of the `main` method at place `p0`. `Z` statments may be executed only at place `p0`. `P` statements may be executed at any place. (We distinguish between `M` and `Z` statements – which are all executed at place 0 – only to give a simple description of the restrictions on `finish`.) These restrictions are checked by the compiler. Each category permits method calls (as long as the body of the method belongs to the same category), and is closed under the sequential control constructs of the language. Figure 1 summarizes the productions for `M`, `Z` and `P`. The productions should be self-evident, given the motivation above; we remark that global array creation is restricted to `M` since it implicitly involves a `finish`.

[3] We have recently relaxed the flatness restriction to permit two-level asyncs as necessary for remote method invocation with return, i.e. in the body of an `async` *A* we permit an `async` back to the place which created *A*. A discussion of this extension is beyond the scope of this paper.

2.2 Programming in FLAT X10

Is FLAT X10 expressive? A surprisingly large number of programs can be expressed within this fragment. For instance the HPC Challenge benchmarks, NAS parallel benchmarks, stencil iteration, etc. In essence, FLAT X10 corresponds to a "PGAS + (flat) Active Messaging" computation model, which permits the expression of programs in PGAS languages such as UPC and programs which use a message-passing API such as MPI or an active-messaging API such as LAPI, GASNet or ARMCI.

We refer the reader to [10] for a detailed discussion on the implementation of the benchmarks, save for one illustration of overlap.

Random Access The kernel of the RandomAccess computation may be expressed by the following `M` method:

```
1. static void RAUpdate(final long NU,
2.   final long LogLocalTableSize,
3.   final LocalTable[.] Table) {
4.  finish ateach(point [p] : UNIQUE) {
5.    long ran=HPCC_starts(p*(NU/NP));
6.    for (long i=0; i<NU/NP; i++) {
7.        final long temp=ran;
8.        final int placeID =
9.           (int) ((ran>>LogLocalTableSize)&MASK);
10.        async(UNIQUE[placeID])
11.           Table[placeID].update(temp);
12.        ran = (ran << 1)^(ran < 0L ? POLY : 0L);
13.    } } }
```

The method executes a `finish`/`ateach`; within the `ateach` remote asyncs are spawned at Line 10. Note that FLAT X10 does not permit the spawning activity to wait for these spawned asyncs to terminate. Instead these `asyncs` are governed by the `finish` in the `M` code at Line 4.

FT–Overlap The HPCC FT computation shows in a different context that a `P`-procedure (called from a `Z`-procedure, Line 3) may spawn a remote inlined `async` (Line 16). `arrayCopy` is a built-in operation that moves bytes from a (possibly remote) array fragments into a local array. The source spawns an `async` to copy the newly computed array while it keeps going with the computation for the next place. All the `asyncs` are governed by the `finish` at Line 2. (See Figure 2.)

CG – Conditional Wait The kernel of the CG (Conjugate Gradient) benchmark uses an all-to-all "butterfly" reduction. This can be expressed through the following `P`-procedure. The procedure spawns a remote inlinable `async` which performs an atomic operation at the destination. The main activity running at the remote place performs a conditional wait (`when (done[i]);`).

In the implementation, such a conditional wait is implemented with a "busy help" strategy: by entering the communication subsystem, processing incoming active messages and subsequently testing the condition. This is a correct strategy because each place is single-threaded and hence the only activity that can change the condition is an activity received over the network. (See Figure 2.)

```
1. static void transposeBA(final Block[:rail] FFT)
2.  finish ateach(point [p]: UNIQUE)
3.    FFT[p].transpose(FFT, false);
4.
5.void transpose(final Block[:rail] FFT,
6.    final boolean a2b) {
7.  final double[:rail] Y= a2b? A: B;
8.  // local transpose
9.  int rowStartA = I*nRows;
10.  for (int k=0; k<NUM_PLACES; ++k) {
11.    int colStartA = k*nRows;
12.    transpose(Y);
13.    for (int i=0; i<nRows;++i) {
14.      final int srcI=(2*(i*SQRTN+colStartA)),
15.                 destI=2*(i*SQRTN+I*nRows);
16.      final int kk=k;
17.      async (UNIQUE[k])
18.          Runtime.arrayCopy(Y, srcI,
19.            (a2b? FFT[kk].B
20.               : FFT[kk].A), destI, 2*nRows);
21.    }}}
```

```
1. void sumReducePiecesComm(LocalVector[.] LV) {
2.  for (point [i] : rowPartner) {
3.   final int k = rowPartner[i];
4.   final int index = I*parent.py+k;
5.   final Vector myParent = parent;
6.   Runtime.arraycopy(e,0,buffer, i*size, size);
7.   async (UNIQUE[index]) {
8.     LocalVector target = LV[index];
9.     Runtime.arraycopy(buffer, i*size,
10.       target.scratch,i*size, size);
11.    atomic target.done[i]=true;
12.  }
13.  when (done[i]);
14.  done[i]=false;
15.  for (int m=0; m<size; ++m)
16.    e[m] += scratch[i*size+m];
17. }
18.}
```

Figure 2. FT and CG related pieces of code

2.3 Runtime for FLAT X10 – Overview

The flatness restrictions permit a particularly simple organization of the runtime library (X10LIB) which we now preview (details in Section 4). FLAT X10 programs are compiled to a single executable. Execution of an FLAT X10 program causes the executable to be launched at the number of places specified by the user. Each place is created with a *single* worker (thread), and this worker proceeds independently until it hits a *global barrier* operation. Three barrier operations are provided: `finishStart`, signaling entry into the body of a `finish` operation ; `finishEnd`, signaling exit from a `finish` and *variable broadcast* permitting a value computed at place `p0` to be broadcast to all other places, where it is stored in a local variable.

Local `asyncs` are executed inline. Remote asyncs are executed by sending an active-message to the target place. Incoming asyncs are handled by a worker when it enters the communication subsystem on a subroutine call through an explicit poll call or because of an outgoing communication. Incoming asyncs are executed inline to completion.[4] Inline execution of `asyncs` does not introduce spurious deadlocks since `asyncs` are non-blocking. Since `asyncs` cannot throw exceptions, the message completion mechanism in the underlying LAPI runtime can be used to communicate termination of the `async` to the originating thread.

Since only one worker is permitted per place, `atomic S` may be implemented as `S`: there is no need to obtain a lock to guarantee atomicity. Similarly, there is no need for sophisticated load-balancing schemes such as parallel depth-first scheduling or work-stealing that are needed by full `X10`.

3. SPMD translation of FLAT X10 programs

We now discuss how FLAT X10 programs are compiled.

After type-checking, the FLAT X10 compiler first verifies that the program is a valid FLAT X10 program and then translates the source program into an SPMD sequential C++ program[5] with calls into the X10LIB runtime. This runtime is responsible for implementing the APGAS abstractions – remote references, inter-process messages, barriers (see Section 4). We use a whole program analysis to do the verification; this involves resolving virtual function targets using class hierarchy analysis [9] (the precision of our analysis can be improved by improving the precision of the CHA).

The design of `X10` ensures that the only code that runs at places other than `p0` ("remote code") is the code in the body of `asyncs` and `ateaches`. The only interaction of the remote code with the statement within which it is embedded is straightforward: (a) the remote code may access `final` variables in the enclosing lexical environment, (b) the remote code may spawn (local or remote) `asyncs`, and (c) the remote code may throw an exception that should be transmitted to the governing `finish` statement.

The design of FLAT X10 ensures that a `finish` is executed only by `p0` and is never nested. Therefore a correct design for the implementation is as follows: (a) The code at places other than `p0` is organized in an "event loop" which waits for an *active message* to arrive, and executes it (the message specifies a function to be executed and the arguments to the function). (b) On executing an `ateach`, the thread at `p0` sends an active message corresponding to the body of the `ateach` to other places. (c) An `async` is executed by sending an active message that is executed inline by the thread at the target of the `async`. (d) On (normal or abrupt) completion of active message corresponding to the body of an `ateach`, and of all `asyncs` spawned by it, a message is sent to `p0` with this termination information. (e) On reaching the end of the statement in the body of a `finish`, the thread at `p0` waits for acknowledgements (sent in from every other thread). (f) Based on this termination status, the code at `p0` determines the next statement to be executed at `p0` and execution continues. In what follows we will refer to this design as the "Message passing" design. With extensions to support full `finish` this design can be extended to handle all of `X10`.

The design can be improved to permit fewer notifications of completion of `ateach` bodies to `p0`. (For discussion of related issues, see the discussion of "fork join" execution vs. SPMD execution in [6].) Consider the code:

```
finish {
  stm1;
  ateach(point p: UNIQUE) stm2;
  stm3;
  ateach(point q: UNIQUE) stm4;
}
```

[4] Thus, in the terminology of messaging systems such as LAPI, GASNet and ARMCI, FLAT X10 semantics does not require an independent progress guarantee for messages from the runtime. It is possible to extend our implementation scheme to permit interrupt-driven handling of messages.

[5] This means that the same program is run in all places.

```
cputime  = -mysecond(); // begin timing
finish ateach(point [p]: UNIQUE)
  for (i=0;i<N;i++) {
    final int placeID= ...;
    async (UNIQUE[placeID])
     Table[placeID].
      array[(int) (temp &Table[placeID].mask)]++;}
cputime = +mysecond(); // end timing
System.out.println (cputime);
```

```
if (here == p0) cputime = -mysecond();
finishStart(0);
Exception z = null;
try { for (i=0;i<N;++i) {
          const int placeID=...;
          LibAsync(...);// code to invoke the async.
      }
} catch (Exception e) {z=e}
finishEnd(z);
if (here == p0) {
  cputime += mysecond();
  System.out.println(cputime); }
```

Figure 3. Core of FLAT X10 RandomAccess program (left) and its SPMD translation into C++ (right).

In the "Message passing" design, notifications will be sent to `p0` from each place on completion of `stm2`. Assume that `stm1` and `stm3` do not throw any exceptions or otherwise exit the `finish` scope (`stm2` and `stm4` are permitted to throw exceptions). Under this condition, all places will execute `stm2` and `stm4`. If we ensure that the bodies of `ateach`s are executed at all places in "program order" (i.e. `stm2` is executed before `stm4` in the example above), the notification of completion of `stm4` will imply the completion of `stm2`. Hence the thread at `p0` does not need to wait for a completion message corresponding to `stm2`.[6]

This motivates the design of the alternative "SPMD" code generator (the topic of this paper). In this design, a single notification of completion is sent to `p0` (from all threads) for each `finish` (as opposed to one notification for each `ateach` within the scope of the `finish`). The statement in the body of a `finish` is translated compositionally to SPMD code. The translation of the sequential constructs of `X10` to C++ is routine and is skipped in what follows except for a few illustrative examples. In some cases for a phrase `X` of syntactic category χ we specify the translation $\sigma_\chi[\![\texttt{X}]\!]$ by specifying the resulting C++ code directly. In other cases we specify it indirectly through the translation of `X` into another FLAT X10 fragment `X'` of category χ' (i.e. $\sigma_\chi[\![\texttt{X}]\!]=\sigma_{\chi'}[\![\texttt{X}']\!]$). The rules are defined carefully so there is no infinite regress.

Main programs (M). We translate the FLAT X10 `finish` statement to a pair of `finishStart` and `finishEnd` statements. The functions `finishStart` and `finishEnd` are global fence routines (Section 4.2).

$\sigma_M[\![$`finish U`$]\!]$ =
```
  if (CS==0)
    CS=finishStart(0);
  Possibly jump rest of this code segment based on CS
  Exception z=null;
  σZ[[U]] // z is inscope in U.
  finishEnd(z);
  CS=0;
```

Array initialization. This is translated into an `ateach` loop, surrounded by a `finish`. We omit the details for lack of space.

Sequential control constructs are dealt with in a straightforward fashion. We illustrate with sequencing, conditionals and while statements. Note: $\sigma_M[\![\texttt{M}_1\ \texttt{M}_2]\!]=\sigma_M[\![\texttt{M}_1]\!]\sigma_M[\![\texttt{M}_2]\!]$.

`if` statements derived from `M` are translated by evaluating the predicate at place `p0` and then broadcasting (using the X10LIB call `sendReceive`) the value to all the other places. This predicate value is used as a guard for executing the body of the `if` statement. Thus:

$\sigma_M[\![$`if (e) M`$]\!]$ =
```
  bool flag;
  if (here==p0) flag=σM[[e]];
  sendReceive(flag);
  if (flag) σM[[M]]
```

`while` statements are translated similarly. Before every iteration of the loop, place `p0` evaluates the predicate of the loop and broadcasts this value to all the other places. The body of the `while` statement is executed (at any place) only if the predicate evaluates to true. (`for` loops are treated similar to the while loops.) Thus:

$\sigma_M[\![$`while (e) M`$]\!]$ =
```
  while (true){
    bool flag;
    if (here==p0) flag=e;
    sendReceive(flag);
    if (!flag) break;
    σM[[M]]
}
```

Place zero programs (Z). FLAT X10 admits `ateach` loops that iterate only over a unique distribution. Thus the translation of `ateach(point p : UNIQUE)P` is the translation of

```
try {
  final point p=[here.id];
  P
} catch (Exception u) {
  push(u, z);
}
```

Here `z` is the local variable introduced by the enclosing `finish` to record exceptions thrown locally.

Clock creation statements are translated to the empty statement. The FLAT X10 language permits in-scope `final` variables to be visible across all places. The initialization of a final variable involves evaluation of an expression *e* at place `p0`. The value is broadcast to all places in case the variable is referenced within an `async`.

A `P`-statement is translated thus:

$\sigma_Z[\![\texttt{P}]\!]$ = `if (here==p0)` $\sigma_P[\![\texttt{P}]\!]$

(Note that `Z` statements must be executed only at `p0`; hence the check.) Final assignments `final T a = e;` are translated in a similar fashion:

$\sigma_Z[\![$`final T a = e;`$]\!]$ =
```
  T a;
  if (here==p0) a=e;
  sendReceive(a)
```

The value is broadcast to other places since it might be referenced in an `async` later in the code; see below for a discussion of optimizations.

Sequential control constructs for *Z* are dealt similar to the translation shown for the sequential control constructs of *M*.

[6] The requirement that `stm1` and `stm3` not exit the `finish` scope permaturely is necessary; if they do, then according to the semantics of `finish`, either `stm1` or `stm3` (or both) should never be executed, and this scheme will fail.

All-place programs (P). `atomic S` is implemented as just `S`. This is possible because the single-threaded runtime implementation at each place executes `S` without any context switches. The `X10` language guarantees that `S` is sequential and non-blocking. The translation of such a statement generates no LAPI calls. In the absence of such calls, the runtime is guaranteed to execute `S` without any context switches (and hence there is no need to obtain locks).

Similary, `next` is translated to `GlobalSync();`, an invocation of a global barrier (Section 4.2).

Asyncs are dealt with thus. We convert the body of an async into a *closure*. The X10LIB routine `LibAsync` is responsible for invoking the closures and passing arguments to it. Thus the `async` statement is replaced with a call to `LibAsync`. The body of the `async` is outlined into a separate function with a uniquely generated name. `LibAsync` uses the name of the function to locate the function and execute it.

$\sigma_P[\![$`async (p) [clocked(`$c_1,\ldots,c_n$`)] P`$]\!]$ =

```
  // clocked arguments if any are ignored
  // environment required by async body
  // is sent as arguments.
  LibAsync(p, func-async1-id, args)
```

with the following code generated on the side:

```
void func-async1 (async-args-t  args) {
 // args explicitly unpacked here,
 σP[[P]]
}
```

The FLAT X10 language design ensures that clocks can be implemented through a global barrier (`GlobalSync`). Hence there is no need to implement a separate mechanism for tracking which activities are registered on which clocks. We omit details for lack of space.

3.1 Optimizations

The rules described above are basic SPMDization rules. Our implementation includes many additional optimizations:

- The value of a predicate is broadcast to all the places only if it is needed (body of the if-condition/while condition contains an `ateach`).
- The `final` variables whose initialization depends only on scalar variables and values and does not have any side effects (e.g. I/O) are not broadcast. Instead, the code is replicated across all the places (this reduces communication overhead).
- The value of a `final` variable is broadcasted/replicated, only if it is used in an `ateach` block or a remote `async`.
- Instead of broadcasting the value of the predicate of an if-statement separately, we use the parameters and the return value of the `finishStart` function to communicate the value of the predicate (besides working as the starting point of a barrier). Thus avoiding the cost of invoking the `sendReceive` function. [7]
- Using `finishStart` function to communicate the success or failure of predicates is an overhead for `finish` statements that are not within any if-statement. Thus we make a special case for `finish` statements that are within an if-statement and those which are not. For the later case, the `finishStart` function is only used as a barrier.
- While translating for-loops and while-loops, we can avoid broadcasting the loop-guard values and instead replicate the computation, if the computation does not involve any place specific data.
- We transform certain runtime calls into distributed array reductions by identifying specific patterns. In our experience, array initialization has been the chief beneficiary of such an optimization.

A related optimization that we perform as part of our compiler is that of merging of remote reductions. Instead of sending each remote reduction separately, our runtime aggregates remote reductions and then sends an aggregated reductions for evaluation. This has an impact on the communication overhead.

[7] We ignore the parameters and the return value in our presentation of rules in this section.

4. The FLAT X10 runtime: X10LIB

4.1 Activities and Messages

A FLAT X10 computation is defined by a FLAT X10 virtual machine, which consists of a collection of processes running on one or more computational nodes connected by a high-performance switch. Each process contains a single application thread and has a unique rank. These threads communicate by means of a messaging layer (and through shared memory). During the lifetime of the computation a thread may execute a large number of activities.

At any time, a thread is in one of two modes. Either it is performing local computations or it is executing an X10LIB call implemented in the messaging layer. There is no thread pre-emption or its related overhead in this model. While executing a library call, the thread may help the runtime library progress by handling incoming messages from other processes. We are concerned with three kinds of messages: *gets*, *puts* and *active messages*. A get message returns some data from the remote processor, and a put message places some data on the remote processor. An active message executes some user-specified code on the remote processor. The *get* and the *put* messages are used to transfer data between different computational nodes. The active messages implement remote inlinable `async`s, discussed in Section 2, and are executed as soon as they arrive.

Messages are not assumed to be delivered in order between pairs of processes or executed in the order in which they are delivered. Indeed, a message may be decomposed at the network layer, the pieces traveling to the final destination through different routes, and the message is then re-composed at the destination. Each message is acknowledged. The acknowledgment for a get message is the response to the get operation. The response for a put message is sent after the put operation has been performed at the remote node. The response for an immediate active message is sent as soon as all the packets for the message have been received and the (inline) completion handler has been run. The response for an eventual active message is sent once the packets for the message have been received (without waiting for the message processing to terminate).

4.2 Fences

The FLAT X10 runtime only provides a *global fence*. All threads must enter a global fence in order for any thread to exit the global fence. At the point in time at which a thread exits from a call to the global fence, the runtime guarantees that all `async`s issued by threads prior to their entry into the global fence have completed. Thus in the context of FLAT X10 programs, Flat `finish`es can be implemented with a global fence. (A more general distributed termination detection scheme is required for general finish.)

The global fence is made up of two functions `finishStart` and `finishEnd` that must be called in pairs (and must bracket the body of the `finish` statement that they implement). Figure 4 describes these functions in detail.

A `finishEnd` is a collective operation that combines a one-way barrier with a fence. Each process must wait (on a `LAPI_Fence`) to ensure that the data transfers associated with all messages sent by it have been completed. (LAPI also guarantees that all inline completion handlers have been executed. Since (inlinable) `async`s are implemented through such handlers, a return from `LAPI_Fence` is also a guarantee that all `async`s initiated by this thread have terminated.) An exception encountered by any process inside the `finish` (the argument of `finishEnd`) is communicated to the parent process (at place 0). The parent process deals with the set of exceptions as a part of the finish barrier.

On receipt of exception information from other processes, the `p0` thread determines where control should flow in all threads. This information (the "continue status") is communicated to other processes by `finishStart` through the `ContinueStatus` field in each child process. This integer carries the jump location for (guarded statements in) SPMD processes based on control flow decisions made in the `p0` thread. Only the `p0` thread writes a non-zero value into the `ContinueStatus` location of any child. The compiler generates code with these jump values, as discussed in section 3.

Figure 4 details the parent and child `finish` implementations. The value of `ContinueStatus` is used as the `ContinueCounter`. We reserve the special value 0 to mean that `ContinueStatus` has not been updated yet.

Thus on return from a global fence, it is guaranteed that the FLAT X10 computation is data quiescent(=no messages in flight) provided that all active messages sent by any process before it entered the global fence were inline messages.

4.3 Messaging Library

The FLAT X10 runtime (X10LIB) uses LAPI (Low Level Application Programming Interface) as its transport layer. X10LIB is designed for use by both programmers and compilers. It provides an API that implements the FLAT X10 constructs. Efficiency is the central challenge in designing a run-time for a high-performance language. X10LIB achieves this in the following ways.

- Polling mode and single thread of execution: LAPI supports both interrupt and polling for communication progress. As the interrupt cost is high, we use the polling mode and disabled the interrupt mode. LAPI also spawns a separate thread for executing the active message handler. We disable that thread and inline the execution in the user thread. This saves thread switching and synchronization overhead, at the same time providing determinate performance guarantees (each LAPI call involves polling).
- Small message optimization: LAPI supports efficient handling of messages that are only one packet in size. The method allows the active message handler to directly read the message from the network FIFO queues, without any intermediate software buffering. X10LIB uses this optimization for short messages.
- Aggregation and Buffering: Instead of immediately dispatching short asyncs, X10LIB aggregates them till a pre-defined threshold is reached (or a barrier is encountered), and sends those asyncs in a single message. X10LIB also provides API calls that avoids buffering of large chunks of data (for example, `asyncs` containing distributed array copies). The provided API call directly sends the user buffer to the remote place. It is the responsibility of the users of X10LIB (in our case the code generated by our compiler) to ensure the safe release of the buffer memory. Since LAPI is a non-blocking communication API, a return from a function does not signify that a buffer is ready for reuse. Our compiler releases or reuses the buffer after the next barrier point. The scheme also avoids the usage of LAPI counters to wait till the messages are sent, thus precluding LAPI from buffering large messages or using other costly techniques to ensure local completion.
- In FLAT X10, a reference is either local or remote. Local references point to objects, which are always word- (4- or 8-byte depending on the target machine) aligned. Remote references are represented by a tagged pointer to a proxy record that stores the actual address and the home place of the reference. The tag makes remote pointer checks efficient (no memory dereference) and uses the least-significant bit, which is conflict-free, since the language disallows interior pointers. (Remote references in X10 cannot be dereferenced directly, but only by spawning a remote async in the appropriate place.) Captured variables are serialized into a buffer during async invocation, and the resulting buffer is transmitted. Serialized primitive values become endianness-agnostic. Serialized remote references are simply the proxy record, which can become a local or remote reference when de-serialized. References to value objects are serialized by contiguously encoding each data member into a serial buffer (applied recursively if the data members refer to value objects). Polymorphic value references are encoded with a unique class identifier to allow dispatching to the de-serialization of the right type, which can be optimized for known hierarchies.

4.4 FLAT X10 on Blue Gene

The Blue Gene/L and Blue Gene/P port of FLAT X10 are built on top of X10LIB. We have implemented a LAPI port on top of a new low level Blue Gene communications library called DCMF ("Deep Computing Messaging Framework"). DCMF is the standard low level communications library in Blue Gene/P and supports higher level message libraries such as MPI, ARMCI and the UPC runtime. Like all Blue Gene communications libraries, DCMF is a user-space library, and relies on the characteristics of the Blue Gene networks such as guaranteed delivery of messages by the hardware, partition of messages into small self-contained packets, low latency, torus interconnect and high ratio of bandwidth to processor speed.

The decision to base the X10LIB port for Blue Gene on top of the LAPI API has the major advantage of abstracting the FLAT X10 development from the hardware. It allows us to run unmodified FLAT X10 programs on both Blue Gene and Power systems, and compare their performance. It simplifies tracking the evolution of the FLAT X10 environment. But it also has the disadvantage of not utilizing some Blue Gene features (such as the global network) which are not easily expressed through LAPI calls.

4.5 Limitations

Our methodology and the implementation have a few limitations: FLAT X10 is a garbage-collected language. The FLAT X10 compiler and runtime do not support garbage-collection. The implementation does not yet fully support separate compilation. Currently verification of valid FLAT X10 programs is fairly conservative: the culprits include a coarse grained termination detection mechanism, dependence of function inlining to enforce the program structure, and a compile-time assumption that implicit exceptions are not thrown in the programs.

These limitations are intended to be removed in future work.

5. Experimental Results

We implemented four programs: Stream, RandomAccess and FT from the HPC Challenge (HPCC) benchmark and FT from NAS parallel benchmark suite (NPB). The programs were implemented in FLAT X10 from first principles, following the HPCC and NPB guidelines. The base HPCC programs are pure C programs, while

```
int finishStart(int CS) {
  if (here == p0) {
   write CS into ContinueStatus for each child;
   LAPI_Fence;
   // Guaranteed that CS was written into every child
   return CS;
  } else {
    Wait on ContinueStatus, while continuing to
    process incoming messages;
    CS = ContinueStatus;
    ContinueStatus = 0;
    return CS;
  }
}
```

```
void finishEnd(const Exception* e) {
  LAPI_Fence;
  if (here == p0) {
    if (e != null) {
        -- append e into Error buffer
    }
    Wait until FinishEnd Counter reaches N-1,
    while continuing to process incoming messages;
    if (Error buffer not empty)
      throw MultipleExceptions(Error buffer);
    }
  } else {
    if (e != null) {
        perform an ActiveMessageSend that
          – appends e into Error buffer on parent.
          – Increment FinishEnd Counter at parent;
    } else {
        Increment FinishEnd Counter at parent;
    }
    LAPI_Fence;
}
```

Figure 4. Finish processing

the NPB FT program is a Fortran program. The FLAT X10 programs are compiled to C++ using the compiler presented in this paper. The base HPCC programs, NPB program and the compiled FLAT X10 programs are compiled using a native XL compiler and linked with IBM MPI and X10LIB respectively. For the sake of a fair comparison, we disable all the exception checking code (for example, null pointer check, array bounds check and so on) in the generated C++ code.

The resulting executables are executed in a cluster of nodes running the AIX operating system. Each node is a shared memory machine with 16 1.9 GHz P575 CPUs and 64 GB of physical memory. The nodes are connected by a high performance interconnection switch (HPS) which supports a user space communication protocol. The cluster has a maximum of 128 nodes. Irrespective of the number of nodes, we always assign equal number of processes to all the nodes and set the maximum number of processes per node to 16.

On Blue Gene/L, we run FLAT X10 programs on the compute nodes, using the light-weight kernel in the "Virtual Node Mode": every node is partitioned into two equivalent processes each of which uses half the 512MB main memory of each node. We also use the GNU tool chain for compilation – unfortunately, this does not take advantage of several optimizations in the dual floating point unit.

The code for Random Access, Stream and HPCC FT used for the performance numbers may be found in [10]. We have compared our results with that of the corresponding hand written programs in MPI on PowerPC clusters, and UPC programs (which perform way better compared to MPI) on Blue Gene/L. A *rack* consists of 2048 processors on these Blue Gene/L systems.

5.1 Stream

Stream is a simple benchmark program that measures sustainable memory bandwidth (in Gbyte/s) and the corresponding computation rate for four simple vector kernels: Copy (c $\leftarrow$ a), Scale (b $\leftarrow \alpha$ c), Add (c $\leftarrow$ a + b) and Triad (a $\leftarrow$ b + α c). Implementation of this benchmark is straight forward. The vectors *a*, *b* and *c* are 1-d distributed arrays. The computation is equally split among the places and does not involve any communication. The comparative results (GB/s) for Stream are shown in Figure 5.

5.2 RandomAccess

The RandomAccess benchmark measures the rate at which random memory locations in a distributed table of 64-bit integers can be updated. The update is a read-modify-write XOR operation. The benchmark reports the results as **G**iga **U**pdates **P**er **S**econd. Updates can either local or remote. In case the update is to a remote memory location, communication is required to transmit the update to the remote processor which owns the fragment of the table. Upon receiving the update, the processor then applies it to its part of the table.

The HPCC benchmark suite distributes an MPI version of the program. This MPI version employs a benchmark-specific aggregation scheme. This aggregation cannot be done inside the MPI library, rather needs to be specially crafted by the programmer. Unfortunately, the distributed benchmark makes too many `MPI_Test` calls to poll for incoming messages. For a fair comparison, we further optimized the MPI version of the benchmark and came up with a version that makes less frequent calls to `MPI_Test`. Our version improved performance by a factor of 5 on a single processor. Others have employed such techniques when dealing with this benchmark [11].

Figure 6 presents the results of our experiments with these two versions of the program. Generally, as we go towards full loading (16 processes) of a node, performance dips significantly. Since FLAT X10 programs demonstrates large GUP/s than MPI, this effect gets visible resulting in an non-smooth graph.

On Blue Gene, we compare with a UPC program that ran on an earlier (faster) version of the low-level messaging layer on Blue Gene. A more recent run of the `X10` Random Access program (as submitted to the HPC Challenge 2008 competition) shows the same performance as the UPC program for 8 racks (16384 processes).

5.3 FT

HPCC FT measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform of a vector of size *m*. In the distributed version, the vector is split equally among the processors. The vector is viewed as a logical two dimensional array, distributed along the *x*-dimension.

The global transposition involves communication among all the processors. Various trade offs like in-place or out-of-place transposition, blocking (the local transposition loop) or not, using collective communication or point-wise communication, overlapping

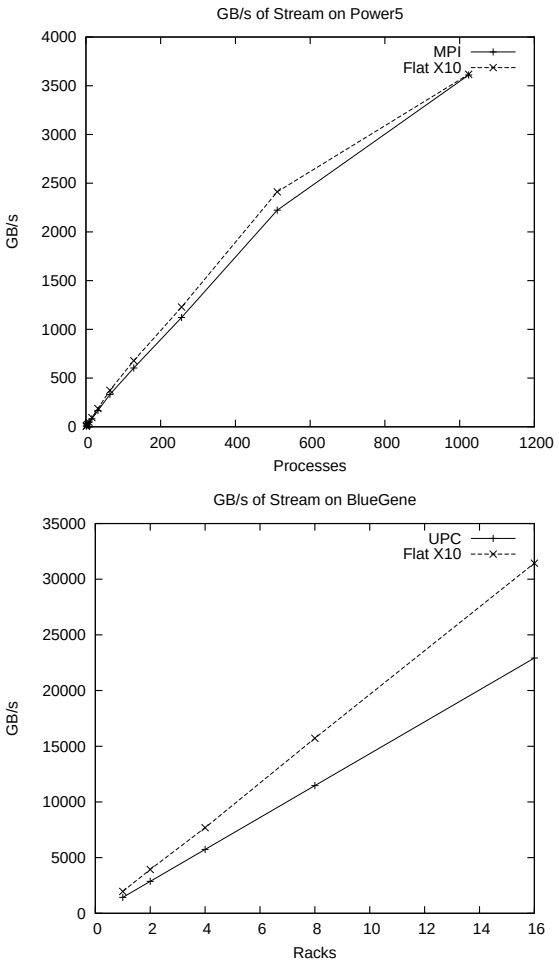

Figure 5. HPCC Stream

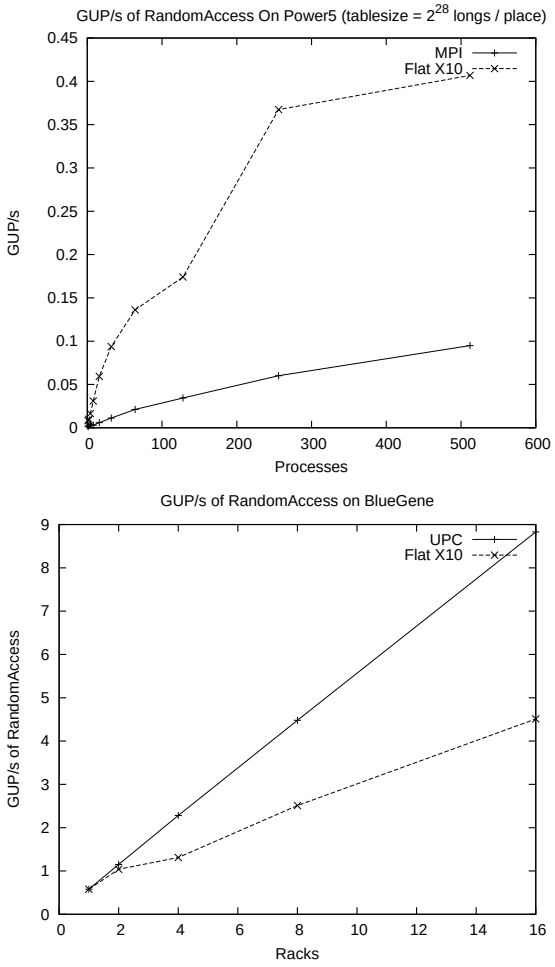

Figure 6. HPCC RandomAccess

(the communication with local transposition) or not, lead to several possible versions of the global transposition step. We present the best performing design alternatives in this section.

In `FX10-VER1`, each processor identifies the chunk to be sent to processor *p*, locally transposes it, and copies it to *p* using one-sided *arrayCopy* interface. A series of *num_procs* local transposition followed by *arrayCopy* completes the global exchange. The local transposition uses blocking for optimal cache utilization. Each array copy is overlapped with the local transposition of the next chunk:

```
finish ateach(UNIQUE) {
  for (int k = 0;  k < N ; k++) {
     perform local transposition of the block k;
     async (k) {  Arraycopy(...); } } }
```

The HPCC version uses MPI_Alltoall, whereas the `FX10-VER1` uses array copies and overlapped transpositions. Thus the comparison between `FX10-VER1` and the HPCC version may not look fair. Hence, we wrote a version of FT in FLAT X10 (`FX10-VER2`) that mimics the HPCC version in all aspects except that it uses a `point-to-point array copies loop` to achieve the same thing that MPI_Alltoall achieves. After the compilation, the point-to-point loop was replaced by MPI_Alltoall by hand. This way, we ensure that both the programs are equivalent. Inclusion of MPI collectives in the `X10` language is an ongoing research topic.

The first graph in Figure 7 compares the Giga-flops of `FX10-VER1` and `FX10-VER2` with those of the HPCC program. The second graph in Figure 7 shows that FLAT X10 (`FX10-VER1`) outperforms the UPC FT on Blue Gene. For a fair comparison, the local Fourier transform is computed using same external FFT routines in all the programs.

5.4 NPB FT

The FT program from the NAS parallel benchmark (NPB) set performs a three dimensional Fourier transform, unlike HPCC FT. The input array is still one-dimensional, but viewed as a logical three dimensional array. The last dimension of the array is distributed among the processors, while the other two dimensions are kept local. Local Fourier transform is applied along the two local dimensions. Then, a transposition is performed to exchange the distributed dimension with one of the local dimensions. Finally, another Fourier transform is applied along that dimension.

The above steps can be modified slightly to allow overlapping of computation and communication (similar to the snippet shown in section 5.3). In this version, for every XY plane, the following steps are applied in every place:

1. Compute the Fourier transform of all the columns of the XY plane.
2. Compute the Fourier transform of a row of the XY plane.

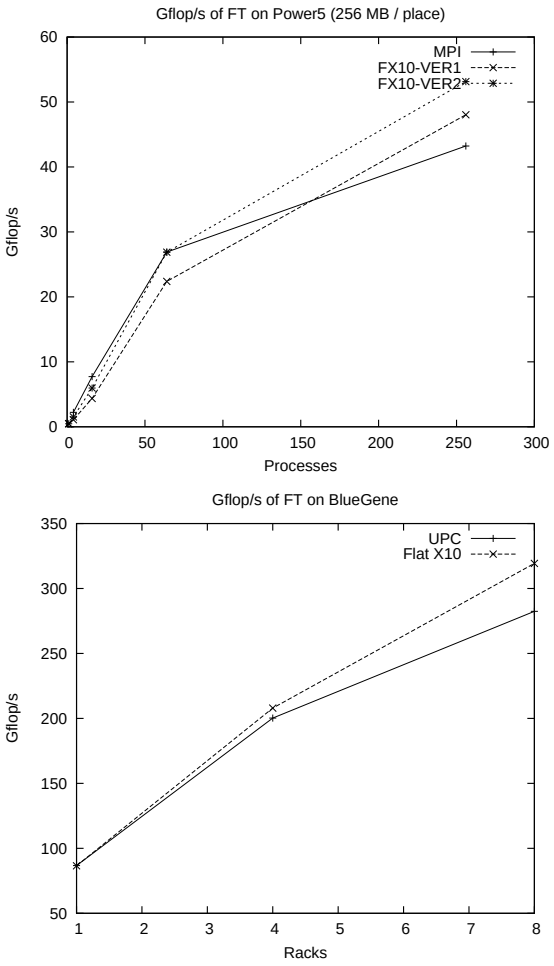

Figure 7. HPCC FT

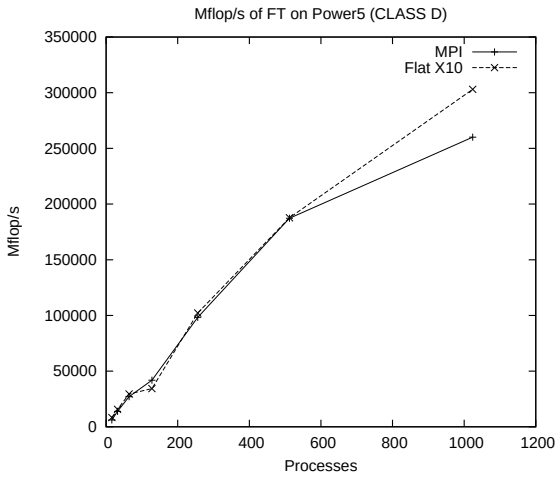

Figure 8. NPB FT

3. Send the row to transposition partner.
4. Repeat steps 2 and 3 for the remaining rows of the XY plane.
5. Compute the Fourier transform of all the columns of XY plane.

The steps 2 and 3 are overlapped which minimizes the communication latency. We applied this optimization to both the FLAT X10 and MPI programs. The experimental results (Megaflops) of FLAT X10 and the NPB MPI FT program is listed in Figure 8.

6. Related Work

Darema et al. [7, 8] present the SPMD parallel model and techniques for direct coding of SPMD programs. SPMD programs are written directly by users with macros and macro-processing support to ease the coding task. No automatic SPMDization of non SPMD programs is carried out. A technique of generating serial sections is described, wherein only one process executes the section while others jump to the end of the section. In our work, serial sections are generated automatically according to this method. Our work goes one step further by communicating jump labels across the machine for conditional control flow. A conditional execution of a parallel section is carried out by communicating the condition's (serial) evaluation result as a control flow variable that decides to the jump label that individual processes skip to.

Cytron et al. [6] present an approach for transforming code written in fork-join style to SPMD code. The approach is specific to nested fork-join parallelism and merges fork-join regions at one nesting level so that the intervening connecting sequential code also executes with (redundant) parallelism. The approach assumes that the execution of parallel loops in the input program is deterministic: that there are no data dependencies among parallel threads so that no race conditions are present and that no synchronization constructs beyond the implicit fork-join barriers are present. Tseng [19] follows up on Cytron et al. in translating fork-join parallel loops into (merged) SPMD regions. Once SPMD regions have been formed, the barrier communications among them are targeted for optimization using communication analysis.

Amarsinghe et al. [1] present distributed-memory SPMD code generation techniques starting with sequential Fortran-77. The compiler generates send and receives, eliminates redundant communications, aggregates small messages, allocates space locally on each processor and translates global addresses to local addresses.

Paalvast et al. [17] describe SPMD code generation for a high-level, parallel, programming language called Booster using a functional calculus called the view calculus. No benchmarks of any Booster implementation are provided. The calculus is used to build an annotation model for generating SPMD programs in [16].

Wallach et al. [21] propose optimistic active messages as a mechanism for allowing arbitrary user code to execute in handlers, with normal constraints on handler-executed code being met by dynamic checking and a run-time mechanism similar to lazy task creation [14] when needed. Code failing to meet handler constraints (e.g. code that reaches a blocked state) is detected and shifted from handler execution to a thread-based task that is created on the fly, or a negative acknowledgment sent back for possible revised active message(s). In contrast to this, our work establishes compliance (or non-compliance) with handler constraints statically and bypasses all run-time costs. This keeps our run-time footprint very light. In addition, our runtime utilizes LAPI [13] which provides a very efficient implementation of Active messages on a variety of networking technologies.

There have been various efforts to realize remote pointers. These include the well known "fat" pointer technique (a struct with address and location of the pointer), the "current memory pointer" approach of Berkeley-UPC [5] run-time (every allocation request at any thread should start from the memory pointer kept at thread 0 and the allocating thread should send the new pointer value back

to the thread 0), and the heavyweight Shared Variable Directory (SVD) approach of Barton et al. [2] (a distributed symbol table SVD has to be maintained across all partitions). The X10lib implementation of remote references is very efficient in that it neither increases the C pointer sizes (like fat pointers) nor introduce communication and synchronization overhead (like current-memory-pointer) nor uses heavy-weight data structures (like SVD).

7. Conclusion and Future work

In this paper we have identified a subset of `X10`, FLAT X10, that is rich enough to express SPMD programs augmented with asynchronous messaging that may be used to overlap communication with computation. We have shown that a set of simple compilation techniques and a simple runtime may be used to implement this language efficiently. We have demonstrated performance comparable to MPI for several benchmarks on a cluster of Power5 SMPs and on the Blue Gene.

Our main objective for future work is to extend the compilation scheme described in this paper to full `X10`. This is concerned with the following major extensions:

- The introduction of an explicit framework for scheduling multiple activities at a single (multi-threaded) place. This will allow the compiler to handle non-inlinable `async`s. We have developed a work-stealing scheduler for `X10`, extending the Cilk work-stealing scheduler [4]. The compilation scheme presented here needs to be extended with a continuation-passing analysis to permit generation of efficient work-stealing code.
- The introduction of non-global barrier operations (multi-place clocks), and nested `finish` operations. This entails significant work in the runtime – extending the single-process JAVA-based implementation in our open source release.

Acknowledgments

We thank Guojing Cong, Nathaniel Nystrom, Mark Stephenson, Vipin Sachdeva, Gheorghe Almasi, Calin Cascaval and Vivek Sarkar for extensive discussions. The HPC Challenge 2008 submission was done jointly with the IBM UPC team led by Calin Cascaval and Gheorghe Almasi. We thank the LAPI team (particularly Hanhong Xue, Chulho Kim, Robert Blackmore, Bill Tuel and Kevin Gildea) for their support. We thank Lauren Smith for efficiently shepherding this submission.

This material is based upon work supported in part by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] Saman P. Amarasinghe and Monica S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 126–138. ACM, 1993.

[2] Christopher Barton, CĆlin Casçaval, George Almási, Yili Zheng, Montse Farreras, Siddhartha Chatterje, and José Nelson Amaral. Shared memory programming for large scale machines. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 108–117. ACM, 2006.

[3] Stephen M. Blackburn, Richard L. Hudson, Ron Morrison, David S. Munro, and John Zigman. Starting with termination: A methodology for building distributed garbage collection algorithms. *Aust. Comput. Sci. Commun*, 23:2001, 2001.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 1995.

[5] UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Laboratory, 2005.

[6] Ron Cytron, Jim Lipkis, and Edith Schonberg. A Compiler-Assisted Approach to SPMD Execution. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 398–406. IEEE Computer Society, 1990.

[7] F. Darema-Rogers, D. A. George, V.A. Norton, and G.F. Pfister. A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN. *Parallel Computing*, 7:11–24, 1988.

[8] F. Darema-Rogers, V.A. Norton, and G.F. Pfister. Using A Single-Program-Multiple-Data Computational Model for Parallel Execution of Scientific Applications. Technical Report RC 11552, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1985.

[9] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 77–101. Springer-Verlag, 1995.

[10] V. Saraswat *et al*. HPC challenge 07: X10, 2007.

[11] R. Garg and Y. Sabharwal. MPI and Communication - Software Routing and Aggregation of Messages to Optimize the Performance of HPCC RandomAccess Benchmark. In *SuperComputing*, Nov 2006.

[12] Paul N. Hilfinger, Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. Titanium Language Reference Manual. Technical report, University of California at Berkeley, 2001.

[13] IBM International Technical Support Organization Poughkeepsie Center. Overview of LAPI. www.redbooks.ibm.com/ redbooks/ pdfs/ sg242080.pdf, 2008.

[14] Eric Mohr, David A. Kranz, and Jr. Robert H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 185–197. ACM, 1990.

[15] R. Numrich and J. Reid. Co-array fortran for parallel programming, 1998.

[16] E. M. Paalvast, L. C. Breebart, and H. J. Sips. An expressive annotation model for generating SPMD programs. In *Scalable High Performance Computing Conference*, pages 208–211. IEEE Computer Society, 1992.

[17] Edwin M. Paalvast, Arjan J. van Gemund, and Henk J. Sips. A method for parallel program generation with an application to the Booster language. *SIGARCH Comput. Archit. News*, 18(3b):457–469, 1990.

[18] Vijay A. Saraswat. `X10` Language Report. Technical report, IBM Research, 2004.

[19] Chau-Wen Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 144–155. ACM, 1995.

[20] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266. ACM, 1992.

[21] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic active messages: a mechanism for scheduling communication with computation. In *Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 217–226. ACM, 1995.