HPCS languages: Fortress, Chapel, and X10

Rhaad Rabbani

& Bingjing Zhang

Outline

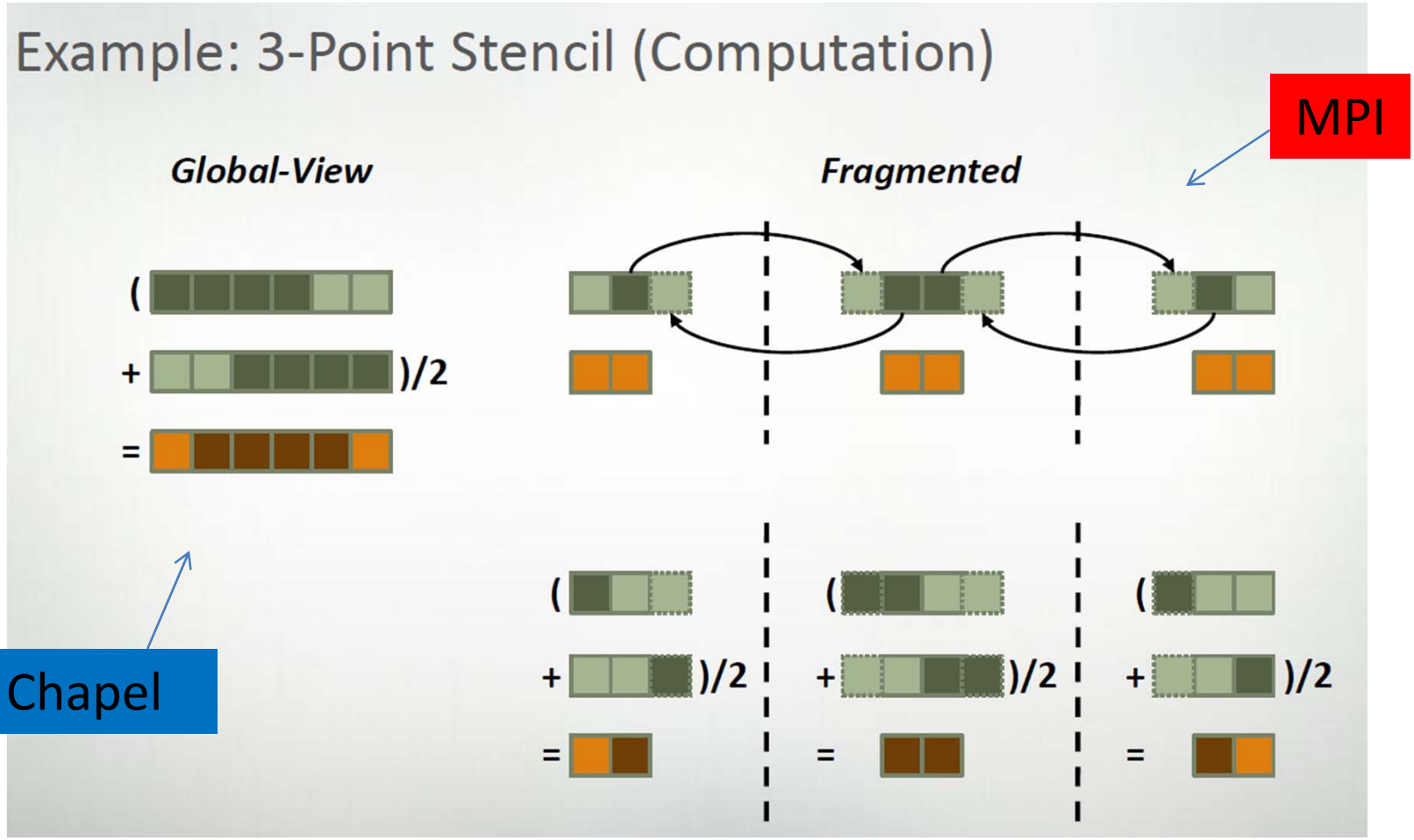
- Introduction to the HPCS programme
  - *HPCS* stands for *High Productivity Computing Systems*
- For each language in {Fortress, Chapel, X10}:
  - Characteristics
  - Syntax / Semantics
  - Task Parallelism
  - Data Parallelism
- The progress and future of HPCS languages

Introduction to HPCS

- High Productivity Computing Systems (HPCS) programme
  - Launched by DARPA in 2002
  - Goal: to build highly productive systems
- Productivity = performance + programmability + portability + robustness
- High performance achieved through parallel computing
- Use of MPI leads to low programmability
- Need to research & design new programming languages to address the lack of programmability

Programmability in Parallel Computing

- Global-View model
  - Programmer writes code to describe computation as a whole
- Fragmented Model
  - Programmer takes point of view of a single processor/thread



Source: http://chapel.cray.com/tutorials/DC2010/DC01-BACKGROUND.pdf

Programmability in Parallel Computing

- Global-View model
  - Programmer writes code to describe computation as a whole
- Fragmented Model
  - Programmer takes point of view of a single processor/thread

Example: 3-Point Stencil (Code)

Assumes p divides n

Global-View

```
def main() {
  var n = 1000;
  var A, B: [1..n] real;

  forall i in 2..n-1 do
    B(i) = (A(i-1)+A(i+1))/2;
}
```

Fragmented

```
def main() {
  var n = 1000;
  var me = commRank(), p = commSize(),
    myN = n/p, myLo = 1, myHi = myN;
  var A, B: [0..myN+1] real;

  if me < p {
    send(me+1, A(myN));
    recv(me+1, A(myN+1));
  } else myHi = myN-1;
  if me > 1 {
    send(me-1, A(1));
    recv(me-1, A(0));
  } else myLo = 2;
  for i in myLo..myHi do
    B(i) = (A(i-1)+A(i+1))/2;
}
```

Source: http://chapel.cray.com/tutorials/DC2010/DC01-BACKGROUND.pdf

Programmability in Parallel Computing

- MPI code size blows up exponentially for higher dimensional Stencil problems
- Important, for programmability, to separate algorithm (tasks and computations on data) from implementation (task and data distribution)
- All 3 HPCS languages use the Global-View model

Common Features of HPCS Languages

- Global-View model
- New syntax and semantics

} Programmability

- Task parallelization
- Data parallelization

} Performance

Fortress

- Intended to stand for ‘secure Fortran’
  - although very different from Fortran in reality
  - not even backwards compatible with Fortran
- Initially funded by Sun, now an Open-Source project under SunLabs
- Supports Unicode and standard mathematical notation
- Implicitly parallel programming language
  - Most language constructs parallel by default
- Currently implemented on the Java VM

Fortress Syntax

$sum\colon \mathbb{R}64 := 0$
for $k \leftarrow 1\colon n$ do
  $a_k := (1-\alpha)b_k$
  $sum \mathrel{+}= c_k\, x^k$
end

```
sum: RR64 := 0
for k<-1:n do
  a[k] := (1-alpha)b[k]
  sum += c[k] x^k
end
```

- Accommodates stylistic rendering of ASCII code
  - Follows standard mathematical notation
- Whitespace can be used to represent multiplication, string concatenation or a function call
  - When in doubt, parenthesize

Source: http://projectfortress.sun.com/Projects/Community/attachment/wiki/WikiStart/reference.pdf?format=raw

Fortress Syntax continued …

$$u = n(n+1)\sin 3n\, x \log\log y$$
$$w = (n(n+1))(\sin(3n)x)(\log(\log y))$$

```
SUM[x <- xs, y <- ys] x y
```

$$\sum_{\substack{x \leftarrow xs \\ y \leftarrow ys}} x\, y$$

```
<| x^2 | x <- xs, x > 43 |>
```

$$\langle\, x^2 \mid x \leftarrow xs, x > 43 \,\rangle$$

Source: http://projectfortress.sun.com/Projects/Community/attachment/wiki/WikiStart/reference.pdf?format=raw

Fortress: Implicit Task Parallelization

- Whenever Fortress sees the potential for parallelization, it splits the current task.
  - 2 separate tasks will evaluate the two operands of the add operator
  - fib must not have side-effects for this to work correctly

```
fib(n: ZZ32): ZZ32 =
    if n < 2 then n
    else fib(n - 1) + fib(n - 2)
    end;
```

- Uses the same work-stealing algorithm as Cilk++
  - Not efficiently implementable on the Java VM

Source: http://software.intel.com/en-us/articles/first-impressions-of-the-fortress-language/

Fortress: Implicit Task Parallelization

- For loops are parallel by default unless the keyword "sequential" is used before the generator (e.g. 1 : 10, that generates a sequence of integers)

```
for i ← 1 : 10 do
    print(i " ")
    end
```

could potentially output 5 4 6 3 7 2 9 10 1 8

- The keyword "atomic" can be used for the atomic update of any shared location inside the loop

```
for i <- 1:n do atomic result := result i end
```

Fortress: Task Parallelization

Potentially Parallel Constructs

- Tuples: `(a,b,c) = (f(x),g(y),h(z))`
- Parallel blocks: `do foo(a) also do foo(b) end`
- Functions, operators, method call recipients, and their arguments: `fail "Division by zero"`, `13-5`, `receiver.method(v)`
- Expressions with generators:

```
> SUM[x <- xs, y <- ys] x y
```

$$\sum_{\substack{x \leftarrow xs \\ y \leftarrow ys}} x\,y$$

```
> <| x^2 | x <- xs, x > 43 |>
```

$$\langle\, x^2 \mid x \leftarrow xs, x > 43 \,\rangle$$

Source: http://projectfortress.sun.com/Projects/Community/attachment/wiki/WikiStart/reference.pdf?format=raw

Fortress: Data Parallelization

- Default types provided include lists, vectors, sets, maps, matrices, multi-dimensional arrays
- A variety of built-in distributions for each data type easily allows data parallelization over any number of nodes
- Any standard reduction operations - like summation of all elements or finding the maximal element – or any user-defined reduction can be performed over such distributed data structures with minimal coding (using the Global-View model)

Chapel

- CHaPeL: Cascade High Productivity Language
- Under development at Cray Inc.
- Not directly based on any existing language
  - Borrows mainly from C, Fortran, Java and Ada
- Mainly built for data parallelization
- Explicit data and task parallelization

Chapel: Data Parallelization

- A domain is a set of indices
- A domain in any dimension may be defined

```
var domn : domain(2) = [1..m, 1..n];
```

- A data array is essentially a map from a domain to data values

```
var matr: [domn] float;
```

- Indices in a domain may be distributed over multiple “locale”s
  - The distribution may be customized by the user

- A data array element is stored at the “locale” that contains the associated domain index

Chapel: Data Parallelization

- Data parallelization is explicit
- “forall” is the parallel version of “for”

```
forall x in domn
    do writeln(“And the next element is ", matr(x));
```

- Atomic sections within the forall are supported
- Reduction operations can be easily performed over a distributed data array

```
var res = max reduce abs(matr);
```

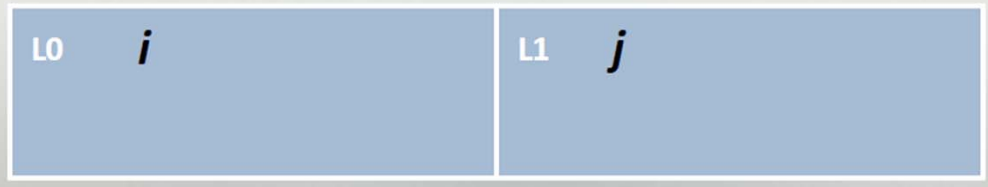
Chapel: Data Parallelization

- Each thread belongs to a specific locale, but can touch data in other locales, but at an increased cost

- Returns the locale on which *expr* is stored

Example

```
var i: int;
on Locales(1) {
  var j: int;
  writeln(i.locale.id, j.locale.id); // outputs 01
}
```



Source: http://chapel.cray.com/tutorials/DC2010/DC06-LOCALITY.pdf

Chapel: Task Parallelization

- Task parallelization is explicit
- cobegin can be used to run multiple tasks concurrently

  ```
  cobegin { fred(a); fred(b); fred(c); }
  ```

- begin can be used to spawn a child thread

  ```
  begin { whatever the child ought to do }
  ```

- Language features, such as sync or single, exist that allow synchronization with the spawned child thread

IBM X10

- X10 is a new language developed in the IBM PERCS project as part of the DARPA program on High Productivity Computing Systems (HPCS)
- X10 is an APGAS (Asynchronous Partitioned Global Address Space) language
- "X10" is meant to represent a "ten-times productivity boost.“
- The design goals for X10 are to create a language which is simple, safe, powerful, scalable, universal.
- IBM researchers claim that X10
  - is more productive than current models.
  - can support high levels of abstraction
  - can exploit multiple levels of parallelism and non-uniform data access
  - is suitable for multiple architectures, and multiple workloads.

Characteristics

- X10 is a strongly typed, concurrent, imperative, object-oriented programming language designed for productivity and performance on modern multi-core and clustered architectures. It is especially good at distributing your application over a cluster of distributed memory machines.

- X10 augments the familiar class-based object-oriented programming model with constructs to support execution across multiple address spaces, including constructs for a global object model, asynchrony and atomicity.

- However, X10 is still in an early stage of development, and is still somewhat unstable.

Concurrency in X10

- X10 supports two levels of concurrency.
- The first level corresponds to concurrency within a single shared-memory process (represented by an X10 Place).
  - Usually, you would use one Place per shared memory multiprocessor.
  - The main construct for concurrency within a Place is the X10 **async** construct.
- The second level of X10 concurrency supports parallelism across processes that do not share memory.
  - Usually, this would correspond to concurrency across nodes in a cluster.
  - The main construct for managing such concurrency in X10 is the **at** construct.
- Additionally, X10 provides various libraries and features to support particular concurrent operations and data structures, such as reductions and distributed arrays.

The first level Concurrency

- Three constructs
  - async {S}
  - finish {S}
  - atomic {S}

- Clocks
  - Generalized barrier for concurrency

async, finish

- async {S}
  - In X10, **async S** asynchronous executes statement **S** located to the place at which the current computation is running .
  - In the meantime, the current thread continues. You can think of **async** as a starting an X10 activity, which is a lightweight thread.

- finish {S}
  - In X10, **finish S** is used to synchronize with all the asynchronous activities that arise during the execution of **S**.
  - Specifically, the finish statement will wait until all **async**s spawned (transitively) by **S** finish.

Code Example

```
public static def main(argv:Rail[String]!) {
    val sums = Rail.make[Int](2, (Int) => 0);
    finish {
      async
        sums(0) = sum(1, 100, (i:Int) => i*i);
      async{
        sums(1) = sum(1, 1000, (i:Int) => i);
      }
    }
    val t = sums(0) + sums(1);
    x10.io.Console.OUT.println("t=" + t);
  }
```

Spawn an activity

wait for finish

Spawn another activity

atomic

- The **atomic** keyword marks operations that will perform atomically.

- it grabs a lock that serializes all atomic operations in a Place. Usually, **atomic** should be used for prototyping, but it will probably not scale well in highly contended code.

- For better performance, the **x10.util.concurrent** library provides various atomic operation and locks, which are implemented more efficiently.
  - e.g. Change from **atomic{v(0) += i; }** to **v.getAndAdd(i);**

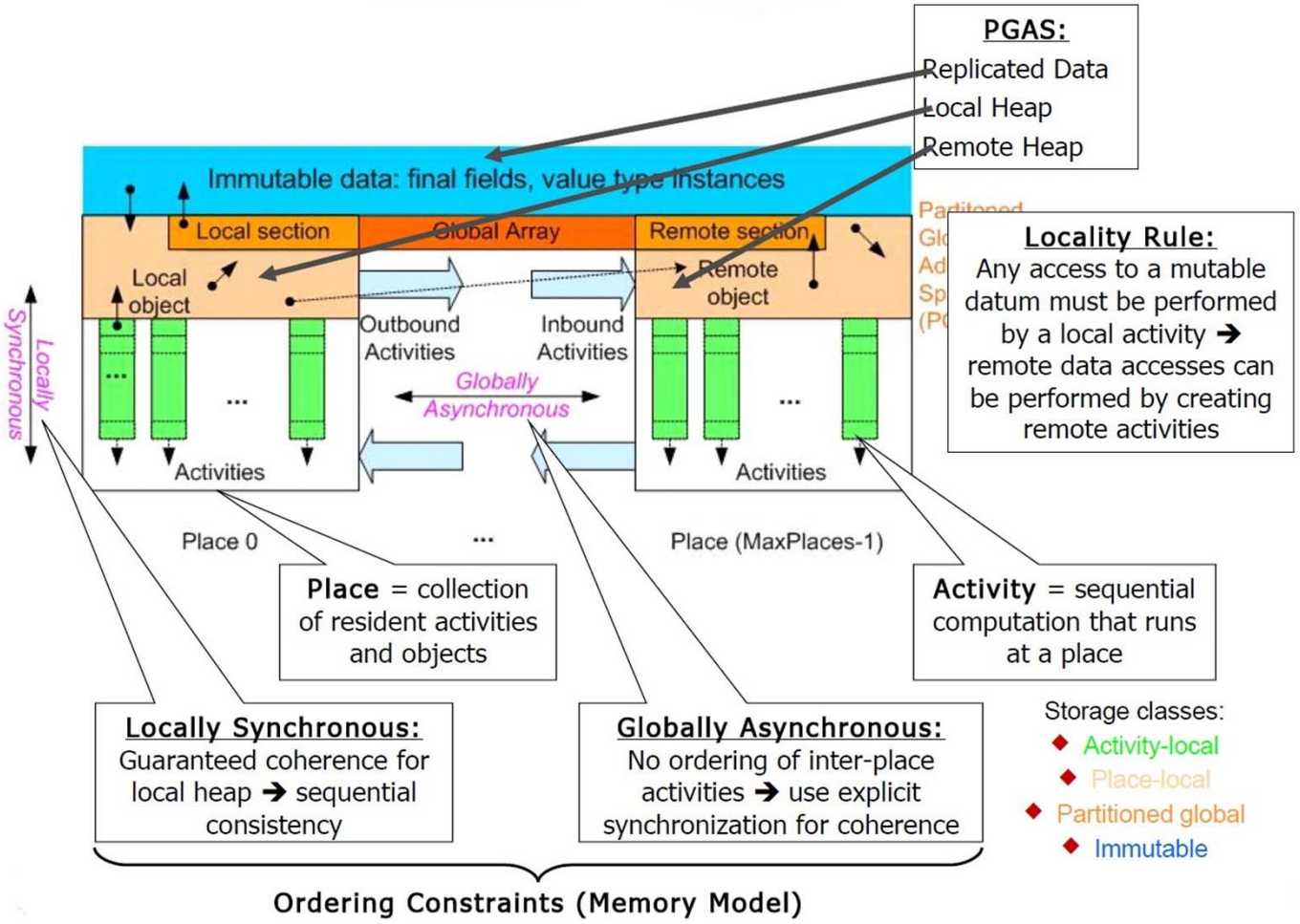
The second level Concurrency

- PGAS (Partitioned Global Address Space)
  - an abstract programming model, which presents an abstraction of a single shared address space, but the address space is partitioned into regions based on an underlying NUMA (non-uniform memory access) architecture.

- APGAS (Asynchronous PGAS) is a variant of PGAS that supports asynchronous operations and control flow.

- X10 is an APGAS language.

X10 Memory Model View



Place

- An X10 **Place** corresponds to a single operating system process. X10 activities can only directly access memory from the Place where they live; they must use constructs like **at** to access memory at remote places.

- Usually, in production, you would run with one X10 **Place** per node in a cluster of workstations. For debugging and development, it is possible to run with multiple Places installed in a single machine.

- A kind of hybrid computing with processes and threads

at

- In X10, **at(p) S** executes statement **S** at the **Place p**.
- Current activity is blocked until S completes.
- **at(p) S** does not start a new activity. It should be thought of as transporting the current activity to **p**, running **S** there, and then transporting it back.
- **async** is the only construct in the language that starts a new activity. In different contexts, each one of the following makes sense:
  - **async at(p) S** (spawn an activity locally to execute **S** at **p**; here **p** is evaluated by the spawned activity)
  - **at(p) async S** (evaluate **p** and then at **p** spawn an activity to execute **S**),
  - **async at(p) async S**

Points and Regions

- A **Point** represents a tuple of integers, or a point in an n-dimensional Cartesian space (n>=1) of integers.

- A **Region** is a collections of points of the same dimension.

- Points and Regions are important in X10, since they are fundamental concepts for X10 Arrays. An n-dimensional X10 Array is defined over an n-dimensional Region.

Distributions and Arrays

- An **x10.array.Dist** supports the definition of distributed arrays.
- Distributions specify mapping of points in a region to places
- One would use a **Dist** to define the layout of an X10 distributed array (**x10.array.DistArray**) .
  - i.e. distributed arrays distribute information according to a **Dist** value passed in as a constructor argument.

Code Example

```
def addTo(a:DistArray[Int], b:DistArray[Int])
{a.dist == b.dist} =  {
val D = a.dist;
for(p in D.places())
  at(p) {
    for(i in D.get(p))
      a(i) += b(i);
  }
}
```

X10 vs. Chapel and Fortress

- Use of global name space (Convenience; All)
- Locales are explicit entities (Essential for managing locality; all but Fortress)
- “Local” and “Global” data accesses are syntactically distinct (Essential for efficient compilation; all)
- Extensibility (Fortress)
- Arrays are distributed (all)
- Rich set of distributions; e.g. block-block, user-defined (Chapel, Fortress)
- Arrays can be redistributed (possible in Fortress via suitable library)

The progress of 3 new languages

- Fortress
  - Funding from DARPA ended in 2006
  - Turns to Open Source
- Chapel and X10 still in Phase III of HPCS project of DARPA
  - Focus on implementation
- Due to come to an end in 2010.
  - All three HPCS languages are still active now.

The future of 3 new languages

- Each of the proposed languages would be an advance over MPI, if properly implemented
- None of the three directly addresses node performance bottlenecks and scaling problems.
- X10, Chapel and especially Fortress require sophisticated compiler technology

However, for compilers…

- It takes **money** to make a good compiler; there is no market for HPC unique optimizations.
- It takes **time** to make a good compiler; there is no funding mechanism that let the development sustain 5 years.
- It takes **people** to make a good compiler; there is no independent compiler company.

Besides…

- The productivity level of a language is also defined by the time required to learn the language and develop the code.
  - X10
    - Similar with Java, accessible to Java programmers.
    - complicated underlying structure
  - Chapel
    - Intuitive and direct way of describing parallel task and data structures.
    - Doesn't require a developer to have detailed knowledge about a machine architecture
  - Fortress
    - Close to standard mathematical notation, accessible to any developer.

- A Good IDE is more than language
  - Only X10 has IDE, X10DT.

Minimal Solution Beyond MPI

- Compiled communication, to avoid software overhead
- Possibly, inlining and optimization of key MPI calls
- Alternatively, simple language extensions for access and update of remote variables
  - The Kanor language, e0@e1 << op << e2@e3 where e4

References

- [1] Chapel, Fortress and X10: novel languages for HPC, M. Weiland, 2007
- [2] Programming Languages for HPC - Is There Life After MPI?, Marc Snir, UIUC, 2006
- [3] High Productivity Language Systems: Next-Generation Petascale Programming, A. Shet, et.al, 2007
- [4] Chapel, Fortress and X10, Didem Unat, 2008
- [5] Optimizing Compilers - Potential Languages of the Future Chapel, Fortress, X10, John Cavazos, University of Delaware, 2009
- [6] First Impressions of the Fortress Language, Pablo Halpern, Intel, 2009
- [7] Official Language Specifications, Tutorials of Fortress, Chapel, X10

X10 Advanced Topics

X10 vs. MPI

- X10 is a higher level programming model than MPI. In general, X10 code should be much more concise than the equivalent MPI.

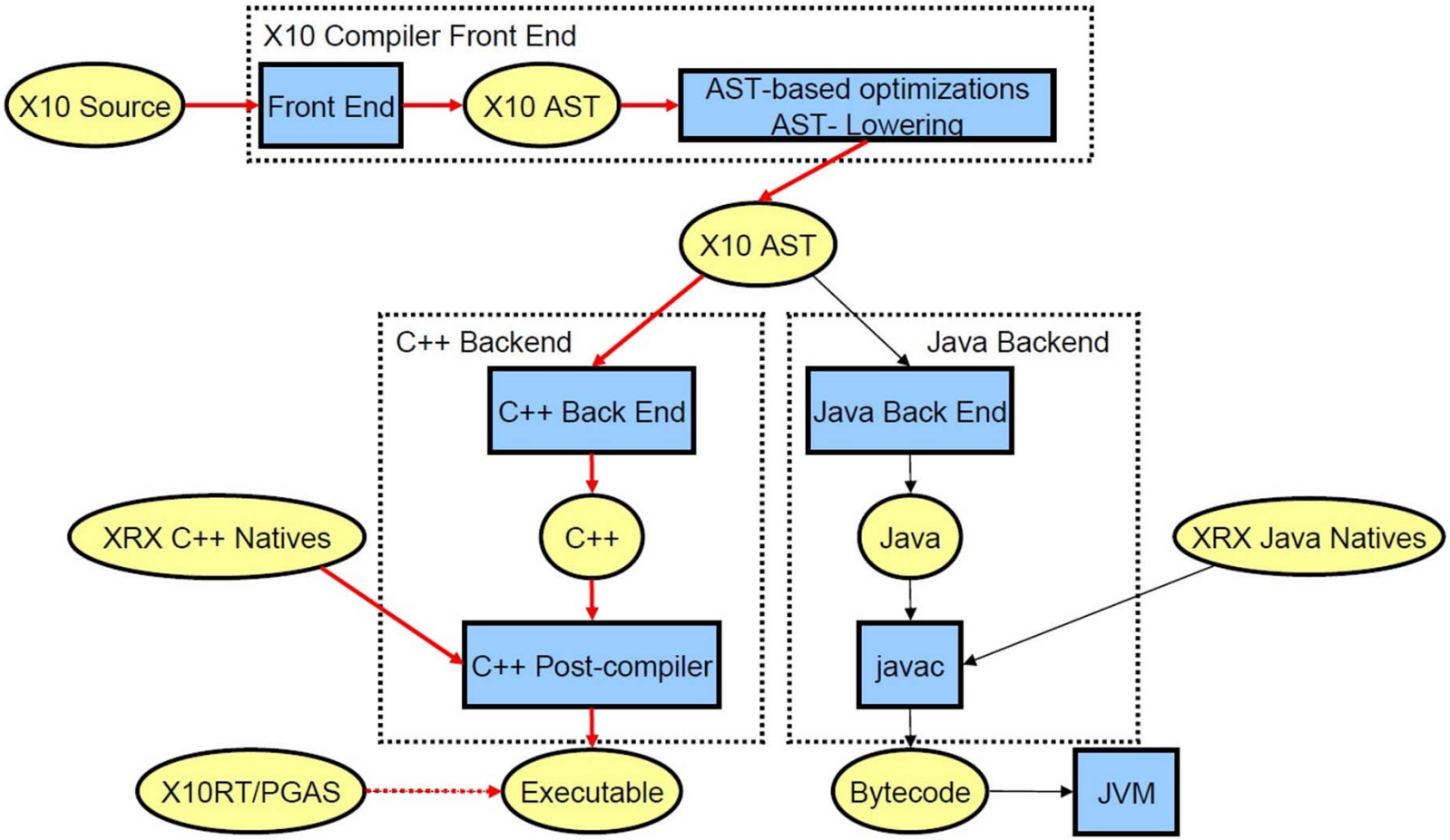
- There are at least two major philosophical differences between the MPI programming model and X10: the control flow, and the memory model.

- Control Flow
  - The MPI control flow model is SPMD (Single-Program Multiple Data): the program begins with a single thread of control in each process.
  - X10 program begins with a single thread of control in the root place, and an X10 program spawns more threads of control across places using **async** and **at**, new activities will be executed based on the local scheduler's decisions.

X10 vs. MPI cont'd

- Memory Model
  - The MPI memory model is a completely distributed memory model. MPI processes communicate via message-passing. There is no shared global address space in MPI, so user code must manage the mappings between local address spaces in different processes.
  - X10 supports a global shared address space. While an X10 activity can only directly access memory in the local Place (address space), it can name a location in a remote place, and the system maintains the mapping between the global address space and each local address space.

X10 Compilation



X10 Platform Support

- Java
  - any platform with Java 5
  - Single process (all places in 1 JVM)
- C++
  - Multi-process (1 place per process)
  - aix, linux, cygwin, solaris
  - x86, x86_64, PowerPC, Sparc
  - x10rt: APGAS runtime (binary only) or MPI (open source)

- The X10 compiler uses the LPG parser generator, the Polyglot compiler framework, and WALA.

The differences of two X10 compile backends

- The X10 compiler can generate either C++ source code or Java source code. The two backends present different tradeoffs on different machines.

- In general, the C++ backend is currently more mature; the Java backend does not currently support execution across multiple JVMs.

- In the medium term, the backends will support different models of interoperability with other languages. In particular, the Java backend will support interoperability with Java code running on JVMs; the C++ backend will support interoperability with certain libraries for hardware accelerators and GPUs.

X10 Clocks

- Clocks correspond to barriers in traditional parallel computing discussions.
- The activities involved may be in the same place or in different places
- You would use a clock when multiple activities are accessing shared storage (producer-consumer), and need to synchronize at the end of a phase before all activities move on to the next phase.

X10 Fault Tolerance Issue

- What happens if a place dies during the execution? Is it possible to detect it and recover the error?

- Currently, the X10 runtime system is not robust with respect to Place failures. The runtime will not fail gracefully if a node dies, and there is no way for user code to detect a failure and recover.