COMP 322: Fundamentals of Parallel Programming

Lecture 36: Comparison of Parallel Programming Models (OpenMP, X10)

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Worksheet #35 solution: Double Checked Locking Idiom in Java

Consider two threads calling the getHelper() method in parallel:

1) Can you construct a possible data race if they call the unoptimized version of getHelper() in lines 3-8?

No race possible (monitor-based synchronization)

2) Can you construct a possible data race if they call the optimized version of getHelper() in lines 12-21?

Yes, thread T1 can assign helper in line 16 while thread T2 reads helper in line 13

3) How will your answer to 2) change if the helper field in line 11 was declared as volatile?

No data race since volatile declaration causes read and write of helper to be (semantically) enclosed in isolated blocks

Worksheet #35 (contd)

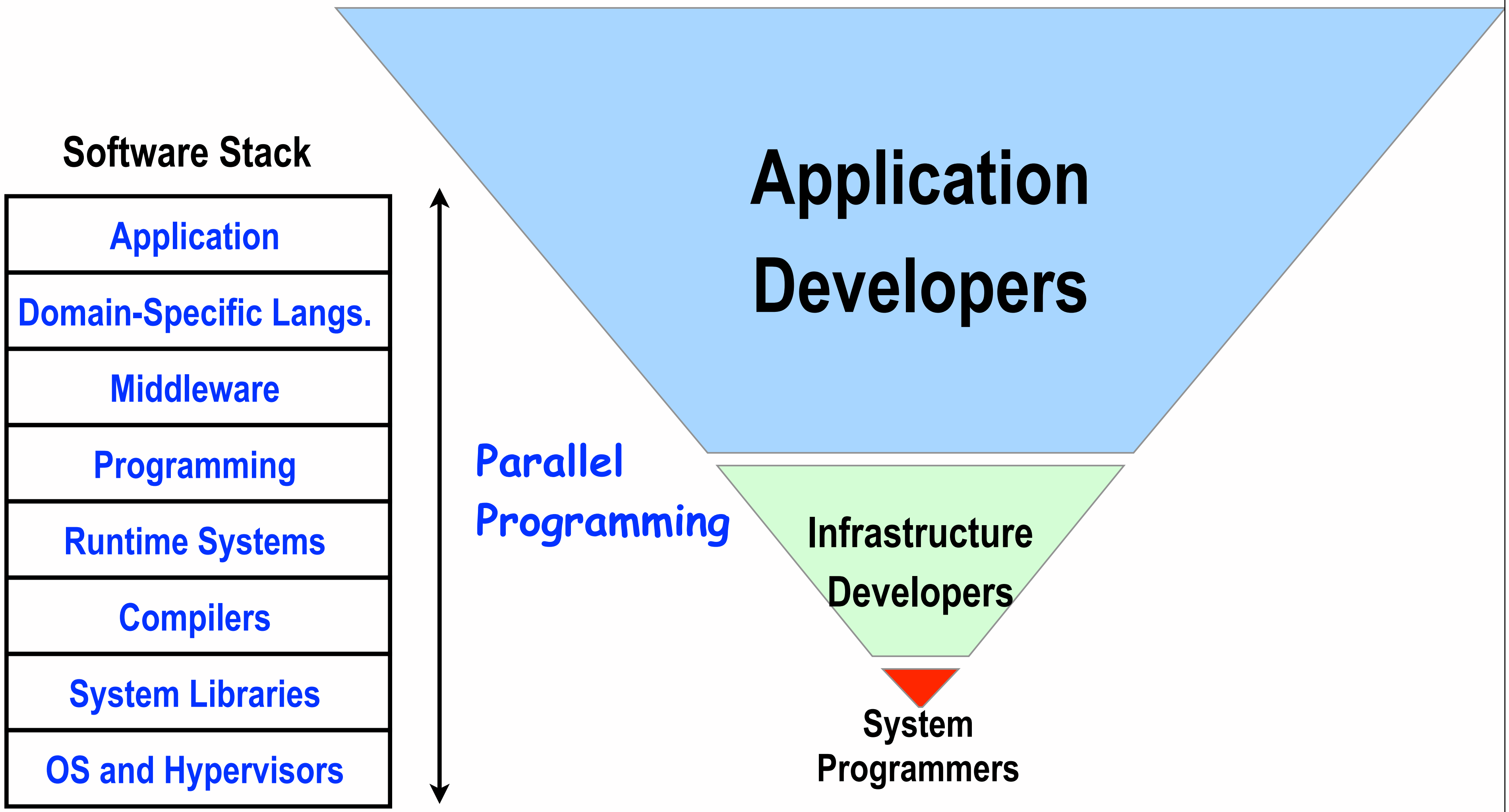
```
class Foo { //unoptimized version
    private Helper helper; // Singleton pattern
    public synchronized Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }
    . . .

class Foo { //Optimized version
    private Helper helper; // Singleton pattern
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
    . . .
```

Acknowledgments

- Slides from COMP 422 course at Rice University
  - http://www.clear.rice.edu/comp422/
- Slides from OpenMP tutorial given by Ruud van der Paas at HPCC 2007
  - http://www.tlc2.uh.edu/hpcc07/Schedule/OpenMP
- "Towards OpenMP 3.0", Larry Meadows, HPCC 2007 presentation
  - http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc07_Larry.ppt
- APGAS Programming in X10 tutorial given by Olivier Tardieu at the Hartree Centre Summer School "Programming for Petascale" in July 2013.

Parallel Programming is a Cross-Cutting Concern

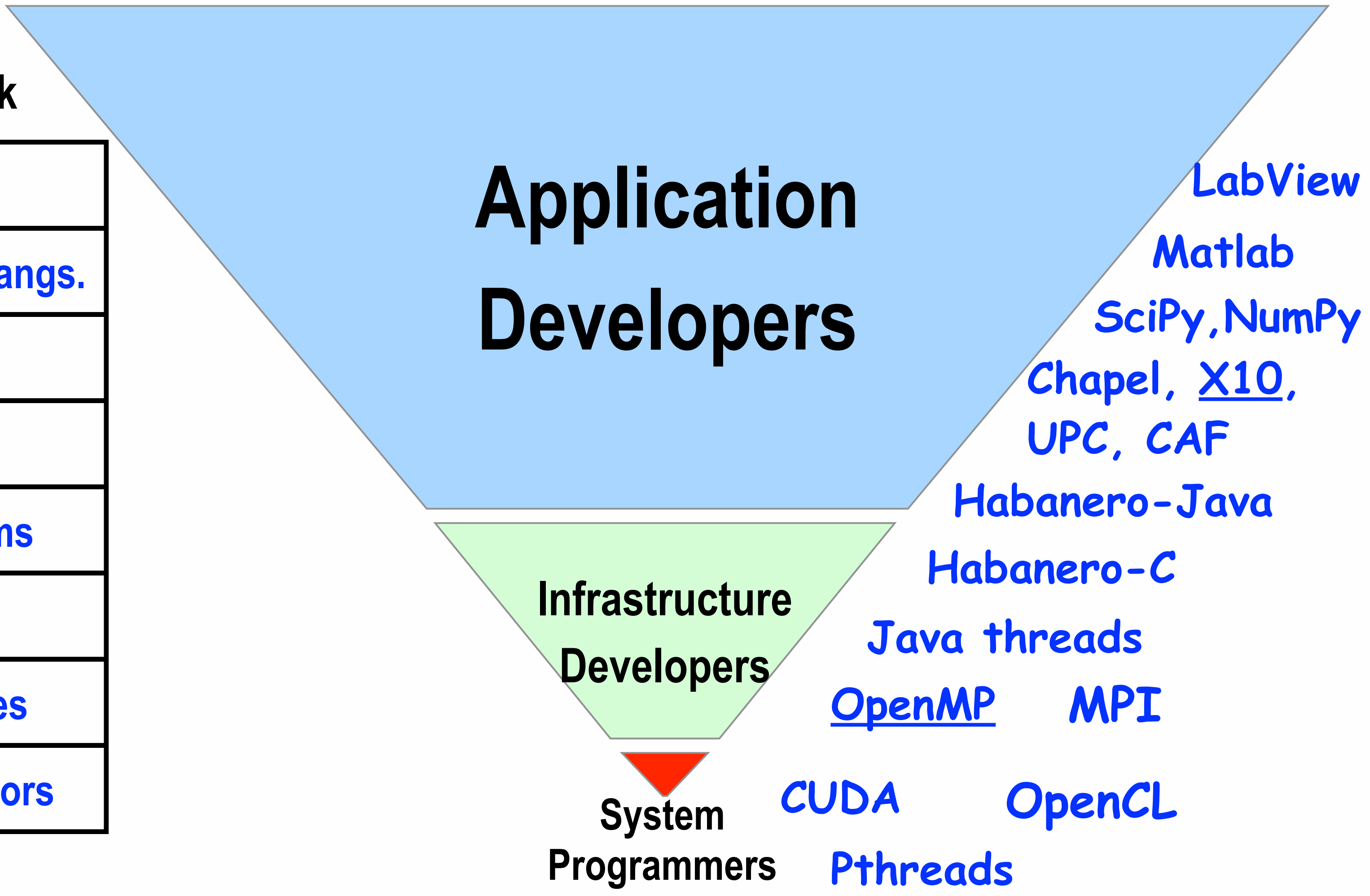
Software Stack



Different Parallel Programming Models for different Levels of Developer Pyramid and Software Stack

Software Stack

Application
Domain-Specific Langs.
Middleware
Programming
Runtime Systems
Compilers
System Libraries
OS and Hypervisors



What is OpenMP?

- Well-established standard for writing shared-memory parallel programs in C, C++ Fortran
  - Open implementation available in gcc
  - Proprietary implementations available from vendors
- Programming model is expressed via
  - —Pragmas/directives (not language extensions)
  - —Runtime routines
  - —Environment variables

—Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)

- —Latest specification: Version 4.0 (July 2013)

A first OpenMP example

For-loop with independent iterations

```
for (i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

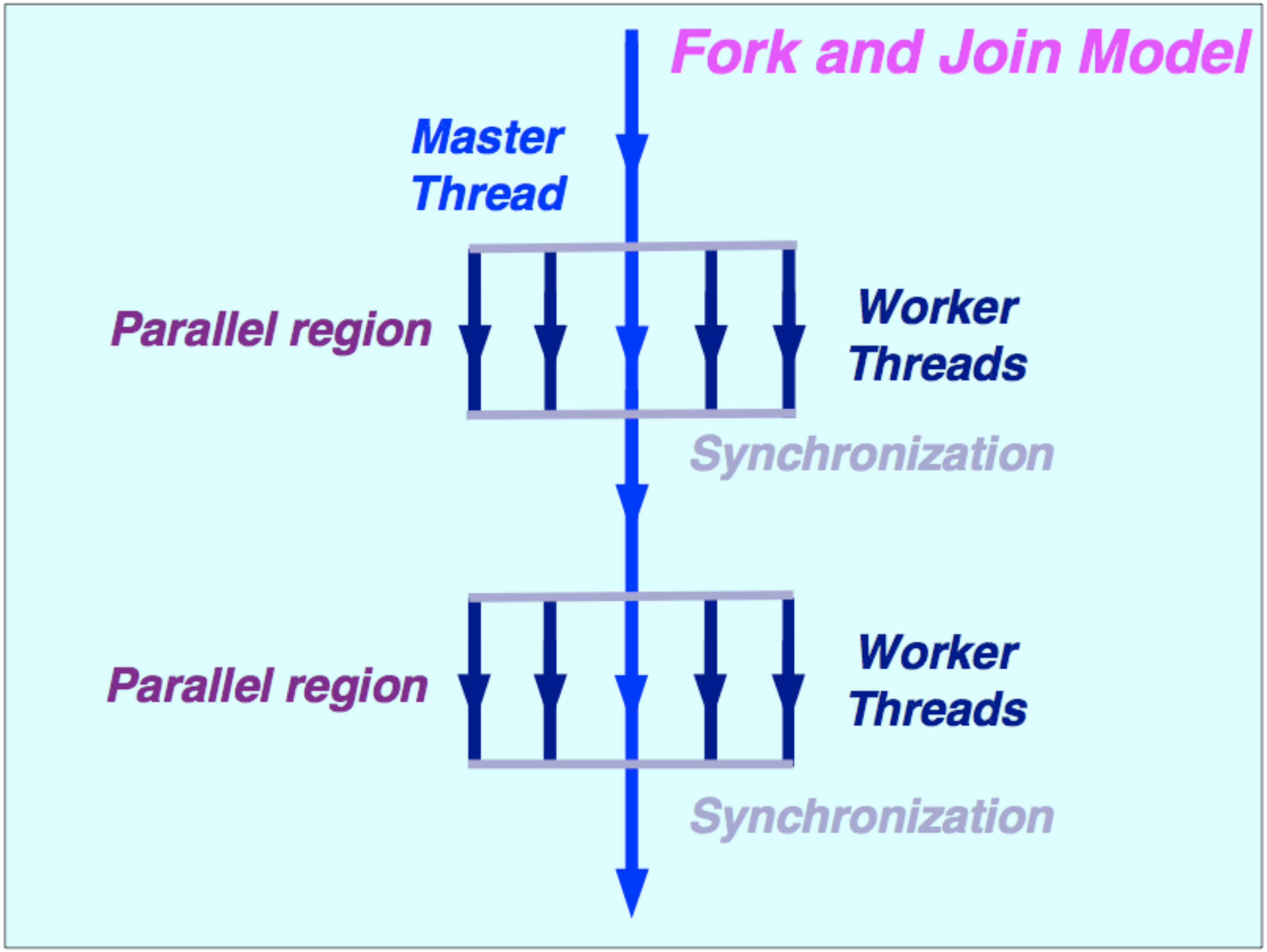
For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for  \
        shared(n, a, b, c)\
        private(i)
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

```
% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 4
% a.out
```

OpenMP parallel for loop is like a forall loop in HJ

The OpenMP Execution Model



Terminology

- OpenMP Team := Master + Workers
- A Parallel Region is a block of code executed by all threads simultaneously (builds on SPMD model)
  - The master thread always has thread ID 0
  - Thread adjustment (if enabled) is only done before entering a parallel region
  - Parallel regions can be nested, but support for this is implementation dependent
  - An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially
- A work-sharing construct divides the execution of the enclosed code region among the members of the team; in other words: they split the work

Parallel Region

```
#pragma omp parallel [clause[[,] clause] ...]
{
    "this is executed in parallel"

} (implied barrier)
```

A parallel region is a block of code executed by multiple threads simultaneously in SPMD mode, and supports the following clauses:

if	*(scalar expression)*	
private	*(list)*	
shared	*(list)*	
default	*(none|shared)*	*(C/C++)*
default	*(none|shared|private)*	*(Fortran)*
reduction	*(operator: list)*	
copyin	*(list)*	
firstprivate	*(list)*	
num_threads	*(scalar_int_expr)*	

Work-sharing constructs in a Parallel Region

```
#pragma omp for
{
    ....
}
```

```
#pragma omp sections
{
        ....
}
```

```
#pragma omp single
{
        ....
}
```

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or none at all
- No implied barrier on entry; implied barrier on exit (unless nowait is specified)
- A work-sharing construct does not launch any new threads

```
#pragma omp parallel
#pragma omp for
   for (...)
```

```
#pragma omp parallel for
for (....)
```

Legality constraints for work-sharing constructs

- **Each worksharing region must be encountered by all threads in a team or by none at all.**
- **The sequence of worksharing regions and barrier regions encountered must be the same for every thread in a team.**

```
#pragma omp parallel
{
  do {
    // c1 and c2 may depend on the OpenMP thread-id
    boolean c1 = … ; boolean c2 = … ;
    . . .
    if (c2) {
      // Start of work-sharing region with no wait clause
      #pragma omp …
      . . . // Worksharing statement
    } // if (c2)
  } while (! c1);
}
```

==> No OpenMP implementation checks for conformance with this rule (unlike HJ's runtime check for phaser single statements)

Example of work-sharing “omp for” loop

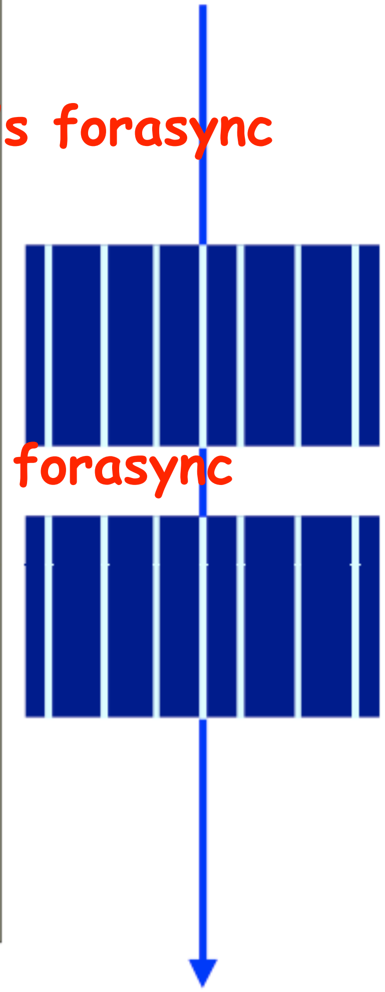
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
         b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
    for (i=0; i<n; i++)
         d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)

Implicit finish

Like HJ's forasync

Like HJ's forasync



task Construct

```
#pragma omp task [clause[[,]clause] ...]
          structured-block
```

where *clause* can be one of:

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none  )
```

Example – parallel pointer chasing using tasks

```
#pragma omp parallel
{
  #pragma omp single
  {
    Node p = listhead ;
    while (p) {
       #pragma omp task
              process (p);
       p= p->next ;
     }
   }
}
```

Spawn call to process(p)

Implicit finish at end of parallel region

Parallel pointer chasing example in HJlib

```
finish( ()-> {
    Node p = listhead ;
    while (p != null) {
        final Node pp = p;
        async(()->{ process(p); });
        p= p->next ;
      }
  }
}
```

Spawn call to process(p)

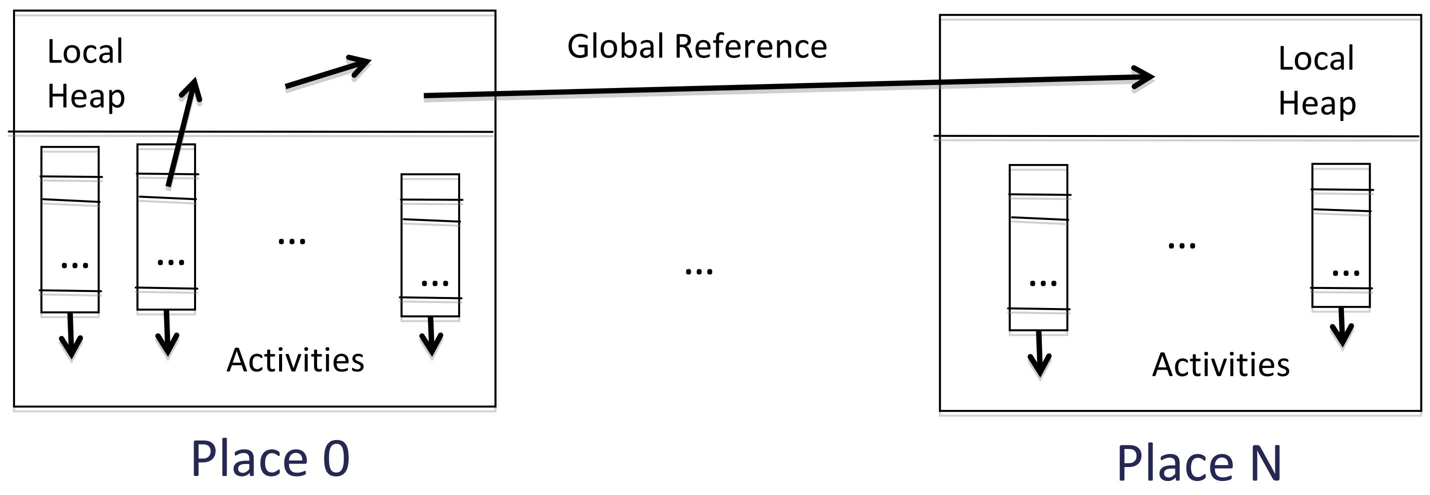
Summary of key features in OpenMP

OpenMP construct	Related HJ/Java constructs
Parallel region #pragma omp parallel	HJ forall (forall iteration = OpenMP thread)
Work-sharing constructs: parallel loops, parallel sections	No direct analogy in HJ or Java
Barrier #pragma omp barrier	HJ forall-next on implicit phaser
Single #pragma omp single	HJ's forall-next-single on implicit phaser (but HJ does not support single + nowait)
Reduction clauses	HJ's finish accumulators (in forall)
Critical section #pragma omp critical	HJ's isolated statement
Task creation #pragma omp task	HJ's async statement
Task termination #pragma omp taskwait	HJ's finish statement

What is X10?

- Open-source parallel programming language led by IBM Research
  - Habanero-Java was built on early version of X10 from 2007
    - "Habanero-Java: the New Adventures of Old X10". Vincent Cave, Jisheng Zhao, Jun Shirako, Vivek Sarkar. 9th International Conference on the Principles and Practice of Programming in Java (PPPJ), August 2011.
- Integrates HJ-style task parallelism and UPC-style PGAS concepts in a new language with Scala-like syntax
- Includes an Eclipse-based IDE (X10DT)
- For more details, see http://x10-lang.org

APGAS in X10: Places and Tasks



Task parallelism

- `async S`
- `finish S`

Place-shifting operations

- `at(p) S`
- `at(p) e`

Concurrency control within a place

- `when(c) S`
- `atomic S`

Distributed heap

- `GlobalRef[T]`
- `PlaceLocalHandle[T]`

APGAS Idioms

- Remote evaluation

```
v = at(p) evalThere(arg1, arg2);
```

- Active message

```
at(p) async runThere(arg1, arg2);
```

- Recursive parallel decomposition

```
def fib(n:Long):Long {
  if(n < 2) return n;
  val f1:Long;
  val f2:Long;
  finish {
    async f1 = fib(n-1);
    f2 = fib(n-2);
  }
  return f1 + f2;
}
```

- SPMD

```
finish for(p in Place.places()) {
  at(p) async runEverywhere();
}
```

- Atomic remote update

```
at(ref) async atomic ref() += v;
```

- Data exchange

```
// swap l() local and r() remote
val _l = l();
finish at(r) async {
  val _r = r();
  r() = _l;
  at(l) async l() = _r;
}
```

Task Parallelism: async and finish

- async S
  - creates a new task that executes S
  - returns immediately
  - S may reference values in scope
    - S may initialize values declared above the enclosing finish
  - S may reference variables declared above the enclosing finish
  - tasks cannot be named or cancelled

- finish S
  - executes S
  - then waits until all transitively spawned tasks in S have terminated
  - rooted exception model
    - trap all exceptions and throw a multi-exception if any spawned task terminates abnormally
    - exception is thrown after all tasks have completed
  - collecting finish combines finish with reduction over values offered by subtasks

Concurrency Control: atomic and when

- atomic S
  - executes statement S atomically
    - atomic blocks are conceptually executed in a serialized order with respect to all other atomic blocks in a place (weak atomicity)
  - S must be non-blocking, sequential, and local
    - no when, at, async

- when(c) S
  - the current task suspends until a state is reached where c is true
  - in that state, S is executed atomically
  - Boolean expression c must be non-blocking, sequential, local, and pure
    - no when, at, async, no side effects

- Gotcha: S in when(c) S is not guaranteed to execute
  - if c is not set to true within an atomic block
  - or if c oscillates

Examples

```
class Account {
  public var value:Int;

  def transfer(src:Account, v:Int) {
    atomic {
       src.value -= v;
       this.value += v;
    }
  }
}
```

```
class Latch {
  private var b:Boolean = false;
  def release() { atomic b = true; }
  def await() { when(b); }
}
```

```
class Buffer[T]{T isref,T haszero} {
  protected var datum:T = null;

  public def send(v:T){v!=null} {
    when(datum == null) {
      datum = v;
    }
  }

  public def receive() {
    when(datum != null) {
         val v = datum;
         datum = null;
         return v;
    }
  }
}
```

Clocks

APGAS barriers

- synchronize dynamic sets of tasks

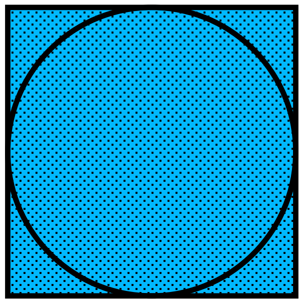
x10.lang.Clock

- anonymous or named
- task instantiating the clock is registered with the clock
- spawned tasks can be registered with a clock at creation time
- tasks can deregister from the clock
- tasks can use multiple clocks
- split-phase clocks
  - clock.resume(), clock.advance()
- compatible with distribution

```
// anonymous clock
clocked finish {
  for(1..4) clocked async {
    Console.OUT.println("Phase 1");
    Clock.advanceAll();
    Console.OUT.println("Phase 2");
  }
}
// named clock
finish {
  val c = Clock.make();
  for(1..4) async clocked(c) {
    Console.OUT.println("Phase 3");
    c.advance();
    Console.OUT.println("Phase 4");
  }
  c.drop();
}
```

Sequential Monte Carlo Pi

```
package examples;
import x10.util.Random;

public class SeqPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0));
    var result:Double = 0;
    val rand = new Random();
    for(1..N) {
      val x = rand.nextDouble();
      val y = rand.nextDouble();
      if(x*x + y*y <= 1) result++;
    }
    val pi = 4*result/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```



Parallel Monte Carlo Pi with Atomic

```
public class ParPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0)); val P = Int.parse(args(1));
    var result:Double = 0;
    finish for(1..P) async {
      val myRand = new Random();
      var myResult:Double = 0;
      for(1..(N/P)) {
        val x = myRand.nextDouble();
        val y = myRand.nextDouble();
        if(x*x + y*y <= 1) myResult++;
      }
      atomic result += myResult;
    }
    val pi = 4*result/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```

Parallel Monte Carlo Pi with Collecting Finish

```
public class CollectPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0)); val P = Int.parse(args(1));
    val result = finish(Reducible.SumReducer[Double]()) {
      for(1..P) async {
        val myRand = new Random();
        var myResult:Double = 0;
        for(1..(N/P)) {
          val x = myRand.nextDouble();
          val y = myRand.nextDouble();
          if(x*x + y*y <= 1) myResult++;
        }
        offer myResult;
      }
    };
    val pi = 4*result/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```

Distribution: Places

An X10 application runs with a fixed number of places decided at launch time

x10.lang.Place

- The available places are numbered from 0 to Place.MAX_PLACES-1
- for(p in Place.places()) iterates over all the available places
- here always evaluates to the current place
- Place(n) is the nth place
- If p is a place then p.id is the index of place p
- *Each place has its own copy of static variables*
- *Static variables are initialized per place and per variable at the first access*

The main method is invoked at place Place(0)

Other places are initially idle

X10 programs are typically parametric in the number of places

Distribution: at

A task can “shift” place using at

- at(p) S
  - executes statement S at place p
  - current task is blocked until S completes
  - S may spawn async tasks
    - at does not wait for these tasks
    - the enclosing finish does

- at(p) e
  - evaluates expression e at place p and returns the computed value

- at(p) async S
  - creates a task at place p to run S
  - returns immediately

HelloWholeWorld.x10

```
class HelloWholeWorld {
  public static def main(args:Rail[String]) {
    finish
      for(p in Place.places())
        at(p) async
          Console.OUT.println(p + " says " + args(0));
    Console.OUT.println("Bye");
  }
}
```

```
$ x10c++ HelloWholeWorld.x10
$ X10_NPLACES=4 ./a.out hello
Place(0) says hello
Place(2) says hello
Place(3) says hello
Place(1) says hello
Bye
```

Distributed Object Model

- Objects live in a single place
  - an object belong to the place of the task that constructed the object
  - objects can only be accessed in the place where they live
  - tasks must shift place accordingly

- Object references are always local
  - rail:Rail[Int] refers to a rail in the current place (if not null)

- Global references (possibly remote) have to be constructed explicitly
  - val ref:GlobalRef[Rail[Int]] = GlobalRef(rail);

- Global references can only be dereferenced at the place of origin “home”
  - at(ref.home) Console.OUT.println(ref());
  - at(ref) Console.OUT.println(ref()); // shorthand syntax
  - ref as GlobalRef[T]{self.home==here} // place cast

Distributed Monte Carlo Pi with Atomic and GlobalRef

```
public class DistPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0));
    val result = GlobalRef[Cell[Double]](new Cell[Double](0));
    finish for(p in Place.places()) at(p) async {
      val myRand = new Random();
      var myResult:Double = 0;
      for(1..(N/Place.MAX_PLACES)) {
        val x = myRand.nextDouble();
        val y = myRand.nextDouble();
        if(x*x + y*y <= 1) myResult++;
      }
      val myFinalResult = myResult;
      at(result) async atomic result()() += myFinalResult;
    }
    val pi = 4*result()()/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```

Distributed Monte Carlo Pi with Collecting Finish

```
public class MontyPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0));
    val result = finish(Reducible.SumReducer[Double]()) {
      for(p in Place.places()) at(p) async {
        val myRand = new Random();
        var myResult:Double = 0;
        for(1..(N/Place.MAX_PLACES)) {
          val x = myRand.nextDouble();
          val y = myRand.nextDouble();
          if(x*x + y*y <= 1) myResult++;
        }
        offer myResult;
      }
    };
    val pi = 4*result/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```

Final Thoughts

- X10 Approach
  - Augment full-fledged modern language with core APGAS constructs
  - Enable programmer to evolve code from prototype to scalable solution

  - Problem selection: do a few key things well, defer many others
  - Mostly a pragmatic/conservative language design (except when it is not)

- X10 2.4 (today) is not the end of the story
  - A base language in which to build higher-level frameworks (Global Matrix Library, Main-Memory Map Reduce, ScaleGraph)
  - A target language for compilers (MatLab, stencil DSLs)

  - APGAS runtime: X10 runtime as Java and C++ libraries
  - APGAS programming model in other languages

Announcements

- Graded midterms can be picked up from Melissa Cisneros in Duncan Hall room 3122 (mcisnero@rice.edu)
- Homework 5 due by 11:55pm today
  - —Send email to comp322-staff@rice.edu if you plan to use slip days
  - —Two more slip days were added for a total of 5 slip days for the semester
- Homework 6 due by 11:55pm on April 25th, penalty-free extension till May 2nd
  - —Slip days can be applied past May 2nd
- No lab this week
- No class on April 23rd
  - —Use the time to complete your homeworks!
- April 25th is last day of classes
  - —Take-home Exam 2 will be handed out on April 25th, due by May 2nd
    - – Will cover lectures 19 - 35

Worksheet #36: Parallel Programming Constructs

Name: ______________________ Netid: ______________________

Which of the following HJ constructs do you like the most? Which one do you like the least? Why?

async, finish, futures, isolated, accumulators, forall, barriers, data-driven tasks/futures, phasers, isolated, object-based isolation, actors , places