HIGH-PERFORMANCE PDC
(PARALLEL AND DISTRIBUTED COMPUTING)

Prasun Dewan
Department of Computer Science
University of North Carolina at Chapel Hill
dewan@cs.unc.edu
GOAL

- High Performance PDC (Parallel and Distributed Computing)
  - Important for traditional problems when computers were slow
  - Back to the future with emergence of data science
- Modern mechanisms:
  - OpenMP (Parallel Computing and Distributed Shared Memory)
  - MapReduce (Distributed Computing)
  - OpenMPI (~ Sockets)
- Connect OpenMP and MapReduce to
  - Each other (reduction)
  - Non HPC PDC
- Derive the design and implementation of both kinds of abstractions
  - From similar but different performance issues that motivate them.
  - Target intersecting classes of algorithms
- Prins 633 course
  - Covers HPC PDC algorithms in detail.
- Don Smith 590 Course
  - Covers MapReduce uses in detail
A Tale of Two Distribution Kinds

- Remotely Accessible Services (Printers, Desktops)
- Replicated Repositories (Files, Databases)
- Collaborative Applications (Games, Shared Desktops)
- Distributed Sensing (Disaster Prediction)
- Computation Distribution (Number, Matrix Multiplication)

Differences between the two groups?
PRIMARY/SECONDARY DISTRIBUTION REASON

Primary

Remote Service
Fault Tolerance
Collaboration
Aggregation of Distributed Data

High-Performance: Speedup

Secondary

Remote service, aggregation,...

Remotely Accessible Services (Printers, Desktops)
Replicated Repositories (Files, Databases)
Collaborative Applications (Games, Shared Desktops)
Distributed Sensing (Disaster Prediction)

Computation Distribution (Number, Matrix Multiplication)
**DIST. VS PARALLEL-DIST. EVOLUTION**

- **Remotely Accessible Services** (Printers, Desktops)
- **Replicated Repositories** (Files, Databases)
- **Collaborative Applications** (Games, Shared Desktops)
- **Distributed Sensing** (Disaster Prediction)
- **Computation Distribution** (Number, Matrix Multiplication)

**Non Distributed Existing Programs**

- **T** Distributed Single-Thread Program
- **T** Distributed Multi-Thread Program
- **T** Distributed Program
- **T** Distributed Program

**Distributed Programs**

- **T** Distributed Program
- **T** Distributed Program
- **T** Distributed Program
- **T** Distributed Program
# Concurrency-Distribution Relationship

<table>
<thead>
<tr>
<th>Remotely Accessible Services (Printers, Desktops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replicated Repositories (Files, Databases)</td>
</tr>
<tr>
<td>Collaborative Applications (Games, Shared Desktops)</td>
</tr>
<tr>
<td>Distributed Sensing (Disaster Prediction)</td>
</tr>
<tr>
<td>Computation Distribution (Number, Matrix Multiplication)</td>
</tr>
</tbody>
</table>

Threads complement blocking IPC primitives by improving responsiveness, removing deadlocks, and allowing certain kinds of distributed applications that would otherwise not be possible.

- e.g. Server waiting for messages from multiple blocking sockets, NIP selector threads sending read data to read thread, creating a separate thread for incoming remote calls.

Thread decomposition replaced by process decomposition and can be present in decomposed processes.

- e.g. Single-process: each row of matrix $A$ multiplied with column of $A$ by separate thread.

- e.g. Multi-process: Each row of matrix assigned to a process, which uses different threads to multiply it with different columns of matrix $B$. 

ALGORITHMIC CHALLENGE

- Computation Distribution (Number, Matrix Multiplication)
- Distributed Sensing (Disaster Prediction)
- Collaborative Applications (Games, Shared Desktops)
- Replicated Repositories (Files, Databases)
- Remotely Accessible Services (Printers, Desktops)

Consistency: How to define and implement correct coupling among existing processes?

How to parallelize/distribute existing single-thread algorithms?
A master process/thread often decomposes problem and combines results computed by slave agents, but decomposer knows about the nature of slaves, which are the service providers.

Clients often talk to each through a central server/master whose code is unaware of specific arbitrary client/slave locations and ports.

Central Mediator

- Remotely Accessible Services (Printers, Desktops)
- Replicated Repositories (Files, Databases)
- Collaborative Applications (Games, Shared Desktops)
- Distributed Sensing (Disaster Prediction)
- Computation Distribution (Number, Matrix Multiplication)
**IMPLEMENTATION CHALLENGE**

- Remotely Accessible Services (Printers, Desktops)
- Replicated Repositories (Files, Databases)
- Collaborative Applications (Games, Shared Desktops)
- Distributed Sensing (Disaster Prediction)
- Computation Distribution (Number, Matrix Multiplication)

How to add to single-process local observable/observer, producer-consumer and synchronization relationships corresponding distributed observable/observer, producer-consumer and synchronization relationships?

How to reuse existing single-thread code in multi-thread/multi-process program?
Communicating threads/processes created on hosts/processors typically under control of one authority, though crowd problem solving is an infrequent exception (UW Condor, Wagstaff primes)
**LIFETIME**

- **Remotely Accessible Services** (Printers, Desktops)
- **Replicated Repositories** (Files, Databases)
- **Collaborative Applications** (Games, Shared Desktops)
- **Distributed Sensing** (Disaster Prediction)

**Long-lived, processes need to be explicitly terminated**

- **Computation Distribution** (Number, Matrix Multiplication)

**Short-lived, terminate when function computation complete**
## Coupling vs High-Performance

<table>
<thead>
<tr>
<th>Coupling/Consistency</th>
<th>High Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent, “long-lived” processes at different autonomous locations made dependent for resource sharing, user collaboration, fault tolerance.</td>
<td>A single short-lived process created to perform some computation process made multi-threaded and/or distributed to increase performance.</td>
</tr>
<tr>
<td>Include consistency algorithms, possibly in separate threads, to define dependency that have producer-consumer relationship with existing algorithms.</td>
<td>Speedup algorithms that replace logic of existing code, with task decomposition.</td>
</tr>
<tr>
<td>May use additional mediating server and other infrastructure code unaware of specific clients.</td>
<td>May involve a central mediating, distributing master code but it can be aware of and creator of specific slave processes.</td>
</tr>
<tr>
<td>Division of labor between client and infrastructure an issue (centralized vs replicated).</td>
<td>Division of labor between master and slave and slave and slaves an issue.</td>
</tr>
</tbody>
</table>
**Example: Service + Speedup**

Client

Client\(^1\)

Client\(^2\)

Server

Grader Server

T\(^S\)

Master

Comp 401 Grader

Slave

Comp 533 Grader

Slave

Comp 533 Grader

Slave

533 Tester

T\(^3\)

T\(^4\)

533 Tester

T\(^5\)

T\(^6\)

T\(^1\)

T\(^2\)
Higher-level abstractions for different classes of speedup algorithms

- Remotely Accessible Services (Printers, Desktops)
- Distributed Repositories (Files, Databases)
- Collaborative Applications (Games, Shared Desktops)
- Distributed Sensing (Disaster Prediction)
- Computation Distribution (Number, Matrix Multiplication)

Threads, IPC, Bounded Buffer
### Complete Automation?

<table>
<thead>
<tr>
<th>Remotely Accessible Services (Printers, Desktops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed Repositories (Files, Databases)</td>
</tr>
<tr>
<td>Collaborative Applications (Games, Shared Desktops)</td>
</tr>
<tr>
<td>Distributed Sensing (Disaster Prediction)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modules of Non-Distributed Program Distributed Transparently</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loader contacts registry to determine if local module loaded or remote module accessed</td>
</tr>
<tr>
<td>Assumes one name space, one instance of each service – cannot handle replication</td>
</tr>
<tr>
<td>Motivates RPC transparency</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallelizing compilers (Kuck and Kennedy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halting problem</td>
</tr>
<tr>
<td>Motivates Declarative Abstractions</td>
</tr>
</tbody>
</table>
**DECLARATIVE ABSTRACTIONS**

- Remotely Accessible Services (Printers, Desktops)
- Distributed Repositories (Files, Databases)
- Collaborative Applications (Games, Shared Desktops)
- Distributed Sensing (Disaster Prediction)
- Computation Distribution (Number, Matrix Multiplication)
- Adding Declarative abstractions for different classes of speedup algorithms
- Threads, IPC, Bounded Buffer
- Distributed Repositories (Files, Databases)
- Collaborative Applications (Games, Shared Desktops)
- Distributed Sensing (Disaster Prediction)
- Computation Distribution (Number, Matrix Multiplication)
- Adding Declarative abstractions for different classes of speedup algorithms
- Threads, IPC, Bounded Buffer
Declarative vs Imperative: Complementing

Declarative: Specify what we want

```java
floats[] floats = {(4.8f, 5.2f, 4.5f)};
```

Imperative: Implement what we want

```java
public static float sum(Float[] aList) {
    float retVal = (float) 0.0;
    for (int i = 0; i < aList.length; i++) {
        retVal += aList[i];
    }
    return retVal;
}
```

Type declaration

Procedural, Functional, O-O, ...

Locality relevant to PDC abstractions - later

Loop
Declarative vs Imperative: Competing

Declarative: Specify what we want

(0|1)*1

Imperative: Implement what we want

Regular expression

FSA (Finite State Automata)

Consistency algorithms are state machines
Declarative vs Imperative PDC: Competing and Complementing

Declarative: Specify what we want

Imperative: Implement what we want

Thread aThread = new Thread(aRunnable);
aThread.start();

aRegistry.rebind(Server.NAME, aServer)

Server aServer = (Server)
aRegistry.lookup(Server.NAME)

Concurrence?

Distribution?

For restricted classes of programs
Declarative vs Imperative Concurrency: Competing and Complementing

- Declarative: Specify what we want
- Imperative: Implement what we want

For restricted classes of programs

Thread aThread = new Thread(aRunnable);
aThread.start();

Concurrency?
Parallel Random

0.6455074599676613
0.14361454218773773

Desired I/O

//omp parallel threadNum(2)

System.out.println(Math.random());

Declarative Concurrency Aware Code?

Imperative Concurrency-Unaware Code

Declarative Concurrency Specification Called a Pragma or Directive
OPENMP

- Language-independent much as RPC and Sockets
- Standard implementations exist for:
  - C/C++
    - #pragma omp parallel num_threads(2)
  - FORTRAN
    - !$OMP PARALLEL NUM_THREADS(2)
- No standard implementation for Java
  - OMP4 implements part of standard (Belohlavek undergrad thesis)
    - //omp parallel threadNum(2)
      - Used in programming examples
- Unified by set of statement attribute values):
  - Parallel (boolean)
  - Number of threads (int)
- Syntax for describing them is similar
- Only certain sets of attributes can be associated with an imperative statement
  - Depends on the kind of statement
### OpenMP Statement Attributes (Abstract)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td></td>
<td>Create multiple threads to execute attributed statement</td>
</tr>
<tr>
<td>NumberOfThreads</td>
<td>int</td>
<td>Number of threads to be created automatically</td>
</tr>
</tbody>
</table>
WHAT PROCESSES THE PRAGMAS?

0.6455074599676613
0.14361454218773773

//omp parallel threadNum(2)

System.out.println(Math.random());

Processed by what kind of program?
**Pre-Compiler Approach**

Source Program in (Concurrency-Unaware/Aware Native Language + Concurrency-Aware New Language) → New Language Precompiler → Source Program in Concurrency-Aware Native Language

Compiled Program in Concurrency-Aware Imperative Language → Native Language Compiler

(Optional) Interpreter

```
omp4j -s Random.java
```

```
 javac Random.java
```

```
 java Random
```
System.out.println(Math.random());

0.6455074599676613
0.14361454218773773
public static void trace(Object... anArgs) {
    System.out.print(Thread.currentThread());
    for (Object anArg : anArgs) {
        System.out.print(" "+anArg);
    }
    System.out.println();
}

//omp parallel threadNum(2)
{
    trace(Math.random());
}
**Java Solution?**

Thread[pool-1-thread-2,5,main] 0.6501712957370558
Thread[pool-1-thread-1,5,main] 0.6907459159093547

```java
//omp parallel threadNum(2)

synchronized (this)
{
    trace(Math.random());
}

public static void trace(Object... anArgs) {
    System.out.print(Thread.currentThread());
    for (Object anArg : anArgs) {
        System.out.print(" " + anArg);
    }
    System.out.println();
}
```

Trace may be called in a static method such as main. OpenMP designed for non OO languages such as C and FORTRAN
OMP Declarative Solution

```
Thread[pool-1-thread-2,5,main]
0.6501712957370558
Thread[pool-1-thread-1,5,main]
0.6907459159093547

//omp parallel threadNum(2)
{
   //omp critical
   trace(Math.random());
}

public static void trace(Object... anArgs) {
    System.out.print(Thread.currentThread());
    for (Object anArg : anArgs) {
        System.out.print(" "+anArg);
    }
    System.out.println();
}
```
## OpenMP Statement Attributes (Abstract)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td></td>
<td>Create multiple threads to execute attributed statement</td>
</tr>
<tr>
<td>NumberOfThreads</td>
<td>int</td>
<td>Number of threads to be created automatically</td>
</tr>
<tr>
<td>Critical</td>
<td></td>
<td>Make auto-threads execute attributed statement atomically</td>
</tr>
</tbody>
</table>
COMPLETE PROGRAM PARALLELISM

```
 Thread[pool-1-thread-2,5,main]
  0.6501712957370558
 Thread[pool-1-thread-1,5,main]
  0.6907459159093547

//omp parallel threadNum(2)
{
  //omp critical
  trace(Math.random());
}

public static void trace(Object... anArgs) {
  System.out.print(Thread.currentThread());
  for (Object anArg : anArgs) {
    System.out.print(" "+ anArg);
  }
  System.out.println();
}
```

Serial + Parallel Program?
**Partial Parallelism**

Thread[main,5,main] Forking
Thread[pool-1-thread-2,5,main] 0.8112103632254872
Thread[pool-1-thread-1,5,main] 0.7339312982272137
Thread[main,5,main] Joined

```
trace("Forking");

//omp parallel threadNum(2)
{
    //omp critical
    trace(Math.random());
}
trace("Joined");
```

**Serial + Parallel Program**

**Which thread(s) will print “Forking” and “Joined”?**
**Abstract Fork-Join**

- **T^0** → Statement
- **T^0** → fork(n)
- **T^1** → Statement
- **T^n** → Statement
- **T^0** → Join

**create thread^{1..n}**

**Make each thread execute forked statement**

**Make creating thread wait for termination of thread^{1..n}**

**More efficient thread creation?**
EQUIVALENT, MORE EFFICIENT ABSTRACT FORK-JOIN

- $T^0$ → Statement
- $T^0$ → fork(n)
- $T^0$ → Statement
- $T^0$ → Join
- $T^0$ → Statement;

create thread$^1..thread^{n-1}$

Original thread could also execute forked statement if it is going to join

Similar to Unix Fork-Join?
**UNIX SINGLE PROCESS FORK-JOIN**

- **$P_0$**
  - **Statement**
  - `childPid = fork()`
  - **Statement**
  - If $(\text{getPid()} \neq \text{childPid})$
  - **join(childPid)**

- **$N$-Slaves?**
  - System call, returns pid of new process in both processes
  - Code executed by both processes
  - Wait for termination of specified child process
**UNIX MULTIPLE process FORK-JOIN**

<table>
<thead>
<tr>
<th><strong>P₀</strong></th>
<th>for (i=1; i &lt; n; i++)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If (getPid() != childPid)</td>
</tr>
<tr>
<td></td>
<td>childPid = fork()</td>
</tr>
<tr>
<td><strong>P₀</strong></td>
<td>Statement</td>
</tr>
<tr>
<td><strong>P₀</strong></td>
<td><strong>P₀</strong></td>
</tr>
<tr>
<td><strong>P₀</strong></td>
<td>If (getPid() != childPid)</td>
</tr>
<tr>
<td><strong>P₀</strong></td>
<td>join()</td>
</tr>
<tr>
<td><strong>P₀</strong></td>
<td><strong>P₀</strong></td>
</tr>
</tbody>
</table>

Procedural (error-prone) code

Wait for termination of all child processes
Thread[main,5,main] Forking
Thread[pool-1-thread-2,5,main] 0.8112103632254872
Thread[pool-1-thread-1,5,main] 0.733931298272137
Thread[main,5,main] Joined

```
trace("Forking");

//omp parallel threadNum(2)
{
    //omp critical
    trace(Math.random());
}
trace("Joined");
```
# OpenMP Statement Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td></td>
<td>Create multiple threads to execute attributed statement, which are killed after execution of statement</td>
</tr>
<tr>
<td>NumberOfThreads</td>
<td>int</td>
<td>Number of threads to be created automatically</td>
</tr>
<tr>
<td>Critical</td>
<td></td>
<td>Make auto-threads execute attributed statement atomically</td>
</tr>
</tbody>
</table>

**Range of concurrent programs supported by attributes?**
SAME CODE, SAME DATA CONCURRENCY

T1

print random#

T2

Same data
SAME CODE, SAME DATA CONCURRENCY

T1

T2

print “hello world”

Same or No data
**SAME CODE, SAME DATA CONCURRENCY**

- `System.out.println(isPrime(toInt(Math.random()));`
- `System.out.println("Hello World");`

Printing the same value or computing the value is not very useful

Cannot think of another example (pipelines considered same code/instruction)

Same data forces some serialization

Other patterns?
## Parallelization Patterns

<table>
<thead>
<tr>
<th></th>
<th>Code</th>
<th>Data (Parts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>SIMD</td>
<td>Same</td>
<td>Different</td>
</tr>
<tr>
<td>MISD</td>
<td>Different</td>
<td>Same</td>
</tr>
<tr>
<td>MIMD</td>
<td>Different</td>
<td>Different</td>
</tr>
</tbody>
</table>

SISD straightforwardly supported using declarative primitives.

We cannot support all possible concurrences with pure declarative primitives.
**OpenMP Statement Attributes (Abstract)**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td></td>
<td>Create multiple threads to execute attributed statement, which are killed after execution of statement</td>
</tr>
<tr>
<td>NumberOfThreads</td>
<td>int</td>
<td>Number of threads to be created automatically</td>
</tr>
<tr>
<td>Critical</td>
<td></td>
<td>Make auto-threads execute attributed statement atomically</td>
</tr>
</tbody>
</table>

Imperative primitives that can be added to these attributes to increase flexibility?
How to integrate alternation with fork-join?
UNIX INSPIRATION

```plaintext
for (i=1; i < n; i++)

If (getPid() != childPid)

childPid = fork()

Statement

If (getPid() != childPid)

join()

Task depends on Id
```
Allow each member of a thread sequence declaratively created execute an imperative step to determine its index.

Different code/data can be executed/accessed by different threads based on their indices.
ThreadNum Example: Same Code Different Data

```java
public static void parallelSumAndToText(float[] aList) {
    // omp parallel threadNum(2)
    {
        if (OMP4J_THREAD_NUM == 0) {
            trace("Sum of rounded:" + sum(aList));
        } else {
            trace("ToText of rounded:" + toText(aList));
        }
    }
}
```
### OpenMP Procedures and Statement Attributes

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td></td>
<td>Create multiple threads to execute attributed statement, which are killed after execution of statement</td>
</tr>
<tr>
<td>NumberOfThreads</td>
<td>int</td>
<td>Number of threads to be created automatically</td>
</tr>
<tr>
<td>Critical</td>
<td></td>
<td>Make auto-threads execute attributed statement atomically</td>
</tr>
</tbody>
</table>

#### Procedures

<table>
<thead>
<tr>
<th>Procedures</th>
<th>Signature</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetThreadNum</td>
<td>→int</td>
<td>Returns index of thread created by Parallel pragma</td>
</tr>
<tr>
<td>GetNumThreads</td>
<td>→int</td>
<td>Returns number of threads created by Parallel pragma</td>
</tr>
</tbody>
</table>
DIFFERENT CODE, SAME DATA CONCURRENCY

T1 \rightarrow \text{Code C1} \rightarrow \text{Same or No data} \rightarrow \text{Code C2} \rightarrow T2
SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY

T1 → Float-List Round → Data (Parts) 1
T2 → Float-List Round → Data (Parts) 2

How to divide data among threads?
CLOSE COUPLING APPROACH

Thread 0 takes elements at even indices
Thread 1 takes elements at even indices
Thread-Aware Float-List Round, Chooses Which Indices to Process Based on ThreadNumber, Number of Threads and Index
Thread Assigner

```java
boolean processIteration(int anIndex, int aStart, int aLimit, int aStepSize, int aThreadNum, int aNumThreads) {
    return aNumThreads == 0 ||
           (anIndex % aNumThreads == aThreadNum);
}
```

Different useful implementations of method (possibly chosen by a factory) possible and discussed later
**Single-Threaded Round**

```
public static void round(float[] aList) {
  trace("Round Started:" + Arrays.toString(aList));
  for (int i = 0; i < aList.length; i++) {
    aList[i] = (float) Math.round(aList[i]);
    trace(aList[i]);
  }
  trace("Round Ended:" + Arrays.toString(aList));
}
```

Put a compound statement for the body and precede it with a parallel pragma

Body of loop is executed based on thread number, index and #threads

`processIteration()` passed these values and returns boolean
public static void parallelRound(float[] aList) {
    // omp parallel threadNum(2)
    {
        trace("Round Started:" + Arrays.toString(aList));
        for (int i = 0; i < aList.length; i++) {
            if (processIteration(i, 0, 1, aList.length, OMP4J_THREAD_NUM, OMP4J_NUM_THREADS)) {
                aList[i] = (float) Math.round(aList[i]);
                trace(aList[i]);
            }
        }
        trace("Round Ended:" + Arrays.toString(aList));
    }
}
Two Independent Problems

T1
T2

Float-List Round

Float-List Round

Number-List Sum

Number-List ToString
**Pipelined Functions: Single-Thread**

```java
class Main {
    public static void main(String[] args) {
        float[] list = {4.8, 5.2, 4.5, 4.75, 4.7};
        roundSumAndToText(list);
    }
}

public static void roundSumAndToText (float[] aList) {
    round(aList);
    trace("Sum of rounded:" + sum(aList);
    trace("ToText of rounded:" + toText(aList));
}
```

Thread[main,5,main] Round Started:[4.8, 5.2, 4.5, 4.75, 4.7]
Thread[main,5,main] Round Ended:[5.0, 5.0, 5.0, 5.0, 5.0]
Thread[main,5,main] Sum of rounded:25.0
Thread[main,5,main] ToText of rounded: 5.0 5.0 5.0 5.0 5.0
Pipelined: Separate Thread Teams

T1 -> Float-List Round

T2 -> Float-List Round

T3 -> Number-List Sum

T4 -> Number-List ToString
public static void parallelRound(float[] aList) {
    // omp parallel threadNum(2)
    {
        trace("Round Started:" + Arrays.toString(aList));
        for (int i = 0; i < aList.length; i++) {
            if (processIteration(i, OMP4J_THREAD_NUM, OMP4J_NUM_THREADS)) {
                aList[i] = (float) Math.round(aList[i]);
                trace(aList[i]);
            }
        }
        trace("Round Ended:" + Arrays.toString(aList));
    }
}
public static void parallelSumAndToText(float[] aList) {
    // omp parallel threadNum(2)
    if (OMP4J_THREAD_NUM == 0) {
        trace("Sum of rounded:" + sum(aList));
    } else {
        trace("ToText of rounded:" + toText(aList));
    }
}
Pipelined: Separate Thread Teams

T1 -> Float-List Round

T2 -> Float-List Round

T3 -> Number-List Sum

T4 -> Number-List ToString

More efficient solution?
PIPELINED: SAME THREAD TEAM

T1

Float-List Round

Number-List Sum

T2

Float-List Round

Number-List ToString
public static void parallelRoundAndSumToText(float[] aList) {
    // omp parallel threadNum(2)
    {
        round(aList[i]);
        if (OMP4J_THREAD_NUM == 0) {
            trace("Sum of rounded:" + sum(aList));
        } else {
            trace("ToText of rounded:" + toText(aList));
        }
    }
}

Thread[pool-1-thread-1,5,main] Round Started:[4.8, 5.2, 4.5, 4.75, 4.7]
    Round Started:[4.8, 5.2, 4.5, 4.75, 4.7]
Thread[pool-1-thread-1,5,main] Round Ended:[5.0, 5.2, 5.0, 4.75, 5.0]
Thread[pool-1-thread-1,5,main] Sum of rounded:24.95
Thread[pool-1-thread-2,5,main] ToText of rounded: 5.0 5.0 5.0 5.0 5.0

New declarative construct to fix problem?
public static void parallelRoundAndSumToText(float[] aList) {
    // omp parallel threadNum(2)
    {
        round(aList[i]);
        // omp barrier
        if (OMP4J_THREAD_NUM == 0) {
            trace("Sum of rounded:" + sum(aList));
        } else {
            trace("ToText of rounded:" + toText(aList));
        }
    }
}

    Round Started:[4.8, 5.2, 4.5, 4.75, 4.7]
Thread[pool-1-thread-1,5,main] Round Ended:[5.0, 5.2, 5.0, 4.75, 5.0]
Thread[pool-1-thread-2,5,main] Round Ended:[5.0, 5.0, 5.0, 5.0, 5.0]
Thread[pool-1-thread-2,5,main] ToText of rounded: 5.0 5.0 5.0 5.0 5.0
Thread[pool-1-thread-1,5,main] Sum of rounded:25.0
### OpenMP Statement Attributes

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td></td>
<td>Create multiple threads to execute attributed statement, which are killed after execution of statement</td>
</tr>
<tr>
<td>NumberOfThreads</td>
<td>int</td>
<td>Number of threads to be created automatically</td>
</tr>
<tr>
<td>Critical</td>
<td></td>
<td>Make auto-threads execute attributed statement atomically</td>
</tr>
<tr>
<td>Barrier</td>
<td></td>
<td>Wait for each sibling created by Parallel pragma to finish preceding statement before proceeding to next statement</td>
</tr>
</tbody>
</table>
**Declarative SumToText?**

```java
public static void parallelRoundAndSumToText(float[] aList) {
    // omp parallel threadNum(2)
    {
        round(aList[i]);
        // omp barrier
        if (OMP4J_THREAD_NUM == 0) {
            trace("Sum of rounded:" + sum(aList));
        } else {
            trace("ToText of rounded:" + toText(aList));
        }
    }
}
```

Can we get rid of the if and use of procedural constructs in this method?
public static void sectionRoundAndSumToText(float[] aList) {
    // omp parallel threadNum(2)
    {
        round(aList[i]);
        // omp barrier
        // omp sections
        {
            // omp section
            trace("Sum of rounded:" + sum(aList));
            // omp section
            trace("ToText of rounded:" + toText(aList));
        } // omp section
    } // omp sections
} // omp parallel
SEQUENTIAL VS CONCURRENT EXECUTION

Begin

T1 → Statement1

T1 → Statement2

T1 → StatementN

End
SEQUENTIAL VS CONCURRENT EXECUTION

Co-Begin

T1
Statement1
T2
Statement2
T1
StatementN
End

A statement/section is executed by a single rather than all threads

Co-Begin can replace Begin if no state Changes (Functional languages)
# OpenMP Statement Attributes

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td></td>
<td>Create multiple threads to execute attributed statement, which are killed after execution of statement</td>
</tr>
<tr>
<td>NumberOfThreads</td>
<td>int</td>
<td>Number of threads to be created automatically</td>
</tr>
<tr>
<td>Critical</td>
<td></td>
<td>Make auto-threads execute attributed statement atomically</td>
</tr>
<tr>
<td>Barrier</td>
<td></td>
<td>Wait for each sibling created by Parallel pragma to finish preceding statement before proceeding to next statement</td>
</tr>
<tr>
<td>Sections</td>
<td></td>
<td>Co-Begin around attributed statement</td>
</tr>
<tr>
<td>Section</td>
<td></td>
<td>Allow attributed sub statement to be executed by any entering thread</td>
</tr>
</tbody>
</table>
CLOSE COUPLING APPROACH

Thread 0 takes elements at even indices
Thread 1 takes elements at even indices
**Thread Assigner**

Interface

```java
boolean processIteration(int anIndex, int aStart, int aLimit, int aStepSize, int aThreadNum, int aNumThreads);
```

Different useful implementations of method (possibly chosen by a factory) possible and discussed later

```java
boolean processIteration(int anIndex, int aStart, int aLimit, int aStepSize, int aThreadNum, int aNumThreads) {
    return aNumThreads == 0 ||
            (anIndex % aNumThreads == aThreadNum);
}
```
**Declarative Parallel Round?**

```java
boolean processIteration(int anIndex, int aStart, int aLimit, int aStepSize, int aThreadNum, int aNumThreads);

public static void parallelRound(float[] aList) {
    // omp parallel threadNum(2)
    {
        for (int i = 0; i < aList.length; i++) {
            if (processIteration(i, OMP4J_THREAD_NUM, OMP4J_NUM_THREADS)) {
                aList[i] = (float) Math.round(aList[i]);
                trace(aList[i]);
            }
        }
    }
}
```

processIteration() is round independent but dependent on loop

Can OMP automatically implement it and call it based on a loop pragma?
**Declarative Parallel Round**

```java
public static void parallelRound(float[] aList) {
    // omp parallel for threadNum(2)
    for (int i = 0; i < aList.length; i++) {
        aList[i] = (float) Math.round(aList[i]);
        trace(aList[i]);
    }
}
```

Conceptually, OpenMP calls processIteration(), actual implementation does not require each thread to determine if it should execute each iteration.

Allow pragma to be associated with every for loop?
Declarative Parallel Round?

```java
boolean processIteration(int anIndex, int aStart, int aLimit, int aStepSize, int aThreadNum, int aNumThreads);

public static void parallelRound(float[] aList) {
    int i = 0;
    // omp parallel for threadNum(2)
    for (; ;) {
        aList[i] = (float) Math.round(aList[i]);
        trace(aList[i]);
        i++;
        if (i == aList.length)
            break;
    }
}
```

processIteration() cannot be passed loop index and other parameters

OMP needs counter-controlled loops: loop with an index variable, increment, start, and limit known at loop entry
# OpenMP Attributes Qualifying Parallel Attributed Loops

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>For</td>
<td></td>
<td>Allow different iterations of a counter-controlled loop to execute in parallel based on counter and other properties</td>
</tr>
</tbody>
</table>
Dividing Iterations: Closest Distance Division

How should iterations be automatically divided among threads?

Closet proximity approach—difference between indices of j\textsuperscript{th} iterations processed by different threads cannot be closer.
Dividing Iterations: Furthest Distance Division

Furthest proximity approach—difference between indices of jth iterations processed by different threads cannot be further.

Comparison based on locality arguments?
**Locality Granularities**

Diagram showing the relationship between threads (T1, T2), processors (P1, P2), cache, memory, and disk. The diagram illustrates locality granularities across different levels of the memory hierarchy.
False sharing: processors writing to different parts of a cache block require a cache coherence algorithm.
Furthest Distance Division
Static (4) Scheduling – Default Loose Coupling

\[
\begin{array}{ccccccc}
I^1 & I^2 & I^3 & I^4 & I^5 & I^6 & I^7 \\
\end{array}
\]

T1

\[
\begin{array}{cccc}
I^1 & I^2 & I^3 & I^4 \\
\end{array}
\]

T2

\[
\begin{array}{ccc}
I^5 & I^6 & I^7 \\
\end{array}
\]
Static (1) Scheduling – Close Coupling

T1

T2

I^1 | I^2 | I^3 | I^4 | I^5 | I^6 | I^7

I^1 | I^3 | I^5 | I^7

I^2 | I^4 | I^6
**Static (2) Scheduling – Variable Iteration Work**

![Diagram of iteration scheduling]

- **T1**:
  - \( I^1 \), \( I^2 \), \( I^5 \), \( I^6 \)

- **T2**:
  - \( I^3 \), \( I^4 \), \( I^7 \)
Dynamic (2) Schedule

\[ \begin{array}{ccccccc}
I^1 & I^2 & I^3 & I^4 & I^5 & I^6 & I^7 \\
\end{array} \]

T1

\[ \begin{array}{ccc}
I^1 & I^2 & I^7 \\
\end{array} \]

T2

\[ \begin{array}{cccc}
I^3 & I^4 & I^5 & I^6 \\
\end{array} \]
Dynamic (2) Schedule

\[
\begin{array}{cccccccccccc}
I^1 & I^2 & I^3 & I^4 & I^5 & I^6 & I^7 & I^8 & I^9 & I^{10} & I^{11} & I^{12} \\
\end{array}
\]

T1

\[
\begin{array}{ccc}
I^1 & I^2 & I^7 \\
\end{array}
\]

T2

\[
\begin{array}{ccccccc}
I^3 & I^4 & I^7 & I^8 & I^{11} & I^{12} \\
\end{array}
\]

T3

\[
\begin{array}{cccc}
I^5 & I^6 & I^9 & I^{10} \\
\end{array}
\]
### Guided (1) Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>I^1</td>
<td>I^2</td>
<td>I^3</td>
</tr>
<tr>
<td>I^5</td>
<td>I^6</td>
<td>I^7</td>
</tr>
<tr>
<td>I^8</td>
<td>I^9</td>
<td>I^10</td>
</tr>
</tbody>
</table>

- **12/3**
  - T1: I^1, I^2, I^3, I^4

- **8/3**
  - T2: I^5, I^6, I^7, I^{11}

- **5/3 3/3 2/3 1/3**
  - T3: I^8, I^9, I^{10}, I^{12}
CHARACTERIZING AUTOMATICALLY PARALLELIZABLE LOOPS

```java
public static void parallelRound(float[] aList) {
    // omp parallel for threadNum(2)
    for (int i = 0; i < aList.length; i++) {
        aList[i] = (float) Math.round(aList[i]);
        trace(aList[i]);
    }
}
```

processIteration() is round independent but dependent on loop

OMP can automatically implement it and call it based on a loop pragma
**Single-Threaded Round**

```java
public static void round(float[] aList) {
    trace("Round Started:" + Arrays.toString(aList));
    for (int i = 0; i < aList.length; i++) {
        aList[i] = (float) Math.round(aList[i]);
        trace(aList[i]);
    }
    trace("Round Ended:" + Arrays.toString(aList));
}
```

*Put a compound statement for the body and precede it with a parallel pragma*

*Body of loop is executed based on thread number, index and #threads*

`processIteration()` passed these values and returns boolean
public static void parallelRound(float[] aList) {
    // omp parallel threadNum(2)
    {
        trace("Round Started:" + Arrays.toString(aList));
        for (int i = 0; i < aList.length; i++) {
            if (processIteration(i, OMP4J_THREAD_NUM, OMP4J_NUM_THREADS)) {
                aList[i] = (float) Math.round(aList[i]);
                trace(aList[i]);
            }
        }
        trace("Round Ended:" + Arrays.toString(aList));
    }
}
SAME CODE, DIFFERENT DATA (PARTS)
CONCURRENCY, SPLIT BUT NO REDUCTION

T1 \rightarrow \text{Float-List Round} \rightarrow \text{Float-List Round} \rightarrow \text{Thread 0 takes elements at even indices}

T2 \rightarrow \text{Float-List Round} \rightarrow \text{Float-List Round} \rightarrow \text{Thread 1 takes elements at even indices}
ALTERNATIVE THREADING

Float-List Round

Number-List Sum

Float-List Round

Number-List ToString
public static void round(float[] aList) {
    trace("Round Started:" + aThreadNum);
    for (int i = 0; i < aList.length; i++) {
        aList[i] = (float) Math.round(aList[i]);
        trace(aList[i]);
    }
    trace("Round Ended:" + aThreadNum);
}
Leveraging Statement-Based Concurrency Attributes?
public static void round(float[] aList, int aThreadNum, int aNumThreads) {
    trace("Round Started:" + aThreadNum);
    for (int i = 0; i < aList.length; i++) {
        if (processIteration(i, aThreadNum, aNumThreads)) {
            if (aThreadNum == 1) {
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            aList[i] = (float) Math.round(aList[i]);
            trace(aList[i]);
        }
    }
    trace("Round Ended:" + aThreadNum);
}
public static void round(float[] aList, int aThreadNum, int aNumThreads) {
    trace("Round Started:" + aThreadNum);
    for (int i = 0; i < aList.length; i++) {
        if (processIteration(i, aThreadNum, aNumThreads)) {
            if (aThreadNum == 1) {
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            aList[i] = (float) Math.round(aList[i]);
            trace(aList[i]);
        }
    }
    trace("Round Ended:" + aThreadNum);
}
Different Code, Same Data Concurrency

T1 → Number-List Sum → Number-List ToString

T2 → Number-List Sum → Number-List ToString
SAME CODE, DIFFERENT DATA (PARTS)
CONCURRENCY, SPLIT BUT NO REDUCTION

T1 \rightarrow \text{Float-List Round} \\
T2 \rightarrow \text{Float-List Round}

Split data among concurrent threads
SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY

T1 \rightarrow Number-List Sum
T2 \rightarrow Number-List Sum

Final Result
Reduce concurrent results
Split data among concurrent threads
SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY

Number (abstract) + is commutative and associative

String + is not commutative and associative
**SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY**

Number (abstract) + is commutative and associative

String + is not commutative and associative
SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY

T1 \rightarrow \text{Number-List Sum} \rightarrow \text{Final Result}

T2 \rightarrow \text{Number-List Sum} \rightarrow \text{Reduce concurrent results} \rightarrow \text{Split without reduction?} \rightarrow \text{Split data among concurrent threads}

10
SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY

T1

Code C

Data (Parts) 1

T2

Code C

Data (Parts) 2
DIFFERENT CODE, DIFFERENT DATA (PARTS) CONCURRENCE

T1

Code C1

Data (Parts) 1

T2

Code C2

Data (Parts) 2
**OpenMP Statement Attributes (Abstract)**

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td></td>
<td>Create multiple threads to execute attributed statement</td>
</tr>
<tr>
<td>Number</td>
<td>OfThreads</td>
<td>int</td>
</tr>
<tr>
<td>Critical</td>
<td></td>
<td>Make auto-threads execute attributed statement atomically</td>
</tr>
</tbody>
</table>

- Can we support all possible concurrencies with pure declarative primitives?
- Imperative primitives that can be added to these attributes to increase flexibility?
LEVERAGING STATEMENT-BASED CONCURRENCY ATTRIBUTES?

T1 → Number-List Sum

T2 → Number-List ToString

---

LEVERAGING STATEMENT-BASED CONCURRENCY ATTRIBUTES?
DIFFERENT CODE, SAME DATA CONCURRENCY

T1 → Number-List Sum

T2 → Number-List ToString

[Diagram showing two processes T1 and T2, connected to Number-List Sum and Number-List ToString respectively.]
Let each member of a thread sequence declaratively created execute an imperative step to determine its index.

Different code/data can be executed/accessed by different threads based on their indices.
ThreadNum Example: Same Code Different Data

```java
public static void parallelSumAndToText(float[] aList) {
    // omp parallel threadNum(2)
    {
        if (OMP4J_THREAD_NUM == 0) {
            float aSum = sum(aList);
            trace("Sum of rounded:" + aSum);
        } else {
            String aString = toText(aList);
            trace("ToText of rounded:" + aString);
        }
    }
}
```
SAME CODE, DIFFERENT DATA (PARTS)
CONCURRENCY, SPLIT BUT NO REDUCTION

Thread 0 takes elements at even indices
Thread 1 takes elements at even indices
OMP Atomic Parallel Random

Thread[pool-1-thread-2,5,main]
0.6501712957370558
Thread[pool-1-thread-1,5,main]
0.6907459159093547

//omp parallel threadNum(2)
{
    //omp critical
    trace(Math.random());
}

public static void trace(Object... anArgs) {
    System.out.print(Thread.currentThread());
    for (Object anArg : anArgs) {
        System.out.print(" "+anArg);
    }
    System.out.println();
}
DIFFERENT CODE, SAME DATA CONCURRENCY

Sum or Print

List

Print List

T1

T2
DIFFERENT CODE, SAME DATA CONCURRENCY

T1 -> Sum List

T2 -> Print List
public interface Counter {
    void increment(int val);
    int getValue() throws RemoteException;
}
public class ACounter implements Counter{
    public ACounter() {
        super();
    }
    Integer value = 0;
    public Object getValue() {
        return value;
    }
    public void increment(int val) {
        value += val;
    }
    public String toString() {
        return "Counter:" + value;
    }
    public boolean equals(Object otherObject) {
        if (!((otherObject instanceof Counter))
            return false;
        return getValue() == ((Counter) otherObject).getValue();
    }
}
ISSUE

How to use RMI for implementing various properties of a collaborative applications?

In particular one implemented using MVC?

Will use our MVC-based uppercasing application as an example
How to use RMI for implementing various properties of a collaborative applications?

In particular one implemented using MVC?
SINGLE-THREAD ➔ PARALLEL ➔ DISTRIBUTED

Non Distributed Single-Thread Process ➔ Non Distributed Multi-Thread Process

Computation Distribution (Number, Matrix Multiplication)

Distributed Process ➔ Distributed Consistent Process
**Single-Thread ➔ Parallel**
**Single Thread Pipeline**

- Float-List Round
- Number-List Sum
- Number-List ToString
Thread Pairs Pipeline

- T1 → Float-List Round
- T3 → Number-List Sum
- T2 → Float-List Round
- T4 → Number-List ToString
More Efficient Two-Thread Threading?

- T1 → Float-List Round
- T3 → Number-List Sum
- T2 → Float-List Round
- T4 → Number-List ToString
**Alternative Threading?**

- **T1**
  - Float-List Round
  - Number-List Sum

- **T2**
  - Float-List Round
  - Number-List ToString
NEW DECLARATIVE PRAGMA?

T1
- Float-List Round
  - Number-List Sum

T2
- Float-List Round
  - Number-List ToString
**Barrier**

T1:
- Float-List Round
- Barrier
- Number-List Sum

T2:
- Float-List Round
- Barrier
- Number-List ToString