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 popular software abstractions:
  - OpenMP (Parallel Computing and Distributed Shared Memory)
  - MapReduce (Distributed Computing)
  - OpenMPI (~ Sockets, not covered)
- Prins 633 course
  - Covers HPC PDC algorithms and OpenMP in detail.
- Don Smith 590 Course
  - Covers MapReduce uses in detail
- Connect OpenMP and MapReduce to
  - Each other (reduction)
  - Non HPC PDC
  - Research (with which I am familiar)
- Derive the design and implementation of both kinds of abstractions
  - From similar but different performance issues that motivate them.
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 among distributed users
  - Aggregation of Distributed Data
  - High-Performance: Speedup

- **Secondary**
  - Remote service, aggregation,...

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

Non-Distributed Existing Program

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

Distributed Program

- Computation Distribution (Number, Matrix Multiplication)

Non-Distributed Existing Single-Thread Program

- Distributed Multi-Thread Program

Distributed Program

- Distributed Program

Collaborative Applications (Games, Shared Desktops)

Replicated Repositories (Files, Databases)

Computation Distribution (Number, Matrix Multiplication)
CONCURRENCY-DISTRIBUTION RELATIONSHIP

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 for speedup and 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**

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

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

How to parallelize/distribute single-thread algorithms?
A central 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 whose code is unaware of specific arbitrary client/slave locations and ports.
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 reuse existing single-thread code in multi-thread/multi-process program?

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

Distributed separation of concerns!
**Security Issues**

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

Communicating processes created independently on typically geographically dispersed autonomous hosts, raising security issues.

Communicating threads/processes created on hosts/processors typically under control of one authority.

Though crowd problem solving is an infrequent exception: UW Condor – part of Cverse/XSEDE, Wagstaff primes.
**Lifetime of Processes/Threads**

- **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 computation complete**
**Coupling vs High-Performance**

**Coupling/Consistency**
- Independent, “long-lived” processes at different autonomous locations made dependent for resource sharing, user collaboration, fault tolerance.
- Include consistency algorithms, possibly in separate threads, to define dependency that have distributed producer-consumer, observer relationship with existing algorithms.
- May use additional mediating server and other infrastructure code unaware of specific clients.
- Division of labor between client and infrastructure an issue (centralized vs replicated).

**High Performance**
- A single short-lived process created to perform some computation made multi-threaded and/or distributed to increase performance.
- Speedup algorithms that replace logic of existing code, with task decomposition.
- May involve a central mediating, distributing master code but it can be aware of and creator of specific slave processes.
- Division of labor among master and slave and slaves an issue.
Example: Service + Speedup

Client

Client$^1$

Client$^2$

Comp 401 Grader

Slave

Assignment-level decomposition

Server

Master

Grader Server

Slave

Comp 533 Grader

Slave

Test-level decomposition

Test

Comp 533 Grader

Slave

Slave

Master

533 Tester

T$^3$

T$^4$

T$^1$

T$^5$

T$^6$
Higher-level abstractions for different classes of speedup algorithms?
COMPLETE AUTOMATION?

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

**Modules of Non-Distributed Program Distributed Transparently**

- Loader contacts registry to determine if local module loaded or remote module accessed
- Assumes one name space, one instance of each service – cannot handle replication
- Motivates RPC transparency

- Computation Distribution (Number, Matrix Multiplication)

**Parallelizing compilers (Kuck and Kennedy)**

- Halting problem
- Motivates Declarative Abstractions
DEclarative ABstractions

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

- Threads, IPC, Bounded Buffer

Adding Declarative abstractions for different classes of speedup algorithms
**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, ...

Loop

Locality relevant to PDC abstractions - later
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

Ease of programming?
Declarative vs Imperative PDC: Competing and Complementing

Declarative: Specify what we want

For restricted classes of programs

Imperative: Implement what we want

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

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

aRegistry.rebind(Server.NAME, aServer)

Concurrency?  Distribution?
Declarative vs Imperative Concurrency: Competing and Complementing

Declarative: Specify what we want

Imperative: Implement what we want

Arguably easier to program.

For restricted classes of programs

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

Concurrency?
Parallel Random

0.6455074599676613
0.14361454218773773

Desired I/O

Declarative Concurrency Specification Called a Pragma or Directive

//omp parallel threadNum(2)

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

Declarative Concurrency Aware Code?

Imperative Concurrency-Unaware Code
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
- Like Java annotations associated with variable, class and method declarations
  - Associated with statements
- 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());
PRE-COMPILIER 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());
Tracing Parallel Random

Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-1,5,main] 0.9310973090994396
0.31647362613936514

```
//omp parallel threadNum(2)
{
    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();
}
```
Java Solution?

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

//omp parallel threadNum(2)

class Main {
    public static void trace(Object... anArgs) {
        synchronized(this) {
            trace(Math.random());
        }
        System.out.print(Thread.currentThread());
        for (Object anArg : anArgs) {
            System.out.print(" "+anArg);
        }
        System.out.println();
    }
    public static void main(String[] args) {
        trace("Hello", "World!");
    }
}

Desired I/O

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

```java
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();
}
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

T₀ → Statement

T₀ → fork(n)

T₁ → Statement
Tₙ → Statement

T₀ → Join

create thread¹..threadⁿ

Make each thread execute forked statement

Make creating thread wait for termination of thread¹..threadⁿ

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

\[ T_0 \rightarrow \text{Statement} \]

\[ T_0 \rightarrow \text{fork}(n) \]

\[ T_0 \rightarrow \text{Statement} \]

\[ T_0 \rightarrow \text{Join} \]

\[ T_0 \rightarrow \text{Statement;} \]

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**

- **P0** ➔ Statement
- **P0** ➔ `childPid = fork()`
- **P0** ➔ Statement
- **P1** ➔ Statement
- **P0** ➔ `If (getPid() != childPid)`
- **P0** ➔ `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

---

The diagram illustrates the process of creating a new process using the `fork()` system call, and then joining with the child process using the `join()` function.
UNIX MULTIPLE PROCESS FORK-JOIN

**Procedural (error-prone) code**

<table>
<thead>
<tr>
<th>P0</th>
<th>for (i=1; i &lt; n; i++)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If (getPid() != childPid)</td>
</tr>
<tr>
<td>P0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>childPid = fork()</td>
</tr>
<tr>
<td>P0</td>
<td>Statement</td>
</tr>
<tr>
<td>P0</td>
<td>If (getPid() != childPid)</td>
</tr>
<tr>
<td>P0</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>join()</td>
</tr>
<tr>
<td></td>
<td>Wait for termination of all child processes</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Thread[main,5,main] Forking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread[pool-1-thread-2,5,main]</td>
</tr>
<tr>
<td>0.8112103632254872</td>
</tr>
<tr>
<td>Thread[pool-1-thread-1,5,main]</td>
</tr>
<tr>
<td>0.7339312982272137</td>
</tr>
<tr>
<td>Thread[main,5,main] Joined</td>
</tr>
</tbody>
</table>

```java
trace("Forking");

#pragma omp parallel threadNum(2)
{
    #pragma 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

T2

print random#

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/Instruction</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 concurrencies 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>NumberThreads</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>
Different Code, Same Data Concurrency

How to integrate alternation with fork-join?
UNIX INSPIRATION

for (i=1; i < n; i++)
If (getPid() != childPid)
childPid = fork()

Statement

if (getPid() != childPid)
join()
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>

<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 | Code C1 | Same or No data
T2 | Code C2

Code C1

T1

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?
**SAME CODE, DIFFERENT DATA (PARTS)**

Thread-Aware Float-List Round, Chooses Which Indices to Process Based on ThreadNumber, Number of Threads and Loop Params

T1

T2
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));
}
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));
    }
}
Thread Assigner

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

boolean processIteration (int aStart, int aLimit, int aStepSize, int aThreadNum, int aNumThreads) {
    return aNumThreads == 0 ||
    (anIndex % aNumThreads == aThreadNum);
}
public static void parallelRound(float[] aList) {
    int i = 0;
    // omp parallel for threadNum(2)
    for (;;) {
        if (processIteration(i, 0, 1, aList.length,
                              OMP4J_THREAD_NUM,
                              OMP4J_NUM_THREADS)) {

            aList[i] = (float) Math.round(aList[i]);
            trace(aList[i]);
            i++;
            if (i == aList.length)
                break;
        }
    }
}

processIteration() parameters may not be explicit parameters of for construct
TWO INDEPENDENT PROBLEMS

- T1 \rightarrow \text{Float-List Round}
- T2 \rightarrow \text{Float-List Round}
- T1 \rightarrow \text{Number-List Sum}
- T2 \rightarrow \text{Number-List ToString}
**PIPEDINED FUNCTIONS: SINGLE-THREAD**

```java
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 paralellSumAndToText(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

More efficient solution?
PIPELINED: SAME THREAD TEAM

- T1
  - Float-List Round
  - Number-List Sum
  - Number-List ToString

- T2
  - Float-List Round
  - Number-List Sum
  - Number-List ToString
SAME TEAM: STEP 1 AND 2

```java
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));
        }
    }
}
```


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>
Can we get rid of the if and use of procedural constructs in this method?
Sections can be executed in parallel rather than sequential execution
SEQUENTIAL VS CONCURRENT EXECUTION

Begin

T1 → Statement1

T1 → Statement2

T1 → StatementN

End
**Sequential vs Concurrent Execution**

- **Co-begin**
  - T1 → Statement1
  - T2 → Statement2
  - T1 → StatementN

- **CoEnd**

**Dijsktra (1968)**

- 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>
PROCEDURAL 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 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?
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 automatically passed loop index and other parameters

OMP for parallelism 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>
SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY

How should iterations be automatically divided among threads?
null
CLOSEST DISTANCE APPROACH

T1
Float-List Round

T2
Float-List Round

Thread 0 takes elements at even indices

Thread 1 takes elements at even indices
**Furthest Distance Division**

Furthest distance approach—difference between indices of $j^{th}$ iterations processed by different threads cannot be further.

Comparison based on locality arguments?
LOCALITY GRANULARITIES

- **T1**
  - **P1**
    - Processor Cache
    - A0
    - A1

- **T2**
  - **P2**
    - Memory
    - A0
    - A1
    - A2
    - A3

- **Disk**
  - A0
  - A1
  - A2
  - A3
  - A4
  - A5
  - A6
  - A7
**Conflicting Replica Changes**

- T1
- T2
- P1
- P2

**True sharing conflict**

- Cache coherence algorithm guarantees consistency (atomic broadcast, two-phase commit)

- Processor Cache
- Memory
- Disk
Closest Distance Division

Consistency is maintained at the cache block level

False sharing! Processors writing to different parts of a cache block execute a cache coherence algorithm
**Furthest Distance Division**

![Diagram of Furthest Distance Division](image)

- **T1** and **T2** are processors with cache memory.
- **P1** and **P2** are processors accessing memory and disk.
- **A0** to **A7** represent memory and disk locations.

- **Processor**: Cache
- **Memory**: A0, A1, A2, A3
- **Disk**: A0, A1, A2, A3, A4, A5, A6, A7
Furthest Distance Division

Page Fault!

Processor Cache

Memory

Disk
Page Fault!

Furthest Distance Division

Processor Cache

Memory

Disk
** Furthest Distance Division **

Page Fault!

Processor Cache

Memory

Disk
**Furthest Distance Division**

Page Fault!

Processor Cache

Memory

Disk

T1

P1

P2

T2

A0

A1

A0

A1

A2

A3

A0

A1

A2

A3

A0

A1

A2

A3

A0

A1

A2

A3

A0

A1

A2

A3

A0

A1

A2

A3

A0

A1

A2

A3

A0

A1

A2

A3

A0

A1

A2

A3
SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY

How should iterations be automatically divided among threads?

Support division parameters set by the programmer
**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>
<tr>
<td>Schedule</td>
<td>Static</td>
<td>All iterations assigned at start of loop</td>
</tr>
<tr>
<td>Step</td>
<td>int</td>
<td>Each thread assigned step number of consecutive iterations</td>
</tr>
</tbody>
</table>
Static (4) Scheduling

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

T1  I^1  I^2  I^3  I^4

T2  I^5  I^6  I^7
Static (1) Scheduling – Close Coupling

T1

I^1 \quad I^3 \quad I^5 \quad I^7

T2

I^2 \quad I^4 \quad I^6
Static (2) Scheduling

Desired locality and load balancing
## 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>
<tr>
<td>Schedule</td>
<td>Static</td>
<td>All iterations assigned at start of loop</td>
</tr>
<tr>
<td>Step</td>
<td>int</td>
<td>Each thread assigned step number of consecutive iterations</td>
</tr>
<tr>
<td>Dynamic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic schedule and reason?
**Static (2) Scheduling – Variable Work**

Local imbalance!
### 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>
| Schedule   | Static/ Dynamic | Static: All iterations assigned at start of loop  
Dynamic: New iterations assigned dynamically based on progress of previous iterations |
| Step       | int        | Each thread assigned step number of consecutive iterations in static and dynamic |
Dynamic (2) Schedule
## Static vs Dynamic

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>No load balancing</td>
<td>Load balancing</td>
</tr>
<tr>
<td>No additional synchronization</td>
<td>After each segment must check a shared data structure leading</td>
</tr>
<tr>
<td>or context switch overhead after each segment</td>
<td>to synchronization and possibly context switch (if multiple threads assigned to processor)</td>
</tr>
</tbody>
</table>
Dynamic (2) Schedule: 3 Threads

T1

\[ I^1, I^2, I^7, I^8 \]

T2

\[ I^3, I^4, I^9, I^{10} \]

T3

\[ I^5, I^6, I^{11}, I^{12} \]

Smaller step size leads to better load balancing

Larger step size leads to less overhead

Variable step size?

T2 does \( I^{10} \) even though T1 and T3 are free!
**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>$I^9$</td>
</tr>
</tbody>
</table>

OMP can automatically implement it and call it based on a loop pragma
GUIDED (1) SCHEDULE

OMP can automatically implement it and call it based on a loop pragma
**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

Thread 0 takes elements at even indices
Thread 1 takes elements at even indices
ALTERNATIVE THREADING

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

- T2
  - Float-List Round
  - Number-List ToString

Diagram showing the flow of operations from T1 and T2 through Float-List Round to Number-List Sum and 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);
}

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
LEVERAGING STATEMENT-BASED CONCURRENTNESS ATTRIBUTES?

T1

Sum or ToString

thread = 0

T2

Number-List Sum

thread = 1

Number-List ToString

T1

T2
**MULTI-THREADED ROUND**

```java
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 ➔

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

T1 → Float-List Round

T2 → Float-List Round

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

T1

Number-List Sum

T2

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 → Number-List Sum
T2 → Number-List Sum

Final Result
Reduce concurrent results
Split without reduction?
Split data among concurrent threads
SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY

T1

Code C

Data (Parts) 1

T2

Code C

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

T1 \rightarrow Code C1 \rightarrow Data (Parts) 1

T2 \rightarrow Code C2 \rightarrow 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>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>

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?
DIFFERENT CODE, SAME DATA CONCURRENCY

T1 → Number-List Sum

T2 → Number-List ToString
LEVERAGING STATEMENT-BASED CONCURRENCY ATTRIBUTES?

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
DIFFERENT CODE, SAME DATA CONCURRENCY

T1 → Sum List

T2 → Print List

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**

Computation Distribution (Number, Matrix Multiplication)

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

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

Non Distributed Single-Thread Process

Non Distributed Multi-Thread Process

T ➔ T
**Single Thread Pipeline**

The diagram represents a single-threaded pipeline with the following steps:

1. **T1**
2. **Float-List Round**
3. **Number-List Sum**
4. **Number-List ToString**

The pipeline starts with input from T1 and proceeds through the steps in sequence.
**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

Number-List Sum

T3

T2 -> Float-List Round

Number-List ToString

T4


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