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)
    Late nineties-
  - MapReduce (Distributed Computing) 2004-
  - 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)

Differences between the two groups?
**Primary/Secondary Distribution Reason**

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

**Primary**
- Remote Service
- Fault Tolerance, Availability
- Collaboration among distributed users
- Aggregation of Distributed Data
- High-Performance: Speedup

**Secondary**
- Speedup
- Remote service, aggregation,...
**Dist. vs Parallel-Dist. Evolution**

- **Non Distributed Existing Single-Thread Program**
- **Distributed Multi-Thread Program**
- **Non Distributed Existing Program**

**Remote Accessible Services (Printers, Desktops)**

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

**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.

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

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

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.
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

Remote Accessible Services (Printers, Desktops)

Replicated Repositories (Files, Databases)

Collaborative Applications (Games, Shared Desktops)

Distributed Sensing (Disaster Prediction)

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.
Fault Tolerance vs Security

Fault Tolerance
- Byzantine Fault Tolerance
  - Block chain
- Non-Byzantine Fault Tolerance
  - Two-Phase Commit
  - Paxos

Usually less autonomy in HPC

Byzantine and general security problems $\propto$ autonomy

Algorithm Manipulation (Adversary)

Message Loss, Computer Failure
**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**
More independent work as goal is to divide common work. Thus faults usually do not propagate. Short-term task can be simply completely or partially restarted.

Faults can be long lived and propagate as goal is to couple processes.
# 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</td>
<td>- A single short-lived process created to perform some computation made multi-</td>
</tr>
<tr>
<td>dependent for resource sharing, user collaboration, fault tolerance.</td>
<td>threaded and/or distributed to increase performance.</td>
</tr>
<tr>
<td>- Include consistency algorithms, possibly in separate threads, to define</td>
<td>- Speedup algorithms that replace logic of existing code, with task decomposition</td>
</tr>
<tr>
<td>dependency that have distributed producer-consumer, observer relationship with</td>
<td>- May involve a central mediating, distributing master code but it can be aware</td>
</tr>
<tr>
<td>existing algorithms</td>
<td>of and creator of specific slave processes</td>
</tr>
<tr>
<td>- May use additional mediating server and other infrastructure code unaware of</td>
<td>- Division of labor among master and slave and slaves an issue</td>
</tr>
<tr>
<td>specific clients.</td>
<td></td>
</tr>
<tr>
<td>- Division of labor between client and infrastructure an issue (centralized vs</td>
<td></td>
</tr>
<tr>
<td>replicated)</td>
<td></td>
</tr>
</tbody>
</table>
EXAMPLE: SERVICE + SPEEDUP

Client

Client\(^1\)

Client\(^2\)

Grader Server

Test-level decomposition

Test-level decomposition

Comp 401 Grader

Slave

Assignment-level decomposition

Comp 533 Grader

Slave

Slave

Master

Slave

Slave

Master

Slave

Comp 533 Grader

Comp 533 Grader

Comp 533 Grader
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

**Parallelizing compilers (Kuck and Kennedy)**
- Halting problem
- Motivates Declarative Abstractions
Adding Declarative 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
**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

Consistency algorithms are state machines

FSA (Finite State Automata)

Ease of programming?
Declarative vs Imperative PDC: 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?

aRegistry.rebind(Server.NAME, aServer)

Distribution?

Server aServer = (Server)
aRegistry.lookup(Server.NAME)
Declarative vs Imperative Concurrency: Competing and Complementing

- **Declarative:** Specify what we want

  Arguably easier to program.

- **Imperative:** Implement what we want

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

  For restricted classes of programs

  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());

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

(Optional) Interpreter

Compiled Program in Concurrency-Aware Imperative Language

Native Language Compiler

omp4j -s Random.java
javac Random.java
java Random
**Imperative Parallel Abstraction?**

```java
{ int nonStatementNonGlobal = 0;
  //omp parallel threadNum(2)
  {nonStatementNonGlobal++;}
}
```

- Statement cannot be compiled individually
- Parallel method can encapsulate statement in method and class
- Interface of class?
- Compile errors at runtime
- Compilation cost at runtime
- Non global (stack) referenced variables outside statement scope and not visible directly to synthesized class
- Need (pre) compiler approach!

```
Thread.parallel(2,"
System.out.println(Math.random());
");
```
Java Precompilation of Parallel

{  
   int nonStatementNonGlobal = 0;  
   //omp parallel threadNum(2)  
   {nonStatementNonGlobal++;}  
}

Attributed statement block encapsulated in Runnable method and Runnable class

Instead of statement run method of new class instance executed threadNum times

Creates a context class that has an instance variable for each non-statement and non-global (stack) variable referenced in statement

Before run method executed non-statement, non-global variables in parent thread stack assigned to corresponding instance variables in context object

Parent thread waits for all threads to finish

Executes next statement

After wait and before next statement instance variables copied back to corresponding statement-non local variables in parent thread stack
```
{ 
    int nonStatementNonGlobal = 0;
    class OMPContext {
        public int local_nonStatementNonGlobal,
    }
    final OMPContext ompContext = new OMPContext();
    ompContext.local_nonStatementNonGlobal = nonStatementNonGlobal;
    final org.omp4j.runtime.IOMPExecutor ompExecutor =
        new org.omp4j.runtime.DynamicExecutor(2);
    for (int ompI = 0; ompI < 2; ompI++) {
        ompExecutor.execute(new Runnable() {
            @Override
            public void run() {
                {ompContext.local_nonStatementNonGlobal++};
            }
        });
    }
    ompExecutor.waitForExecution();
    nonStatementNonGlobal = ompContext.local_nonStatementNonGlobal;
}
```
Relative Expressibility of Two Abstractions

Can $A^1$ implement $A^2$?

- Regular Expressions
- Finite State Automata
- Push Down Automata
- Monitors
- Semaphores
- RMI + Threads
- NIO + Threads

- Theory of Computation
- OS: Thread Coordination
- IPC
**OTHER CONSIDERATIONS**

- **Performance**
- **Ease of Programming (Level)**

Usually involves experimental data and arguments rather than proofs.
```java
{ int nonStatementNonGlobal = 0;
  class OMPContext {
    public int local_nonStatementNonGlobal;
  }
  final OMPContext ompContext = new OMPContext();
  ompContext.local_nonStatementNonGlobal = nonStatementNonGlobal;
  final org.omp4j.runtime.IOMPExecutor ompExecutor =
    new org.omp4j.runtime.DynamicExecutor(2);
  for (int ompI = 0; ompI < 2; ompI++) {
    ompExecutor.execute(new Runnable() {
      @Override
      public void run() {
        {ompContext.local_nonStatementNonGlobal++};
      }
    });
  }
  ompExecutor.waitForExecution();
  nonStatementNonGlobal = ompContext.local_nonStatementNonGlobal;
}
```
Migrating from C to Java: Case Study

- Pthread_create confused with Unix fork
- Java API and uses looked up
- Thread start() confused with run()
- Long debug time!

#pragma omp parallel num_threads(2)
//omp parallel threadNum(2)
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)

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();
}
OMP Declarative Solution

```java
//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

```java
//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] 0.811210363254872
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");
```

Serial + Parallel Program

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

- **T⁰** → Statement
- **Fork(n)** → Statement
- **T¹ → Tⁿ** → Statement
- **T⁰** → Fork(n)
- **T⁰** → Join
- **T⁰** → Statement

- 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$^1..\text{thread}^{n-1}$

Make all thread execute forked statement if it is going to join

Make creating thread wait for termination of thread$^1..\text{thread}^{n-1}$

Implementation requires Statement code to be replicated in both the (in Java, Runnable) code executed by new threads and also in the code executed by the existing thread

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

- **P₀**
  - Statement

- **P₀**
  - childPid = fork()
  - Code executed by both processes

- **P₀**
  - If (0 ! = childPid)
  - join(childPid)

- **P₁**
  - Statement

- **N-Slaves?**
  - System call, returns pid of new process in parent and 0 in child

- **Wait for termination of specified child process**
**UNIX MULTIPLE PROCESS FORK-JOIN**

<table>
<thead>
<tr>
<th>Process</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>for ($i=1; i &lt; n; i++$)</td>
</tr>
<tr>
<td>$P_0$</td>
<td>if ($0 != childPid$)</td>
</tr>
<tr>
<td>$P_0$</td>
<td>$childPid = fork()$</td>
</tr>
<tr>
<td>$P_n$</td>
<td>Statement</td>
</tr>
<tr>
<td>$P_0$</td>
<td>if ($0 != childPid$)</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$join()$</td>
</tr>
</tbody>
</table>

- Procedural (error-prone) code
- Concept of fork-join at least as old as Unix (60s)
- Comparison with OpenMP Parallel Attribute?
- Wait for termination of all child processes
SAME CODE/INSTRUCTION, SAME DATA CONCURRENCY

T1

print random#

T2

Same data
SAME CODE/INSTRUCTION, SAME DATA CONCURRENCY

T1  →  print “hello world”  →  T2

Same or No data
Range of Same Code/Instruction, Same Data Concurrency?

Code C

T1

T2

Same or No data

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 other examples

Same data forces some serialization

Range of realistic applications?

Other patterns?
# Parallelization Classes

<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>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>

Imperative primitives that can be added to these attributes to increase flexibility?
How to integrate alternation with fork-join which requires a single piece of code?

Motivated by Unix?
### UNIX INSPIRATION

<table>
<thead>
<tr>
<th>$P_0$</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>$P_0$</td>
<td>Statement</td>
</tr>
<tr>
<td>$P_n$</td>
<td>Statement</td>
</tr>
<tr>
<td>$P_0$</td>
<td>If (getPid() ! = childPid)</td>
</tr>
<tr>
<td>$P_1$</td>
<td>join()</td>
</tr>
<tr>
<td>$P_0$</td>
<td></td>
</tr>
</tbody>
</table>

Task depends on Id
ID-AWARE IMPERATIVE CODE

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: Different Code Same 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);
        }
    }
}
```

OMP4J_THREAD_NUM is a predefined runtime variable with a different value for each forked thread.
# OpenMP Operations and 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, 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>Operations</th>
<th>Signature</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetThreadNum</td>
<td>( \rightarrow ) int</td>
<td>Returns index of thread created by Parallel pragma</td>
</tr>
<tr>
<td>GetNumThreads</td>
<td>( \rightarrow ) int</td>
<td>Returns number of threads created by Parallel pragma</td>
</tr>
</tbody>
</table>
Different Code, Same Data Concurrency

T1 \rightarrow Code C1

Code C2 \rightarrow T2

Same or No data
SAME CODE, DIFFERENT DATA (PARTS)
CONCURRENCY

T1 -> Float-List Round -> Data (Parts) 1

T2 -> Float-List Round -> Data (Parts) 2

Need 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

What do we need besides thread ID to do the division?
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));
}

float vs Float → Garbage collection

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

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

Parallelized version

processIteration() passed these values and returns Boolean based on whether a thread should process an iteration
MULTI-THREADED ROUND

```java
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

**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 aStart, int aLimit, int aStepSize, int aThreadNum, int aNumThreads) {
    return aNumThreads == 0 ||
           (anIndex % aNumThreads == aThreadNum);
}
```
**Alternate For**

```java
public static void parallelRound(float[] aList) {
    int i = 0;
    // omp parallel threadNum(2)
    for (;;) {
        if (processIteration(i, 0, aList.length, 1,
                                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

Declarative alternative will not have this property
TWO INDEPENDENT PROBLEMS

T1 -> Float-List Round

T2 -> Float-List Round

T1 -> Number-List Sum

T2 -> Number-List ToString
**Pipelined 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
PIPIPIPIPIPELINEPPPELINE: SEPSEPSEPSEPSEPSEPSEPSEPSEPSEPSEPSEPSEPSEPSEPARATE TTHHHTTHHHTHRREEEEDD TTEAMMSMMSMMSMMSMMSMMSMMSMMSMMSMMSMMSMMSMMSMMSMMSMMSMMSMMSS

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, 0, aList.length, 1,
                                OMP4J_THREAD_NUM,
                                OMP4J_NUM_THREADS)) {
                aList[i] = (float) Math.round(aList[i]);
                trace(aList[i]);
            }
        }
    }
    trace("Round Ended:" + Arrays.toString(aList));
}

Separate Thread Team: Step 2

```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));
    }
}
```
PIPEDINED: SEPARATE THREAD TEAMS

T1 -> Float-List Round

T2 -> Float-List Round

T3 -> Number-List Sum

T4 -> Number-List ToString

More resource-efficient solution?
PIPEDLINE: SAME THREAD TEAM

T1 -> Float-List Round

T2 -> Float-List Round

Number-List Sum

Number-List ToString
public static void roundwithProcessIteration(float[] aList) {
  
  trace("Round Started:" + Arrays.toString(aList));
  for (int i = 0; i < aList.length; i++) {
    if (processIteration(i, 0, aList.length, 1,
                         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 parallelRoundAndSumToText(float[] aList) {
    // omp parallel threadNum(2)
    {
        roundWithProcessIteration(aList);
        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]
Thread[pool-1-thread-2,5,main] 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] 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

New declarative construct to fix problem?
public static void parallelRoundAndSumToText(float[] aList) {
    // omp parallel threadNum(2)
    {
        roundWithProcessIteration(aList);
        // 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 active Parallel pragma to finish preceding statement before proceeding to next statement</td>
</tr>
</tbody>
</table>

More declarative constructs for patterns we have seen?
Can we get rid of the if - use of imperative constructs in this method?
Sections can be executed in parallel rather than sequentially.
SEQUENTIAL VS CONCURRENT EXECUTION

Begin

Statement1

Statement2

StatementN

End
Sequential vs Concurrent Execution

CoBegin

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 programming and eager evaluation)
## 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 thread once</td>
</tr>
</tbody>
</table>
**Imperative 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 aStart, int aLimit, int aStepSize, int aThreadNum, int aNumThreads) {
    return aNumThreads == 0 ||
           (anIndex % aNumThreads == aThreadNum);
}
```

Suppose the programmer does not care about how iterations are assigned to threads
Declarative Parallel Round?

```java
public static void parallelRound(float[] aList) {
    // omp parallel threadNum(2)
    {
        for (int i = 0; i < aList.length; i++) {
            if (processIteration(i, 0, aList.length, 1,
                                  OMP4J_THREAD_NUM,
                                  OMP4J_NUM_THREADS)) {
                aList[i] = (float) Math.round(aList[i]);
                trace(aList[i]);
            }
        }
    }
}
```

Can OMP automatically implement processIteration and call it based on a loop pragma?

processIteration() is problem independent but dependent on 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) {
    // 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 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 properties of counter-controlled loops</td>
</tr>
</tbody>
</table>
SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY

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

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

How should iterations be automatically divided among threads?
Our 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 aStart, int aLimit, int aStepSize, int aThreadNum, int aNumThreads) {
    return aNumThreads == 0 ||
            (anIndex % aNumThreads == aThreadNum);
}
```
CLOSEST DISTANCE APPROACH

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

T2 \rightarrow \text{Float-List Round} \rightarrow \text{Diagram}
Comparison based on locality arguments?
Locality Granularities

T1
P1
A0
A1
P2
A0
A1
A2
A3
T2
A0
A1
A2
A3
P1
A0
A1
Processor Cache
Memory (1 Page)
Disk
CONFLICTING REPLICA CHANGES

Processor Cache
Memory (1 Page)
Disk
CONFLICTING REPLICABLE CHANGES

True sharing conflict

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

Processor Cache

Memory (1 Page)

Disk

T1

P1

A0

A1

T2

P2

A0

A1

A0

A1

A2

A3

A0

A1

A2

A3

A0

A1

A2

A3

A4

A5

A6

A7
**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.

- **Processor Cache**
  - A0
  - A1

- **Memory (1 Page)**
  - A0
  - A1
  - A2
  - A3

- **Disk**
  - A0
  - A1
  - A2
  - A3
  - A4
  - A5
  - A6
  - A7
Furthest Distance Division (T1’s Page)
Furthest Distance Division (T2’s Page)

Page Fault!

T1
P1

A0
A1

A4
A5

A4
A5

A0
A1
A2
A3
A4
A5
A6
A7

Processor Cache
Memory (1 Page)
Disk
Furthest Distance Division (T1’s Page)

Page Fault!

T1

P1

T2

A0
A1

A0
A1
A2
A3

A0
A1
A2
A3
A4
A5
A6
A7

Processor Cache

Memory (1 Page)

Disk
SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY

How should iterations be automatically divided among threads?

Support division parameters set by the programmer