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)
- Computation Distribution (Number, Matrix Multiplication)

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

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

- **Secondary**
  - Speedup
  - 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)**

**Non Distributed Existing Program**

**Distributed Program**

**Non Distributed Existing Program**

**Distributed Program**

**Non Distributed Multi-Thread Program**

**Distributed Program**

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

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

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.
**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)

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
  - Algorithm Manipulation (Adversary)
- Non-Byzantine Fault Tolerance
  - Two-Phase Commit
  - Paxos

Byzantine and general security problems $\propto$ autonomy

Usually less autonomy in HPC
**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**
**Fault Tolerance**

Faults can be long lived and propagate as goal is to couple processes.

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.

- Remotely Accessible Services (Printers, Desktops)
- Replicated Repositories (Files, Databases)
- Collaborative Applications (Games, Shared Desktops)
- Distributed Sensing (Disaster Prediction)
- Computation Distribution (Number, Matrix Multiplication)
## 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 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 distributed producer-consumer, observer 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 among master and slave and slave and slaves an issue</td>
</tr>
</tbody>
</table>
**Example: Service + Speedup**

- **Client** 1
- **Client** 2
- **Comp 533 Grader**
- **Comp 401 Grader**
- **Grader Server**
- **Master**
- **Slave**

**Assignment-level decomposition**

**Test-level decomposition**
ABSTRACTIONS

- 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

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)
- Computation Distribution (Number, Matrix Multiplication)

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

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

Declarative Abstractions
Declarative vs Imperative: Complementing

Declarative: Specify what we want

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

Imperative: Implement what we want

```
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

```
Type declaration
```

Loop

```
Loop
```

Locality relevant to PDC abstractions - later

```
Locality relevant to PDC abstractions - later
```
**Declarative vs Imperative: Competing**

- **Declarative**: Specify what we want
  
  \[(0|1)^*1\]

- **Imperative**: Implement what we want

- **Consistency algorithms are state machines**

- **FSA (Finite State Automata)**

- **Ease of programming?**

- **Regular expression**
**Declarative vs Imperative PDC: Competing and Complementing**

Declarative: Specify what we want

Imperative: Implement what we want

Concurrency?

Distribution?

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

aRegistry.rebind(Server.NAME, aServer)

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

For restricted classes of programs

```
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();
PARALLEL RANDOM

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

Desired I/O

0.6455074599676613
0.14361454218773773

Declarative Concurrency Specification Called a Pragma or Directive

//omp parallel threadNum(2)

Imperative Concurrency Unaware Code

Declarative Concurrency Aware 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-COMPILE 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++;}
}

Thread.parallel(2,"
```

```
System.out.println(Math.random());
"
```
Java Precompilation of Parallel

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

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.

After wait and before next statement instance variables copied back to corresponding statement-non local variables in parent thread stack.

Attributed statement block encapsulated in Runnable method and Runnable class.

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

Parent thread waits for all threads to finish.

Executes next statement.
Assuming non-stack variables are shared by all threads, so a single context for all threads.
**Relative Expressibility of Two Abstractions**

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

- Regular Expressions
- Finite State Automata
- Push Down Automata
- Finite State 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
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)

Imperative
Declarative
TRACING PARALLEL RANDOM

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

```java
//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**

```java
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.

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

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();
}
```
**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

Serial + Parallel Program

Which thread(s) will print “Forking” and “Joined”?  

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

trace("Forking");

```

```
trace("Joined");
```
**Abstract Fork-Join**

- \( T^0 \) \rightarrow \text{Statement}
- \( T^0 \) \rightarrow \text{Fork}(n)
- \( T^1 \) \rightarrow \text{Statement}
- \( T^n \) \rightarrow \text{Statement}
- \( T^0 \) \rightarrow \text{Join}
- \( T^0 \) \rightarrow \text{Statement}

- create thread\(^1..\text{thread}\)^n
- Make each thread execute forked statement
- Make creating thread wait for termination of thread\(^1..\text{thread}\)^n
- More efficient thread creation?
EQUIVALENT, MORE EFFICIENT ABSTRACT FORK-JOIN

- **T⁰** → Statement
- **T⁰** → fork(n)
- **T⁰** → Statement
- **Tⁿ⁻¹** → Statement
- **T⁰** → Join
- **T⁰** → Statement;

- create thread¹..threadⁿ⁻¹
- Make all thread execute forked statement if it is going to join
- Make creating thread wait for termination of thread¹..thread⁻¹
  - Implementation requires Statement code to be replicated in both the (in Java, Runnable) code executed by new threads and also in the code execute by the existing thread
- Similar to Unix Fork-Join?
UNIX SINGLE PROCESS FORK-JOIN

- **P₀**
  - Statement
  - `childPid = fork()`
  - Statement
  - If (0 != childPid)
  - `join(childPid)`

- **P₀**
  - System call, returns pid of new process in parent and 0 in child

- **P₀**
  - Code executed by both processes

- **P₁**
  - **P₀**
  - **P₁**
  - **P₀**
  - **P₁**

- **N-Slaves?**

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

- for (i=1; i < n; i++)
  - if (0 != childPid)
  - childPid = fork()
  - Statement
  - if (0 != childPid)
  - join()

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

T1 and T2 both print random number, which are the same data due to concurrency.
SAME CODE/INSTRUCTION, SAME DATA CONCURRENCY

print "hello world"

T1

T2

Same or No data
RANGE OF SAME CODE/INSTRUCTION, SAME DATA CONCURRENCY?

Code C

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
# 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>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?
DIFFERENT CODE, SAME DATA CONCURRENCY

How to integrate alternation with fork-join which requires a single piece of code?

Motivated by Unix?
UNIX INSPIRATION

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: 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>➔ 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}

T2 \rightarrow \text{Code C2}
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.
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 $\rightarrow$ Garbage collection

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

Parallelized version

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

Body of loop is executed based on thread number, #threads, and loop parameters
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);
}
```
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
PIPPLED: 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, 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 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 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));
        }
    }
}

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>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>
<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?
public static void sectionRoundAndSumToText(float[] aList) {
  // omp parallel threadNum(2)
  {
    roundWithProcessIteration(aList);
    // omp barrier
    // omp sections
    {
      // omp section
      trace("Sum of rounded:" + sum(aList));
      // omp section
      trace("ToText of rounded:" + toText(aList));
    }
  }
}

Sections can be executed in parallel rather than sequentially.
**SEQUENTIAL vs CONCURRENT EXECUTION**

```
Begin

T1 → Statement1
T1 → Statement2
T1 → 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
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, 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?
Declarative Parallel Round

```java
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}

T2 \rightarrow \text{Float-List Round}
**Furthest Distance Division**

Comparison based on locality arguments?
LOCALITY GRANULARITIES

Processor Cache
Memory (1 Page)
Disk

T1
P1
A0
A1

P2
A0
A1
A2
A3

T2
A0
A1
A2
A3

A0
A1
A2
A3
A4
A5
A6
A7
CONFLICTING REPLICATING CHANGES

T1

P1

A0
A1

A0
A1
A2
A3

A0
A1
A2
A3
A4
A5
A6
A7

P2

T2

Processor Cache

Memory (1 Page)

Disk
CONFLICTING REPLICA 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
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

Memory (1 Page)

Disk
Furthest Distance Division (T1’s Page)
Furthest Distance Division (T2’s Page)

Page Fault!

Processor Cache

Memory (1 Page)

Disk
Furthest Distance Division (T1’s Page)
SAME CODE, DIFFERENT DATA (PARTS) CONCURRENCY

How should iterations be automatically divided among threads?

Support division parameters set by the programmer
MEMORY ACCESS MODEL

- Data moves from disk to main memory to per-processor cache.
- Disk is larger and slower than main memory which is larger and slower than a cache.
- Main memory is divided into units called pages.
- A cache is divided into units called cache lines or blocks.
- If data accessed by a process is not in main memory, an event called a page fault occurs, which is processed by writing some existing page in main memory to disk and loading the accessed data from disk to the freed saved page.
- Parts of main memory pages accessed by a processor are loaded into cache lines and accessed from cache lines subsequently if they are loaded in cache lines.
- Programs run faster if they have fewer transfers between disk and main memory and main memory and cache.
CACHE CONFLICTS

- Two processors can load cache lines holding the same region of main memory.
- Concurrent **writes** to cache lines replicating the same memory blocks results in execution of a cache consistency algorithm a la two phase commit and atomic broadcast.
- False sharing occurs from concurrent writes to replicas of the same memory region, and result in unnecessary consistency algorithm executions.
- True sharing occurs from concurrent writes to same words in cache replicas of the same memory block— the overhead of consistency algorithms is necessary now.
- Reading from cache lines does not cause these algorithms to be executed.
- Fewer execution of concurrent consistency algorithms makes program faster.
CLOSE VS. FAR DISTANCE SHARING

- Far distance sharing can result in threads concurrently accessing different pages, and thus potentially causing more page faults depending on size of memory compared to size of data accessed.
- Close distance sharing can result in more false sharing conflicts as threads access different addresses in cache blocks.
- Moral: Need threads to work with on same pages but not write to overlapping loaded cache lines.
- Reading from overlapping cache lines does not cause any problem.
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 properties of counter-controlled loops</td>
</tr>
<tr>
<td>Schedule</td>
<td>(Static, Step)</td>
<td>All iterations assigned at start of loop</td>
</tr>
<tr>
<td>Step Parameter</td>
<td>int</td>
<td>Each thread assigned step number of consecutive iterations (chunk) at a time</td>
</tr>
</tbody>
</table>
Static (5) Scheduling

**T1**

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

**T2**

I^6 | I^7

Bad load balancing
Static (1) Scheduling – Close Coupling

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

T1
I^1  I^3  I^5  I^7

T2
I^2  I^4  I^6

Ever reasonable given false sharing?

Adjacent reads do not conflict
Static (2) Scheduling

Balancing locality and load balancing
Static (2) Scheduling – Variable Work

T2 is Free and T1 is busy
Load 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 properties of counter-controlled loops</td>
</tr>
</tbody>
</table>
| Schedule     | (Static/ Dynamic, Step) | Static: All iterations assigned at start of loop  
|              |                      | Dynamic: New iterations assigned dynamically based on progress of previous iterations |
| Step Parameter | int                 | Each thread assigned step number of consecutive iterations (chunk) in static and dynamic |
Dynamic (2) Schedule
**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>
| Schedule         | (Static/Dynamic, int) | Static: All iterations assigned at start of loop  
                     Dynamic: New iterations assigned dynamically based on progress of previous iterations |
| Step Parameter   | int            | Each thread assigned step number of consecutive iterations (chunk) in static and dynamic |

Why not always use dynamic?
### Static vs Dynamic: Variable Iteration Cost

<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 or context switch overhead after each</td>
<td>- After each segment must check a shared data structure leading to</td>
</tr>
<tr>
<td>segment</td>
<td>synchronization and possibly context switch (if multiple threads</td>
</tr>
<tr>
<td></td>
<td>assigned to processor)</td>
</tr>
</tbody>
</table>

**Static**

- No load balancing
- No additional synchronization or context switch overhead after each segment

**Dynamic**

+ Load balancing
- After each segment must check a shared data structure leading to synchronization and possibly context switch (if multiple threads assigned to processor)
Dynamic (2) Schedule: 3 Threads

T1

T2

T3

Solution: smaller step/chunk size?

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!
## 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>
| Schedule         | (Static/ Dynamic/ Guided, Step) | Static: All iterations assigned at start of loop  
                           Dynamic: New iterations assigned dynamically based on progress of previous iterations  
                           Guided: Step size changes dynamically based on remaining iterations after each assignment |
| Step Parameter   | int              | Each thread assigned step number of consecutive iterations (chunk) in static and dynamic and at least step number in guided |
**GUIDED (1) SCHEDULE**

Chunk size decreases dynamically $\propto$ Remaining Iterations/#Threads

Chunk Size at least specified size

<table>
<thead>
<tr>
<th>$I^1$</th>
<th>$I^2$</th>
<th>$I^3$</th>
<th>$I^4$</th>
<th>$I^5$</th>
<th>$I^6$</th>
<th>$I^7$</th>
<th>$I^8$</th>
<th>$I^9$</th>
<th>$I^{10}$</th>
<th>$I^{11}$</th>
<th>$I^{12}$</th>
</tr>
</thead>
</table>

T1

12/3

$I^1$ $I^2$ $I^3$ $I^4$

8/3 3/3

T2

$I^5$ $I^6$ $I^7$ $I^{10}$ $I^{12}$

5/3

T3

$I^8$ $I^9$ $I^{11}$
**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>
| Schedule     | (Static/ Dynamic/ Guided, Step) | Static: All iterations assigned at start of loop  
Dynamic: New iterations assigned dynamically based on progress of previous iterations  
Guides: Step size changes dynamically based on remaining iterations after each assignment |
| Step Parameter | int           | Each thread assigned step number of consecutive iterations (chunk) in static and dynamic and at least step number in guided                  |
Dynamic/Guided Application

- Grader assignments and tests
- Irregular algorithms in general
public static void parallelRound(float[] aList) {
    // omp parallel for threadNum(2) schedule (static, 128)
    for (int i = 0; i < aList.length; i++) {
        aList[i] = (float) Math.round(aList[i]);
        trace(aList[i]);
    }
}
public static void parallelSum(float[] aList) {
    float retVal = (float) 0.0;
    // omp parallel for threadNum(2)
    for (int i = 0; i < aList.length; i++) {
        retVal += aList[i];
    }
    return retVal;
}
**Parallel Atomic Sum**

```java
class ParallelAtomicSum {
    public static void parallelSum(float[] aList) {
        float retVal = (float) 0.0;
        // omp parallel for threadNum(2)
        for (int i = 0; i < aList.length; i++) {
            // omp critical
            retVal += aList[i];
        }
        return retVal;
    }
}
```

Computations are serialized!

Alternative algorithm?
Reduction: Divide and Conquer

\[ 2 + 3 + 4 + 6 \]

\[ (2 + 3 ) \]
\[ (4 + 6 ) \]

\[ (5 + 10 ) \]

Partial reduction
Final reduction
Multi-Level Parallel Reduction of List: Sum

Final reduction: result computed from thread-specific private variables by one or more threads

Reduced sum in private variable of thread

Partial Reduction: Partial solution of problem, reducing it

Replicate a reducible shared variable (but not its value) for wait-free / lockless coordination!
**Multi-Level Parallel Reduction of List: **

**ToString**

- **String**
- **Number-List Sum**
- **String1**
- **String2**
- **Number-List ToString**
- **Number-List ToString**
- **A1**
- **A2**
- **A3**
- **A4**

- **Number (abstract) + is commutative and associative**
- **String + is not commutative and associative**
public static void parallelSum(float[] aList) {
    float retVal = 0.0;
    // omp parallel for threadNum(2)
    for (int i = 0; i < aList.length; i++) {
        // omp critical
        retVal += aList[i];
    }
    return retVal;
}
2-Level Sum Reduction with Parallel For

```java
public static void parallelSum(float[] aList) {
    float retVal = 0.0f;
    float[] aSums = {0.0f, 0.0f};
    // omp parallel for threadNum(2))
    for (int i = 0; i < aList.length; i++) {
        int aThreadNum = 0;
        aSums[OMP4J_THREAD_NUM] += aList[i];
    }
    for (float aSum:aSums) {
        retVal += aSum;
    }
    return retVal;
}
```

Cache issues in allocation of aSums?

False sharing as all threads write to adjacent array slots!
public static void parallelSum(float[] aList) {
    float retVal = 0.0f;
    // omp parallel private (aSum)
    {
        float aSum = 0.0f; // private to each thread
        for (int i = 0; i < aList.length; i++) {
            if (processIteration(i, 0, aList.length, 1,
                OMP4J_THREAD_NUM,
                OMP4J_NUM_THREADS)) {
                aSum += aList[i]; // replaced retVal += aList[i]
            }
        }
        // omp critical
        retVal += aSum; // second reduction
    }
    return retVal;
}
# 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>
<tr>
<td>Private/Shared</td>
<td>String</td>
<td>Variable declared in statement private or shared by threads that execute it</td>
</tr>
</tbody>
</table>
public static void parallelSum(float[] aList) {
    float retVal = 0.0f;
    // omp parallel private (aSum)
    {
        float aSum = 0.0f; // private to each thread
        for (int i = 0; i < aList.length; i++) {
            if (processIteration(i, 0, aList.length, 1, OMP4J_THREAD_NUM, OMP4J_NUM_THREADS)) {
                aSum += aList[i]; // replaced retVal += aList[i]
            }
        }
        // omp critical
        retVal += aSum; // second reduction
    }
    return retVal;
}
public static void parallelSum(float[] aList) {
    float retVal = 1.0f;
    // omp parallel private (aSum)
    {
        float aProd = 0.0f; // private to each thread
        for (int i = 0; i < aList.length; i++) {
            if (processIteration(i, 0, aList.length, 1,
                                  OMP4J_THREAD_NUM,
                                  OMP4J_NUM_THREADS)) {
                aProd *= aList[i];
            }
        }
        // omp critical
        retVal *= aProd; // second reduction
    }
    return retVal;
}
**Change How For Multi-Level Reduction?**

```java
public static void parallelSum(float[] aList) {
    float retVal = 0.0;
    // omp parallel for threadNum(2) reduction(+:retVal)
    for (int i = 0; i < aList.length; i++) {
        retVal += aList[i];
    }
    return retVal;
}
```

- Private variable created before attributed statement entered for reduction variable (retVal)
- All occurrences of reduction variable replaced with private variable in for loop
- Initialization of private variable?
- Private variable initialized to value dependent on reduction operator (+)
- Reduction operator (+) repeatedly applied to private variables and resulting reduced value stored in reduction variable
Parallel Word Count

Words: [the, a, an, the, a]
Counts: {the=2, a=2, an=1}

```java
public static void add (Map<String, Integer> aMap, String aKey, Integer aValue) {
    Integer anOriginalValue = aMap.get(aKey);
    Integer aNewValue = aValue +
        (anOriginalValue == null?0:anOriginalValue);
    aMap.put(aKey, aNewValue);
}

public static Map<String, Integer> wordCount(String[] aWords) {
    Map<String, Integer> aWordCount = new HashMap();
    for (int i = 0; i < aWords.length; i++) {
        add (aWordCount, aWords[i], 1);
    }
    return aWordCount;
}
```

Desired I/O

Integer vs int space issues?

Each Integer addition creates a new object!

Reducible for parallelism?

Both operands of reducible add should be of the same type
PARALLEL WORD COUNT

Words: [the, a, an, the, a]
Counts: {the=2, a=2, an=1}

Before for loop, private variable initialization

```java
public static void mapPlus(Map<String, Integer> aMap,
Map<String, Integer> anAddition) {
    for (String aKey: anAddition.keySet()) {
        add(aMap, aKey, anAddition.get(aKey));
    }
}
```

```java
public static Map<String, Integer> reducableWordCount(String[] aWords) {
    Map<String, Integer> aWordCount = new HashMap();
    for (int i = 0; i < aWords.length; i++) {
        Map<String, Integer> anAddedValue = new HashMap();
        anAddedValue.put(aWords[i], 1);
        mapPlus(aWordCount, anAddedValue);
    }
    return aWordCount;
}
```

After for loop, second level reduction carried out by calling mapPlus with the shared variable being passed in the position of omp_out and private variable being called in position of omp_in
// omp declare reduction
// (<Operation Name>:<Operand Type?:omp_out=<Method>(<omp_out or omp_in>,<omp_out or omp_in))
// initializer(omp_priv= <Initializing Expression>)

Before the for loop each private variable is assigned
<Initializing Expression>

<Operation Name> is used in the for attribute set

The <Method> to which it is bound called after the for

Parameter mp_out is replaced with the for reduction variable and mp_in is replaced with the private variable created from it
## OpenMP Attributes Qualifying For

<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>
| Schedule           | Static/ Dynamic/ Guided, Step | Static: All iterations assigned at start of loop  
Dynamic: New iterations assigned dynamically based on progress of previous iterations  
Guides: Step size changes dynamically based on remaining iterations after each assignment |
| Step Parameter     | int                 | Each thread assigned step number of consecutive iterations (chunk) in static and dynamic and at least step number in guided               |
| Reduction          | String: String      | Multi-level reduction using the operator identified by 1\textsuperscript{st} parameter of value stored by variable identified by 2\textsuperscript{nd} parameter |
## 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>

**How many threads to create?**
DEFAULT VALUES

- **NumberOfThreads**
  - `Runtime.getRuntime().availableProcessors()`

- **Schedule**
  - Static

- **Step Size**
  - Static
    - Number of iterations/number of threads
    - Each thread gets one chunk
  - Dynamic/Guided
    - 1
How Many Threads?

Runtime.getRuntime().availableProcessors()
Multiple Applications

P1  P2  P3  P4

A1  T11  T12  T13  T14  T15
A2  T21  T22  T23
A3  T31  T32

Multi-Process Scheduling Policy?
**Common Global Thread Queue**

Threads when time quantum allocated for current threads expires (assuming no blocking)
Does it matter how threads are mapped in the second context switch?
SECOND THREAD SWITCH: PROCESSOR AGNOSTIC

Cache blocks of P1 and P2 may still hold data of $T^{14}$ and $T^{15}$ when they were scheduled last on them.
SECOND CONTEXT SWITCH: PROCESSOR AFFINITY

Affinity-based: Schedule threads on processors on which they were scheduled earlier
**Processor Agnostic vs Affinity-based Scheduling**

- Affinity-based can increase performance, if no load does not get unbalanced, as in our scheme.
- Other scheduling schemes that do affinity based scheduling (Linux) need a load balancing step.
Applications with more threads get more processor time

Reason: Single queue

Fair?

Common Global Thread Queue
APPLICATION-AND THREAD-LEVEL QUEUE
**Thread Switch**

- A^2
- A^3

- T^{11}
- T^{12}
- T^{13}
- T^{14}
- T^{15}

- T^{21}
- T^{22}
- T^{23}

- T^{31}
- T^{32}
Threads of next application(s) can use processors not used by current application
**Single-Level vs Multi-Level Queue**

- **Single-level**
  - Single queue of threads associated with a single time quantum $t$.
  - After $t$ units, thread switch.
  - Applications with more threads get more processor time.

- **Multi-level**
  - Level 1 queue of processes/applications with time quantum $T$.
  - Level 2 queue with threads of current application with time quantum.
  - Every $t$ units threads in level-2 queue switched.
  - Every $T$ units processes in level-1 queue switched, which results in new processes' threads to be loaded in second level queue and $n$ threads of next processes’ in level 1 queue if $(n = \# \text{ of processors} - \# \text{threads of current process}) > 0$.
  - The total time a process gets depends on its position in the queue, if it is behind a thread-stingy process, it gets time during its time quantum and also possibly time quantum of processes’ in front of it.

- Can use processor agnostic or affinity based scheduling in both
Better Use of Cache?

Divide processors rather than time among applications
Thread spinning on test and set does no work during its time quantum as thread holding the lock is in the queue.
TIME VS SPACE MULTIPLEXING

Space scheduling
- $\#\text{processors} - \#\text{applications}$ put in application queue.
- Processors assigned to active applications with some applications getting more than others if they do not divide evenly.
- An active application continues to use its processors until it completes.
- Threads of an application use time multiplexing to share its applications

Throughput
- Better use of cache.

Response time
- Applications in queue must wait until current applications finish
UNCOORDINATED ATOMICITY-AWARE SCHEDULING

Thread spinning on test and set does no work during its time quantum as thread holding the lock is in the queue.
COORDINATED ATOMICITY-AWARE SCHEDULING

Threads in critical region given more precedence and those spinning given less precedence.
COORDINATED VS UNCOORDINATED ATOMICITY-AWARE SCHEDULING

- Uncoordinated
  - Scheduler and application are independent

- Coordinated
  - Application tells scheduler if it is spinning or in critical section.
  - If language/library abstractions support atomicity then they can automatically inform the scheduler which may be implemented by the language/library
Should the scheduler and application coordinate on number of threads?

Can an application go faster if it has fewer threads than it can use?

Given N processors, how many threads should be scheduled at one time?
SPEEDUP: THREAD VS. PROCESSORS

Why is speedup not linear when there are enough processors?

Thread coordination overhead – cost of splitting and combining work

One active application with n threads or two applications with n/2 processors?

Why does speedup go down when #threads > #processors?

Cache invalidation, lack of coordination, thread switching

Keep # of threads = # processors

COORDINATED VS UNCOORDINATED THREAD-CONTROLLING SCHEDULING

- **Uncoordinated**
  - Application gets no information about number of threads it should create

- **Coordinated**
  - The scheduler keeps \# of threads = \# of processors.
  - Application gets callback about desired number of threads.
  - In dynamic scheduling, it can react to callback by terminating some threads when they finish their current chunk.
  - Moral: dynamic scheduling is better in multi-application computers!

- Two applications running \(n/2\) threads will go faster than one application running \(n\) threads as speedup decreases with number of threads
DIST. VS PARALLEL-DIST. EVOLUTION

Remote 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 Program

Distributed Program

Non Distributed Existing Program

Distributed Program

Non Distributed Multi-Thread Program

Distributed Program

Distributed Program
PARALLEL WORD COUNT

Words:[the, a the, an, an, the, a]
Counts:{the=3, a=2, an=1}

```java
// omp declare reduction \
// (mapAdd:java.util.Map<String, Integer>:omp_out=mapPlus(omp_out,omp_in)) \
// initializer(omp_priv= new java.util.HashMap())
public static void mapPlus (Map<String, Integer> aMap,
    Map<String, Integer> anAddition) {
    for (String aKey:anAddition.keySet()) {
        add(aMap, aKey, anAddition.get(aKey));
    }
}
public static Map<String, Integer> reducableWordCount(String[] aWords) {
    Map<String, Integer> aWordCount = new HashMap();
    // omp parallel for threadNum(aNumThreads) reduction(mapAdd:aWordCount)
    for (int i = 0; i < aWords.length; i++) {
        Map<String, Integer> anAddedValue = new HashMap();
        anAddedValue.put(aWords[i], 1);
        mapPlus(aWordCount, anAddedValue);
    }
    return aWordCount;
}
```
Distributing OpenMP?

Need a software implementation of shared memory

The Fork-Join model requires input and data to start and end at the for loop one process/thread

Inconsistent with big data, where all data may not fit in the memory of one process and even it does, the process becomes a bottleneck

MapReduce provides a competitor for Fork-Join loop-based reduction (not sections, non-loop parallel, ...
DISTRIBUTED OPENMP ARCHITECTURE
MapReduce

Input

Inter-Process Input Splitting

Intra-Process Input Splitting

Mapping

Intra-Process Shuffling

Partial Reduction

Inter-Process Shuffling

Final Reduction

Output

Mapping and Optionally Partially Reducing Processes

Reducing Processes
**MapReduce Key Ideas**

- Assume data fits in a set of input files rather than a single process.
- Work split to different processes by giving them different portions of the files without first loading them into one process.
- Data-based division rather than loop-based division.
- Work combined by multiple processes who directly write to output files rather than sending them to one process.
**MapReduce Data-based Division**

- **External Data Structure**: File Text lines
- **In Memory Data Structure**: <Key, Value> pairs (Map)
- Can simulate any collection: array, list, set, matrix
OpenMP Reduction vs MapReduce Model

OpenMP:
- A binary commutative and associative reduce operation known to OpenMP
- A counter-controlled loop known to OpenMP
- Output variable in forking thread known to OpenMP that is an operand to operation on each iteration.
- An expression computed by each loop iteration that is the other operand to operation in iteration

MapReduce:
- An n-ary reduce function that reduces a list of values bound to a key to a single value using an unknown commutative and associative binary function.
- Input text files known to MapReduce
- A map function known to MapReduce that converts each input line to (Key, Value) pairs processed by a reduce function.
## OpenMP Reduction vs MapReduce

<table>
<thead>
<tr>
<th>OpenMP</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work divided among different threads/processes consists of reduction</td>
<td>Work divided among different threads/processes consists of mapping and reduction.</td>
</tr>
<tr>
<td>OpenMP splits work to multiple threads/processes based on loop index</td>
<td>Splits application lines among different mappers processes.</td>
</tr>
<tr>
<td>Feeds the results of child/ calls the reduction function to combines the reduced results of child threads/processes in one thread</td>
<td>Splits mapper-process lines among different mapper-threads, which call the mapper function</td>
</tr>
<tr>
<td></td>
<td>Optionally feeds the results of mapper-threads among reduction threads of mapper process, which call the reduction function.</td>
</tr>
<tr>
<td></td>
<td>Splits the results of reduction threads of mapper-process among different reduction processes, which are responsible for combining values of non overlapping sets of keys.</td>
</tr>
<tr>
<td></td>
<td>Split keys received by reduction process among different reduction threads, which directly write the output for each key</td>
</tr>
</tbody>
</table>
MAP REDUCE ARCHITECTURE: WORD COUNT

1, (the a)
2, (a)

the, 1
a, 1
a, 1

the, 1
the, 1
a, 2

the, 2

the, 1

a, 2

the, 2
a, 2
an, 1

an, 1
an, 1
an, 1

3, the
4, an
public class WordCount {
    public static class Map extends MapReduceBase {
        implements Mapper<LongWritable, Text, Text, IntWritable> {
            private final static IntWritable one = new IntWritable(1);
            private Text word = new Text();
            public void map(LongWritable key, Text value,
                OutputCollector<Text, IntWritable> output, Reporter reporter)
                throws IOException {
                String line = value.toString();
                StringTokenizer tokenizer = new StringTokenizer(line);
                while (tokenizer.hasMoreTokens()) {
                    word.set(tokenizer.nextToken());
                    output.collect(word, one);
                }
            }
        }
    }
}
public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
Map Reduce: Min Temp
MAP REDUCE: SUM

1, 5
2, 3
3, 3

S, 5
S, 3
S, 3

S, 11

S, 11
S, 3

S, 14

S, 14

S, 2
S, 1

4 (2 1)
**Map Reduce: ROUND**

1, 5.2
2, 3.3
3, 3.4

1.1, 5
2.1, 3
3.1, 3

1.1, 5
2.1, 3
3.1, 3

4.1, 2
4.2, 1

4 (2.2 1.3)

1.1, 5
2.1, 3
3.1, 3
4.1, 2
4.2, 1
MAP REDUCE: ROUND + SUM

1, 5.2
2, 3.3
3, 3.4

S, 5
S, 3
S, 3
S, 11
S, 11
S, 3
S, 11
S, 3
S, 14
S, 14

4 (2.2 1.3)
Value reduction must involve a commutative and associative operation!
**Map Reduce: Common Friends**

1, A>B C
2, C>A B

AB, BC
AC, BC
AC, AB
BC, AB

AB, ACD
BC, ACD
BD, ACD

3, B>A C D

AB, BC
AC, B
BC, AB

AB, B C
AB, ACD

AB, C

AB, C

AB, ACD
BC, ACD
AC, B
**Map/Reduce Functions**

- **Mapper Function:**
  - Infrastructure can call map in parallel on different input lines
  - Gets as input a line of text (value parameter) and its offset in the input file (key), which is like the index in a loop
  - Produces as output key-value pairs
  - Each iteration free to determine how input is transformed to output.

- **Reduce Functions:**
  - Infrastructure can call reduce function in parallel with different keys
  - Takes as input a key and a list of map values produced for that key by different map functions, which correspond to local result variables in OpenMP
  - Produces as output a single result reduction value for that key.
**Map Reduce Architecture**

- **Mapper Node and Phase:**
  - Executes in parallel the map function on lines assigned to it.
  - Feeds the output <keyvalue> pairs to the local reduce function, as an optional step.
  - Sends each output <Key, Value> Pairs produced from map and the optional reduce phase to a reducer that handles it.

- **Reducer Node:**
  - Bound with some target set of keys $K^1$ to $K^n$
  - Different reducers handle different sets of keys
  - Execute the reduce function on data received from remote mappers.
  - Output of this function written to one or more distributed files.
**Map Reduce Infrastructure**

- Automation provided based on Map data structure rather than loop
  - Can simulate any collection, bean, or array
- Provides a distributed file system in which different parts of the output can be placed from different locations
- Automatically determines the number and location of mappers and reducers based on input file distribution.
  - Process goes to the data rather than vice versa.
- Automatically splits the input data among the mappers.
- Automatically splits keys (as opposed to iterations) among the reducers.
- Computed results can be combined by multiple reducers
- Reads input data assigned to a mapper and feeds it into a programmer-defined map method.
- Transmits each <Key, Value> output by this method to a local reducer and then a remote reducer that handles it.
- Calls a programmer-defined reduce method in a remote reducer and supplies it with a (key, value) sequence.
- Combines the output of the reduce methods.
ASSIGNMENT

Like MapReduce distributes computation among multiple processes and threads without assuming distributes shared memory.

Like MapReduce splits work based on data rather than a loop.

Like OpenMP assumes a single master process is source and sink of input and output data, respectively.

Number of slave threads and processed input is determined through interactive input processed by an MVC architecture.

Uses a bounded buffer of (Key, Value) pairs to distribute work to slave threads of master process a la dynamic scheduling.

A slave thread can delegate work to a free client registered dynamically with the server.
Parallel-Distributed Assignment

Words: [the, a, a, the, an]
Counts: {the=2, a=2, an=1}

Client

R

a, 2

a, 1

S

a, 1

the, 1

day, 1

a, 1

Server

M

the, 2

a, 2

an, 1

S

da, 1

day, 1

the, 1

an, 1

null, null

null, null
More on MapReduce Details

//Run the application:

$ bin/hadoop jar /usr/joe/wordcount.jar mapreduce.WordCount /usr/joe/wordcount/input /usr/joe/wordcount/output

//Output:

$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-00000
the 2
an 1
a 2
**MAP CLASS**

**Usage**
Assuming HADOOP_HOME is the root of the installation and HADOOP_VERSION is the Hadoop version installed, compile WordCount.java and create a jar:

$$
$ \text{mkdir wordcount\_classes} \\
$ \text{javac -classpath {}\}/hadoop-
\text{HADOOP\_VERSION\}-core.jar -d wordcount\_classes}
WordCount.java \\
$ \text{jar -cvf /usr/joe/wordcount.jar -C wordcount\_classes/} .
$$
MAP CLASS

Assuming that:

/usr/joe/wordcount/input - input directory in HDFS
/usr/joe/wordcount/output - output directory in HDFS

Sample text-files as input:

$ bin/hadoop dfs -ls /usr/joe/wordcount/input/
/usr/joe/wordcount/input/file01
/usr/joe/wordcount/input/file02

$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file01
the, a, an
$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file02
the,a
public class WordCount {
    public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
            }
        }
    }
}
public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {
public void reduce(Text key, Iterator<IntWritable> values,
    OutputCollector<Text, IntWritable> output, Reporter reporter)
    throws IOException {
    int sum = 0;
    while (values.hasNext()) {
        sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
}
}
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
}