COMP 322: Fundamentals of Parallel Programming

Lecture 26: Parallel Programming Patterns

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

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

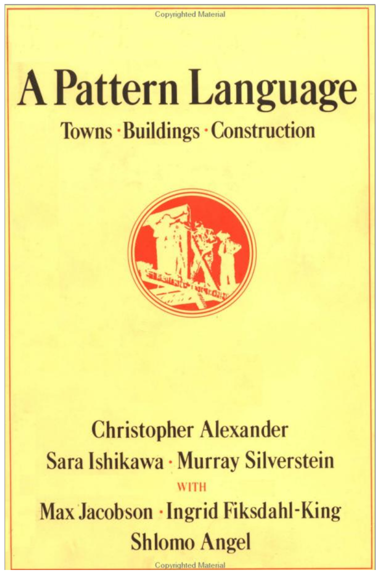
Acknowledgments for Today's Lecture

- "A Pattern language for Parallel Programming"
  - —Presentation by Beverly Sanders, U. Florida
  - —http://www.cise.ufl.edu/research/ParallelPatterns/sasplas.ppt
- "Parallel Programming with Microsoft .NET"
  - —Book published by Microsoft; free download available at http://parallelpatterns.codeplex.com
- "Introduction to Concurrency in Programming Languages", Matthew J. Sottile, Timothy G. Mattson, and Craig E Rasmussen
  - —Book published by CRC press; slides available from book web site, http://www.parlang.com/
- "Parallel Programming Patterns", ME964 lecture, Oct 2008, U. Wisconsin
  - —http://sbel.wisc.edu/Courses/ME964/2008/.../me964Oct16.ppt

Outline

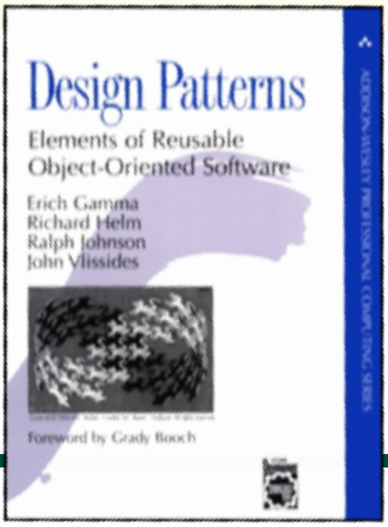
- <u>Introduction to Design Patterns</u>
- Algorithm Design Patterns from "Parallel Programming with Microsoft .NET", illustrated using HJ
- Supporting Design Patterns from "Introduction to Concurrency in Programming Languages", illustrated using HJ

Design Patterns = formal discipline of design


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

- **Christopher Alexander's approach to (civil) architecture:**
  - **A design pattern "describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."** *Page x, A Pattern Language,* **Christopher Alexander, 1977**
- **A pattern language is an organized way of tackling an architectural problem using patterns**

- The gang of 4 used patterns to bring order to the chaos of object oriented design.
- The book "overnight" turned object oriented design from "an art" to a systematic design discipline.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

GOF Pattern Example

- Behavioral Pattern: Visitor
  - Separate the structure of an object collection from the operations performed on that collection.
  - Example: Abstract syntax tree in a compiler
    - Multiple node types (declaration, command, expression, etc.)
    - Action during traversal depends on both type of node and compiler pass (type checking, code generation)
    - Can add new functionality by implementing new visitor without modifying AST code.

Parallel Programming

- Can a pattern language/taxonomy providing guidance for the entire development process make parallel programming easier?
  - Need to identify basic patterns, along with refinements (usually for efficiency)
- By relating HJ constructs to parallel programming patterns, you can apply HJ concepts to any parallel programming model you encounter in the future

Algorithm Patterns vs. Supporting Patterns

- Algorithm Patterns
  - Selection of task and data decompositions to solve a given problem in parallel
    - Task decomposition = identification of parallel steps
    - Data decomposition = partitioning of data into task-local vs. shared storage classes (with ownership specified for local data)
  - Examples: Parallel Loops, Parallel Tasks, Reductions, Dataflow, Pipeline
- Supporting Patterns
  - Selection of execution model for a given algorithm pattern
    - Execution model specifies scheduling and synchronization of tasks
    - May be implemented using code and/or configuration parameters
  - Examples of supporting patterns for task scheduling: Master-worker, Single Instruction Multiple Data (SIMD), Single Program Multiple Data (SPMD)
  - Examples of supporting patterns for mutual exclusion: Isolated, Object-based isolation, Atomic variables, Concurrent collections, Actors
- Boundary between Algorithm & Supporting Patterns can sometimes be blurred

Outline

- Introduction to Design Patterns
- Algorithm Design Patterns from "Parallel Programming with Microsoft .NET", illustrated using HJ
- Supporting Design Patterns from "Introduction to Concurrency in Programming Languages", illustrated using HJ

Selecting the Right Pattern
(adapted from page 9, Parallel Programming w/ Microsoft .Net)

Application characteristics	**Algorithmic pattern**	**Relevant HJ constructs**
Sequential loop with independent iterations	1) Parallel Loop	forall, forasync
Independent operations with well-defined control flow	2) Parallel Task	async, finish
Aggregating data from independent tasks/iterations	3) Parallel Aggregation (reductions)	finish accumulators, atomic variables
Ordering of steps based on data flow constraints	4) Futures	futures, data-driven tasks
Divide-and-conquer algorithms with recursive data structures	5) Dynamic Task Parallelism	async, finish
Repetitive operations on data streams	6) Pipelines	streaming phasers (deterministic), actors (non-deterministic)

1) Example of Parallel Loop Pattern using HJ forall statement (Lecture 10)

```
double[] myVal = new double[n+2];
. . . // Initialize myVal
for (point [iter] : [0:iterations-1]) {
  // Compute MyNew array as function of input array, MyVal
  myNew = new double[n+2];
  // PARALLEL LOOP PATTERN
  forall (point [j] : [1:n]) { // Create n tasks
     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
  } // forall
  // myNew becomes input array for next iteration
  myVal=myNew;
} // for
```

Partitioning/Chunking refinement of Parallel Loops for Efficiency (Lecture 10)

```
double[] myVal = new double[n+2];
. . . // Initialize myVal
for (point [iter] : [0:iterations-1]) {
  // Compute MyNew array as function of input array, MyVal
  myNew = new double[n+2];
  // PARALLEL LOOP PATTERN WITH CHUNKING
  forall (point [jj]:[0:Cj-1]) { // Create Cj tasks
    for (point [j]:getChunk([1:n],Cj,jj)) // Execute chunk jj
      // Loop body is unchanged from before
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
  } // forall
  // myNew becomes input array for next iteration
  myVal=myNew;
} // for
```

Double Buffering for Further Efficiency (Lecture 10)

```
double[] myVal = new double[n+2];
double[] myNew = new double[n+2];
. . . // Initialize myVal
for (point [iter] : [0:iterations-1]) {
  // Compute MyNew array as function of input array, MyVal
  // PARALLEL LOOP PATTERN WITH CHUNKING AND DOUBLE BUFFERING
  forall (point [jj]:[0:Cj-1]) { // Create Cj tasks
    for (point [j]:getChunk([1:n],Cj,jj)) // Execute chunk jj
      // Loop body is unchanged from before
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
  } // forall
  // myNew becomes input array for next iteration
  // DOUBLE BUFFERING: swap myVal & myNew
  temp=myVal; myVal=myNew; myNew=temp;
} // for
```

2) Example of Parallel Task Pattern using HJ async and finish statements (Lecture 1)

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
finish {
  async { // Task T1 computes sum of upper half of array
    for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
  }
  // Continue in T0 and compute sum of lower half of array
  for(int i=0; i < X.length/2; i++) sum1 += X[i];
} // finish
// Task T0 waits for Task T1 (join)
return sum1 + sum2;
```

Sequential clause refinement to Parallel Task Pattern for Efficiency (Lecture 9)

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
finish {
  async seq(X.length < 100000) {
    // Only create async if array length is >= 100000
    for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
  } // async
  // Continue in T0 and compute sum of lower half of array
  for(int i=0; i < X.length/2; i++) sum1 += X[i];
 } // finish
// Task T0 waits for Task T1 (join)
return sum1 + sum2;
```

3) Example of Parallel Aggregation Pattern using HJ finish accumulators (Lecture 12)

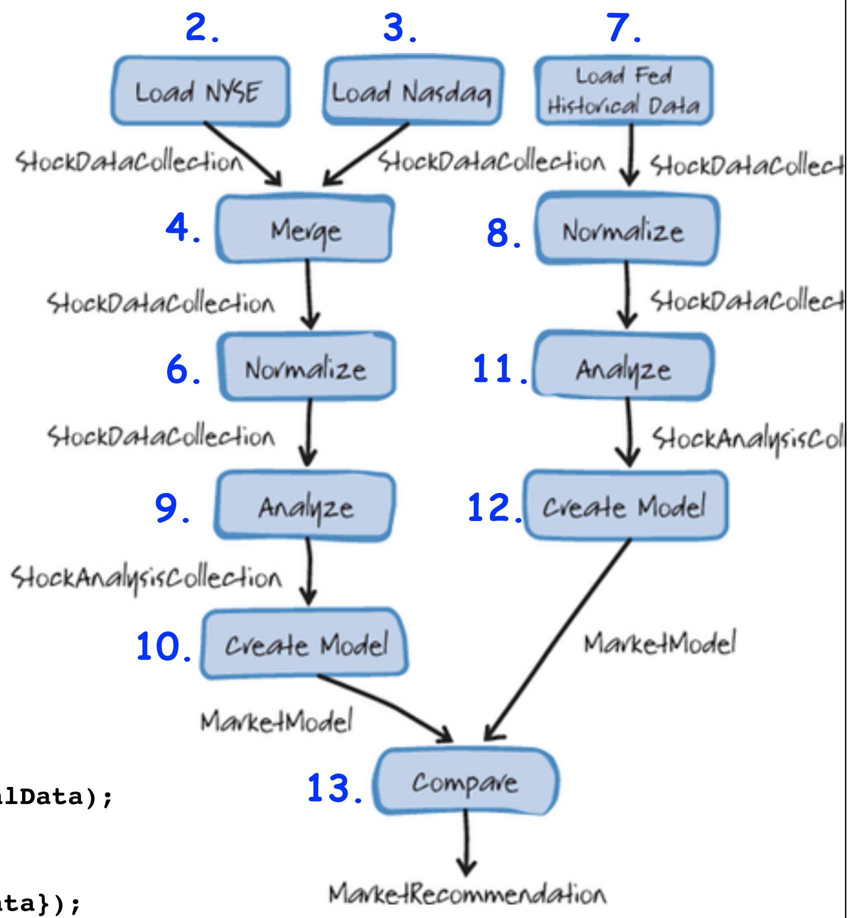
```
static accumulator a;
. . .
a = accumulator.factory.accumulator(SUM, int.class);
finish(a) nqueens_kernel(new int[0], 0);
System.out.println("No. of solutions = " + a.get().intValue())
. . .
void nqueens_kernel(int [] a, int depth) {
  if (size == depth) a.put(1);
  else
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) async {
      /* allocate a temporary array and copy array a into it */
      int [] b = new int [depth+1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = i;
      if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
    } // for-async
} // nqueens_kernel()
```

Refinement of Parallel Aggregation Pattern using Atomic Variables for Efficiency (Lecture 6)

```
import java.util.concurrent.atomic.AtomicInteger;
. . .
AtomicInteger count = new AtomicInteger();
finish nqueens_kernel(new int[0], 0);
. . .
void nqueens_kernel(int [] a, int depth) {
  if (size == depth) count.addAndGet(1);
  else
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) async {
      /* allocate a temporary array and copy array a into it */
      int [] b = new int [depth+1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = i;
      if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
    } // for-async
} // nqueens_kernel()
```

“Adatum Dashboard” Example: Sequential Version (Lecture 8)

```
1.public MarketRecommendation DoAnalysisSequential() {
2.  StockDataCollection nyseData = LoadNyseData();
3.  StockDataCollection nasdaqData = LoadNasdaqData();
4.  StockDataCollection mergedMarketData =
5.    MergeMarketData(new[]{nyseData, nasdaqData});
6.  StockDataCollection normalizedMarketData =
      NormalizeData(mergedMarketData);
7.  StockDataCollection fedHistoricalData =
      LoadFedHistoricalData();
8.  StockDataCollection normalizedHistoricalData =
      NormalizeData(fedHistoricalData);
9.  StockAnalysisCollection analyzedStockData =
      AnalyzeData(normalizedMarketData);
10. MarketModel modeledMarketData = RunModel(analyzedStockData);
11. StockAnalysisCollection analyzedHistoricalData =
      AnalyzeData(normalizedHistoricalData);
12. MarketModel modeledHistoricalData = RunModel(analyzedHistoricalData);
13. MarketRecommendation recommendation =
14.    CompareModels(new[] {modeledMarketData, modeledHistoricalData});
15.  return recommendation;
16.}
```



Source: http://programming4.us/enterprise/3004.aspx

4) Example of Future pattern using HJ future tasks

```
public MarketRecommendation DoAnalysisParallelDDT() {
  future<StockDataCollection> nyseData = async<StockDataCollection> {return LoadNyseData();};
  future<StockDataCollection> nasdaqData = async<StockDataCollection> {return LoadNasdaqData();};
  future<StockDataCollection> mergedMarketData =
    async<StockDataCollection> {return MergeMarketData(new[]{nyseData.get(), nasdaqData.get()});};
  future<StockDataCollection> normalizedMarketData =
    async<StockDataCollection> {return NormalizeData(mergedMarketData.get());};
  future<StockDataCollection> fedHistoricalData = async<StockDataCollection> {return LoadFedHistoricalData();};
  future<StockDataCollection> normalizedHistoricalData =
    async<StockDataCollection> {return NormalizeData(fedHistoricalData.get());};
  future<StockAnalysisCollection> analyzedStockData =
    async<StockAnalysisCollection> {return AnalyzeData(normalizedMarketData.get());};
  future<MarketModel> modeledMarketData = async<MarketModel> {return RunModel(analyzedStockData.get());};
  future<StockAnalysisCollection> analyzedHistoricalData =
    async<StockAnalysisCollection> {return AnalyzeData(normalizedHistoricalData.get());};
  future<MarketModel> modeledHistoricalData = async<MarketModel>{return RunModel(analyzedHistoricalData.get());};
  MarketRecommendation recommendation =
    CompareModels(new[] {modeledMarketData.get(), modeledHistoricalData.get()});
  return recommendation;
}
```

Refinement of Future Pattern using Data-Driven Tasks for Efficiency (Lecture 8)

```
public MarketRecommendation DoAnalysisParallelDDT() {
  async nyseData.put(LoadNyseData());
  async nasdaqData.put(LoadNasdaqData());
  async await(nyseData, nasdaqData)
    mergedMarketData.put(MergeMarketData(new[]{nyseData.get(), nasdaqData.get()}));
  async await(mergedMarketData) normalizedMarketData.put(NormalizeData(mergedMarketData.get()));
  async fedHistoricalData.put(LoadFedHistoricalData());
  async await(fedHistoricalData) normalizedHistoricalData.put(NormalizeData(fedHistoricalData.get()));
  async await(normalizedMarketData) analyzedStockData.put(AnalyzeData(normalizedMarketData.get()));
  async await(analyzedStockData) modeledMarketData.put(RunModel(analyzedStockData.get()));
  async await(normalizedHistoricalData) analyzedHistoricalData.put(AnalyzeData(normalizedHistoricalData.get()));
  async await(analyzedHistoricalData) modeledHistoricalData.put(RunModel(analyzedHistoricalData.get()));
  MarketRecommendation recommendation =
    CompareModels(new[] {modeledMarketData.get(), modeledHistoricalData.get()});
  return recommendation;
}
```

5) Example of Dynamic Task Pattern using HJ async and finish constructs (Lecture 11)

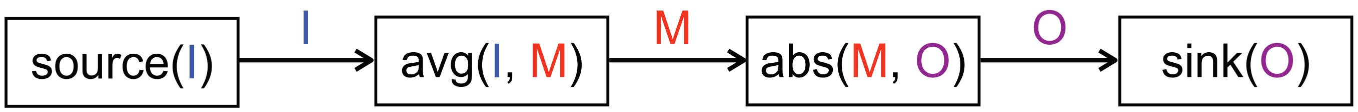
```
static void quicksort(int[] A, int M, int N) {
  if (M < N) {
    // partition() selects a pivot element in A[M…N]
    // to partition A[M…N] into A[M…J] and A[I…N]
    point p = partition(A, M, N);
    int I=p.get(0); int J=p.get(1);
    async quicksort(A, M, J);
    async quicksort(A, I, N);
  }
} //quicksort
  . . .
  finish quicksort(A, 0, A.length-1);
```

Sequential clause refinement to Dynamic Task Pattern for Efficiency

```
static void quicksort(int[] A, int M, int N) {
  if (M < N) {
    // partition() selects a pivot element in A[M…N]
    // to partition A[M…N] into A[M…J] and A[I…N]
    point p = partition(A, M, N);
    int I=p.get(0); int J=p.get(1);
    async seq(J-M < 10000) quicksort(A, M, J);
    async seq(N-I < 10000) quicksort(A, I, N);
  }
} //quicksort
. . .
finish quicksort(A, 0, A.length-1);
```

6) Example of Pipeline Pattern using HJ bounded phasers and accumulators (Lecture 15)

```
void Pipeline() {
    phaser phI     = new phaser(SIG_WAIT, bnd);
    accumulator I = new accumulator(phI, accumulator.ANY);
    phaser phM     = new phaser(SIG_WAIT, bnd);
    accumulator M = new accumulator(phM, accumulator.ANY);
    phaser phO     = new phaser(SIG_WAIT, bnd);
    accumulator O = new accumulator(phO, accumulator.ANY);
    async phased (phI<SIG>)               source(I);
    async phased (phI<WAIT>, phM<SIG>) avg(I,M);
    async phased (phM<WAIT>, phO<SIG>) abs(M,O);
    async phased (phO<WAIT>)              sink(O);
}
void avg(accumulator I, accumulator M) {
    while(...) {
        wait; wait;         // wait for two elements on I
        v1 = I.get(0);      // read first element
        v2 = I.get(-1);     // read second element (offset = -1)
        M.put((v1+v2)/2); // put result on M
        signal;
} }
```



Outline

- Introduction to Design Patterns
- Algorithm Design Patterns from "Parallel Programming with Microsoft .NET", illustrated using HJ
- <u>Supporting Design Patterns from "Introduction to Concurrency in Programming Languages", illustrated using HJ</u>

Supporting Patterns

1) Master-worker

— A process or thread (the master) sets up a task queue and manages other threads (the workers) as they grab a task from the queue, carry out the computation, and then return for their next task. This continues until the master detects that a termination condition has been met, at which point the master ends the computation.

2) Single Instruction Multiple Data (SIMD)

— A supporting pattern for data parallelism, in which a single instruction stream is applied to multiple data elements in parallel.

3) Single Program Multiple Data (SPMD)

— Multiple copies of a single program are launched typically with their own view of the data. The path through the program for each copy is determined in part based on a unique ID (a rank).

1) Master-Worker pattern

- A master manages a collection of tasks (often with a queue) and schedules them for execution on by a collection of workers.
- Workers request more tasks as they finish their assigned work,
- The result … the computational work is automatically balanced between a collection of workers.

Master Worker: Pseudo-code for the master

```
Global SharedQueue tasks ;
Global Results R;

master {
int i, taskCount ;

// First , generate the set of tasks and
// place them in the shared queue .

    for (i=0;i< taskCount ;i++) {
        Task t = generateSubTask ();
        tasks .add(t);
    }
// Next , consume the set of results produced
// by the workers .

    for (i=0;i< taskCount ;i++) {
        Result res = R.pop ();
        consumeResult (res );
    }
    shut_down();
}
```

Create data structures shared between workers and the master:

- A Queue to hold the tasks
- A Queue to hold the results

Fill the queue with a set of tasks

Pull results off the results-queue and consume them.

Shut down system results from all tasks have been consumed

Master Worker: Pseudo-code for the worker

```
Global SharedQueue tasks ;
Global Results R;

worker {

    while ( true ) {
        // Get a task
            task = tasks .pop ();
        // Perform the work
            Result res = performWork ( task );
        // Put the result in the result queue
            R. push (res );
    }
}
```

Access key shared data structures:

- A Queue to hold the tasks
- A Queue to hold the results

Grab next available task

Do the work

Store the result

This version of the worker loops until the master shuts down the entire computation

Pi Calculation Example (Lecture 22) --- Master Actor

```
class Master extends Actor<Object> {
  private double result = 0; private int nrMsgsReceived = 0;
  private Worker[] workers;
  Master(nrWrkrs, nrEls, nrMsgs) {...} // constructor
  void start() {
    super.start(); // Starts the master actor
    // Create and start workers
    workers = new Worker[nrWrkrs];
    for (int i = 0; i < nrwrkrs; i++) {
      workers[i] = new Worker();
      workers[i].start();
    }
    // Send messages to workers
    for (int j = 0; j < nrMsgs; j++) {
      someWrkr = ... ; // Select worker for message j
      someWrkr.send(new Work(...));
    }
  } // start()
```

Pi Calculation --- Master Actor (contd)

```
  void exit() {
    for (int i = 0; i < nrWrkrs; i++) workers[i].send(new Stop());
    super.exit(); // Terminates the actor
  } // exit()
  void process(final Object msg) {
    if (msg instanceof Result) {
      result += ((Result) msg).result;
      nrMsgsReceived += 1;
      if (nrMsgsReceived == nrMsgs) exit();
    }
    // Handle other message cases here
  } // process()
} // Master
. . .
// Main program
Master master = new Master(w, e, m);
finish master.start();
println("PI = " + master.getResult());
```

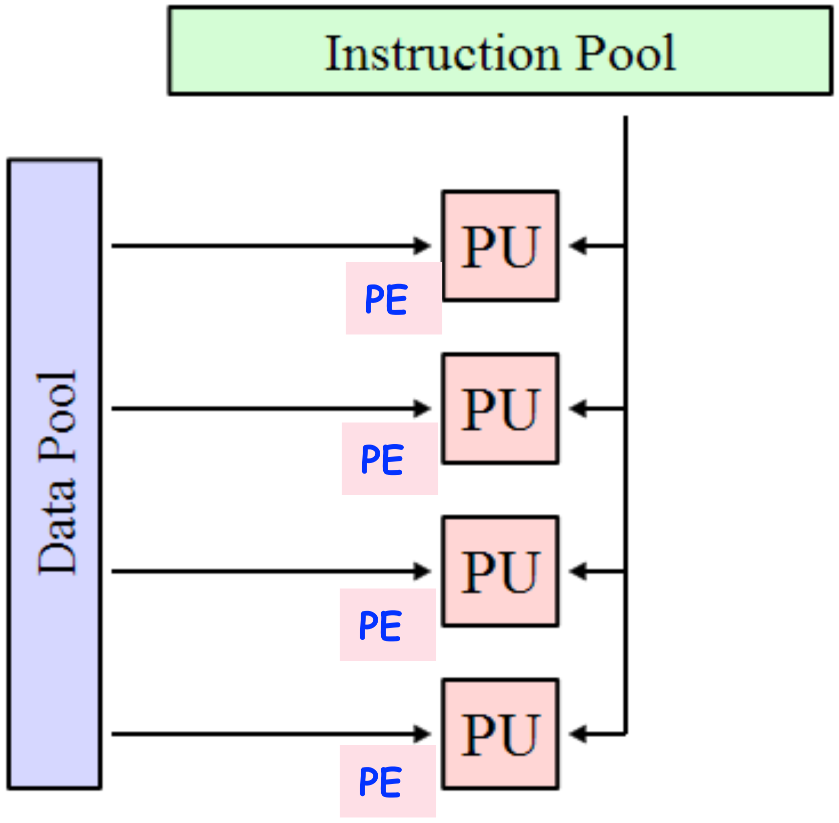
Pi Calculation --- Worker Actor

```
class Worker extends Actor<Object> {
  void process(Object msg) {
    if (msg instanceof Stop) exit();
    else if (msg instanceof Work) {
      Work wm = (Work) msg;
      double result = calculatePiFor(wm.start, wm.end)
      master.send(new ResultMessage(result));    }
  } // process()

  private double calculatePiFor(int start, int end) {
    double acc = 0.0;
    for (int i = start; i < end; i++) {
      acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1);
    }
    return acc;
  }
} // Worker
```

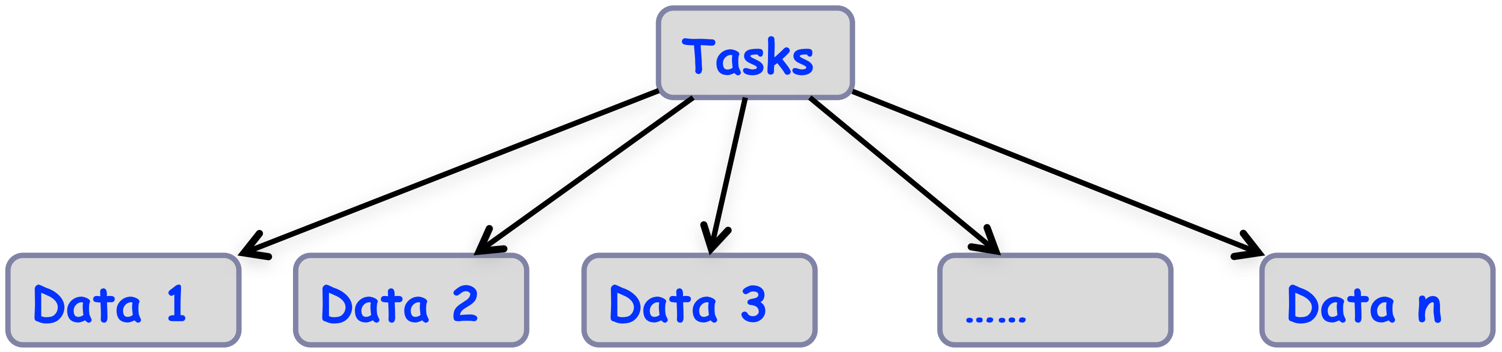
2) SIMD pattern: Single Instruction Multiple Data

- A supporting pattern for data parallelism algorithmic pattern.
- Definition: A single instruction stream is applied to multiple data elements.
  - One program text
  - One instruction counter
  - Distinct data streams per Processing Element (PE)



Data Parallelism Algorithmic Pattern

- **Use when:**
  - Your problem is defined in terms of collections of data elements operated on by a similar (if not identical) sequence of instructions; i.e. the concurrency is in the data.
- **Solution**
  - Define collections of data elements that can be updated in parallel.
  - Define computation as a sequence of collective operations applied together to each data element.



Matrix Multiplication (serial)

```
double[][] a, b, c;  // three 2D arrays : a,b,c
int n; // Assume that all arrays are of size n*n

for(point[i,j] : [0:n-1,0:n-1] {
        c[i][j] = 0.0
        for(int k=0; k<n; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
}
```

This operation is called a dot product of the two vectors a and b to produce a scalar value for c(i,j)

Matrix Multiplication (with multiple sources of parallelism)

```
double[][] a, b, c;  // three 2D arrays : a,b,c
int n; // Assume that all arrays are of size n*n

forall(point[i,j] : [0:n-1,0:n-1] {

          c[i][j] = sum(a[i][0:n-1] * b[0:n-1][j]);

}
```

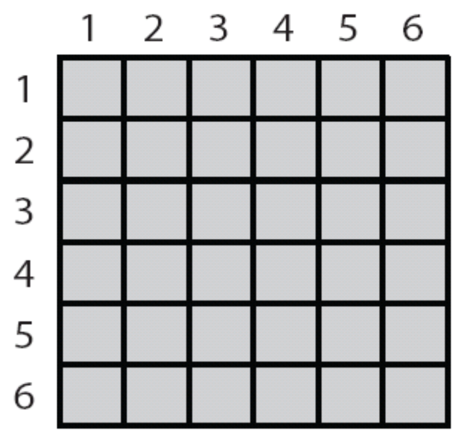
Loop parallelism

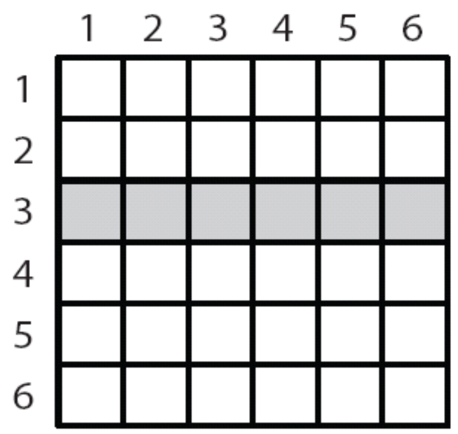
Dot product is expressed as SIMD parallelism

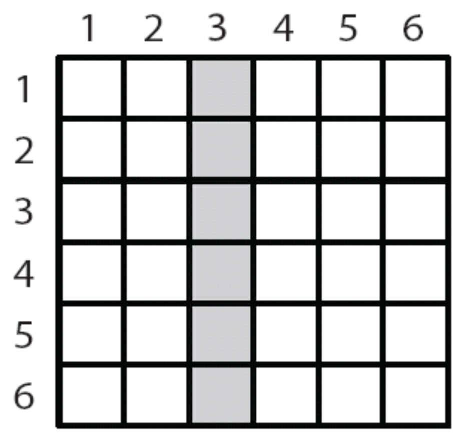
(This is pseudocode, not real HJ code)

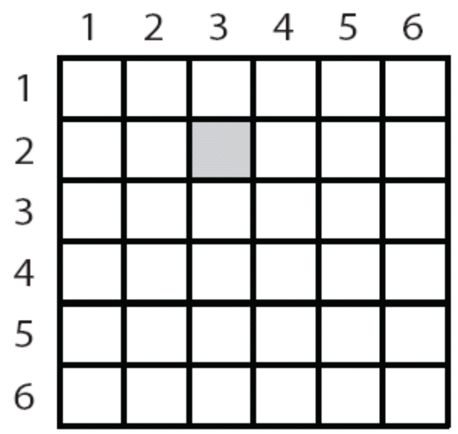
Array slice notation

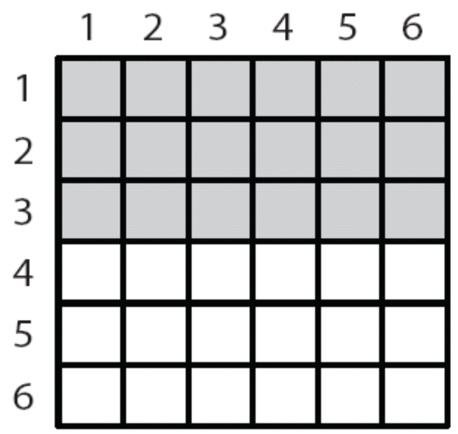
- Designating different slices of an array.

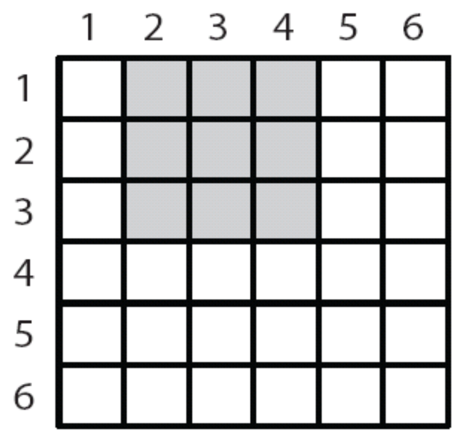

A[:][:]


A[3][:]


A[:][3]


A[2][3]


A[1:3][:]


A[1:3][2:4]

Flynn’s Taxonomy for Parallel Computers

	Single Instruction	Multiple Instructions
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Single Instruction, Single Data stream (SISD)

A sequential computer which exploits no parallelism in either the instruction or data streams. e.g., old single processor PC

Single Instruction, Multiple Data streams (SIMD)

A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. e.g. graphics processing unit

Multiple Instruction, Single Data stream (MISD)

Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. e.g. the Space Shuttle flight control computer.

Multiple Instruction, Multiple Data streams (MIMD)

Multiple autonomous processors simultaneously executing different instructions on different data. e.g. a PC cluster memory space.

3) SPMD Supporting Pattern

- SPMD: Single Program Multiple Data
- Run the same program on P processing elements (PEs)
- Use the "rank" … an ID ranging from 0 to (P-1) … to determine what computation is performed on what data by a given PE
- Different PEs can follow different paths through the same code (unlike the SIMD pattern)
- Convenient pattern for hardware platforms that are not amenable to efficient forms of dynamic task parallelism
  - General-Purpose Graphics Processing Units (GPGPUs)
  - Distributed-memory parallel machines
- Key design decisions --- what data and computation should be replicated or partitioned across PEs?

Typical SPMD Program Phases

- **Initialize**
  - Establish localized data structure and communication channels
- **Obtain a unique identifier**
  - Each thread acquires a unique identifier, typically range from 0 to N=1, where N is the number of threads.
  - Both OpenMP and CUDA have built-in support for this.
- **Distribute Data**
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- **Run the core computation**
  - More details in next slide…
- **Finalize**
  - Reconcile global data structure, prepare for the next major iteration

SPMD Example #1

- Assign a chunk of iterations to each thread
  - The last thread also finishes up the remainder iterations

```
num_steps  = 1000000;

i_start = my_id * (num_steps/num_threads);
i_end = i_start + (num_steps/num_threads);
if (my_id == (num_threads-1))  i_end = num_steps;

for (i = i_start; i < i_end; i++) {
….
}
Reconciliation of results across threads if necessary.
```

SPMD Example #2: Iterative Averaging Example (Slide 9, Lecture 13)

```
double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
gVal[n+1] = 1; // Boundary condition
int Cj = Runtime.getNumOfWorkers();
forall (point [jj]:[0:Cj-1]) { // SPMD computation
  double[] myVal = gVal; double[] myNew = gNew; // Local copy
  for (point [iter] : [0:numIters-1]) {
    // Compute MyNew as function of input array MyVal
    for (point [j]:getChunk([1:n],[Cj],[jj]))
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    next; // Barrier before executing next iteration of iter loop
    // Swap myVal and myNew (replicated computation)
    double[] temp=myVal; myVal=myNew; myNew=temp;
    // myNew becomes input array for next iter
  } // for
} // forall
```