Session:

Parallel Programming Patterns
Objectives

At the end of this session, you will be able to:

• Describe the concepts behind design patterns and parallel design patterns

• Given serial code or algorithms, choose the better Algorithm Structure design pattern (either Task Parallelism or Geometric Decomposition) to be used in threading the code and defend your choices

• Given serial code or algorithms, choose the better Supporting Structure design pattern (either SPMD, Loop Parallelism or Boss/Worker) to be used in threading the code and defend your choices
Agenda

Pattern Language Structure

The Task Parallelism Pattern

The Geometric Decomposition Pattern

Supporting Structures

Summary
Agenda

Pattern Language Structure

The Task Parallelism Pattern

The Geometric Decomposition Pattern

Supporting Structures

Summary
Design Patterns and Pattern Languages

A design pattern is:

- A “solution to a problem in a context.”
- A structured description of high quality solutions to recurring problems
- A quest to encode expertise so all designers can capture that “quality without a name” that distinguishes truly excellent designs

A pattern language is:

- A structured catalog of design patterns that supports design activities as “webs of connected design patterns.”
Reference

Details a pattern language for parallel algorithm design

Examples in MPI, OpenMP and Java are given

Represents the author's hypothesis for how programmers think about parallel programming

*Patterns for Parallel Programming*, Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, Addison-Wesley, 2005, ISBN 0321228111
Pattern Language’s Structure

A software design can be viewed as a series of refinements. Consider the process in terms of 4 Design Spaces:

- Add progressively lower level elements to the design.

<table>
<thead>
<tr>
<th>Design Space</th>
<th>The Evolving Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finding Concurrency</td>
<td>Tasks, shared data, partial orders</td>
</tr>
<tr>
<td>Algorithm Structure</td>
<td>Thread/process structures, schedules</td>
</tr>
<tr>
<td>Supporting Structures</td>
<td>Source Code organization, Shared data</td>
</tr>
<tr>
<td>Implementation Mechanisms</td>
<td>Messages, synchronization, spawn</td>
</tr>
</tbody>
</table>
Finding Concurrency Design Space

Finding the scope for parallelization:
- Begin with a sequential application that solves the original problem
- Decompose the application into tasks or data sets
- Analyze the dependency among tasks before decomposition

Decomposition Analysis
- Domain decomposition
- Task decomposition

Dependency Analysis
- Group tasks
- Order groups
- Data sharing
Finding Concurrency

From serial code:

1. Use profiler, such as Intel® VTune™ Performance Analyzer
2. Identify hotspots in an application
3. Examine the code in hotspots
4. Determine whether the tasks within the hotspots can be executed independently

From a design document:

- Examine the design components
- Find components that contain independent operations
Algorithm Structure Design Space

Structure used for organizing computations to support parallelization

- How is the computation structured?
  - Organized by data
    - Linear?
      - Geometric Decomposition
    - Recursive?
      - Recursive Data
  - Organized by tasks
    - Linear?
      - Task Parallelism
    - Recursive?
      - Divide and Conquer
  - Organized by flow of data
    - Regular?
      - Pipeline
    - Irregular?
      - Event-based Coordination
Forces Affecting Algorithm Structures

The following forces affecting the design goals must be kept in mind:

- **Efficiency**
  - The parallel program must run quickly and make good use of processing resources

- **Simplicity**
  - Simple algorithms (and easy-to-understand code) are easier to develop, debug, verify, and maintain

- **Portability**
  - Programs should run on the widest range of parallel computers

- **Scalability**
  - Parallel algorithms should be effective on a wide range of numbers of threads and cores, and sizes of data sets

There may be conflicts between two or more of these forces

- Balance of forces is critical to getting best parallel code
Supporting Structures Design Space

High-level constructs used to organize the source code
Categorized into program structures and data structures

- Program Structure
  - SPMD
  - Loop parallelism
  - Boss/Worker
  - Fork/join

- Data Structures
  - Shared data
  - Shared queue
  - Distributed array
Implementation Mechanisms Design Space

Low level constructs implementing specific constructs used in parallel computing
- Not proper design patterns; included to make the pattern language self-contained

- UE* Management
  - Thread Control
  - Process Control
- Synchronization
  - Memory sync/fences
  - Mutual Exclusion
  - Barriers
- Communications
  - Message Passing
  - Collective Comm
  - Other Comm
Agenda

Pattern Language Structure

The Task Parallelism Pattern

The Geometric Decomposition Pattern

Supporting Structures

Summary
How is the computation structured?

Organized by data
- Linear?
- Recursive?
- Geometric Decomposition
- Recursive Data

Organized by tasks
- Linear?
- Recursive?
- Task Parallelism
- Divide and Conquer

Organized by flow of data
- Regular?
- Irregular?
- Pipeline
- Event-based Coordination
Task Parallelism Pattern

Problem:
- How do you exploit concurrency expressed in terms of a set of distinct tasks?

Examples:
- **Ray tracing** – the computation of each ray is independent of any other ray computation
- **Molecular dynamics** – Update of forces on atoms is independent; combining of all partial force computations from each thread is done after all forces are done
Task Parallelism Pattern Context

Every parallel algorithm is a collection of concurrent tasks
- Can often be found through inspection of code
- Common factor is decomposing computation into collection of concurrent tasks
  - Tasks may be completely independent
  - Tasks may have dependencies that need to be satisfied

Tasks are usually known at outset of computation
Tasks may arise dynamically (e.g., branch-and-bound tree search)
Typically must wait for all tasks to complete before problem is done
- May be able to terminate when solution found before all tasks complete
  - Example: Searching – if item is found, no need to continue searching; if item not present, must finish search to know for sure
Task Parallelism Pattern Forces

Size of tasks
  • Small size to balance load vs. large size to reduce scheduling overhead

Managing dependencies
  • Without destroying efficiency, simplicity, or scalability
Task Parallelism Pattern Solutions

Three key elements for Task Parallelism pattern designs

- Tasks and how they are defined
- The dependencies between the tasks
- Scheduling the tasks to threads
Defining Tasks

Decomposing serial algorithm with task decomposition should meet two criteria:

1. There should be at least as many tasks as there are threads (or cores)
   - Preferably there should be many more tasks than threads
   - Gives greater flexibility in scheduling
   - Assist in achieving a balanced load between threads

2. Amount of computation within a task must be large enough to offset overhead of managing task and thread

If initial decomposition does not meet these criteria, alternate decompositions should be considered
ACTIVITY: defining task example

Consider an image processing application, where each pixel is independent

What is the appropriate level of task decomposition?

- Task == one pixel
- Task == one line
- Task == blocks within image
Managing Dependencies Within Task Parallelism Pattern Solutions

Ordering constraints between tasks can be handled by forcing groups of tasks to execute in order.
Managing Dependencies Within Task Parallelism Pattern Solutions

Solving data dependencies can be more complicated

- Simplest case is no dependencies (embarrassingly parallel)

- Strategies for handling data dependencies between tasks:
  - removing them (replicating data)
  - transforming induction variables
  - exposing reductions
  - explicitly protecting (shared data pattern)
Removable Dependencies

Not a true dependency, but an apparent dependence that can be eliminated by code transformation

The simplest solution would be to create, initialize, and use local variables
  • May involve the replication of work within threads

Thread 1

```
Compute X
Compute Y
Compute Z
For i = 1, 20
  Compute a[i] = F(X,Y,Z,b[i])
```

Thread 2

```
Compute global X
Compute local X
Compute local Y
Compute global Z
For i = 1, 10
  Compute a[i] = F(X,Y,Z,b[i])
Wait for X,Y,Z to be ready
Compute local Z
For i = 11, 20
  Compute a[i] = F(X,Y,Z,b[i])
```
Removable Dependencies

Induction variables are incremented on each trip through the loop

```
i1 = 4;
i2 = 0;
for (I = 0; I < N; I++)
{
    B[i1++] = ... ;
i2 = i2 + I;
    A[i2] = ... ;
}
```

Fix by replacing increment statements with closed-form expressions involving loop index

```
for (I = 0; I < N; I++)
{
    B[I+4] = ...;
    A[((I+1)*I)/2)] = ...;
}```
“Separable” Dependencies

Dependence on shared data structure used for accumulation of data computed in each thread

- Pull the accumulation out of the concurrency:
  - Replicate the data structure to each thread, initialize as needed
  - Execute the tasks using the local copy of data structure
  - Upon completion, combine all copies in global data structure

Most common example: Reduction

- A collection of data is reduced to a single value by applying a binary operation to all elements of the collection
  - Binary operation must be associative and commutative

```c
for (int i = 0; i < N; i++)
    sum += a[i] * b[i];
```
Explicit Protection of Shared Data

If dependency cannot be removed or separated from tasks, then shared data that is updated must be protected.

Implement a critical region around code that accesses shared data:

- If any updates are made, all accesses (both read and write) must be in a critical region.
- Add programming logic to enforce mutual exclusion to critical region.
- Use appropriate synchronization object to create mutual exclusion.
Scheduling Tasks

Tasks must be assigned to threads for execution

Schedule tasks for execution with balanced load

Two classes of schedules are used

- **Static scheduling**
  - Tasks are assigned to threads at beginning of computation and does not change
  - Easy to program and incurs the least overhead
  - Should be used whenever possible

- **Dynamic scheduling**
  - Tasks are assigned as the computation proceeds
Static Scheduling Considerations

Work may be differentiated with unique thread id for each thread

If tasks are collections of separate, independent function calls
  • Group calls into tasks that are assigned to threads

If tasks are loop iterations, two possible schedules:
  1. Divide the total number of iterations by the number of threads and assign block(s) of iterations to threads
  2. Start each thread’s set of iteration by the thread id number (0..N-1), each thread then computes every Nth iteration
Dynamic Scheduling Considerations

Used when task execution length is variable and is unpredictable

Possible methods for dynamic scheduling of tasks

• Use a shared structure to hold tasks
  – If tasks are able to be encapsulated into a structure, create a global queue that is used to hold and dispatch tasks when threads need more work
  – If tasks are stored in a file, threads share access to reading tasks from file

• Some tasks may take pre-processing in order to be ready for assignment to computation thread
  – Use an extra thread to get tasks ready for threads and assign tasks to threads when requested (Master/Worker pattern)
  – Use extra thread to place tasks into shared queue for other threads to remove as needed (Producer/Consumer pattern)

• If tasks are indexed, create shared counter to keep track of and assign next task for execution
CASE STUDY: Avoiding Airplane Disasters

Snapshot of current set of airplanes around an airport

Are any pair of planes within dangerous proximity to each other?

- If so, issue an immediate warning and log the event
- Keep track of the number of warnings generated by a plane in the snapshot
  - Will be used to determine which flight may be most in danger

Planes have flight number and current location in 3-D space

- X and Y are location with respect to the ground (map)
- Z is current altitude

Pseudo-code on next slide

- Simple distance calculation used to determine if potential for warning
function AirplanePositions (N, Flights)
    int const N;                       //number of planes
    Array of Airplane_t :: Flights(N); // array of planes structs

    Real :: distX, distY, distZ, distance

    loop [j = 0, N]
        loop [k = j+1, N]
            distX = (Flights[j].Xcoord - Flights[k].Xcoord) ^ 2
            distY = (Flights[j].Ycoord - Flights[k].Ycoord) ^ 2
            distZ = (Flights[j].Zcoord - Flights[k].Zcoord) ^ 2
            distance = sqrt(distX + distY + distZ)
            if (distance < SAFETY_MARGIN)
                PrintWarning(Flights[j],Flights[k])
                Flights[j].numWarnings++
                Flights[k].numWarnings++
            
        end loop[k]
    end loop[j]
end function AirplanePositions
function AirplanePositions (N, Flights)
    int const N;                      //number of planes
    Array of Airplane_t :: Flights(N); // array of planes structs

    Real :: distX, distY, distZ, distance

    loop [j = 0 .. N]
        loop [k = j+1 .. N]
            distX = (Flights[j].Xcoord - Flights[k].Xcoord) ^ 2
            distY = (Flights[j].Ycoord - Flights[k].Ycoord) ^ 2
            distZ = (Flights[j].Zcoord - Flights[k].Zcoord) ^ 2
            distance = sqrt(distX + distY + distZ)
            if (distance < SAFETY_MARGIN)
                PrintWarning(Flights[j],Flights[k])
                Flights[j].numWarnings++
                Flights[k].numWarnings++
            end loop[k]
        end loop[j]
    end function AirplanePositions

Computation of distance between each plane is a task
CASE STUDY: Airplane Positioning code - Dependencies

```c
function AirplanePositions (N, Flights)
  int const N;                       //number of planes
  Array of Airplane_t :: Flights(N); // array of planes structs
  Real :: distX, distY, distZ, distance
  
  Each thread will need local
  copy of loop indices j, k
  of these work variables

  loop [j = 0 .. N]
    loop [k = j+1 .. N]
      distX = (Flights[j].Xcoord - Flights[k].Xcoord) ^ 2
      distY = (Flights[j].Ycoord - Flights[k].Ycoord) ^ 2
      distZ = (Flights[j].Zcoord - Flights[k].Zcoord) ^ 2
      distance = sqrt(distX + distY + distZ)
      
      if (distance < SAFETY_MARGIN)
        PrintWarning(Flights[j].FlightID, Flights[k].FlightID)
        Flights[j].numWarnings++
        Flights[k].numWarnings++
      endif
    end loop[k]
  end loop[j]
end function AirplanePositions
```

Update of shared
variables will need to
be protected

Each thread will need local
copy of loop indices j, k
of these work variables
CASE STUDY: Airplane Positioning code How Should Tasks be Scheduled?

```plaintext
loop [j = 0, N]
    loop [k = j+1, N]
        distX = (Flights[j].Xcoord - Flights[k].Xcoord) ^ 2
        distY = (Flights[j].Ycoord - Flights[k].Ycoord) ^ 2
        distZ = (Flights[j].Zcoord - Flights[k].Zcoord) ^ 2
        distance = sqrt(distX + distY + distZ)
        if (distance < SAFETY_MARGIN)
            LogWarning(Flights[j], Flights[k])
            Flights[j].numWarnings++
            Flights[k].numWarnings++
        endif
    end loop[k]
end loop[j]
```

**Static Schedule**

Divide up N planes and assign to threads

- Assign N/(# of threads) planes to a thread
- Assign every (# of threads)\(^{th}\) plane to each thread, start with thread id

**Dynamic Schedule**

Set up global counter (0..N-1)

Threads access counter for next [j], increment counter

- Must protect access and increment of global counter
What are the tasks within the pseudo-code?
Are there any dependencies? How should they be resolved?
How should the tasks be scheduled to run on threads?
ACTIVITY: Molecular Dynamics code

Collections of atoms computes nonbonded forces between pairs of atoms

Only pairs of atoms that are with a radius equal to some preset limit
  • Neighbor list for each atom determines which other atoms are within radius
  • Newton’s Third Law: Force of Atom[i] on Atom[j] is negative of Atom[j] on Atom[i]
    – If Atom[j] is on the neighbor list of Atom[i], then Atom[i] will not be on list for Atom[j]

Pseudo-code on next slide
  • Physics computations not needed for this exercise
  • How neighbor list updates are made is not shown
function non_bonded_forces_all (N, Atoms, neighbors, Forces)
  Int const N // number of atoms
  Array of Real :: atoms(3,N) // 3D coordinates
  Array of Real :: Forces(3,N) // force in each dimension
  Array of List :: neighbors(N) // atoms in cutoff volume
  Real :: forceX, forceY, forceZ

  loop [i] over atoms
    loop [j] over neighbors(i)
      forceX = non_bonded_force_pair(atoms(1,i), atoms(1,j))
      forceY = non_bonded_force_pair(atoms(2,i), atoms(2,j))
      forceZ = non_bonded_force_pair(atoms(3,i), atoms(3,j))
      Forces(1,i) += forceX; Forces(1,j) -= forceX;
      Forces(2,i) += forceY; Forces(2,j) -= forceY;
      Forces(3,i) += forceZ; Forces(3,j) -= forceZ;
    end loop[j]
  end loop [i]
end function non_bonded_forces_all
Agenda

Pattern Language Structure

The Task Parallelism Pattern

The Geometric Decomposition Pattern

Supporting Structures

Summary
Algorithm Structure Design Space

How is the computation structured?

Organized by data
- Linear?
- Recursive?
  - Geometric Decomposition
  - Recursive Data

Organized by tasks
- Linear?
- Recursive?
  - Task Parallelism
  - Divide and Conquer

Organized by flow of data
- Regular?
- Irregular?
  - Pipeline
  - Event-based Coordination
Geometric Decomposition Pattern

Problem:
- How can a parallel algorithm be organized around a data structure that can be decomposed into independent or concurrently updatable “chunks”?

Examples:
- **Weather prediction** – Divide area of interest into discrete, non-overlapping sections; compute effects of wind, land, sun, humidity, weather in neighboring sections, etc. iteratively
- **Newton-Raphson fractals** – For each point within the complex plane, compute number of iterations required to reach root of polynomial; color point appropriately
Geometric Decomposition Pattern Context

Many problems can be best understood as a sequence of operations on a core data structure
- All elements of the structure are updated or used for computation

Structure is divided into contiguous substructures or sub-regions
- Arrays: divide along one or more dimensions
  
- Lists: define sublists of discrete elements

- Graphs: construct subgraphs
We will use the generic term “chunk” to refer to a sub-region.

Decomposition of data into chunks implies a division of the computation into tasks that operate on elements from each chunk:
- Tasks will execute concurrently.
- Each task will update the associated chunk.
- Tasks may require data from “neighboring” chunks that must be shared.
  - Neighboring chunks contain data that was “nearby” in the original data structure.
Geometric Decomposition Pattern Forces

Goals: simplicity, portability, scalability, and efficiency

Chunks of data must be assigned to threads
- Load balance can be a factor, especially when using chunks of variable sizes
- Associated computation must be executable by the thread

Ensure that data needed to update chunks is available when needed
- Data from within chunk is readily available
- Accessing or retrieving essential data from neighboring chunks may require coordination between threads
The following key activities must be considered:

1. Data decomposition
   - Partitioning the global data structure into chunks
   - Consider granularity and shape of chunks

2. Exchange operation
   - Ensuring each task has access to all data needed for update
   - Does “external” data need to be available?

3. Update (computation) operation
   - Updating elements within the chunk
   - Typically the bulk of the computation

4. Data distribution and task scheduling
   - How to map chunks and associated computation to threads
1. Data Decomposition - Granularity

Granularity of data decomposition will have significant impact on performance and efficiency of the application

- **Coarse-grained decomposition**
  - Smaller number of large chunks
  - Requires lower amount of data sharing (synchronization/communication)

- **Fine-grained decomposition**
  - Larger number of small chunks
  - May have (many) more chunks than threads; easier for load balancing
  - Larger amount of data sharing and synchronization

Optimal granularity may be hard to derive mathematically at outset of computation

- Experiment to find best decomposition size
- If workloads can change from run to run, decomposition size should be tunable parameter within application
  - Good for scalability of application
1. Data Decomposition – Shape (Array)

Shape of chunks can affect amount of communication and synchronization needed
- Data to be shared is typically boundary values from chunks
- Shared data scales with the surface area of the chunks
- Computation scales with volume of chunks (number of data elements in chunk)

Maximize Surface-to-Volume ratio to maximize computation to synchronization ratio

Size and shape of chunks may be affected by other factors
- Cache line size, row-major vs. column-major, source code before/after
1. Data Decomposition – Shape (Graph)

Surface of subgraph may be cut width to divide into subgraphs
- Assumes data must be shared across edges

Better division of nodes to threads to minimize neighbors assigned to different thread

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>
Describe decomposition schemes for the following data sets:

- Square grid representing land mass and coastline of Australia
- Catalog of library books
- Graph representing map of 3000 US cities (nodes) and roads between them (edges)
2. Exchange Operation

Key factor in using Geometric Decomposition pattern correctly
- Ensuring the efficient exchange of data between nearby chunks
- Copy data to local structures or access data as needed?

Copy to local data structure
- If data available before update operation and won’t change during copy
- Requires extra memory per thread, but no contention on copies

Access data as needed
- Takes advantage of shared memory between threads
- Delays coordination between threads until data is needed

Must coordinate between threads to ensure data is available and won’t change during exchange
3. Update Operation

Execution of the associated tasks will concurrently update the data structure

Coding considerations for exchange and update interaction

- If all data is available at beginning of the tasks and will not change during computation, parallelization will be easier and likely efficient
- If non-local data will be copied from other chunks before update computations, add data gathering phase before start of update
- If non-local data will be accessed throughout update operations, add code to ensure correct data is found and ensure proper access protocol is observed (mutual exclusion?)
  - Mixing exchange and update computations can complicate the logic of the application, especially to ensure correct data is retrieved
4. Data Distribution and Task Scheduling

Determine what threads will update what data chunks

Static distribution is simplest

- Coordination for exchange will be straightforward to determine and implement
- Most appropriate when amount of computation within tasks is uniform

Dynamic distribution useful for load balance

- Requires (many) more tasks than threads
- Exchange operation will be more complex
  - Must ensure updates of non-local data are completed

What about dynamic redistribution of initial static distribution?

- If amount of work per chunk changes over the course of computation
  - Example: matrix operation that eliminates rows/columns
- Application can reconfigure task/data distribution for load balance
- Gain in performance must overcome added overhead for redistribution
CASE STUDY: Matrix Multiplication

```python
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        C[i][j] = 0.0;

for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < L; k++)
            C[i][j] += A[i][k] * B[k][j];
```

What can be computed independently?

- Individual elements of C
- Rows of C (using all of B)
- Columns of C (using all of A)
- Blocks of C
What large data structure needs to be updated?

How should the data structure be decomposed? What granularity will be best?

What data needs to be shared between tasks associated with the data chunks? Is there any non-local data that needs to be exchanged?

How should we assign chunks to threads? Static or Dynamic?
What large data structure needs to be updated?
  The C matrix

How should the data structure be decomposed? What granularity will be best?
  Individual elements seems a bit too fine-grained
  Groups of rows fits row-major access and is coarser-grained
  Blocks could be used with eye on cache size and utilization

What data needs to be shared between tasks associated with the data chunks? Is there any non-local data that needs to be exchanged?
  Elements of A and B arrays are shared (but not updated)
  If these fit in memory, they can be shared without exchange

How should we assign chunks to threads? Static or Dynamic?
  Since no exchange of data is needed, static distribution is easiest
CASE STUDY: Matrix Multiplication

Modify bounds of “i” loop, based on unique thread id, in order to assign groups of rows to individual threads

```c
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        C[i][j] = 0.0;

for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < L; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Each thread will require local copies of all three loop index variables
What is the large data set that can be updated concurrently? How should this data be decomposed? How should the data and tasks be scheduled to run on threads? What considerations need to be taken into account for load balance?
Agenda

Pattern Language Structure

The Task Parallelism Pattern

The Geometric Decomposition Pattern

Supporting Structures

- SPMD
- Loop Parallelism
- Boss/Worker

Summary
Supporting Structures Design Space

High-level constructs used to organize the source code
Categorized into program structures and data structures

Program Structure
- SPMD
- Loop parallelism
- Boss/Worker
- Fork/join

Data Structures
- Shared data
- Shared queue
- Distributed array
SPMD Pattern

Problem
- How do you structure a parallel program to make interactions between threads manageable yet easy to integrate with the core computation?

Context
- Interactions between threads must be well orchestrated for efficient and correct solutions
- Most parallel algorithms have similar operations carried out by each thread, but there may be slightly different requirements for a subset of threads or data
- Only the most complex algorithms need very different instruction and data streams on each thread

Forces
- It will be easier to balance the conflicting needs of scalability and maintainability within a single source than across multiple source files
SPMD Pattern – Solution

Single application will create threads that execute code and functions within the source

- Due to nature of multithreaded programming, SPMD pattern is default

Coding Elements to implement SPMD Pattern

- **Obtain unique identifier**
  - Assign ID number (rank) to each thread, usually from 0..N-1
- **Use ID to differentiate behavior of different threads**
  - Branching statements can be used to assign blocks of code
  - Use ID in loop index calculations to divide iterations among threads
- **Distribute data**
  - Decompose global data into chunks and access chunks based on ID
- **Finalize**
  - If globally relevant data was distributed among threads, partial results may need to be recombined
SPMD Pattern – Discussion

SPMD programs are more closely aligned with distributed, message-passing environments

Single program that spawns threads is the default for threaded programming

• Threads within a single process sharing code and data
• More relevant with general threading models than OpenMP
Loop Parallelism Pattern

Problem
- How do you structure a parallel program from a serial program where a set of computationally intensive loops dominate the execution time?

Context
- An overwhelming number of scientific and technical applications make use of iterative constructs (loop-based)
- Goal is to “evolve” sequential code into parallel code by transforming loops to execute separate iterations on threads
  - Loop iterations must work well as parallel tasks
    - Computationally intensive
    - Express sufficient concurrency
    - Mostly independent
  - Remove loop-carried dependencies, if present

This pattern is particularly relevant for using OpenMP, Intel TBB
Loop Parallelism Pattern - Forces

Sequential Equivalence
- Program that yields identical results running with one or with many threads
  - Round-off error variations are acceptable
- Sequentially equivalent code is easier to write and maintain

Incremental Parallelism
- Easier to maintain correctness if the loop transformations can be done and tested one at a time
- Can be difficult with general threading model and distribution of data to local chunks (“whole code” transformation needed)

Memory Utilization
- Data access patterns must fit well with memory hierarchy
- Dividing iterations for threads may ruin sequential access patterns
  - Restructuring of loops for good access may be required
Loop Parallelism Pattern - Solution

Programming style needed is aligned with OpenMP, Intel TBB
- General threading models can be used with programmer supplied assignment of loop iterations

Coding Elements to implement Loop Parallelism Pattern
- **Find the hotspots**
  - Identify the most computationally intensive loops by inspection of code, understanding of algorithm, or use of profiling tool
- **Eliminate loop-carried dependencies**
  - Find dependencies between iterations or data conflicts
  - Remove dependencies or protect data with synchronization
- **Parallelize the loops**
  - Split up the iterations among threads
- **Optimize the loop schedule**
  - Schedule loop iterations to thread to ensure load balance
Loop Parallelism Pattern - Considerations

Compute time within iterations must be sufficient to overcome overhead of marshalling threads and assigning work to threads

- Solution: Merge Loops
  - Given a sequence of loops with consistent index ranges, merging of these loops into a single loop will increase the work per iteration of the resulting loop

The larger the number of iterations available, the more flexibility there will be for scheduling decisions

- Solution: Coalesce nested loops
  - Combining nested loops can yield a single loop with a larger combined iteration count
  - Larger iteration counts may also afford better chances to overcome overhead
Prime number of iterations will never be perfectly load balanced.

Parallelize inner loop? Are there enough iterations to overcome thread management overhead?

Larger number of iterations gives better opportunity for load balance and hiding overhead.

DIV and MOD with inner loop upper bound value.

These computations are overhead.
Splitting Loop Iterations - OpenMP

```c
#define N 23
#define M 1000
...

#pragma omp for private(k,j) schedule(static, 500)
for (kj = 0; kj < N*M; kj++) {
    k = kj / M;
    j = kj % M;
    w_new[k][j] = DoSomeWork(w[k][j], k, j);
}
```

Add OpenMP worksharing construct pragma to divide loop iterations across threads

- Add data environment and schedule clauses, as needed
### Splitting Loop Iterations – Divide into Chunks

```c
#define N 23
#define M 1000
#define numThreads 4
```

Compute start and end points for loop iterations based on number of threads and the thread’s ID number:

```c
int start = ((N*M)/numThreads) * myID;
int end = ((N*M)/numThreads) * (myID+1);
```

If `myID == (numThreads-1)`, then:

```c
end = N*M;
```

For loop:

```c
for (kj = start; kj < end; kj++) {
    k = kj / M; j = kj % M;
    w_new[k][j] = DoSomeWork(w[k][j], k, j);
}
```

These must be private copies for each thread.

Be sure all iterations are included.
23 * 5323 = 122,429 iterations

122,429 / 16 = 7651.8125

$
\text{int start} = \left(\frac{N \times M}{\text{numThreads}}\right) \times \text{myID};
\text{int end} = \left(\frac{N \times M}{\text{numThreads}}\right) \times (\text{myID}+1);
$

# define N 23
# define M 5323
# define numThreads 16


13 extra iterations!

Software and Services

Copyright © 2014, Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. * Other brands and names are the property of their respective owners.
Splitting Loop Iterations – Round Robin

```c
#define N 23
#define M 1000
#define numThreads 4

for (kj = myID; kj < N*M; kj+=numThreads) {
    k = kj / M;
    j = kj % M;
    w_new[k][j] = DoSomeWork(w[k][j], k, j);
}
```

Each thread does every (numThreads)th iteration

Range of myID values is [0..numThreads-1]
Good load balance on number of iterations
• Off by ±1 between any two threads
• Easily coded, no chance of missing iterations

Be aware of potential cache and memory access problems
ACTIVITY: Loop Parallelism

What type of assignment would be used on the following computations to divide iterations among threads?

Compute the next frame of a computer-animated film

Loop over rows
  Loop over pixels in a row (columns)
  Compute pixel

Search through library catalog to find how many books have “loop” in the title

Loop over letters of alphabet
  Loop over authors with first letter in name
  Loop over books by author
  Increment global counter if title has “loop”
The Boss/Worker Pattern

Problem

- How should a program be organized when the design is dominated by the need to dynamically balance the work on a set of tasks?

Context

- Balancing task loads dominates even serial portion and overhead
  - Task workloads are variable or unpredictable
  - Computationally intense portions of code don’t map to simple loops
  - Capabilities of cores across system are varied, changeable, unpredictable
- Particularly relevant for Task Parallelism pattern

Forces

- Unpredictable work per task requires dynamic assignment logic
- Load balancing operations create synchronization overhead
  - Fewer and larger tasks may reduce this, but restricts options
- Trade-off between optimal load balance and maintainable code
The Boss/Worker Pattern - Solution

Boss Thread
- Initiates the computation
- Controls distribution of tasks to be assigned worker threads

Worker Threads
- Loop while more tasks to be done
  - Accept tasks from Boss thread
  - Execute each task assigned

Programming Considerations
- Task distribution
- Detecting completion and disposition of workers
The Boss/Worker Pattern Task Distribution

Rendezvous
- Two threads “meet” to exchange data
  - Symmetrical waiting (first thread to arrive must wait for second)
  - Single two-way transaction is completed
  - Mutual exclusion of other threads attempting rendezvous

Bag of tasks
- Shared data structure (queue) of tasks
  - Boss thread loads up tasks into shared structure
  - Worker threads remove new task when more work is needed
- Producer/Consumer: Boss continues to add tasks while Workers run

Shared Monotonic Counter
- Boss sets up indexing of tasks
- Workers access (and increment) counter to get next task
The Boss/Worker Pattern Detecting Completion

Rendezvous
- Send termination task (*poison pill*) to each worker
- Upon task receipt, workers check for new work or termination task

Bag of tasks
- If all tasks initially loaded into bag, then empty bag signals computation is complete
- Add termination task into queue after all computation tasks
  - For simplicity, add one termination task per worker thread

Shared Monotonic Counter
- Counter greater than number of known tasks signals completion
CASE STUDY: Rendezvous Implementation Boss Code

LockSyncObject(BWLock); // sync access to num_waiting counter
num_waiting++;
if (num_waiting != 2) { // if no worker waiting...
    UnlockSyncObject(BWLock);
    SleepUntilWorkerArrives(); // ...wait
}
else {
    WakeUpSleepingWorkerThread(); // wake up worker
    num_waiting = 0; // reset for next rendezvous
    UnlockSyncObject(BWLock);
}

<TRANSFER DATA, ASSIGN NEW TASK OR SEND TERMINATION>
LockSyncObject(WorkerLock);  // enforce exclusion of other workers
LockSyncObject(BWlock);  // sync access to num_waiting counter
num_waiting++;
if (num_waiting != 2) {  // if boss not waiting...
    UnlockSyncObject(BWLock);
    SleepUntilBossArrives();  // ...wait
}
else {
    WakeUpSleepingBossThread();  // wake up boss
    num_waiting = 0;  // reset for next rendezvous
    UnlockSyncObject(BWLock);
}
TRANSFER DATA, ASSIGN NEW TASK OR SEND TERMINATION>
UnlockSyncObject(WorkerLock);  // allow next worker to rendezvous
What type of task distribution method would be used on the following computations to divide iterations among threads?

Compute Mandlebrot set

Loop over rows
  Loop over pixels in a row (columns)
  Compute pixel

Search through library catalog to find how many books have “loop” in the title

Loop over letters of alphabet
  Loop over authors with first letter in name
  Loop over books by author
  Increment global counter if title has “loop”
Agenda

Pattern Language Structure

The Task Parallelism Pattern

The Geometric Decomposition Pattern

Supporting Structures

Summary