Parallel Programming
in C with MPI and OpenMP

OpenMP

- OpenMP: An application programming interface (API) for parallel programming on multiprocessors
  - Compiler directives
  - Library of support functions
- OpenMP works in conjunction with Fortran, C, or C++
Shared-memory Model

Processors interact and synchronize with each other through shared variables.

Fork/Join Parallelism

- Initially only master thread is active
- Master thread executes sequential code
- Fork: Master thread creates or awakens additional threads to execute parallel code
- Join: At end of parallel code created threads die or are suspended
Fork/Join Parallelism

Shared-memory Model vs. Message-passing Model (#1)

- Shared-memory model
  - Number active threads 1 at start and finish of program, changes dynamically during execution
- Message-passing model
  - All processes active throughout execution of program
Incremental Parallelization

- Sequential program a special case of a shared-memory parallel program
- Parallel shared-memory programs may only have a single parallel loop
- Incremental parallelization: process of converting a sequential program to a parallel program a little bit at a time

Shared-memory Model vs. Message-passing Model (#2)

- Shared-memory model
  - Execute and profile sequential program
  - Incrementally make it parallel
  - Stop when further effort not warranted
- Message-passing model
  - Sequential-to-parallel transformation requires major effort
  - Transformation done in one giant step rather than many tiny steps
Parallel for Loops

- C programs often express data-parallel operations as `for` loops
  
  ```c
  for (i = first; i < size; i += prime)
      marked[i] = 1;
  ```

- OpenMP makes it easy to indicate when the iterations of a loop may execute in parallel
- Compiler takes care of generating code that forks/joins threads and allocates the iterations to threads

Pragmas

-Pragma: a compiler directive in C or C++
- Stands for “pragmatic information”
- A way for the programmer to communicate with the compiler
- Compiler free to ignore pragmas
- Syntax:
  ```c
  #pragma omp <rest of pragma>
  ```
### Code transformation

- Every time the compiler finds a `#pragma omp parallel` directive creates a new function in which the code belonging to the scope of the pragma itself is **moved**.

- The directive is replaced with a call to a runtime function that is responsible for **forking** new threads (in a thread-based implementation) or for **loading parallel code** onto the slave processors.

- Once the parallel region has been executed, threads/processors need to **synchronize**.
Parallel for Pragma

- Format:
  
  ```c
  #pragma omp parallel for
  for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
  ```

- Compiler must be able to verify the runtime system will have information it needs to schedule loop iterations
Es #1 - pthreads

- Current implementation of GCC OpenMP runtime environment (libgomp) is basically a wrapper around the pthreads library.

- Master forks new worker threads with a call to the runtime function `GOMP_parallel_start`

- After parallel region master joins workers with a call to the runtime function `GOMP_parallel_end`
Current implementation of GCC OpenMP runtime environment (libgomp) is basically a wrapper around the pthreads library.

- Master forks new worker threads with a call to the runtime function `GOMP_parallel_start`.
- After parallel region, master joins workers with a call to the runtime function `GOMP_parallel_end`.

If `num_threads = 0`, determine the number of worker threads. Fork worker threads. Wait for all threads to be ready before starting parallel region.

We need to synchronize threads with a barrier at the end of a parallel region. Join worker threads and suspend them. Wait for all threads to be ready before starting parallel region.
Es #2 - MPARM

```c
void parallel_routine() {
    // sequential code
    for (i=0; i<N; i++)
        for (j=i; j<N; j++)
            A[i][j] = 1;
    // sequential code
}

void main() {
    initenv();
    if (cpuID == MASTER) {
        // gather workers on barrier
        start();
        // release workers
    } else {
        // spin until work provided
        parallel_routine();
        // spin until work provided
    }
}

void doall() {
    // release workers
    parallel_routine();
    // gather workers on barrier
    // Synchronization facilities
    // Lock Implementation
    // Barrier Implementation
}

int start() {
    // sequential code
    do_all();
    // sequential code
}
```

---

Functions for SPMD-style Programming

- The parallel pragma allows us to write SPMD-style programs
- In these programs we often need to know number of threads and thread ID number
- OpenMP provides functions to retrieve this information
Function omp_get_thread_num

- This function returns the thread identification number
- If there are $t$ threads, the ID numbers range from 0 to $t-1$
- The master thread has ID number 0

    int omp_get_thread_num (void)

Function omp_get_num_threads

- Function omp_get_num_threads returns the number of active threads
- If call this function from sequential portion of program, it will return 1

    int omp_get_num_threads (void)
for Pragma

- The **parallel** pragma instructs every thread to execute all of the code inside the block.
- If we encounter a **for** loop that we want to divide among threads, we use the **for** pragma.

```c
#pragma omp for
```

---

**Make shared data visible to all processors/threads**

- Replace uses of shared variables with corresponding field in shared data struct.

---

**GCC OPENMP EXPANSION DUMP**

- Replace uses of shared variables with corresponding field in shared data struct.
GCC OPENMP EXPANSION DUMP

Call runtime to determine number of threads

Call runtime to determine thread ID

Create work sharing by splitting loop iterations between threads

GCC OPENMP EXPANSION DUMP

COMPUTE LOWER AND UPPER BOUNDS FOR EACH THREAD

HOW?
It depends on what the schedule clause specifies (see after)

Initialize induction variable to lower bound

Create work sharing by splitting loop iterations between threads

Check termination condition on upper bound
Canonical Shape of for Loop Control Clause

\[
\text{for(index = start; index } \geq \begin{cases} < \\
\leq \\
> \end{cases} \text{ end;)} \begin{cases}
\text{index ++ } \\
\text{++index}
\end{cases}
\]

\[
\begin{cases}
\text{index -- } \\
\text{index } = \text{ inc}
\end{cases}
\]

\[
\begin{cases}
\text{index -- } = \text{ inc}
\end{cases}
\]

\[
\text{index } = \text{ index } + \text{ inc}
\]

\[
\text{index } = \text{ inc } + \text{index}
\]

\[
\text{index } = \text{ index } - \text{ inc}
\]

Shared and Private Variables

- Shared variable: has same address in execution context of every thread
- Private variable: has different address in execution context of every thread
- A thread cannot access the private variables of another thread
Shared and Private Variables

```c
int main (int argc, char *argv[]) {
    int b[3];
    char *cptr;
    int i;

    cptr = malloc(1);
    #pragma omp parallel for
    for (i = 0; i < 3; i++)
        b[i] = i;
}
```

---

Declaring Private Variables

```c
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
    for (j = 0; j < n; j++)
        a[i][j] = MIN(a[i][j], a[i][k]+tmp);
```

- Either loop could be executed in parallel
- We prefer to make outer loop parallel, to reduce number of forks/joins
- We then must give each thread its own private copy of variable j
private Clause

- Clause: an optional, additional component to a pragma
- Private clause: directs compiler to make one or more variables private

`private ( <variable list> )`

Example Use of private Clause

```c
#pragma omp parallel for private(j)
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
    for (j = 0; j < n; j++)
        a[i][j] = MIN(a[i][j], a[i][k] + tmp);
```
firstprivate Clause

- Used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered
- Variables are initialized once per thread, not once per loop iteration
- If a thread modifies a variable’s value in an iteration, subsequent iterations will get the modified value
Variable `tmp` is only accessible by the master thread. It has an initial value that slaves need to know. Slave threads will have a private copy of `tmp` initialized with this value. The init value is made visible to slaves through the shared data struct.

```c
...;
...
int i = ...
...;
...

data_0.i.a = ...
...;
...

data_0.i.b = ...
...;
...

_private data_0.i.a = ...
...;
...

_private data_0.i.b = ...
...;
...

_private data_0.i.c = ...
...

return D_298d;
...;
...
```

Private copy of `tmp` is initialized with this value.
lastprivate Clause

- Sequentially last iteration: iteration that occurs last when the loop is executed sequentially
- lastprivate clause: used to copy back to the master thread’s copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration

Variable tmp is only accessible by the master thread.

Its value will be written by the last thread that works on its private copy.
Its value will be written by the last thread that works on its private copy. Variable tmp is only accessible by the master thread. After all iterations have been executed...

...local value of tmp is copied into the shared data struct...

...and through this copied into the master's copy of tmp...
Critical Sections

double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
    x += (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;

Race Condition

- Consider this C program segment to compute π using the rectangle rule:

double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
Race Condition (cont.)

- If we simply parallelize the loop...

```c
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Race Condition (cont.)

- ... we set up a race condition in which one process may “race ahead” of another and not see its change to shared variable area

```
area 15.230  Answer should be 18.995
```

Thread A 15.432  Thread B 15.230

```
area += 4.0/(1.0 + x*x)
```
Race Condition Time Line

<table>
<thead>
<tr>
<th>Value of area</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.667</td>
<td></td>
<td>+ 3.765</td>
</tr>
<tr>
<td>11.667</td>
<td>+ 3.765</td>
<td></td>
</tr>
<tr>
<td>15.432</td>
<td></td>
<td>+ 3.563</td>
</tr>
<tr>
<td>15.230</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

critical Pragma

- Critical section: a portion of code that only thread at a time may execute
- We denote a critical section by putting the pragma

```c
#pragma omp critical
```

in front of a block of C code
Correct, But Inefficient, Code

double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
#pragma omp critical
    area += 4.0/(1.0 + x*x);
}
pi = area / n;

THIS IS DONE AT EVERY LOOP ITERATION!
Source of Inefficiency

- Update to area inside a critical section
- Only one thread at a time may execute the statement; i.e., it is sequential code
- Time to execute statement significant part of loop
- By Amdahl’s Law we know speedup will be severely constrained

Reductions

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to parallel for pragma
- Specify reduction operation and reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop
reduction Clause

- The reduction clause has this syntax:
  \[ \text{reduction} \ (<\text{op}> : \text{variable}>)? \]

- Operators
  - + Sum
  - * Product
  - & Bitwise and
  - | Bitwise or
  - ^ Bitwise exclusive or
  - && Logical and
  - || Logical or

π-finding Code with Reduction Clause

double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for 
    private(x) reduction(+:area)
for (i = 0; i < n; i++) {
    x = (i + 0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
Shared variable is updated at every iteration. This is NOT necessary

Shared variable is only updated at the end of the loop, when its final value is known

UNOPTIMIZED CODE
__sync_fetch_and_add(&omp_data_i->area, area);

This is a single atomic write. Target architecture may provide such an instruction

Shared variable only updated at the end of the loop, when its final value is known

UNOPTIMIZED CODE
__sync_fetch_and_add(i.omp_data_i->area, area);

This is a single atomic write. Target architecture may provide such an instruction

Shared variable only updated at the end of the loop, when its final value is known
Performance Improvement #1

- Too many fork/joins can lower performance
- Inverting loops may help performance if
  - Parallelism is in inner loop
  - After inversion, the outer loop can be made parallel
  - Inversion does not significantly lower cache hit rate

Performance Improvement #2

- If loop has too few iterations, fork/join overhead is greater than time savings from parallel execution
- The `if` clause instructs compiler to insert code that determines at run-time whether loop should be executed in parallel; e.g.,

  ```c
  #pragma omp parallel for if(n > 5000)
  ```
Check condition to determine whether to parallelize the loop or not

If it is true set NTHR = 0, otherwise set it to 1

Pass NTHR to runtime: if it equals 1 only one thread will execute it.

---

Performance Improvement #3

- We can use schedule clause to specify how iterations of a loop should be allocated to threads
- Static schedule: all iterations allocated to threads before any iterations executed
- Dynamic schedule: only some iterations allocated to threads at beginning of loop’s execution. Remaining iterations allocated to threads that complete their assigned iterations.
Static vs. Dynamic Scheduling

- Static scheduling
  - Low overhead
  - May exhibit high workload imbalance
- Dynamic scheduling
  - Higher overhead
  - Can reduce workload imbalance

Chunks

- A chunk is a contiguous range of iterations
- Increasing chunk size reduces overhead and may increase cache hit rate
- Decreasing chunk size allows finer balancing of workloads
USING THE `schedule` CLAUSE

- A parallel region has at least one *barrier* at its end, and may have additional barriers within it.
- At each barrier the other members of the team must *wait* for the last thread to arrive.
- To minimize this wait time shared work should be distributed so that all threads arrive at the barrier at about the *same time*.
- The choice of a schedule for a *for* construct is also determined by characteristics of the *memory* system (presence of caches, uniform access times, etc.).

```c
#pragma omp parallel
{
  #pragma omp for schedule (static)
  for (i=0; i<n; i++)
    a[i] = work1(i);

  #pragma omp for schedule (static)
  for (i=0; i<n; i++)
    if (i>=k) a[i] = work2(i);
}
```

**Assigning same iterations to same threads may improve data reuse.**
schedule Clause

- Syntax of schedule clause
  \texttt{schedule (<type>[,<chunk>] )}
- Schedule type required, chunk size optional
- Allowable schedule types
  - static: static allocation
  - dynamic: dynamic allocation
  - guided: guided self-scheduling
  - runtime: type chosen at run-time based on value of environment variable OMP\_SCHEDULE

Scheduling Options

- \texttt{schedule(static)}: block allocation of about \(n/t\) contiguous iterations to each thread
- \texttt{schedule(static,C)}: interleaved allocation of chunks of size \(C\) to threads
- \texttt{schedule(dynamic)}: dynamic one-at-a-time allocation of iterations to threads
- \texttt{schedule(dynamic,C)}: dynamic allocation of \(C\) iterations at a time to threads
SPLITTING LOOP ITERATIONS – schedule (static)

- The static schedule is appropriate for a parallel region containing a single for construct, with each iteration requiring the same amount of work.
- Loop boundaries are statically determined at compile time. No interaction with the runtime is required.

**Example:** N=10, Nthr=4.

<table>
<thead>
<tr>
<th>TID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>C * TID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>UB</td>
<td>min(C * (TID + 1), N)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Calculations:

- Compute chunk \( C = \text{ceil}(\frac{N}{Nthr}) \)
- Compute \( LB = C \times TID \)
- Compute \( UB = \text{min}\{C \times (TID + 1), N\} \)

**NOTE:** There may be LOAD IMBALANCE between threads (i.e. they operate on data chunks of different sizes).

---

**GCC OPENMP EXPANSION DUMP**

```
Compute chunk \( C = \text{ceil}(\frac{N}{Nthr}) \)
```

```
Compute \( LB = C \times TID \)
```

```
Compute \( UB = \text{min}\{C \times (TID + 1), N\} \)
```
Scheduling Options

- schedule(static): block allocation of about n/t contiguous iterations to each thread
- schedule(static,C): interleaved allocation of chunks of size C to threads
- schedule(dynamic): dynamic one-at-a-time allocation of iterations to threads
- schedule(dynamic,C): dynamic allocation of C iterations at a time to threads

**IN GENERAL:** Small chunks allow finer grained control on workload

```c
int main() {
    int a[24];
    int i;
    #pragma omp parallel
    #pragma omp for schedule (static)
    for (i = 0; i < 24; i++)
        a[i] = i;
}
```

```c
int main() {
    int a[24];
    int i;
    #pragma omp parallel
    #pragma omp for schedule (static, 2)
    for (i = 0; i < 24; i++)
        a[i] = i;
}
```

```c
int main() {
    int a[24];
    int i;
    #pragma omp parallel
    #pragma omp for schedule (static, 4)
    for (i = 0; i < 24; i++)
        a[i] = i;
}
```
Scheduling Options

- schedule(static): block allocation of about n/t contiguous iterations to each thread
- schedule(static,C): interleaved allocation of chunks of size C to threads
- schedule(dynamic): dynamic one-at-a-time allocation of iterations to threads
- schedule(dynamic,C): dynamic allocation of C iterations at a time to threads

SPLITTING LOOP ITERATIONS – schedule (dynamic, C)

- The **dynamic** schedule is appropriate for the case of a for construct with the iterations requiring varying, or even unpredictable, amounts of work
- Iterations are assigned one at a time to threads as they become available. This requires a strict cooperation with the runtime
- Runtime overhead can be reduced by specifying a chunk size \( k \) greater than 1, so that threads are assigned \( k \) at a time until fewer than \( k \) remain
SPLITTING LOOP ITERATIONS – schedule (dynamic, C)

• The dynamic schedule is appropriate for the case of a for construct with the iterations requiring varying, or even unpredictable, amounts of work.

• Iterations are assigned one at a time to threads as they become available.

• Runtime overhead can be reduced by specifying a chunk size $k$ greater than 1, so that threads are assigned $k$ at a time until fewer than $k$ remain.

Retrieve lower and upper bounds for current thread’s first iteration.

Call runtime

Iteration step

Chunk size

Absolute lower and upper bounds

Retrieve lower and upper bounds for current thread’s first iteration

Execute loop body over this iteration space..

..then compute next chunk’s iteration space
SPLITTING LOOP ITERATIONS – schedule (guided, C)

- The **guided** schedule is appropriate for the case in which the threads may arrive at varying times at a **for** construct with each iterations requiring about the same amounts of work.

- This can happen if, for example, the **for** construct is preceded by one or more **for** constructs with **nowait** clauses.

- The interaction with the runtime works much like the dynamic schedule, but the size of chunks is computed dividing remaining iterations among threads, and considering C as a minimum size for the chunk.
Scheduling Options (cont.)

- schedule(guided, C): dynamic allocation of chunks to tasks using guided self-scheduling heuristic. Initial chunks are bigger, later chunks are smaller, minimum chunk size is C.
- schedule(guided): guided self-scheduling with minimum chunk size 1
- schedule(runtime): schedule chosen at run-time based on value of OMP_SCHEDULE; Unix example:
  ```
  setenv OMP_SCHEDULE "static,1"
  ```

More General Data Parallelism

- Our focus has been on the parallelization of `for` loops
- Other opportunities for data parallelism
  - processing items on a “to do” list
  - `for` loop + additional code outside of loop
Processing a “To Do” List

Sequential Code (1/2)

```c
int main (int argc, char *argv[]) {
    struct job_struct *job_ptr;
    struct task_struct *task_ptr;

    ...
    task_ptr = get_next_task (&job_ptr);
    while (task_ptr != NULL) {
        complete_task (task_ptr);
        task_ptr = get_next_task (&job_ptr);
    }
    ...
}
```
Sequential Code (2/2)

```c
char *get_next_task(struct job_struct **job_ptr) {
    struct task_struct *answer;

    if (*job_ptr == NULL) answer = NULL;
    else {
        answer = (*job_ptr)->task;
        *job_ptr = (*job_ptr)->next;
    }
    return answer;
}
```

Parallelization Strategy

- Every thread should repeatedly take next task from list and complete it, until there are no more tasks
- We must ensure no two threads take same task from the list; i.e., must declare a critical section
Use of `parallel` Pragma

```c
#pragma omp parallel private(task_ptr)
{
    task_ptr = get_next_task (&job_ptr);
    while (task_ptr != NULL) {
        complete_task (task_ptr);
        task_ptr = get_next_task (&job_ptr);
    }
}
```

Critical Section for `get_next_task`

```c
char *get_next_task(struct job_struct **job_ptr) {
    struct task_struct *answer;
    #pragma omp critical
    {
        if (*job_ptr == NULL) answer = NULL;
        else {
            answer = (*job_ptr)->task;
            *job_ptr = (*job_ptr)->next;
        }
    }
    return answer;
}
```
Example Use of for Pragma

```c
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        printf ("Exiting (%d)\n", i);
        break;
    }
#pragma omp for
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i])/b[i];
}
```

single Pragma

- Suppose we only want to see the output once
- The `single` pragma directs compiler that only a single thread should execute the block of code the pragma precedes
- Syntax:

```c
#pragma omp single
```
Use of single Pragma

```c
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        #pragma omp single
        printf ("Exiting (%d)\n", i);
        break;
    }
    #pragma omp for
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i])/b[i];
}
```

nowait Clause

- Compiler puts a barrier synchronization at end of every parallel for statement
- In our example, this is necessary: if a thread leaves loop and changes low or high, it may affect behavior of another thread
- If we make these private variables, then it would be okay to let threads move ahead, which could reduce execution time
Use of nowait Clause

```c
#pragma omp parallel private(i,j,low,high)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        #pragma omp single
        printf ("Exiting (%d)\n", i);
        break;
    }
    #pragma omp for nowait
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i])/b[i];
}
```

Functional Parallelism

- To this point all of our focus has been on exploiting data parallelism
- OpenMP allows us to assign different threads to different portions of code (functional parallelism)
Functional Parallelism Example

v = alpha();
w = beta();
x = gamma(v, w);
y = delta();
printf(”%6.2f\n”, epsilon(x,y));

May execute alpha, beta, and delta in parallel

parallel sections Pragma

- Precedes a block of $k$ blocks of code that may be executed concurrently by $k$ threads
- Syntax:
  
  #pragma omp parallel sections
section Pragma

- Precedes each block of code within the encompassing block preceded by the parallel sections pragma
- May be omitted for first parallel section after the parallel sections pragma
- Syntax:

  ```c
  #pragma omp section
  ```

Example of parallel sections

```c
data
#pragma omp parallel sections
{
  #pragma omp section /* Optional */
  v = alpha();
  #pragma omp section
  w = beta();
  #pragma omp section
  y = delta();
}
  x = gamma(v, w);
  printf ("%6.2f\n", epsilon(x,y));
```
Another Approach

Execute alpha and beta in parallel.
Execute gamma and delta in parallel.

sectionsPragma

- Appears inside a parallel block of code
- Has same meaning as the `parallel sections` pragma
- If multiple `sections` pragmas inside one parallel block, may reduce fork/join costs
Use of sections Pragma

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        v = alpha();
        #pragma omp section
        w = beta();
    }
    #pragma omp sections
    {
        x = gamma(v, w);
        #pragma omp section
        y = delta();
    }
}
printf ("%.6f\n", epsilon(x,y));
```
49

.. specifying number of sections..

Call runtime to obtain consecutive scheduling ids

switch these ids to make execution jump to the code corresponding to the relative #pragma omp section

---

Summary (1/3)

- OpenMP an API for shared-memory parallel programming
- Shared-memory model based on fork/join parallelism
- Data parallelism
  - parallel for pragma
  - reduction clause
Summary (2/3)

- Functional parallelism (parallel sections pragma)
- SPMD-style programming (parallel pragma)
- Critical sections (critical pragma)
- Enhancing performance of parallel for loops
  - Inverting loops
  - Conditionally parallelizing loops
  - Changing loop scheduling

Summary (3/3)

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>OpenMP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suitable for multiprocessors</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Suitable for multicomputers</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Supports incremental parallelization</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Minimal extra code</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Explicit control of memory hierarchy</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>