Programming with Shared Memory
PART II

HPC Fall 2007
Prof. Robert van Engelen
Overview

- Parallel programming constructs
- Dependence analysis
- OpenMP
- Autoparallelization
- Further reading
Parallel Programming Constructs

- Declaring shared data, when private is implicit

  \[
  \text{shared int } x; \quad \text{A shared variable}
  \]
  \[
  \text{shared int } *p; \quad \text{Private pointer to a shared value}
  \]

- Declaring private data, when private is explicit

  \[
  \text{private int } x;
  \]
  \[
  \text{private int } *p; \quad \text{Would this make any sense?}
  \]
Parallel Programming
Constructs

- The `par` construct

```
par {
    S1;
    S2;
    ...
    Sn;
}
```

*Statements in the body are executed concurrently*
Parallel Programming Constructs

- The **forall** construct (also called **parfor**)

```c
forall (i=0; i<n; i++) {
    S1;
    S2;
    ...
    Sm;
}
```

Statements in the body are executed in serial order by n threads
Sequential Consistency

- **Sequential consistency**: the result of a parallel program is always the same as the sequential program, irrespective of the statement interleaving that is a result of parallel execution.

```
x=1;
...  
y=x+3;
...  
z=x+y;
```

```
x=
...  
a=x;
...  
z=x+y;
```

```
b=x;
...  
```

Output a, b, x, y, z

Any order
Bernstein’s Conditions

- $I_i$ is the set of memory locations read by process $P_i$
- $O_j$ is the set of memory locations altered by process $P_j$

- Processes $P_1$ and $P_2$ can be executed concurrently if all of the following conditions are met

\[
I_1 \cap O_2 = \emptyset \\
I_2 \cap O_1 = \emptyset \\
O_1 \cap O_2 = \emptyset
\]
Dependence Analysis

- Dependence analysis performed by a compiler determines that Bernstein’s conditions are not violated when optimizing and/or parallelizing a program

\[
\begin{align*}
\text{independent} & \quad \text{RAW} & \quad \text{WAR} & \quad \text{WAW} \\
\begin{array}{c}
P_1: \ A = x + y; \\
P_2: \ B = x + z; \\
I_1 \cap O_2 = \emptyset \\
I_2 \cap O_1 = \emptyset \\
O_1 \cap O_2 = \emptyset
\end{array}
& & \begin{array}{c}
P_1: \ A = x + y; \\
P_2: \ B = x + A; \\
I_1 \cap O_2 = \emptyset \\
I_2 \cap O_1 = \{A\} \\
O_1 \cap O_2 = \emptyset
\end{array}
& & \begin{array}{c}
P_1: \ A = x + B; \\
P_2: \ B = x + z; \\
I_1 \cap O_2 = \emptyset \\
I_2 \cap O_1 = \emptyset \\
O_1 \cap O_2 = \emptyset
\end{array}
& & \begin{array}{c}
P_1: \ A = x + y; \\
P_2: \ A = x + z; \\
I_1 \cap O_2 = \emptyset \\
I_2 \cap O_1 = \emptyset \\
O_1 \cap O_2 = \{A\}
\end{array}
\end{align*}
\]
Dependence Analysis

- Dependence analysis performed by a compiler determines that Bernstein’s conditions are not violated when optimizing and/or parallelizing a program.

\[
\begin{align*}
\text{independent} \\
P_1: & \quad A = x + y; \\
P_2: & \quad B = x + z; \\
I_1 \cap O_2 & = \emptyset \\
I_2 \cap O_1 & = \emptyset \\
O_1 \cap O_2 & = \emptyset
\end{align*}
\]

\[
\text{par} \{ \\
P_1: & \quad A = x + y; \\
P_2: & \quad B = x + z; \\
\}
\]

Instruction scheduling for instruction-level parallelism (ILP)
Dependence Analysis

- Dependence analysis performed by a compiler determines that Bernstein’s conditions are not violated when optimizing and/or parallelizing a program

\[
\forall i, j: 1 \leq i < j \leq 4 \\
I_i \cap O_j = \emptyset \\
I_j \cap O_i = \emptyset \\
O_i \cap O_j = \emptyset
\]

forall (I=1; I<5; I++) {
  \[P_1: A[I] = B[I];\]
}


No dependence

The forall loop should not have any cross-iteration dependences
**Dependence Analysis**

- Thus, a forall loop is valid when any interleaving of its parallel executed body yields the same result.

   ```
   forall (I=4; I<7; I++)
   ```


Dependence Analysis

- To understand dependences in loops, unroll

```
for (I=3; I<7; I++) {
  S1:  A[I] = B[I];
  S2:  C[I] = A[I-2];
}
```

- Flow dependence from $S1(3)$ and $S2(5)$
- Flow dependence from $S1(4)$ to $S2(6)$
Dependence Analysis

To understand dependences in loops, unroll

```
for (I=3; I<7; I++) {
  S1:  A[I] = B[I];
  S2:  C[I] = A[I-2];
}
```

```
for (I=3; I<7; I++) {
  S1:  A[I] = B[I];
  par {
    S2:  C[I] = A[I-2];
    S1:  A[I] = B[I];
  }
}
```


Any reordering is valid as long as dependences are not changed.

Flow dependence from S1(3) and S2(5)

Flow dependence from S1(4) to S2(6)
Dependence Analysis

To understand dependences in loops, unroll

```
for (I=3;I<7;I++) {
  S1: A[I] = B[I];
  S2: C[I] = A[I-2];
}
```

```
forall (I=3;I<7;I++) {
  S1: A[I] = B[I];
  S2: C[I] = A[I-2];
}
```

Is this correct?

Any reordering is valid as long as dependences are not changed

Flow dependence from S1(3) and S2(5)

Flow dependence from S1(4) to S2(6)
OpenMP

- OpenMP is a portable implementation of common parallel constructs for shared memory machines

- OpenMP in C

  ```
  #pragma omp directive_name
  statement_block
  ```

- OpenMP in Fortran

  ```
  !$OMP directive_name
  statement_block
  !$OMP end directive_name
  ```
OpenMP

OpenMP Constructs

- OpenMP language extensions
  - parallel control structures
  - work sharing
  - data environment
  - synchronization
  - runtime functions, env. variables

- parallel directive
- do/parallel do directive
- shared and private clauses
- critical and atomic directives
- barrier directive

- omp_set_num_threads()
- omp_get_thread_num()
- OMP_NUM_THREADS
- OMP_SCHEDULE
OpenMP Parallel

- The `parallel` construct

```
#pragma omp parallel
{
   S1;
   S2;
   ...
   Sm;
}
```

*Starts a team of threads to execute the body statements and joins them when done*

Parallel region

```
One thread
       \\
        \\
threads
S1; S2; ... Sm;
S1; S2; ... Sm;
... 
S1; S2; ... Sm;
One thread
```
OpenMP Parallel

- The `parallel` construct

```
#pragma omp parallel default(none) shared(vars)
{
    S1;
    S2;
    ... 
    Sm;
}
```

This specifies that variables should not be assumed to be shared by default

Parallel region

```
threads
  ▾ S1; S2; ... Sm;
  ▾ S1; S2; ... Sm;
  ▾ ... 
  ▾ S1; S2; ... Sm;
```

One thread

One thread
OpenMP Parallel

- The `parallel` construct

```
#pragma omp parallel private(n, i)
{
    n = omp_get_num_threads();
    i = omp_get_thread_num();
    ...
}
```

*Use `private` to declare private data*

- `omp_get_num_threads()` returns the number of threads that are currently being used

- `omp_get_thread_num()` returns the thread id (0 to n-1)
OpenMP Parallel

- The `parallel` construct

\[ y = \sin(x); \]

```c
#pragma omp parallel firstprivate(y)
{
    ... y ...
}
```

*Use `firstprivate` to declare private variables that are initialized with the master thread’s value of the variables*

*Likewise, use `lastprivate` to declare private variables whose values are copied back out to master thread’s variables*
OpenMP Parallel

- The `parallel` construct

```c
#pragma omp parallel reduction(operator:vars)
{
    var = expr;
    ...
}
```

Performs a global reduction over privatized variables and assigns final value to master’s private variable(s) or to the shared variable(s) when shared
OpenMP Parallel

- The \texttt{parallel} construct

```c
#pragma omp parallel num_threads(n)
{
  S1;
  S2;
  ...
  Sm;
}
```

Alternatively, use \texttt{omp_set_num_threads()} or set environment variable \texttt{OMP_NUM_THREADS}

```c
S1; S2; ...
Sm;
```

One thread \hspace{2cm} n threads \hspace{2cm} One thread

Number of threads
OpenMP Parallel Sections

- The **sections** construct is for *work-sharing*, where a current team of threads is used to execute statements concurrently.

```c
#pragma omp parallel
...
#pragma omp sections
{
    #pragma omp section
    S1;
    #pragma omp section
    S2;
    ...
    #pragma omp section
    Sm;
}
```

*Statements in the sections are executed concurrently*

- **n threads** executing **m sections** with a **barrier**
OpenMP Parallel Sections

- The sections construct is for work-sharing, where a current team of threads is used to execute statements concurrently.

```
#ifdef omp parallel
...#pragma omp sections nowait
{
  #pragma omp section
  S1;
  #pragma omp section
  S2;
  ...#pragma omp section
  Sm;
}
```

Use `nowait` to remove the implicit barrier.

$n$ threads executing $m$ sections

```
SSS
```

Use nowait to remove the implicit barrier.
OpenMP Parallel Sections

- The `sections` construct is for *work-sharing*, where a current team of threads is used to execute statements concurrently.

```c
#pragma omp parallel sections
{
    #pragma omp section
    S1;
    #pragma omp section
    S2;
    ...
    #pragma omp section
    Sm;
}
```

*Use parallel sections to combine parallel with sections*

*One thread*

*n threads executing m sections*

S1

S2

Sm
The `for` construct (do in Fortran) is for *work-sharing*, where a current team of threads is used to execute a loop concurrently.

```
#pragma omp parallel
...
#pragma omp for
for (i=0; i<=k; i++)
{
  S1;
  S2;
  ...
  Sm;
}
```

*Loop iterations are executed concurrently by n threads*

- n threads executing k iterations
  - $i=0; S1; S2; \ldots Sm$ $\rightarrow$
  - $i=1; S1; S2; \ldots Sm$ $\rightarrow$
  - $\ldots$ $\rightarrow$
  - $i=k; S1; S2; \ldots Sm$ $\rightarrow$
OpenMP For/Do

- The `for` construct is for work-sharing, where a current team of threads is used to execute a loop concurrently.

```c
#pragma omp parallel
...
#pragma omp for reduction(operator:vars)
for (i=0; i<=k; i++)
{
    S1;
    S2;
    ...
    Sm;
}
```

Performs a global reduction over privatized variables and assigns final value to master’s private variable(s) or to the shared variable(s) when shared.
OpenMP For/Do

- The **for** construct is for *work-sharing*, where a current team of threads is used to execute a loop concurrently.

```c
#pragma omp parallel
...
#pragma omp for schedule(dynamic)
for (i=0; i<=k; i++)
{
    S1;
    S2;
    ...
    Sm;
}
```

*When k>n, threads execute randomly chosen loop iterations until all iterations are completed.*

$n$ threads executing $k$ iterations
OpenMP For/Do

- The `for` construct is for work-sharing, where a current team of threads is used to execute a loop concurrently.

```plaintext
#pragma omp parallel
...
#pragma omp for schedule(static)
for (i=0; i<4; i++)
{
    S1;
    S2;
    ...
    Sm;
}
```

When $k > n$, threads are assigned to $\lceil k/n \rceil$ chunks of the iteration space.

2 threads executing 4 iterations:

- $i=0$: S1; S2; ... Sm;
- $i=1$: S1; S2; ... Sm;
- $i=2$: S1; S2; ... Sm;
- $i=3$: S1; S2; ... Sm;
OpenMP For/Do

- The `for` construct is for work-sharing, where a current team of threads is used to execute a loop concurrently.

```c
#pragma omp parallel
...
#pragma omp for schedule(static, 2)
for (i=0; i<8; i++)
{
    S1;
    S2;
    ...
    Sm;
}
```

2 threads executing 8 iterations using chunk size 2 in a round-robin fashion
OpenMP For/Do

- The `for` construct is for work-sharing, where a current team of threads is used to execute a loop concurrently.

```c
#pragma omp parallel
...
#pragma omp for schedule(guided, 4)
for (i=0; i<=k; i++)
{
    S1;
    S2;
    ...
    Sm;
}
```

Exponentially decreasing chunk size, for example: 4, 2, 1
OpenMP For/Do Scheduling Comparison

0  Loop iteration index

guided, 4

static, 4

static, 2

dynamic, 1

0  Loop iteration index
OpenMP For/Do Scheduling with Load Imbalances

Cost per iteration

0 Loop iteration index

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

static, 4

<table>
<thead>
<tr>
<th>load</th>
<th>load</th>
<th>load</th>
<th>load</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>20</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

time = 12

time = 20

time = 8

time = 32

dynamic, 2

<table>
<thead>
<tr>
<th>load</th>
<th>load</th>
<th>load</th>
<th>load</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>10</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

time = 4

time = 8

time = 10

time = 14

time = 18

time = 20

time = 22

time = 26
OpenMP For/Do

- The `for` construct is for *work-sharing*, where a current team of threads is used to execute a loop concurrently.

```c
#pragma omp parallel
...
#pragma omp for schedule(run_time)
for (i=0; i<=k; i++)
{
    S1;
    S2;
    ...
    Sm;
    // Controlled by environment variable OMP_SCHEDULE:
    setenv OMP_SCHEDULE "dynamic"
    setenv OMP_SCHEDULE "static"
    setenv OMP_SCHEDULE "static,2"
}```
OpenMP For/Do

- The `for` construct is for work-sharing, where a current team of threads is used to execute a loop concurrently.

```c
#pragma omp parallel for
for (i=0; i<k; i++)
{
    S1;
    S2;
    ...
    Sm;
}
```

Use `parallel for` to combine parallel with `for`.

- `n` threads executing `k` iterations.

```
> i=0; S1; S2; ... Sm;
> i=1; S1; S2; ... Sm;
> ...
> i=k; S1; S2; ... Sm;
```

One thread
OpenMP Single

- The **single** construct selects one thread of the current team of threads to execute the body.

```c
#pragma omp parallel
...
#pragma omp single
{
    S1;
    S2;
    ...
    Sm;
}
```

One thread executes the body
OpenMP Master

- The **master** construct selects the master thread of the current team of threads to execute the body

```plaintext
#pragma omp parallel
...
#pragma omp master
{
    S1;
    S2;
    ...
    Sm;
}
```

The “**master**” thread executes the body, no barrier is inserted
OpenMP Critical

- The `critical` construct defines a critical section

```c
#pragma omp parallel
...
#pragma omp critical name
{
  S1;
  S2;
  ...
  Sm;
}
```

*Mutual exclusion is enforced on the body using a named lock*

S1; S2; ... Sm; **wait** S1; S2; ... Sm;
OpenMP Critical

- The `critical` construct defines a critical section

```c
#pragma omp parallel
...
#pragma omp critical qlock
{ enqueue(job); }
...
#pragma omp critical qlock
{ dequeue(job); }
```

- `enqueue(job)`
- `dequeue(job)`
- `wait`
OpenMP Critical

- The critical construct defines a critical section

```
#pragma omp parallel
...
#pragma omp critical
{
  S1;
  S2;
  ...
  Sm;
}
```

Mutual exclusion is enforced on the body using an anonymous lock
OpenMP Barrier

- The **barrier** construct synchronizes the current team of threads

```c
#pragma omp parallel
...
#pragma omp barrier
```
OpenMP Atomic

- The **atomic** construct executes an expression atomically (expressions are restricted to simple updates)

```c
#pragma omp parallel
...
#pragma omp atomic
expression;
```
OpenMP Atomic

- The `atomic` construct executes an expression atomically (expressions are restricted to simple updates)

```
#pragma omp parallel
...
#pragma omp atomic
n = n+1;
...
#pragma omp atomic
n = n-1;
```

One thread is here

Another thread is here

\[ \Rightarrow n = n+1; \quad \Rightarrow n = n-1 \]
OpenMP Flush

- The `flush` construct flushes shared variables from local storage (registers, cache) to shared memory.
- OpenMP adopts a *relaxed consistency model* of memory.

```c
#pragma omp parallel
...
#pragma omp flush(variables)
```

\[
b = 3, \text{ but there is no guarantee that } a \text{ will be } 3
\]

flush

\[
\begin{align*}
n &= 3; \\
a &= n \\
b &= n
\end{align*}
\]
OpenMP Relaxed Consistency Memory Model

- *Relaxed consistency* means that memory updates made by one CPU may not be immediately visible to another CPU
  - Data can be in registers
  - Data can be in cache and no cache coherence protocol is used
- Therefore, the updated value of a shared variable that was set by a thread may not be available to another
- An OpenMP flush is automatically performed at
  - Entry and exit of *parallel* and *critical*
  - Exit of *for*
  - Exit of *sections*
  - Exit of *single*
  - Barriers
OpenMP Thread Scheduling

- Controlled by environment variable **OMP_DYNAMIC**
- When set to **FALSE**
  - Same number of threads used for every parallel region
- When set to **TRUE**
  - The number of threads is adjusted for each parallel region
  - `omp_get_num_threads()` returns actual number of threads
  - `omp_get_max_threads()` returns `OMP_NUM_THREADS`

```
Determine optimal number of threads
```

```
Parallel region
```

```
Determine optimal number of threads
```

```
Parallel region
```
OpenMP Threadprivate

- The `threadprivate` construct declares variables in a global scope private to a thread across multiple parallel regions

  - Must use when variables should stay private, even outside of the current scope, e.g. across function calls

  ```
  int counter;  // Global counter
  #pragma threadprivate(counter)

  #pragma omp parallel
  { counter = 0;  // Each thread has a local copy of counter
    ...
  }
  ...
  #pragma omp parallel
  { counter++;  // Each thread has a local copy of counter
    ...
  }
  ```
OpenMP Locks

- Mutex locks, with additional “nestable” versions of locks

```c
omp_lock_t lck;
omp_init_lock(&lck);
omp_set_lock(&lck);
...
... critical section ...
...
omp_unset_lock(&lck);
omp_destroy_lock(&lck);
```

- `omp_lock_t`:
  - the lock type

- `omp_init_lock()`:
  - initialization

- `omp_set_lock()`:
  - blocks until lock is acquired

- `omp_unset_lock()`:
  - releases the lock

- `omp_destroy_lock()`:
  - deallocates the lock
Compiler Options for OpenMP

- GOMP project for GCC 4.2 (C and Fortran)

- Use `#include <omp.h>`
  - Note: the `_OPENMP` define is set when compiling with OpenMP

- Intel compiler:
  - `icc -openmp ...`
  - `ifort -openmp`

- Sun compiler:
  - `suncc -xopenmp ...`
  - `f95 -xopenmp`
Autoparallelization

- Compiler applies dependence analysis and parallelizes a loop (or entire loop nest) automatically when possible
  - Typically parallelizes *outer loops* (more parallel work), possibly after loop interchange, fusion, etc.
  - Similar to adding `#pragma parallel for` to loop(s), with appropriate `private` and `shared` clauses

- Intel compiler:
  - `icc -parallel ...
  - `ifort -parallel ...

- Sun compiler:
  - `suncc -xautopar ...
  - `f95 -xautopar ...
Further Reading

- [PP2] pages 248-271

Optional:

OpenMP tutorial at Lawrence Livermore
http://www.llnl.gov/computing/tutorials/openMP/