Programming with Shared Memory
PART II

HPC Spring 2017
Prof. Robert van Engelen
Overview

- Sequential consistency
- Parallel programming constructs
- Dependence analysis
- OpenMP
- Autoparallelization
- Further reading
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.

```plaintext
Parallel program (a, b, x, y, z are shared)
```

```
a=5;
x=1;
...
y=x+3;
...
z=x+y;
```

```
a=5;
x=1;
...
y=x+3;
...
z=x+y;
```

```
x=1;
y=x+3;
...
a=5;
```

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

Output a, b, x, y, z

Any order of permitted statement interleavings
Data Flow: Implicitly Parallel

\[ a = 5; \]
\[ x = 1; \]
\[ y = x + 3; \]
\[ z = x + y; \]
\[ \text{...} \]
\[ \text{output } a, b, x, y, z \]

Flow dependences determine the parallel execution schedule: each operation waits until operands are produced.
Explicit Parallel Programming Constructs

- Declaring shared data, when private is implicit

  ```
  shared int x;  // A shared variable
  shared int *p;
  ```

- Declaring private data, when private is explicit

  ```
  private int x;
  private int *p;  // Would this make any sense?
  ```
Explicit Parallel Programming Constructs

- The `par` construct

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

Statements in the body are executed concurrently
Explicit 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 `i=0..n-1` in parallel.
Explicit Parallel: Many Choices, Which is Safe?

Think about data flow: each operation requires completion of operands first

Data dependences preserved  sequential consistency guaranteed
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:

\[
\begin{align*}
I_1 \cap O_2 &= \emptyset \\
I_2 \cap O_1 &= \emptyset \\
O_1 \cap O_2 &= \emptyset
\end{align*}
\]
Bernstein’s Conditions Verified by Dependence Analysis

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

<table>
<thead>
<tr>
<th>independent</th>
<th>RAW</th>
<th>WAR</th>
<th>WAW</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1: A = x + y ); ( P_2: B = x + z );</td>
<td>( P_1: A = x + y ); ( P_2: B = x + A );</td>
<td>( P_1: A = x + B ); ( P_2: B = x + z );</td>
<td>( P_1: A = x + y ); ( P_2: A = x + z );</td>
</tr>
<tr>
<td>( I_1 \cap O_2 = \emptyset )</td>
<td>( I_1 \cap O_2 = \emptyset )</td>
<td>( I_1 \cap O_2 = {B} )</td>
<td>( I_1 \cap O_2 = \emptyset )</td>
</tr>
<tr>
<td>( I_2 \cap O_1 = \emptyset )</td>
<td>( I_2 \cap O_1 = {A} )</td>
<td>( I_2 \cap O_1 = \emptyset )</td>
<td>( I_2 \cap O_1 = \emptyset )</td>
</tr>
<tr>
<td>( O_1 \cap O_2 = \emptyset )</td>
<td>( O_1 \cap O_2 = \emptyset )</td>
<td>( O_1 \cap O_2 = \emptyset )</td>
<td>( O_1 \cap O_2 = {A} )</td>
</tr>
</tbody>
</table>
Bernstein’s Conditions Verified by Dependence Analysis

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

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

\[
\begin{align*}
\text{par} & \{ \\
P_1: \quad & A = x + y; \\
P_2: \quad & B = x + z; \\
\} \\
\text{Recall:} & \\
\text{instruction scheduling for} \\
\text{instruction-level parallelism} & (ILP)
\end{align*}
\]
Bernstein’s Conditions in Loops

A parallel loop is valid when any ordering of its parallel body yields the same result

\[
\text{forall } (I=4; I<7; I++)
\]

\[
\]
OpenMP

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

- OpenMP in C

```c
#pragma omp directive_name
statement_block
```

- OpenMP in Fortran

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

- The parallel construct (in OpenMP is not the same as `par`)

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

A team of threads all execute the body statements and join when done

Parallel region

One thread (master thread)
OpenMP Parallel

The parallel construct

```c
#include <omp.h>

#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

One thread (master thread)
OpenMP Parallel

- The `parallel` construct

```c
#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)

What happens if `n` and/or `i` are not declared private?
Do Bernstein’s conditions still hold?
OpenMP Parallel with Reduction

- The parallel construct with reduction clause

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

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

```
var = expr;
var = expr;
...
var = expr;
```

One thread
OpenMP Parallel

- The parallel construct

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

Alternatively, use `omp_set_num_threads()` or set environment variable `OMP_NUM_THREADS`
OpenMP Parallel Sections

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

```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
OpenMP Parallel Sections

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

  ```
  #pragma 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

  ![Diagram showing n threads executing m sections]
OpenMP Parallel Sections

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

```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*

*n threads executing m sections*

One thread

S1

S2

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

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

Loop iterations are executed concurrently by $n$ threads.

Use `nowait` to remove the implicit barrier.

```
<table>
<thead>
<tr>
<th>n threads</th>
<th>barrier</th>
</tr>
</thead>
<tbody>
<tr>
<td>executing $k$ iterations</td>
<td></td>
</tr>
<tr>
<td>$i=0$: S1; S2; ... Sm;</td>
<td></td>
</tr>
<tr>
<td>$i=1$: S1; S2; ... Sm;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$i=k-1$: S1; S2; ... Sm;</td>
<td></td>
</tr>
</tbody>
</table>
```
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.*

```
\begin{array}{l}
\text{n threads executing k iterations}\ \\
\text{barrier}
\end{array}
```

```
\begin{array}{l}
\text{i=0: S1; S2; ... Sm;}
\text{i=1: S1; S2; ... Sm;}
\text{i=k-1: S1; S2; ... Sm;}
\end{array}
```
OpenMP For/Do

- The \textbf{for} construct is for \textit{work-sharing}, where a current team of threads is used to execute a loop concurrently

\begin{verbatim}
#pragma omp parallel
...
#pragma omp for schedule(static)
for (i=0; i<4; i++)
{
    S1;
    S2;
    ...
    Sm;
}
\end{verbatim}

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

2 threads executing 4 iterations

\begin{align*}
    i=0; & \quad S1; \quad S2; \quad \ldots \quad Sm; \\
    i=1; & \quad S1; \quad S2; \quad \ldots \quad Sm; \\
    i=2; & \quad S1; \quad S2; \quad \ldots \quad Sm; \\
    i=3; & \quad S1; \quad S2; \quad \ldots \quad Sm;
\end{align*}

\begin{tikzpicture}
    \node at (0,0) {2 threads executing 4 iterations};
    \node at (5,0) {barrier};
    \node at (0,-1) {$i=0; \quad S1; \quad S2; \quad \ldots \quad Sm;$};
    \node at (0,-2) {$i=1; \quad S1; \quad S2; \quad \ldots \quad Sm;$};
    \node at (0,-3) {$i=2; \quad S1; \quad S2; \quad \ldots \quad Sm;$};
    \node at (0,-4) {$i=3; \quad S1; \quad S2; \quad \ldots \quad Sm;$};
\end{tikzpicture}
OpenMP For/Do

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

```
#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, in this example: 4, 2, 1*
OpenMP For/Do Scheduling Comparison

0  Loop iteration index

- guided, 4
- static, 7
- static, 2
- dynamic, 1

P1  P2  P3  P4
OpenMP For/Do Scheduling with Load Imbalances

0 Loop iteration index

Cost per iteration

static, 4

load = 12
load = 20

load = 8
load = 22
load = 20
load = 8
load = 4

time = 12
time = 20
time = 8
time = 32

dynamic, 2

load = 4
load = 8
load = 10
load = 6
load = 2
load = 12
load = 10
load = 16
load = 4
load = 4
load = 4
load = 2
load = 2

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"
```
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(+:s)
for (i=0; i<k; i++)
{
    s += a[i];
}
```

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

- \( i=0: s += a[0]; \)
- \( i=1: s += a[1]; \)
- ...  
- \( i=k-1: s += a[k-1]; \)

Operation: +, *, -, &, ^, |, &&, ||
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

One thread

$n$ threads
executing $k$ iterations

- $i=0$: $S1; S2; \ldots Sm$
- $i=1$: $S1; S2; \ldots Sm$
- $\ldots$
- $i=k-1$: $S1; S2; \ldots Sm$
OpenMP Firstprivate and Lastprivate

The parallel construct with firstprivate and/or lastprivate clause

```c
x = ...;
#pragma omp parallel firstprivate(x) lastprivate(y)
{  x = x + ...;
   #pragma omp for
   for (i=0; i<k; i++)
   {  ...  
      y = i;
   }
}
... = y;
```

Use firstprivate to declare private variables that are initialized with the main thread’s value of the variables.

Likewise, use lastprivate to declare private variables whose values are copied back out to main thread’s variables by the thread that executes the last iteration of a parallel for loop, or the thread that executes the last parallel section.
OpenMP Single

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

```plaintext
#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

```cpp
#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

- Acquire
- Release

Diagram:

```
wait  S1; S2; ... Sm;  acquire  release
```
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);
}
```

One thread is here

Another thread is here
OpenMP Critical

- The critical construct defines a critical section

```bash
#pragma omp parallel
...

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

*Mutual exclusion is enforced on the body using an anonymous lock*

![Diagram showing the critical section with acquire and release operations]
OpenMP Barrier

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

```
#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;
```

indivisible

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*
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)
```

```
n = 3;
a = n
b = n
```

*flush*

*b = 3, but there is no guarantee that a will be 3*
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 (cache coherence protocol is slow or non-existent)
- 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`

![Diagram](image)

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

```c
int counter;                 // Global counter
#pragma omp 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

```
omp_lock_t lck;
omp_init_lock(&lck);
omp_set_lock(&lck);
...
... critical section ...
... omp_unset_lock(&lck);
omp_destroy_lock(&lck);
```
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 task-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/