Open Multi-Processing: Basic Course

Jerry Eriksson, Mikael Rännar and Pedro Ojeda

HPC2N,
Umeå University,
901 87, Sweden.

May 26, 2015
Table of contents

1 Overview of Parallelism
   - Parallelism Importance
   - Partitioning Data
   - Distributed Memory
   - Working on Abisko

2 OpenMP
   - Pragmas/Sentinels in OpenMP

3 Workshare constructs
   - Constructs, Parallel, For/Do, Section, Single

4 Synchronization constructs
   - Master, Critical, Barrier, Atomic

5 Data sharing
   - Runtime library
Application of Parallel algorithms

Molecular Dynamics

Simulations of Galaxies properties

Figure: AdK enzyme in water.

Figure: Galaxies [Nat., 509, 177 (2014)].
solution of this equation requires the knowledge of an array of particles’ positions and velocities

\[ \mathbf{F} = -\nabla U \quad \text{Newton's Law} \quad (1) \]

\[ \mathbf{X} = ( x_1^1, x_2^1, x_3^1, x_1^2, x_2^2, x_3^2 \ldots x_1^N, x_2^N, x_3^N ) \quad (2) \]

\[ \mathbf{V} = ( v_1^1, v_2^1, v_3^1, v_1^2, v_2^2, v_3^2 \ldots v_1^N, v_2^N, v_3^N ) \quad (3) \]
Working with arrays

\[ \mathbf{F} = -\nabla U \]  \hspace{1cm} \text{Newton’s Law} \hspace{1cm} (1)

solution of this equation requires the knowledge of an array of particles’ positions and velocities

\[ \mathbf{X} = (x_1^1, x_2^1, x_3^1, x_1^2, x_2^2, x_3^2 \ldots x_1^N, x_2^N, x_3^N) \] \hspace{1cm} (2)

\[ \mathbf{V} = (v_1^1, v_2^1, v_3^1, v_1^2, v_2^2, v_3^2 \ldots v_1^N, v_2^N, v_3^N) \] \hspace{1cm} (3)
Distributed Memory vs. Share Memory Systems

- Each **process** has a separate address space
- Processes communicate by explicitly sending and receiving messages

**Figure:** Distributed memory.

**Figure:** Shared memory.
Running jobs on Abisko

- Load Modules
- Compiling and linking
- Testing MPI programs
- Job submission
# OpenMPI for the PathScale compiler
module load psc

# OpenMPI for the GCC compiler
module load gcc

# OpenMPI for the Portland compiler
module load pgi

# OpenMPI for the Intel compiler
module load intel
Compiling and linking

• Compile with the appropriate OpenMP flag

Example:

# Executable: run.x
gcc/gfortran -fopenmp -o run.x main.c
Job submission

Template for a job script (script.sbatch):

```
#!/bin/bash
#SBATCH -A SNIC2015-7-15
#SBATCH --reservation SNIC2015-7-15
#SBATCH -n 1
#SBATCH --time=00:30:00
#SBATCH --error=job-%J.err
#SBATCH --output=job-%J.out
echo "Starting at ‘date‘"
srun ./run.x
echo "Stopping at ‘date‘"
```

Job submission:

```
sbatch script.sbatch
```
# Get the status of all your jobs
squeue -u <user>

# Get the predicted start of your queued jobs
squeue -u <user> --start

# Cancel a job
scancel <jobid>
OpenMP

A portable fork-join parallel model for architectures with shared memory

- Portable, Fortran, C/C++ bindings
- Many implementations
- Fork-join model
- Shared memory
- Ease of use, significant improvement with 3 or 4 directives
- Task parallelism and loop parallelism
OpenMP Resources

- www.openmp.org

- www.openmp.org/presentations/miguel/F95_OpenMPv1_v2.pdf
OpenMP Directive Format

```plaintext
#pragma omp name [clause[,[,] clause]...]```

- Each directive begins with `#pragma omp`
- followed by the `name` of the directive
- and a possibly empty list of `clauses`.
- The directive must end with a new line.
- Long directives may be split into multiple source lines by appending a backslash to continued lines.
OpenMP Constructs

Definition (Construct)

A **construct** consists of an **executable directive** and the associated **loop**, **statement**, or **structured block**.

Example:

```c
#pragma omp parallel
{
    // ..inside parallel construct..
    subroutine();
}
void subroutine( void )
{
    // ..outside parallel construct..
}
```
Example (Fortran):

```fortran
PROGRAM HELLO
$OMP PARALLEL

PRINT *, 'Hello World'

$OMP END PARALLEL

END
```
Parallel Constructs

Example (C):

```c
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    #pragma omp parallel
    {
        printf("Hello\nWorld\n");
    }
}
```
#pragma omp for [clauses]
for( init-exp ; test-exp ; inc-exp )
{ // ..loop body..  }

- Parallelizes a for loop or a for loop nest
- **Restrictions apply** to the three for loop expressions (Hint: The iteration count must be possible to compute before the loop (nest) is entered)
- The iterations must be **independent** (assumed and not checked)
- The mapping of iterations to threads can be influenced using the **schedule** clause. Schedules:
  - static, dynamic, guided, auto, and runtime
Parallel Constructs: Pi calculation (wrong)

Example (C):

```c
int main(void){
    double pi, x;
    int i, N;
    pi=0.0;
    N=1000;
    #pragma omp parallel for private(x)
    for(i=0; i<N; i++){
        x = (double)i/N;
        pi += 4 / (1 + x*x);
    }
    pi = pi/N;
    printf("Pi is \%f\n", pi);
}
```
Parallel Constructs: Pi calculation correct

Reduction option of "For/Do" loop Example (C):

```c
int main(void){
    double pi,x;
    int i,N;
    pi=0.0;
    N=1000;
    #pragma omp parallel for private(x) reduction(+:pi)
    for(i=0;i<N;i++){
        x=(double)i/N;
        pi+=4/(1+x*x);
    }
    pi=pi/N;
    printf("Pi is \%f \n",pi);
}
```
Parallel Constructs: Do loop

Example (C):

```c
X=0.0D0

!$OMP PARALLEL

!$OMP DO

DO I=1,NLIN
  DO J=1,NLIN
    X(I)=X(I)+I*J*1.0D0
    ENDDO
  ENDDO

!$OMP END DO

!$OMP END PARALLEL
```

Jerry Eriksson, Mikael Rännar and Pedro Ojeda

Open Multi-Processing: Basic Course
Parallel Constructs: Do loop wrong

Example (Fortran):

```fortran
!$OMP PARALLEL
!$OMP DO
  DO I=2,NLIN
    A(I)=2.0D0*A(I-1)
    PRINT *, 'EL.NR.',I,A(I)
  ENDDO
!$OMP END DO
!$OMP END PARALLEL
```
Parallel Constructs: Do loop correct

Example (Fortran):

```fortran
!$OMP PARALLEL
!$OMP DO ORDERED
DO I=2,NLIN
!$OMP ORDERED
A(I)=2.0D0*A(I-1)
!$OMP END ORDERED
PRINT *, 'EL.NR.',I,A(I)
ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

Jerry Eriksson, Mikael Rännar and Pedro Ojeda
Open Multi-Processing: Basic Course
Parallel Constructs: SAXPY

Example (Fortran):

```fortran
A=1.0; Y=1.0
DO I=1,N
   X(I)=1.0*I
ENDDO

!$OMP PARALLEL DO
DO I=1,N
   Z(I)=A*X(I)+Y
ENDDO

!$OMP END PARALLEL DO

!$OMP PARALLEL DO
DO I=1,N
   WRITE(6,*) Z(I)
ENDDO

!$OMP END PARALLEL DO
```

Jerry Eriksson, Mikael Rännar and Pedro Ojeda

Open Multi-Processing: Basic Course
Nested parallel regions

- Parallel regions can be **nested** in the sense that one parallel region is contained within another.
- Some implementations support it and some don’t.
- One major application of **nested parallelism** is to support parallel libraries in parallel programs.
Nested Parallel

Example (Fortran):

```fortran
PROGRAM HELLO
  !$OMP PARALLEL

  PRINT *, 'Hello'

  !$OMP PARALLEL

  PRINT *, 'Hi'

  !$OMP END PARALLEL

END
```

Jerry Eriksson, Mikael Rännar and Pedro Ojeda

Open Multi-Processing: Basic Course
Nested Parallel

Example (C):

```c
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    #pragma omp parallel
    {
        printf("Hello\n");
        #pragma omp parallel
        {
            printf("Hi\n");
        }
    }
}
```
Sections

Each thread can do an independent task for each section

Example (Fortran):

```fortran
PROGRAM HELLO
$OMP SECTIONS clauses...
$OMP SECTION
  ...task
$OMP SECTION
  ...task
$OMP SECTION
  ...task
$OMP END SECTIONS end_clauses
END
```
Only one thread can execute the task enclosed by this directive

Example (Fortran):

```fortran
PROGRAM HELLO

$OMP SINGLE clauses...

...task

$OMP END SINGLE end_clauses

END
```
To serialize some part of a parallel region, use the `master` directive.

**Examples:**

```c
#pragma omp master
{
    // ..only the master thread..
}
```
Critical sections

- OpenMP provides a construct for **critical sections** (mutual exclusion)
- Two forms: Unnamed and named
- Two critical sections with different names are unordered.
- All critical sections of the unnamed form use the same hidden lock and are ordered.

**Directive format:**

```
#pragma omp critical [name]
{
    // ..critical section..
}
```
Critical sections: Example

```c

task_t dequeue( void )
{
void run( void )
{
#pragma omp parallel
  { while( true ) {
    task_t task;
#pragma omp critical
    task = dequeue( );
    execute( task ); }
  }
}

Critical construct synchronize accesses to a shared queue.
```
Barrier

It is a construct to synchronize explicitly all the threads

!$OMP\ BARRIER\ \ \ (Fortran)$

`#pragma omp barrier\ (C)`
Atomic operations

- The atomic construct ensures atomic accesses to a specific storage location.
- Lightweight alternative to critical sections via critical or explicit locks in some situations.
- Probably mapped by the OpenMP implementation directly onto fast hardware atomic operations.

**Directive format (alt 1 of 2):**

```
#pragma omp atomic [type] expression-statement
```

where the optional type is one of:

`read`, `write`, `update`, `capture`
Atomic operations: Expression statements

An **expression statement** takes the form:

```plaintext
// If type=read
v = x;

// If type=write
x = expr;

// If type=update
x++; ++x; x--; --x;

x binop= expr; x = x binop expr;

// If type=capture
v = x++; v = x--; v = ++x; v = --x;

v = x binop= expr;
```
The atomic construct guarantees atomic operations regardless of the native word size. Expected to map to fast hardware atomic operations when available.

- atomic `read` performs an atomic read
- atomic `write` performs an atomic write
- atomic `update` performs an atomic read-modify-write update
- atomic `capture` performs an atomic read-modify-write update while also capturing the old or new value of the variable
#pragma omp atomic read
private = shared;

#pragma omp atomic update
counter += 1;

#pragma omp atomic capture
new_count = counter += 1;
Atomic example

Example (Fortran):

```fortran
PROGRAM ATOMIC
IMPLICIT NONE
INTEGER :: I
INTEGER, PARAMETER :: NLIN=10000000
REAL*8 :: X

X=0.0D0
!

$OMP PARALLEL DO
DO I=1,NLIN
    !$OMP ATOMIC
    X= X + I*1.0
ENDDO
!

$OMP END PARALLEL DO
WRITE(*,*) 'SUM=', X
END
```
Data sharing: Shared and private variables

- Variables are either **shared**, **private**, or **thread-private** (but more on thread-private variables later)
- The default can be specified using the `default` clause
- A **shared** variable is accessible to all threads and accesses must be synchronized if the shared variable is modified. Concurrent reads are okay.
- A **private** variable is accessible only to one thread.
- A **private** variable can be **reduced** to a new value in the master thread at the end of a region.
- A **private** variable can be initialized from the enclosing data environment with the `firstprivate` clause.
- A **private** variable can update the enclosing data environment with the `lastprivate` clause.
int k;
#pragma omp for
for( k = 0; k < 10; ++k )
{
    // ..k implied private by parallel for..
}
Data sharing: Example

```c
int k = 42;
#pragma omp parallel firstprivate(k)
{
    // ..k = 42 and private..
}
```
Data sharing: Example

```c
int k = 0;
#pragma omp parallel reduction(+: k)
{
    // ..k = 0 and implied private..
    k = omp_get_thread_num( );
}
// ..k = sum from 0 to nth-1..
```
A **thread-private variable** provides one instance of a variable for each thread.

The variable refers to a unique storage block in each thread.

Enables **persistent** private variables.

**Directive syntax:**

```c
int a, b;
#pragma omp threadprivate(a, b)
// .a and b are thread-private..
```
Thread-private variables: Example
Example A.27.1.c from OpenMP 3.1 spec

//
// Provides a per-thread counter.
//

int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter( void )
{
  ++counter;
  return counter;
}
OpenMP run-time library

Execution environment routines

- OMP_SET_NUM_THREADS
- OMP_GET_NUM_THREADS
- OMP_GET_MAX_THREADS
- OMP_GET_THREAD_NUM
- OMP_GET_NUM_PROCS
- OMP_SET_DYNAMIC
- OMP_SET_NESTED
OpenMP run-time library

Environment variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_DYNAMIC
- OMP_NESTED
The End!