

Programming Multi-Core Systems with OpenMP

Clemens Grelck

University of Amsterdam

UvA / SurfSARA

High Performance Computing and Big Data Course

June 2014



UNIVERSITY OF AMSTERDAM



Programming Multi-Core Systems with OpenMP

OpenMP at a Glance

Loop Parallelization

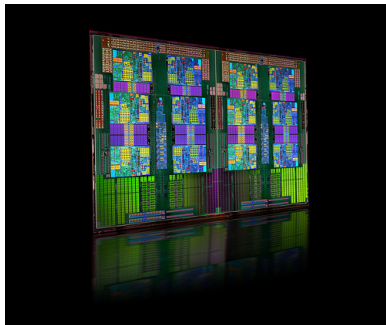
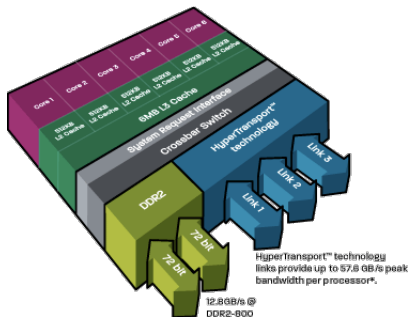
Scheduling

Outlook

Target Multi-core Systems

Small-scale general-purpose (x86) multicore processors:

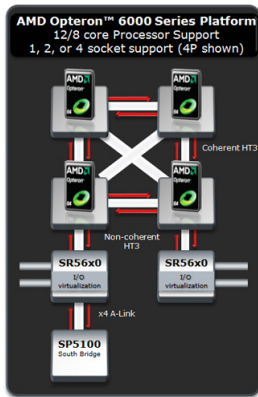
- ▶ Intel / AMD commodity processors with 2, 4, 6 or 8 cores
- ▶ potentially hyperthreaded



Target Multi-core Systems

Medium-scale server systems:

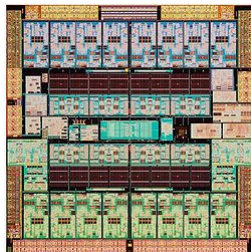
- ▶ multiple (2 or 4 in practice) identical processors
- ▶ each processor with several cores
- ▶ high bandwidth data path between processors



Target Multi-core Systems

Large-scale shared address space compute systems:

- ▶ large number of slightly simpler cores
- ▶ SUN Microsystems / Oracle Niagara / UltraSparc T series
- ▶ up to 512 hardware threads (T3-4 server)



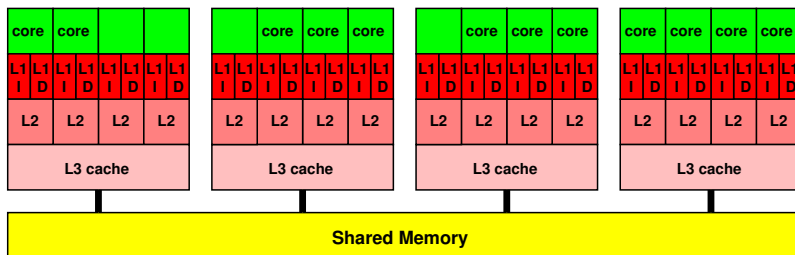
OpenMP Architectural Model

Multiple hardware threads:

Multiple cores:

Multiple processors:

Shared memory with cache coherence:



Design Rationale of OpenMP

Ideal:

- ▶ Automatic parallelisation of sequential code.
- ▶ No additional parallelisation effort for development, debugging, maintenance, etc.

Design Rationale of OpenMP

Ideal:

- ▶ Automatic parallelisation of sequential code.
- ▶ No additional parallelisation effort for development, debugging, maintenance, etc.

Problem:

- ▶ Data dependences are difficult to assess.
- ▶ Compilers must be conservative in their assumptions.

Design Rationale of OpenMP

Ideal:

- ▶ Automatic parallelisation of sequential code.
- ▶ No additional parallelisation effort for development, debugging, maintenance, etc.

Problem:

- ▶ Data dependences are difficult to assess.
- ▶ Compilers must be conservative in their assumptions.

Way out:

- ▶ Take or write ordinary sequential program.
- ▶ Add annotations/pragmas/compiler directives that guide parallelisation.
- ▶ Let the compiler generate the corresponding code.

OpenMP at a Glance

OpenMP as a programming interface:

- ▶ Compiler directives
- ▶ Library functions
- ▶ Environment variables

C/C++ version:

```
#pragma omp name [clause]*  
structured block
```

Fortran version:

```
!$ OMP name [clause [, clause]*]  
code block  
!$ OMP END name
```


Hello World with OpenMP

```
#include "omp.h"
#include <stdio.h>

int main()
{
    printf( "Starting execution with %d threads:\n",
           omp_get_num_threads());

    #pragma omp parallel
    {
        printf( "Hello world says thread %d of %d.\n",
               omp_get_thread_num(),
               omp_get_num_threads());
    }

    printf( "Execution of %d threads terminated.\n",
           omp_get_num_threads());

    return( 0);
}
```


Hello World with OpenMP

Compilation:

```
gcc -fopenmp hello_world.c
```

Output using 4 threads:

```
Starting execution with 1 threads:  
Hello world says thread 2 of 4.  
Hello world says thread 3 of 4.  
Hello world says thread 1 of 4.  
Hello world says thread 0 of 4.  
Execution of 1 threads terminated.
```


Hello World with OpenMP

Using 4 threads:

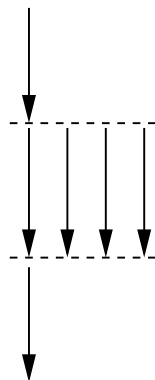
```
Starting execution with 1 threads:  
Hello world says thread 2 of 4.  
Hello world says thread 3 of 4.  
Hello world says thread 1 of 4.  
Hello world says thread 0 of 4.  
Execution of 1 threads terminated.
```

Who determines number of threads ?

- ▶ Environment variable: `export OMP_NUM_THREADS=4`
- ▶ Library function: `void omp_set_num_threads(int)`

OpenMP Execution Model

Classical Fork/Join:



Master thread executes serial code.

Master thread encounters parallel directive.

**Master and slave threads concurrently
execute parallel block.**

Implicit barrier, wait for all threads to finish.

Master thread resumes serial execution.

Programming Multi-Core Systems with OpenMP

OpenMP at a Glance

Loop Parallelization

Scheduling

Outlook

Simple Loop Parallelisation

Example: element-wise vector product:

```
void elem_prod( double *c, double *a, double *b, int len)
{
    int i;

    #pragma omp parallel for

    for (i=0; i<len; i++)
    {
        c[i] = a[i] * b[i];
    }
}
```


Simple Loop Parallelisation

Example: element-wise vector product:

```
void elem_prod( double *c, double *a, double *b, int len)
{
    int i;

    #pragma omp parallel for

    for (i=0; i<len; i++)
    {
        c[i] = a[i] * b[i];
    }
}
```

Prerequisite:

- ▶ No data dependence between any two iterations.

Simple Loop Parallelisation

Example: element-wise vector product:

```
void elem_prod( double *c, double *a, double *b, int len)
{
    int i;

    #pragma omp parallel for

    for (i=0; i<len; i++)
    {
        c[i] = a[i] * b[i];
    }
}
```

Prerequisite:

- ▶ No data dependence between any two iterations.
- ▶ **Caution: YOU claim this property !!**

Directive `#pragma omp parallel for`

What the compiler directive does for you:

- ▶ It starts additional worker threads depending on `OMP_NUM_THREADS`.
- ▶ It divides the iteration space among all threads.
- ▶ It lets all threads execute loop restricted to their mutually disjoint subsets.
- ▶ It synchronizes all threads at an implicit barrier.
- ▶ It terminates worker threads.

Directive `#pragma omp parallel for`

What the compiler directive does for you:

- ▶ It starts additional worker threads depending on `OMP_NUM_THREADS`.
- ▶ It divides the iteration space among all threads.
- ▶ It lets all threads execute loop restricted to their mutually disjoint subsets.
- ▶ It synchronizes all threads at an implicit barrier.
- ▶ It terminates worker threads.

Restrictions:

- ▶ The directive must directly precede for-loop.
- ▶ The for-loop must match a constrained pattern.
- ▶ The trip-count of the for-loop must be known in advance.

Shared and Private Variables

Example:

```
#pragma omp parallel for  
  
for (i=0; i<len; i++)  
{  
    res[i] = a[i] * b[i];  
}
```

- ▶ Shared variable: one instance for **all** threads
- ▶ Private variable: one instance for **each** thread

Shared and Private Variables

Example:

```
#pragma omp parallel for  
  
for (i=0; i<len; i++)  
{  
    res[i] = a[i] * b[i];  
}
```

Who decides that res, a, b, and len are shared variables, whereas i is private ??

Shared and Private Variables

Example:

```
#pragma omp parallel for  
  
for (i=0; i<len; i++)  
{  
    res[i] = a[i] * b[i];  
}
```

Who decides that **res**, **a**, **b**, and **len** are shared variables, whereas **i** is private ??

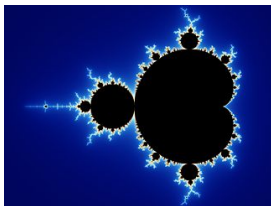
Default rules:

- ▶ All variables are **shared**.
- ▶ Only loop variables of parallel loops are **private**.

Parallelisation of a Less Simple Loop

Mandelbrot set:

```
double x, y;  
int i, j, max = 200;  
int depth[M,N];  
...  
for (i=0; i<M; i++) {  
    for (j=0; j<N; j++) {  
        x = (double) i / (double) M;  
        y = (double) j / (double) N;  
        depth[i,j] = mandelval( x, y, max);  
    }  
}
```



Parallelisation of a Less Simple Loop

Mandelbrot set:

```
double x, y;
int i, j, max = 200;
int depth[M,N];
...
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    } }
```

Properties to check:

- ▶ No data dependencies between loop iterations ?

Parallelisation of a Less Simple Loop

Mandelbrot set:

```
double x, y;
int i, j, max = 200;
int depth[M,N];
...
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    } }
```

Properties to check:

- ▶ No data dependencies between loop iterations ? **YES !**
- ▶ Trip-count known in advance ?

Parallelisation of a Less Simple Loop

Mandelbrot set:

```
double x, y;
int i, j, max = 200;
int depth[M,N];
...
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    } }
```

Properties to check:

- ▶ No data dependencies between loop iterations ? **YES !**
- ▶ Trip-count known in advance ? **YES !**
- ▶ Function `mandelval` without side-effects ?

Parallelisation of a Less Simple Loop

Function mandelval:

```
int mandelval( double xx, double yy, int max)
{
    int i =0;
    double x = xx, y = yy;

    while (x*x + y*y <= 4.0 && i < max) {
        x = x*x - y*y + xx;
        y = x*y + x*y + yy;
        i++;
    }

    return i;
}
```


Parallelisation of a Less Simple Loop

Mandelbrot set:

```
double x, y;
int i, j, max = 200;
int depth[M,N];
...
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    }
}
```

Properties to check:

- ▶ No data dependencies between loop iterations ? **YES !**
- ▶ Trip-count known in advance ? **YES !**
- ▶ Function `mandelval` without side-effects ? **YES !**
- ▶ Only loop variable `i` needs to be private ? **NO !!!!**

Check `x,y,j`

Parallelisation of a Less Simple Loop

Mandelbrot set:

```
double x, y;
int i, j, max = 200;
int depth[M,N];
...
#pragma omp parallel for private( x, y, j) shared( M, N, max)
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    }
}
```

Private clause:

- ▶ Directives may be refined by *clauses*.
- ▶ Private clause allows us to tag any variable as private.
- ▶ **Caution:** private variables are **not** initialised outside parallel section !!
- ▶ Shared clause allows us to explicitly mark shared variables.

Parallelisation of a Less, Less Simple Loop

Mandelbrot set with additional counter:

```
int total = 0;
...
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
        total = total + depth[i,j];
    }
}
```


Parallelisation of a Less, Less Simple Loop

Mandelbrot set with additional counter:

```
int total = 0;
...
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
        total = total + depth[i,j];
    }
}
```

Problems:

- ▶ New variable `total` introduces data dependence.
- ▶ Data dependence could be ignored due to associativity.
- ▶ New variable `total` must be shared.
- ▶ Incrementation of `total` must avoid race condition.

Parallelisation of a Less, Less Simple Loop

Mandelbrot set with additional counter:

```
int total = 0;
...
#pragma omp parallel for private( x, y, j)
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);

        #pragma omp critical
        {
            total = total + depth[i,j];
        }
    }
}
```


Critical Regions

The critical directive:

- ▶ Directive must immediately precede new statement block.
- ▶ Statement block is executed without interleaving.
- ▶ Directive implements critical region.

Equivalence:

```
#pragma omp critical  
{  
    <statements>  
}
```



```
pthread_mutex_lock( &lock);  
    <statements>  
pthread_mutex_unlock( &lock);
```


Critical Regions

The critical directive:

- ▶ Directive must immediately precede new statement block.
- ▶ Statement block is executed without interleaving.
- ▶ Directive implements critical region.

Equivalence:

```
#pragma omp critical
{
    <statements>
}
```



```
pthread_mutex_lock( &lock);
    <statements>
pthread_mutex_unlock( &lock);
```

Disadvantage:

- ▶ All critical regions in entire program are synchronised.
- ▶ Unnecessary overhead.

Critical Regions

The named critical directive

- ▶ Critical regions may be associated with names.
- ▶ Critical regions with identical names are synchronised.
- ▶ Critical regions with different names are executed concurrently.

Equivalence:

```
#pragma omp critical (name)
{
    <statements>
}
```



```
pthread_mutex_lock( &name_lock);
    <statements>
pthread_mutex_unlock( &name_lock);
```


Reduction Operations

Specific solution: reduction clause

```
#pragma omp parallel for private( x, y, i, j) \  
                                reduction(+:total)  
for (i=0; i<M; i++) {  
    for (j=0; j<N; j++) {  
        x = (double) i / (double) M;  
        y = (double) j / (double) N;  
        depth[i,j] = mandelval( x, y, max);  
        total = total + depth[i,j];  
    }  
}
```

Properties:

- ▶ Reduction clause only supports built-in reduction operations: $+$, $*$, $^$, $\&$, $|$, $\&\&$, $||$.
- ▶ User-defined reductions only supported via critical regions.
- ▶ Bit accuracy not guaranteed.

Shared and Private Variables Reloaded

Shared variables:

- ▶ One instance shared between sequential and parallel execution.
- ▶ Value unaffected by transition.

Private variables:

- ▶ One instance during sequential execution.
- ▶ One instance per worker thread during parallel execution.
- ▶ No exchange of values.

Shared and Private Variables Reloaded

Shared variables:

- ▶ One instance shared between sequential and parallel execution.
- ▶ Value unaffected by transition.

Private variables:

- ▶ One instance during sequential execution.
- ▶ One instance per worker thread during parallel execution.
- ▶ No exchange of values.

New: Firstprivate variables:

- ▶ Like private variables, but ...
- ▶ Worker thread instances initialised with master thread value.

Shared and Private Variables Reloaded

Example:

```
int a=1, b=2, c=3

#pragma omp parallel for private( a) \
                        firstprivate( b) \
                        shared(c)

for (i=0; i<10; i++) {
    // before first iteration:
    //   a : ?? | b : ?? | c : ??
    a++; b++; c=i;
}

//   a : ?? | b : ?? | c : ??
```


Shared and Private Variables Reloaded

Example:

```
int a=1, b=2, c=3

#pragma omp parallel for private( a) \
                        firstprivate( b) \
                        shared( c)

for (i=0; i<10; i++) {
    // before first iteration:
    //  a : undef | b : 2 | c : undef
    a++; b++; c=i;
}

//  a : 1 | b : 2 | c : undef
```


Conditional Parallelisation

Problem:

- ▶ Parallel execution of a loop incurs overhead:
 - ▶ creation of worker threads
 - ▶ scheduling
 - ▶ synchronisation barrier
- ▶ This overhead must be outweighed by sufficient workload.
- ▶ Workload depends on
 - ▶ loop body,
 - ▶ trip count.

Conditional Parallelisation

Problem:

- ▶ Parallel execution of a loop incurs overhead:
 - ▶ creation of worker threads
 - ▶ scheduling
 - ▶ synchronisation barrier
- ▶ This overhead must be outweighed by sufficient workload.
- ▶ Workload depends on
 - ▶ loop body,
 - ▶ trip count.

Example:

```
if (len < 1000) {  
    for (i=0; i<len; i++)  
    {  
        res[i] = a[i] * b[i];  
    }  
}  
else {  
    #pragma omp parallel for  
    for (i=0; i<len; i++)  
    {  
        res[i] = a[i] * b[i];  
    }  
}
```


Conditional Parallelisation

Introducing the if-clause:

```
if (len < 1000) {  
    for (i=0; i<len; i++) {  
        res[i] = a[i] * b[i];  
    }  
}  
else {  
    #pragma omp parallel for  
    for (i=0; i<len; i++) {  
        res[i] = a[i] * b[i];  
    }  
}
```



```
#pragma omp parallel for if (len >= 1000)  
for (i=0; i<len; i++) {  
    res[i] = a[i] * b[i];  
}
```


Programming Multi-Core Systems with OpenMP

OpenMP at a Glance

Loop Parallelization

Scheduling

Outlook

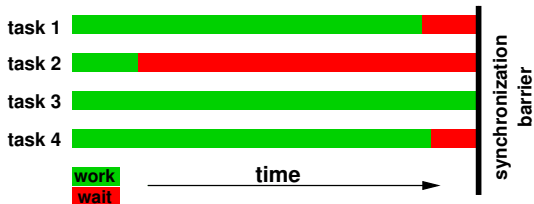
Loop Scheduling

Definition:

- ▶ Loop scheduling determines which iterations are executed by which thread.

Aim:

- ▶ Equal workload distribution



Loop Scheduling

Problem:

- ▶ Different situations require different techniques

The `schedule` clause:

```
#pragma omp parallel for schedule( <type> [, <chunk>])  
for (...)  
{  
    ...  
}
```

Properties:

- ▶ Clause selects one out of a set of scheduling techniques.
- ▶ Optionally, a chunk size can be specified.
- ▶ Default chunk size depends on scheduling technique.

Loop Scheduling

Static scheduling:

```
#pragma omp parallel for schedule( static)
```

- ▶ Loop is subdivided into as many chunks as threads exist.
- ▶ Often called **block scheduling**.

Loop Scheduling

Static scheduling:

```
#pragma omp parallel for schedule( static)
```

- ▶ Loop is subdivided into as many chunks as threads exist.
- ▶ Often called **block scheduling**.

Static scheduling with chunk size:

```
#pragma omp parallel for schedule( static, <n>)
```

- ▶ Loop is subdivided into chunks of n iterations.
- ▶ Chunks are assigned to threads in a round-robin fashion.
- ▶ Also called **block-cyclic scheduling**.

Loop Scheduling

Dynamic scheduling:

```
#pragma omp parallel for schedule( dynamic, <n>)
```

- ▶ Loop is subdivided into chunks of n iterations.
- ▶ Chunks are dynamically assigned to threads on their demand.
- ▶ Also called **self scheduling**.
- ▶ Default chunk size: 1 iteration.

Properties:

- ▶ Allows for dynamic load distribution and adjustment.
- ▶ Requires additional synchronization.
- ▶ Generates more overhead than static scheduling.

Loop Scheduling

Dilemma of chunk size selection:

- ▶ Small chunk sizes mean good load balancing, but high synchronisation overhead.
- ▶ Large chunk sizes reduce synchronisation overhead, but result in poor load balancing.

Loop Scheduling

Dilemma of chunk size selection:

- ▶ Small chunk sizes mean good load balancing, but high synchronisation overhead.
- ▶ Large chunk sizes reduce synchronisation overhead, but result in poor load balancing.

Rationale of guided scheduling:

- ▶ In the beginning, large chunks keep synchronisation overhead small.
- ▶ When approaching the final barrier, small chunks balance workload.

Loop Scheduling

Guided scheduling:

```
#pragma omp parallel for schedule( guided, <n>)
```

- ▶ Chunks are dynamically assigned to threads on their demand.
- ▶ Initial chunk size is implementation dependent.
- ▶ Chunk size decreases exponentially with every assignment.
- ▶ Also called **guided self scheduling**.
- ▶ Minimum chunk size: n (default: 1)

Example:

- ▶ Total number of iterations: 250
- ▶ Initial / minimal chunk size: 50 / 5
- ▶ Current chunk size: 80% of last chunk size:
50 – 40 – 32 – 26 – 21 – 17 – 14 – 12 – 10 – 8 – 6 – 5 – 5 – 4

Programming Multi-Core Systems with OpenMP

OpenMP at a Glance

Loop Parallelization

Scheduling

Outlook

What's More ?

More in OpenMP-2:

- ▶ Decouple parallel regions from work sharing
- ▶ Control synchronisation barriers
- ▶ Task parallel sections
- ▶ Low-level locks and condition variables
- ▶ ...

What's More ?

More in OpenMP-2:

- ▶ Decouple parallel regions from work sharing
- ▶ Control synchronisation barriers
- ▶ Task parallel sections
- ▶ Low-level locks and condition variables
- ▶ ...

More in OpenMP-3:

- ▶ Nested parallel regions
- ▶ Spawning and synchronisation of tasks
- ▶ ...

What's More ?

More in OpenMP-2:

- ▶ Decouple parallel regions from work sharing
- ▶ Control synchronisation barriers
- ▶ Task parallel sections
- ▶ Low-level locks and condition variables
- ▶ ...

More in OpenMP-3:

- ▶ Nested parallel regions
- ▶ Spawning and synchronisation of tasks
- ▶ ...

More information:

- ▶ www.openmp.org

The End: Questions ?

