University of Zurich[UZH]

High Performance Computing Lecture 6

Douglas Potter
Jozef Bucko
Noah Kubli

The OpenMP Library


Introduction


Basic principles


Work sharing


Synchronization


Performance


Conclusion

Introduction and history

Multi-threading using directives:

- Many vendors were using their own multi-threading directives (CRAY, NEC, IBM…) during the era of vector processing (mid-90s).

- On October 28th 1997, many vendors met and adopted the *Open Multi Processing* standard.
- OpenMP is specified by the Architecture Review Board (ARB).
- OpenMP 2.0: Nov 2000: modern Fortran constructs
- OpenMP 3.0: May 2008: task-based computing
- OpenMP 4.0: July 2013: external devices
- OpenMP 5.0: November 2018
- OpenMP 5.2: November 2021

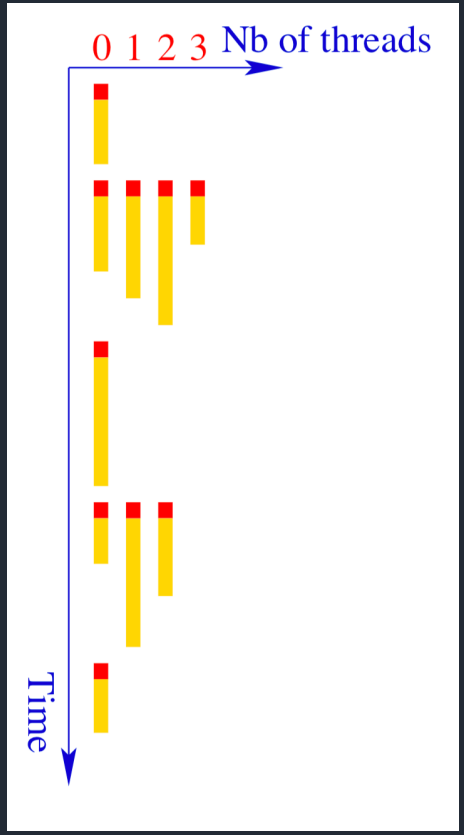
General concepts

Multi-threading

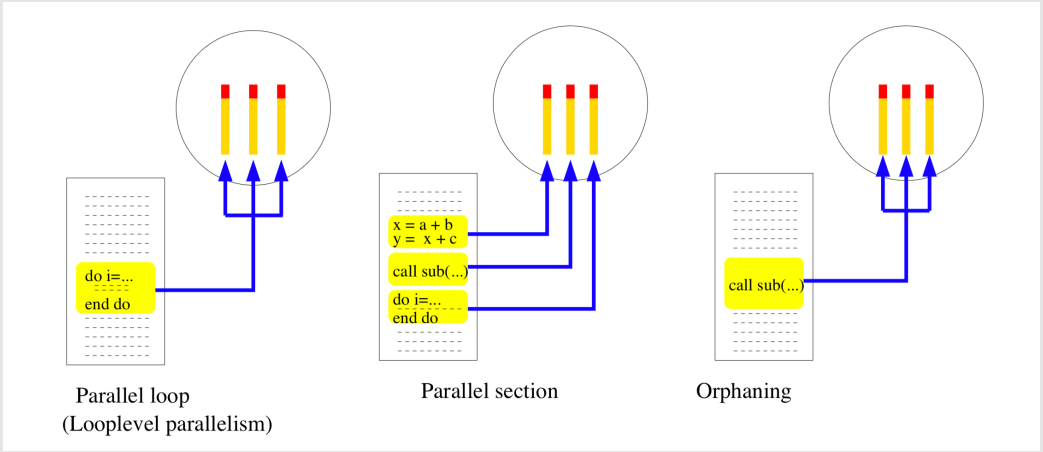
- An OpenMP program is executed by only one process, called the Master thread
- The master thread activates light-weight processes, called the workers or slave threads at the entry of a *parallel region*.
- Each thread executes a task corresponding to a block of instructions
- During the execution of the task, variables can be read from or updated in memory:
  - The variable can be defined in the local memory of the thread, called the stack memory, the variable is called a *private variable*.
  - The variable can be defined in the main shared (RAM) memory. It is called a *shared variable*.

Parallel regions

- A *sequential region* of the program is executed only by the master thread 0.
- A *parallel region* of the program is executed by several thread in parallel, the master and several workers.



Work sharing strategies



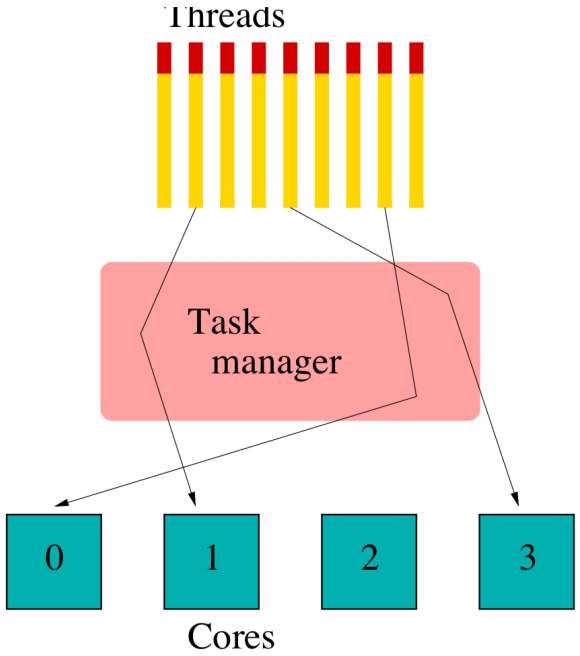
OpenMP uses three main work sharing paradigms

- Executing a loop by dividing up the basic operations between threads
- Executing different code sections, assigning one per thread
- Executing multiple instance of the same procedure, one per thread

Threads versus cores ?

The OpenMP runtime system launches threads that are mapped onto computing core by the operating system (Linux).

- Optimally, there is one thread per computing core (including the master thread)
- Worst case scenario, all threads are executed by only one core. The program is quasi-sequential.
- In practice, the situation is complex. Threads can be migrated across cores by the operating system.
- The runtime system can pin threads to cores, according to the rank (system dependent).



OpenMP structure

Compilation directives and clauses:

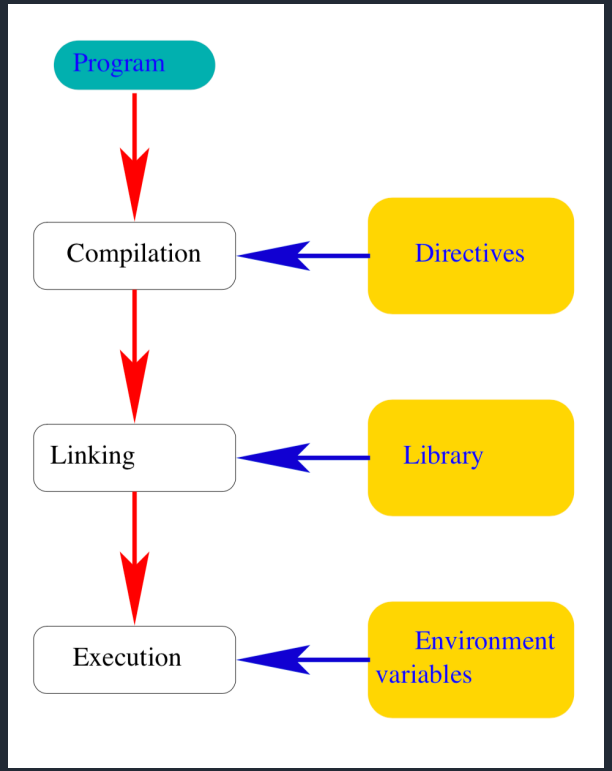
- They are put in the program to create the threads, define the work and data sharing strategy, and synchronize shared variables
- They are considered by the C, C++ or Fortran compilers as mere comment lines unless one specifies –openmp or –fopenmp on the compilation command line

Functions and routines:

- OpenMP contains several dedicated functions (like MPI). They are part of the OpenMP library than can be linked at link time.

Environment variables:

- OpenMP has several environment variables that can be set at execution time and changed the parallel computing behavior.



OpenMP structure

Documentation

- http://www.openmp.org/
- Book: Using OpenMP, by Chapman et al.

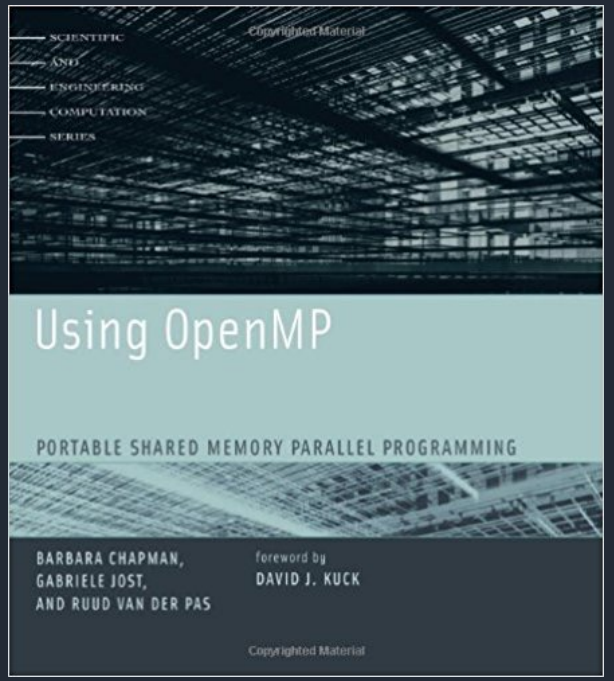
Example

- Compilation:

```
icc -openmp prog.c
gcc -fopenmp prog.c
```

- Environment variable:

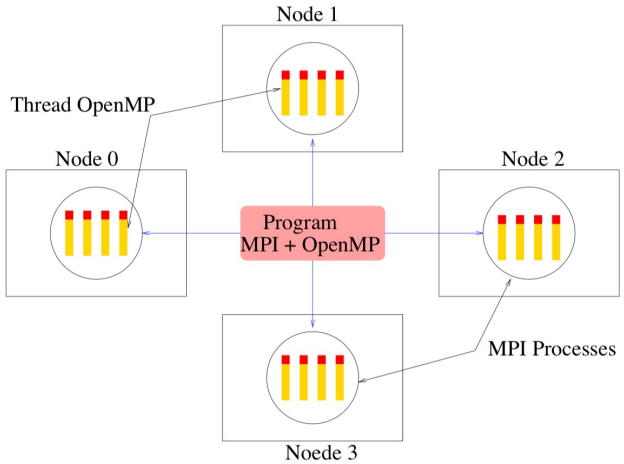
```
export OMP_NUM_THREADS=4
```



OpenMP versus MPI

- MPI is a multi-process model, for which communications between processes are explicit and under the responsibility of the programmer.
- OpenMP is a multithread model, within a single process. Communications between threads are implicit . The management of communications is under the responsibility of the compiler (and the operating system).

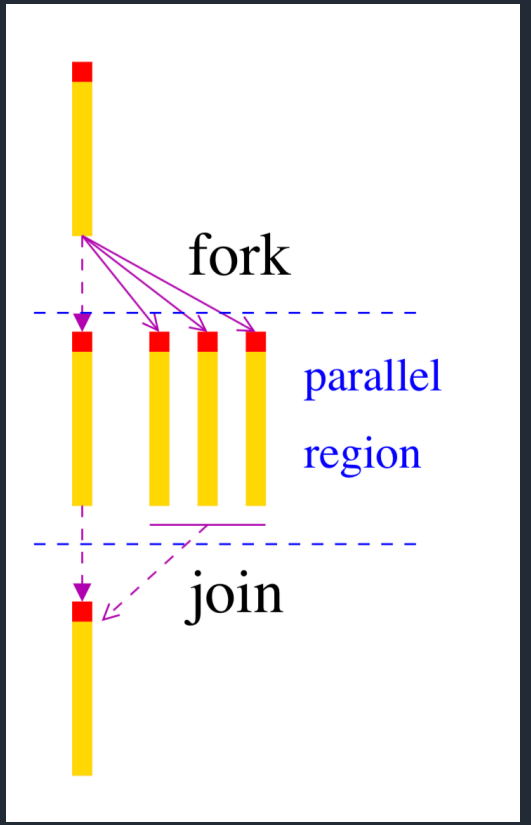
- MPI is used on distributed memory architectures (clusters with Infiniband network).
- OpenMP is used on shared-memory, multi-core architectures.
- On a cluster of many large shared-memory nodes, an hybrid approach (OpenMP within nodes and MPI across nodes) can be optimal



Basic principles of OpenMP

OpenMP uses directives

- The developer introduces these directives into their code where they see fit.
- At execution time, when the program hit a directive, the operating system creates a parallel regions, and spawns multiple threads. This is the so-called “fork-join” model.
- On entry, the master thread activates a team of children threads. On exit, these threads disappear, or hibernate, while only the master thread continue the execution.
- Careful: activating threads is expensive. Similar to the latency in MPI.



Directive Syntax

OpenMP directives for C/C++ are specified with `#pragma` directives. The syntax of an OpenMP directive is as follows:

> **`#pragma omp`** directive-name [clause[[,] clause] ...] new-line

Each directive starts with `#pragma omp`. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. In particular, white space can be used before and after the #, and sometimes white space must be used to separate the words in a directive. Preprocessing tokens following `#pragma omp` are subject to macro replacement.

Some OpenMP directives may be composed of consecutive `#pragma` directives if specified in their syntax.

Directives are case-sensitive.

Each of the expressions used in the OpenMP syntax inside of the clauses must be a valid *assignment-expression* of the base language unless otherwise specified.

OpenMP directives for Fortran are specified as follows:

> sentinel directive-name [clause[[,] clause]...]

All OpenMP compiler directives must begin with a directive sentinel. The format of a sentinel differs between fixed form and free form source files, as described in Section 2.1.1 on page 111 and Section 2.1.2 on page 112.

Directives are case insensitive. Directives cannot be embedded within continued statements, and statements cannot be embedded within directives.

Each of the expressions used in the OpenMP syntax inside of the clauses must be a valid *expression* of the base language unless otherwise specified.

In order to simplify the presentation, free form is used for the syntax of OpenMP directives for Fortran in the remainder of this document, except as noted.

- The directives are ignored by the compiler if OpenMP is disabled
  - The C compiler ignores unknown pragmas
  - The ”sentinel” in Fortran looks like a comment
- Disabled if unsupported or if “-fopenmp” not specified
- To use OpenMP functions one must include <omp.h> in C, or use the OMP_LIB module in Fortran. This is not required for directives.

Parallel Region

```
#include <stdio.h>
int main() {
    float a = 92290;
    #pragma omp parallel
    printf("a=%g\n",a);
    printf("done\n");
    return 0;
}
```

- Region is the statement after “parallel”
  - Use braces {} for multiple statements
- All threads execute the same code in parallel
- Variables are “shared” by default
- Implicit barrier after the region

```
> g++-10 -fopenmp -o parallel1
parallel1.cxx
> ./parallel1
a=92290
a=92290
a=92290
a=92290
a=92290
a=92290
a=92290
a=92290
a=92290
a=92290
a=92290
a=92290
a=92290
a=92290
a=92290
a=92290
done
```

Parallel Region

```c
#include <stdio.h>
int main() {
    float a = 92290;
    #pragma omp parallel
    printf("a=%g\n",a);
    printf("done\n");
    return 0;
}
```

```
> export OMP_NUM_THREADS=4
> ./parallel1
a=92290
a=92290
a=92290
a=92290
done
> OMP_NUM_THREADS=2 ./parallel1
a=92290
a=92290
done
```

- Use OMP_NUM_THREADS to set the number of threads in the parallel region.
- Default is usually the number of cores.

Parallel Region

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int main(int argc,char *argv[]) {
    float a = 92290;
    bool p = false;
    #pragma omp parallel
    {
#ifdef _OPENMP
        p = omp_in_parallel();
#endif
        printf("a=%g; p=%d\n",a,p);
    }
    return 0;
}
```

```
> g++-10 -fopenmp -o parallel2
parallel2.cxx
> OMP_NUM_THREADS=6 ./parallel2
a=92290; p=1
a=92290; p=1
a=92290; p=1
a=92290; p=1
a=92290; p=1
a=92290; p=1
> OMP_NUM_THREADS=1 ./parallel2
a=92290; p=0
> g++-10 -o parallel2
parallel2.cxx
> OMP_NUM_THREADS=6 ./parallel2
a=92290; p=0
```

Shared/Private

```
#include <stdio.h>
int main() {
    float a = 92290;
#pragma omp parallel default(none)
    {
        a = a + 290;
        printf("a=%g\n",a);
    }
    return 0;
}
```

- You can change the default to “none”
- Error for variables that are not explicitly set
- Helps catch errors
- Can be verbose

```
> g++-10 -fopenmp -o parallel3
parallel3.cxx
parallel3.cxx: In function 'int
main()':
parallel3.cxx:6:8: error: 'a'
not specified in enclosing
'parallel'
    6 |  a = a + 290;
      |      ~~^~~~
parallel3.cxx:4:13: error:
enclosing 'parallel'
    4 |     #pragma omp parallel
default(none)
      |             ^~~
```

private

```
#include <stdio.h>
int main() {
    float a = 92290;
    #pragma omp parallel\
         default(none) private(a)
    {
        a = a + 290;
        printf("a=%g\n",a);
    }
    return 0;
}
```

- **private** variables have a different value for each thread (thread local memory)
- May not be initialized by the compiler

```
> g++-10 -fopenmp -o parallel4
parallel4.cxx
> OMP_NUM_THREADS=8 ./parallel4
a=290
a=290
a=290
a=290
a=290
a=290
a=290
a=290
```

firstprivate

```
#include <stdio.h>
int main() {
    float a = 92290;
    #pragma omp parallel\
                default(none)\
                firstprivate(a)
    {
        a = a + 290;
        printf("a=%g\n",a);
    }
    return 0;
}
```

- Variables with **firstprivate** inherit the value of the global variable, but are then private.

```
> g++-10 -fopenmp -o parallel5
parallel5.cxx
> OMP_NUM_THREADS=8 ./parallel5
a=92580
a=92580
a=92580
a=92580
a=92580
a=92580
a=92580
a=92580
```

Extending region

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
void function(void) {
    bool p = false;
#ifdef _OPENMP
    p = omp_in_parallel();
#endif
    printf("%s\n", p ?
        "parallel" : "serial");
}
int main(int argc,char *argv[]) {
#pragma omp parallel
    function();
    return 0;
}
```

```
> OMP_NUM_THREADS=8 ./parallel6
parallel
parallel
parallel
parallel
parallel
parallel
parallel
parallel
> OMP_NUM_THREADS=1 ./parallel6
Serial
```

- The scope of the parallel region extends to the code that is lexically contained in this region (static extent) as well as to the code of the called routine (dynamical extent).

Extending region

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
void function(void) {
    bool p = false;
#ifdef _OPENMP
    p = omp_in_parallel();
#endif
    printf("%s\n", p ?
        "parallel" : "serial");
}
int main(int argc,char *argv[]) {
#pragma omp parallel
    function();
    function();
    return 0;
}
```

```
> OMP_NUM_THREADS=8 ./parallel6
parallel
parallel
parallel
parallel
parallel
parallel
parallel
parallel
serial
> OMP_NUM_THREADS=1 ./parallel6
serial
serial
```

Extending region

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int function(int x) {
    // Should guard with _OPENMP
    int v = x+omp_get_thread_num();
    return v;
}
int main(int argc,char *argv[]) {
    #pragma omp parallel
    {
        int y = function(92000);
        printf("y=%d\n",y);
    }
    return 0;
}
```

```
> g++-10 -fopenmp -o parallel7
parallel7.cxx
> OMP_NUM_THREADS=8 ./parallel7
y=92001
y=92002
y=92006
y=92000
y=92004
y=92005
y=92007
y=92003
```

- Some variables are always **private**
  - Local variables inside subroutines
  - Variables declared in the parallel region

Dynamical Forking

```c
#include <stdio.h>
int main(int argc,char *argv[]) {
    #pragma omp parallel num_threads(2)
    printf("hello\n");
    #pragma omp parallel num_threads(3)
    printf("hi\n");
    return 0;
}
```

- Use **num_threads** to set the number of threads inside a parallel region to override **OMP_NUM_THREADS**.
- You can call **omp_get_num_threads()** to get the number of threads in the parallel region.

```
> OMP_NUM_THREADS=8 ./parallel8
hello
hello
hi
hi
hi
> OMP_NUM_THREADS=2 ./parallel8
hello
hello
hi
hi
hi
> ./parallel8
hello
hello
hi
hi
hi
```

Sharing work in OpenMP

General ideas

- With all we have seen so far, you can program your own parallel code in a spirit similar to MPI.
- OpenMP proposes simple directives to distribute automatically work and data across threads.
- Loop level parallelism
- OpenMP directives to execute code only for one thread.

Parallel loop

- A parallel loop is a for-loop where each iteration is independent from the others
- Work decomposition by distributing the loop iterations
- The parallel loop is the one that follows immediately after a **for** directive
- Loops without loop indices or while loop are not supported by OpenMP
- The distribution strategy is set by a **schedule** clause.
- Proper scheduling can optimize the load-balancing of the work.
- Loop indices are always private integer variables.
- By default, the runtime sets a global synchronization the end of the loop
- The **nowait** clause can remove the barrier
- One can set many for directives inside a **parallel** region.

Parallel Loop

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int main(int argc,char *argv[]) {
    int n = 4096;
    #pragma omp parallel
    {
        int rank = omp_get_thread_num();
        #pragma omp for
        for(int i=0; i<n; ++i) {
            printf("%2d %5d\n",rank,i);
        }
    }
    return 0;
}
```

```
> OMP_NUM_THREADS=4 ./loop | more
 1  1024
 1  1025
 1  1026
 0     0
 0     1
 0     2
 2  2048
 2  2049
 2  2050
 2  2051
 2  2052
 2  2053
 2  2054
 2  2055
 2  2056
 2  2057
 2  2058
 2  2059
 2  2060
 3  3072
 3  3073
 3  3074
 3  3075
 3  3076
```

“parallel for”

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int main(int argc,char *argv[]) {
    int n = 4096;
    #pragma omp parallel for
    for(int i=0; i<n; ++i) {
        int rank = omp_get_thread_num();
        printf("%2d %5d\n",rank,i);
    }
    return 0;
}
```

- Combines **for** loop inside **parallel** region
- Common pattern and more compact
- Remember starting a parallel region is “expensive”
  - Might be better to have a region with several loops inside

```
> OMP_NUM_THREADS=4 ./loop | more
 2  2048
 2  2049
 2  2050
 1  1024
 1  1025
 1  1026
 1  1027
 0     0
 2  2051
 2  2052
 1  1028
 1  1029
 1  1030
 1  1031
 1  1032
 1  1033
 1  1034
 1  1035
 1  1036
 1  1037
 1  1038
 1  1039
 3  3072
 3  3073
```

Static Scheduling

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int main(int argc,char *argv[]) {
    int n = 4096;
    #pragma omp parallel
    {
        int rank = omp_get_thread_num();
        #pragma omp for schedule(static,128)
        for(int i=0; i<n; ++i) {
            printf("%2d %5d\n",rank,i);
        }
    }

    return 0;
}
```

```
> OMP_NUM_THREADS=4 ./loop
 0     0
 3   384
 3   385
 3   386
 2   256
 2   257
 2   258
 2   259
 2   262
 2   263
 0     1
 1   128
 1   129
…
 0   512
 0   513
…
 1   640
 1   641
…
 2   768
 2   769
…
 3   896
 3   897
```

Work scheduling

- The **schedule(static,N)** directive consists of dividing the loop into chunks of size N (except perhaps for the last one). The chunks are assigned to threads in a cyclic manner (round-robin way)
- The **schedule(runtime)** directives consists of postponing the choice of the work scheduling to execution time.
  - The adopted strategy is set at run time using the environment variable **OMP_SCHEDULE**
  - The optimal strategy depends on the problem size or on the simulation parameters, and can be decided just before execution
  - The optimal strategy is a balance between the size of each task (the bigger the better in term of overheads) and the granularity of the tasks (the smaller the better because of load balancing).
- The directive **schedule(dynamic,N)** consists of dividing the loop into chunks of size N, but they are assigned whenever a thread is available.
- The directive **schedule(guided,N)** consists of dividing the loop into chunks of exponentially decreasing size, but larger than N.

Parallel Loop

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int main(int argc,char *argv[]) {
    int n = 4096;
    int rank;
    #pragma omp parallel private(rank)
    {
        rank = omp_get_thread_num();
        #pragma omp for schedule(runtime)
        for(int i=0; i<n; ++i) {
            printf("%2d %5d\n",rank,i);
        }
    }

    return 0;
}
```

```
> export OMP_SCHEDULE="guided,16"
> OMP_NUM_THREADS=4 ./loop3 | more
 0     0
 1  2368
 0     1
 0     2
 3  1024
 3  1025
 2  1792
```

Reduction Operations

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int main(int argc,char *argv[]) {
    int n = 5;
    int s=1, p=1, r=1;
    #pragma omp parallel
    {
        #pragma omp for reduction(+:s)\
                reduction(*:p,r)
        for(int i=0; i<n; ++i) {
            s = s + 1;
            p = p * 2;
            r = r * 3;
        }
    }
    printf("s=%d p=%d r=%d\n",s,p,r);
    return 0;
}
```

```
> g++-10 -fopenmp -o reduce reduce.cxx
> ./reduce
s=6 p=32 r=243
```

- In a parallel loop, if one needs to perform a global operation, one uses the **`reduction`** clause.
- The operations can be:
  - Arithmetic: +, *, -
  - Logical/Boolean: &, |, ^, &&, ||
  - Predefined: max, min
  - User defined
- Each thread computes its partial result, which is then combined to the others at the end of the parallel loop.

Parallel loop: final details

- Shared variables can be declared as **private** to a loop only (not necessarily the entire parallel region) using **private(a)**
- Variables declared inside the parallel section are **private**
- Using **firstprivate(a)**, shared variable a is declared as **private** for the loop only, and it is assigned the value it had just before entering the loop, when it was still **shared**.
- Using **lastprivate(a)**, shared variable a is declared as **private** inside the loop only, and the value it got at the end of the last iteration of the loop is passed to its shared value after the loop.
- **parallel for** is the union of a **parallel** directive and a **for** directive. Be careful, the **parallel** directive is expensive because threads are spawned by the operating system.
- The end of a **parallel for** is always a barrier – **nowait** directive not possible – because you are leaving a parallel section at the end.

Exclusive execution

- The **sections** directive contains a set of consecutive program block, each block being assigned to different thread. Block are identified by a **section** construct. One can add a **nowait** clause.
- The **single** and **master** directives can be used to execute a code block with only one thread, either a random one or the master one.
- The end of a single block is a barrier for the other thread, unless a **nowait** clause is specified. One can broadcast a private variable using the **copyprivate** clause.
- The **master** clause executes the code block by the master thread only. There is no additional clause allowed, and there is no barrier.

Section

```
#include <omp.h>
#include <stdio.h>
int main() {
  #pragma omp parallel default(none)
  {
    printf("running on thread %d\n",omp_get_thread_num());
    #pragma omp sections
    {
      #pragma omp section
      {printf("Section 1 on thread %d\n",omp_get_thread_num());}
      #pragma omp section
      {printf("Section 2 on thread %d\n",omp_get_thread_num());}
      #pragma omp section
      {printf("Section 3 on thread %d\n",omp_get_thread_num());}
    }
    #pragma omp master
    {printf("master run on thread %d\n",omp_get_thread_num());}
    #pragma omp single
    {printf("single run on thread %d\n",omp_get_thread_num());}
  }
  return 0;
}
```

```
> OMP_NUM_THREADS=10 ./sections
running on thread 2
running on thread 4
Section 2 on thread 4
running on thread 1
Section 3 on thread 4
running on thread 5
running on thread 3
running on thread 8
Section 1 on thread 2
running on thread 0
running on thread 7
running on thread 6
running on thread 9
single run on thread 1
master run on thread 0
```

Synchronization with OpenMP

General concepts

- Synchronization is necessary to ensure that all threads have reached a given line of the program (global barrier).
- Synchronization is required when different threads are affecting the same shared variables in main memory.

- We have already seen implicit barriers at the end of many OpenMP constructs, these can be removed using the **nowait** clause.
- The directive **#pragma omp barrier** forces the synchronization of all threads within a parallel region.
- The directives **atomic** and **critical** can be used to force a serial variable update and avoid race conditions.

Atomic Update

- The **atomic** directive forces the shared variable access or update to be performed by one thread at a time.
- This is usually done with hardware support (special atomic instructions).
- This limits the operations that can be performed.

```
#pragma omp atomic [clause[[,] clause] ... ]  new-line
      statement
```

where *statement* is a statement with one of the following forms:

- If *atomic-clause* is **read** then *statement* is *read-expr-stmt*, a read expression statement that has the following form:

```
v = x;
```

- If *atomic-clause* is **write** then *statement* is *write-expr-stmt*, a write expression statement that has the following form:

```
x = expr;
```

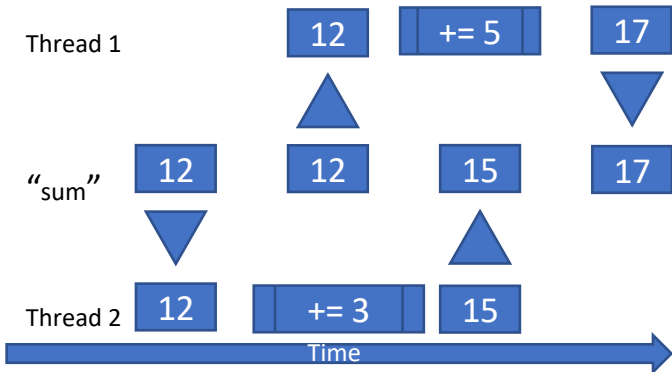
- If *atomic-clause* is **update** then *statement* can be *update-expr-stmt*, an update expression statement that has one of the following forms:

```
x++;
x--;
++x;
--x;
x binop= expr;
x = x binop expr;
x = expr binop x;
```

Update shared variable

```
#include <omp.h>
#include <stdio.h>
int main() {
  int counter = 92290;
  #pragma omp parallel
  {
    ++counter;
  }
  printf("Final counter is %d\n",counter);
  return 0;
}
```

```
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92389
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
```



Atomic update

```
#include <omp.h>
#include <stdio.h>
int main() {
  int counter = 92290;
  #pragma omp parallel
  {
    #pragma omp atomic
    ++counter;
  }
  printf("Final counter is %d\n",counter);
  return 0;
}
```

```
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
> OMP_NUM_THREADS=100 ./atomic
Final counter is 92390
```

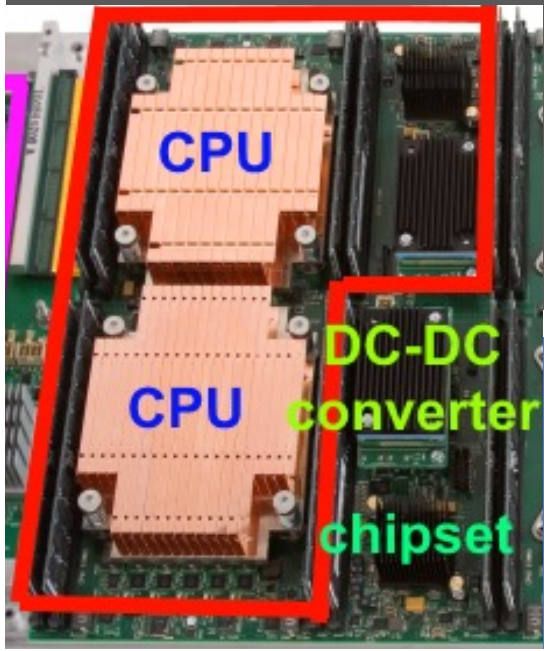
Critical region

```
#include <omp.h>
#include <stdio.h>
int main() {
  int counter = 92290;
  #pragma omp parallel
  {
    #pragma omp critical
    {
      ++counter;
    }
  }
  printf("Final counter is %d\n",counter);
  return 0;
}
```

```
> OMP_NUM_THREADS=100 ./critical
Final counter is 92390
> OMP_NUM_THREADS=100 ./critical
Final counter is 92390
> OMP_NUM_THREADS=100 ./critical
Final counter is 92390
> OMP_NUM_THREADS=100 ./critical
Final counter is 92390
…
```

- More general than **atomic**; anything in region.
- Order of execution is non-deterministic, but one thread at a time.
- Region is the next statement (or multiple statements when enclosed with braces).
- Less efficient that **atomic** for simple operations as it uses locking instead of hardware support.

Memory Issues



Memory access can be the dominate cost. Good OpenMP programs will be designed to access data in a way that maximizes the memory hierarchy. From fastest to slowest:

- CPU registers, L1 cache
- L2/L3 cache
- Main memory (RAM) on socket
- Main memory on a different socket

Conflicts between threads can lead to poor cache memory management (the so-called cache misses). Level 1 and 2 cache memory management is key to OpenMP performance.

Local cache memory is faster than main RAM memory. Mapping your data in memory is a key performance factor. On Linux, the memory is mapped to a given socket on a “first touch” basis. This is very important for “Non Uniform Memory Access” machines for which the main memory is not really shared but distributed across the node.

Performance analysis with OpenMP can be done using several functions provided by OMP_LIB to measure time. The most useful is:

time = omp_get_wtime() gives the elapsed time in seconds.

Conclusions

- OpenMP requires a multi-processor machine with shared memory.
- Parallelization is relatively easy to implement, starting from a sequential program (incremental parallelization)
- Within parallel regions, the work can be shared within loops or between sections.
- Parallel regions are extending to all subroutines called within.
- Synchronization is sometime necessary
- Data sharing is the key to good and robust parallelism.
- Performance is often related to the memory layout and the cache memory management.