Introducing OpenMP

Jonathan Ferland

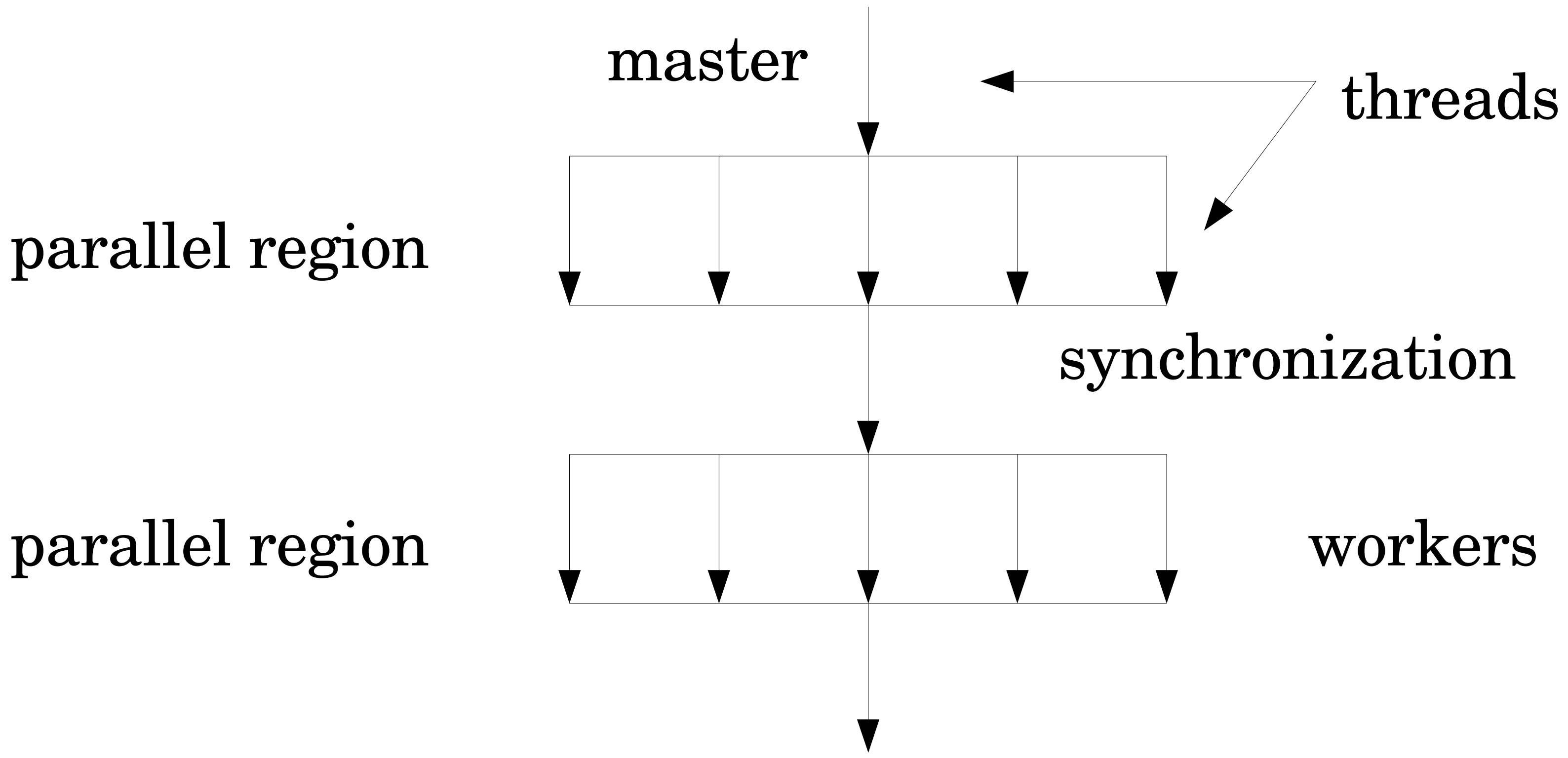
Scientific computing analyst

OpenMP objectives

- Standardization : available on all shared memory architectures.
- Simple interface : reduced instructions set (3 to 4) usually sufficient for parallelization.
- Ease of use : progressive integration possible.
- Portable: supported in many languages :

  Fortran (77,90,95), C (C90,C99) and C++

OpenMP model



OpenMP model cont.

- Master
  - Executes sequential part of the program
  - Participates in parallel region
  - Exists for program lifetime
- Workers/Children
  - In general, all of them execute the same task on a different part of the data
- Number of threads is independent on the number of processors.

First code

- To use OpenMP, the following is needed:

  #include <omp.h> (C)

  use omp_lib (Fortran 90,95)

  include 'omp_lib.h' (Fortran 77)

- A parallel region is defined by a delimiter (known by the compiler) and a "parallel" directive.

  #pragma omp parallel {} (C)

  !$omp parallel (Fortran)

  !$omp end parallel

First code cont.

- It might be useful to know a thread rank or the total number of threads inside the parallel region.

  total = omp_get_num_threads()

  rank = omp_get_thread_num()

- Variable scope is important.

  private(rank) shared(total)

- Let's now look at a simple “hello world” example using OpenMP.

Hello world! (C)

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif

int main (int argc, char ** argv)
{
    int total =1, rank=0;

#pragma omp parallel private(rank) shared(total)
  {
#ifdef _OPENMP
    total = omp_get_num_threads();
    rank = omp_get_thread_num();
#endif
    printf("Hello from thread %d in a group of %d threads\n",rank,total);
  }
}
```

Hello world! (Fortran)

```
program hello
!$ use omp_lib

integer total, rank

total=1
rank=0
!$omp parallel private(rank) shared(total)
!$   total = omp_get_num_threads()
!$   rank = omp_get_thread_num()
print *, "Hello from thread ", rank, " in a group of  ", total, " threads"
!$omp end parallel
end program hello
```

Observations

- Must include omp.h or use “use omp_lib”.
- Thread rank ranges from 0 up to total-1.
- The macro _OPENMP is defined when compiling with -openmp in C to exclude/include code. Sentinel syntax in Fortran allows for conditional inclusion of code.
  - Sentinel can be : !$ or C$ or *$ (fixed format)

    !$ (free form format)
- All threads inside a parallel region are doing the same work, but using different data.

Compilation and execution

- To compile an openMP application, extra options are needed.

```
ifort/icc hello.[f90,c] -o prog_hello  -openmp
```

- Total number of threads inside parallel region can be set via environment variable.

```
setenv OMP_NUM_THREADS 4

./prog_hello

Hello from thread 0 in a group of 4 threads
Hello from thread 1 in a group of 4 threads
Hello from thread 3 in a group of 4 threads
Hello from thread 2 in a group of 4 threads
```

Work sharing

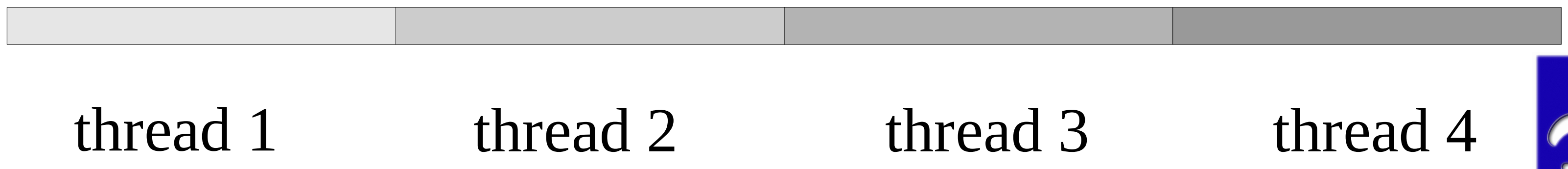
- To assign given task to a specific thread.
- To use do/for loops, one must uses directives:

  #pragma omp for (C)

  !$omp do

  !$omp end do (Fortran)

- By default, the loop's iterations are equally distributed on all threads.



Loop (C)

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
#define N 20

int main (int argc, char ** argv)
{
    int i, rank=0;

#pragma omp parallel private(rank)
  {
#ifdef _OPENMP
    rank = omp_get_thread_num();
#endif
#pragma omp for
    for (i=0; i<N; i++)
      printf("Element %d done by thread %d \n", i, rank);
  }
}
```

Loop (Fortran)

```
program loop
!$ use omp_lib

integer  i, rank, N
N=20
rank=0
!$omp parallel private(rank)
!$   rank = omp_get_thread_num()
!$omp do
do i=1, N
  print *, "Element",i," done by thread ",rank
end do
!$omp end parallel
end program loop
```

Observations

- Loop index is private.
- the do/for loop must immediately follow the do/for sentinel declaration.
- Must be inside a parallel region.
- There is a short form of the do/for sentinel.

  #pragma omp parallel for (C)

  !$omp parallel do

  !$omp end parallel do (Fortran)

- The sentinel directive "**!$omp end do**" is optional.

Work sharing cont.

- Each thread can execute its own work, which can be different from what other threads are doing.
- The following is used:

C

```
#pragma omp sections
{
#pragma omp section
{ ... }
#pragma omp section
{ ... }
}
```

Fortran

```
!$omp sections
!$omp section
...
!$omp section
...
!$omp end sections
```

Sections (C)

```
...
int main (int argc, char ** argv)
{
    int rank=0;

#pragma omp parallel private(rank)
  {
#ifdef _OPENMP
    rank = omp_get_thread_num();
#endif
#pragma omp sections
  {
#pragma omp section
   printf("Section 1 is executed by thread %d\n", rank);
#pragma omp section
   printf("Section 2 is executed by thread %d\n", rank);
  }
}
```

Sections (Fortran)

```
...
integer rank

rank=0
!$omp parallel private(rank)
!$   rank = omp_get_thread_num()
!$omp sections
!$omp section
  print *, "Section 1 is executed by thread" ,  rank
!$omp section
  print *, "Section 2 is executed by thread" ,  rank
!$omp end sections
!$omp end parallel
...
```

Observations

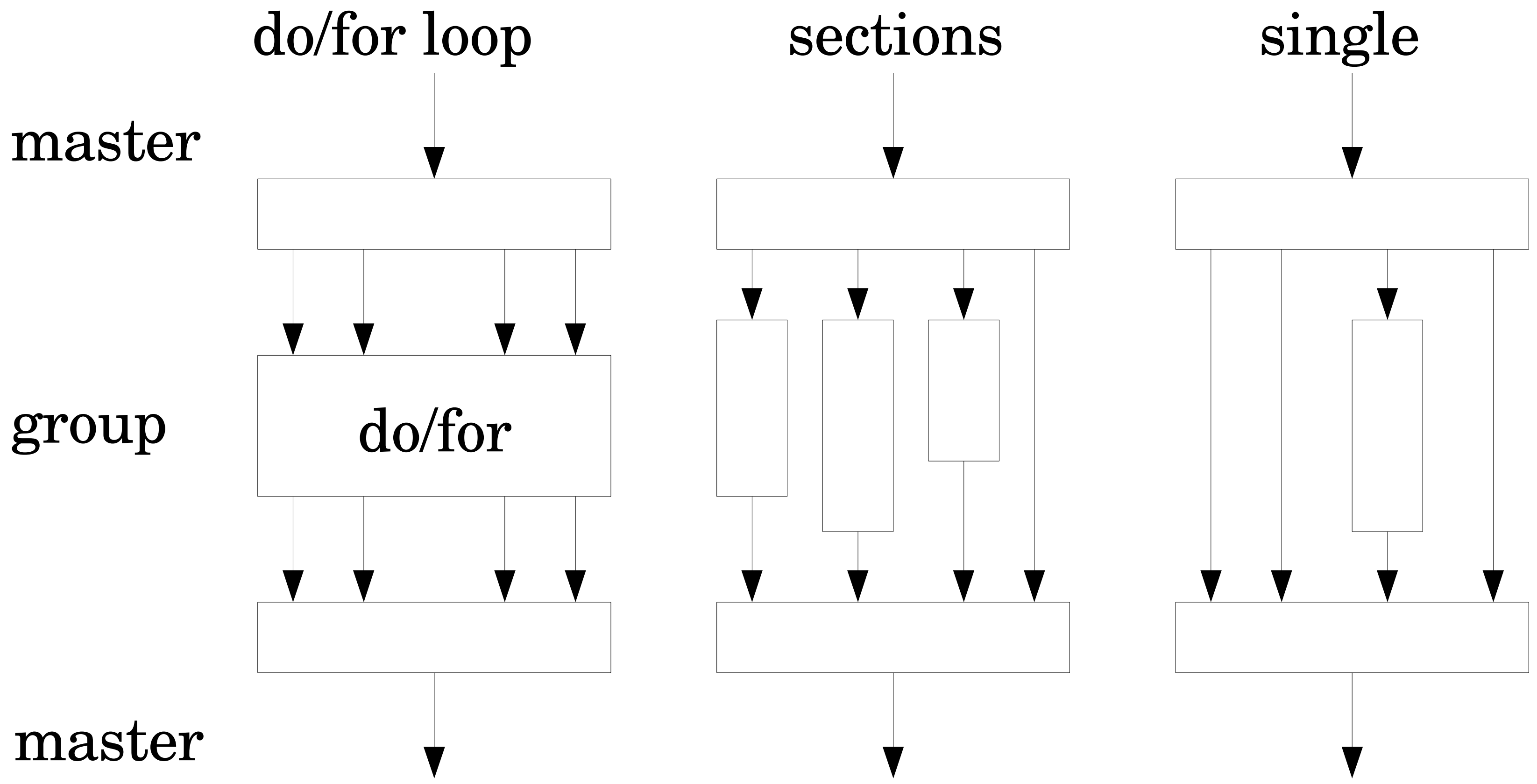
- Each section is executed by a different thread.
- Load balancing between sections is important.
- There is a short form for the sections sentinel:

#pragma omp parallel sections (C)

!$omp parallel sections

!$omp end parallel sections (Fortran)

Work sharing cont.



Reduction

- To carry out an operation (sum, multiplication, etc.) involving all threads. Some logical operation are also supported.
- In addition to shared and private, a variable's scope can be "reduction".

Naive reduction (C)

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
#define N 20

int main (int argc, char ** argv)
{
   int i, sum = 0, val =0, elements[N];
#pragma omp parallel for shared(elements)
   for (i=0; i<N; i++)   elements[i] = i+1;

#pragma omp parallel private(val) shared (elements,sum)
#pragme omp for
    for (i=0; i<N; i++)
      val += elements[i];
#pragma omp critical
      sum += val;

   printf ("The sum is %d\n", sum);
}
```

Naive reduction (Fortran)

```
program reduction
!$ use omp_lib
integer  i, sum, val, elements(20)
sum = 0
val = 0
!$omp parallel do shared(elements)
do i=1,20
  elements(i) = i
end do
!$omp parallel private(val) shared(elements,sum)
!$omp do
do i=1,20
  val = val + elements(i)
end do
!$omp end do
!$omp critical
sum = sum+val
!$omp end critical
!$omp end parallel
print *, "The sum is ", sum
end program reduction
```

Reduction (C)

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
#define N 20

int main (int argc, char ** argv)
{
   int i, sum = 0, elements[N];
#pragma omp parallel for shared(elements)
   for (i=0; i<N; i++)
      elements[i] = i+1;

#pragma omp parallel for shared (elements) reduction (+:sum)
    for (i=0; i<N; i++)
      sum += elements[i];

   printf ("The sum is %d\n", sum);
}
```

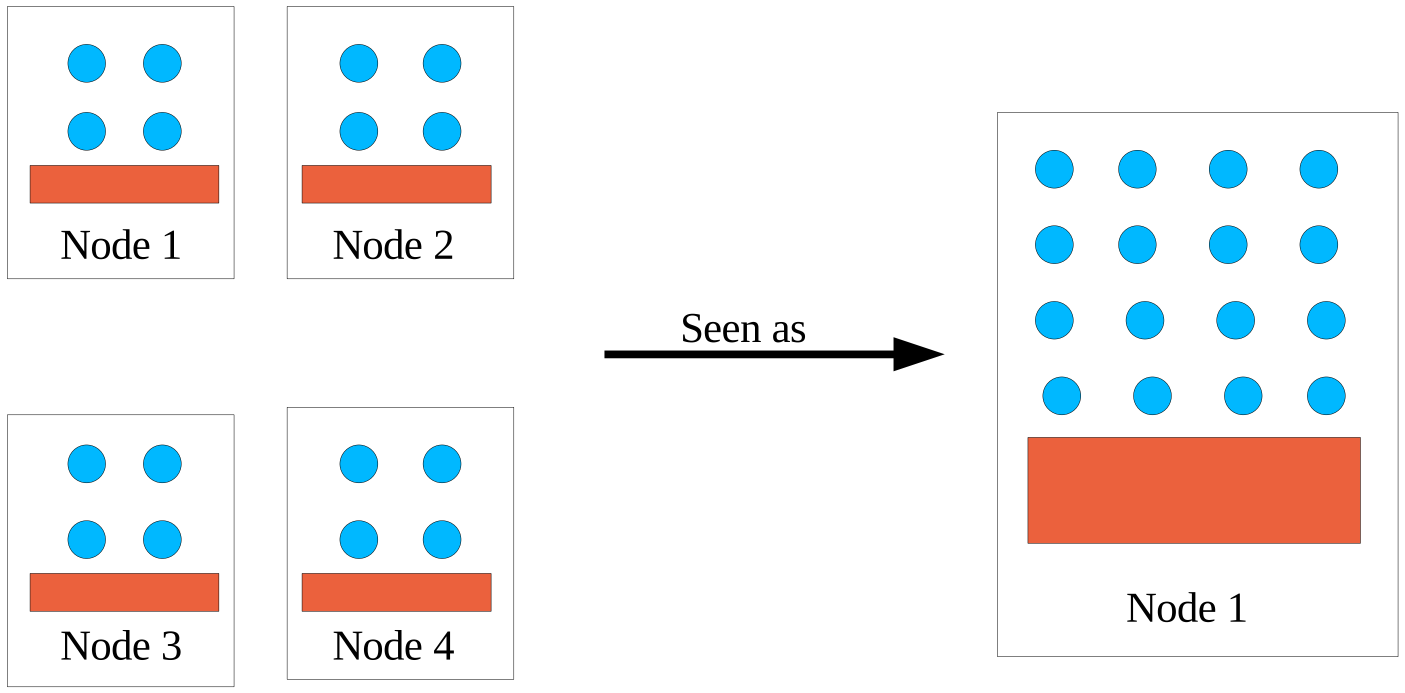
Reduction (Fortran)

```
program reduction
!$ use omp_lib

integer  i, sum, elements(20)
sum = 0
!$omp parallel do shared(elements)
do i=1,20
  elements(i) = i
end do

!$omp parallel do shared(elements) reduction(+:sum)
do i=1,20
  sum = sum + elements(i)
end do
!$omp end parallel do

print *, "The sum is ", sum
end program reduction
```

Observations

- Available operators are +, -, * and some boolean operators. In Fortran min and max operator are also available for reduction.
- Multiple reduction operations can be realized in the same loop.
- Results may differ from those obtained during serial execution due to round-off errors.
- Reductions also work with section constructions.
- Initializing tables in parallel is better (on Altix system, the performance is better).

Altix system



Multiple hosts are interconnected by a fast network and are configured in a way that makes them look like a single big node. In such a system, memory access isn't uniform. “dplace” is a tool to pin processes on a given underlying node in order to have fast memory accesses.

Performance

```
program reduction
use iflport
!$ use omp_lib
integer  i, sum, elements(1000000)
real*8 begin, end

do i=1,1000000
  elements(i) = irand()
end do

begin = dclock()
!$omp parallel do shared(elements) reduction(+:sum)
do i=1,1000000
  sum = sum + elements(i)
end do
!$omp end parallel do

end = dclock()
print *, "The sum is ", sum," and took",(end-begin)*1000.0d0," m sec"
end program reduction
```

Performance cont.

- OpenMP is not always a good solution. There is an overhead when using it.

Size	Serial	OpenMP(4)
1 000 000	0.5 ms	3.4 ms
100 000 000	115 ms	55 ms

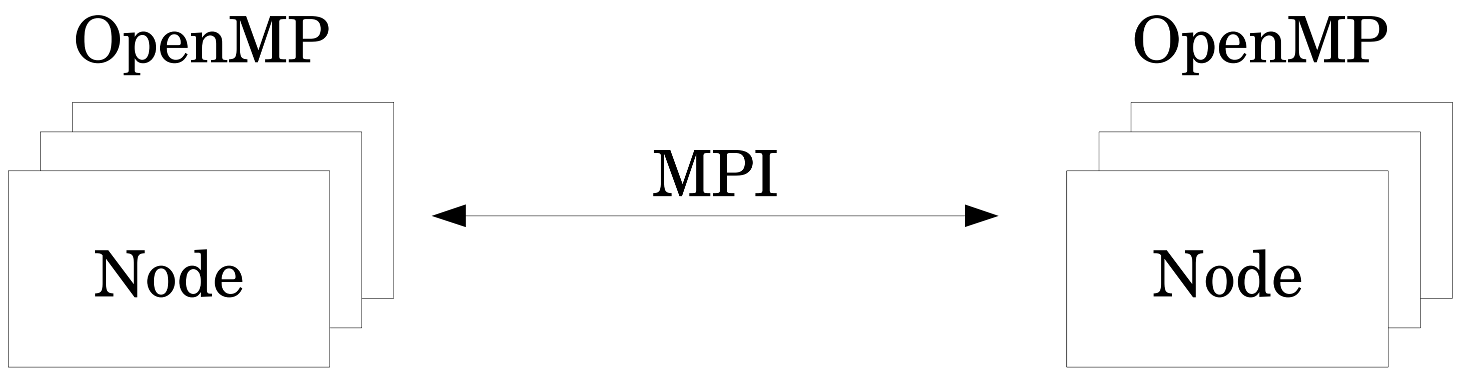
- OpenMP might be a good choice if the amount of work to do in each thread is large or if the total time spent inside one thread is high.
- It is very important to profile your application in order to decide where OpenMP might be useful.

References

- http://www.openmp.org
- *Parallel Programming in Openmp*, Chandra, et al., Morgan Kaufmann publisher, 2001.
- https://computing.llnl.gov/tutorials/openMP

Hybrid MPI/OpenMP

- The inter-node communication uses MPI from the openMP master thread on each node.
- Calculations are executed on multiple openMP threads on every node.



Pthreads

- Posix threads allow you to spawn a new concurrent process flow. It defines ways to schedule, synchronize, etc.
- A standard available on all platforms (C/C++).
- Very flexible.
- Hard to debug and to design on large system.
- Pthreads: Programming with POSIX Threads, David R. Butenhof, Addison-Wesley Professional computing series, 1997.

Others

Compilers

- UPC (*Unified parallel C*): http://www.gwu.edu/~upc/
- Co-array Fortran: http://www.co-array.org/

Libraries

- SHMEM: man intro_shmem (Altix)
- Global arrays: http://www.emsl.pnl.gov/docs/global/