Introduction to OpenMP

Fabio Affinito

Parallel paradigms in a nutshell

- Distributed memory
  - Each PE has its own private memory and exchange data buffers using messages; implemented, for example using MPI (Message Passing Interface)
- Shared memory
  - PEs can access the same memory address space; implemented in OpenMP (and pthreads for example).

Limitations of MPI

- Each MPI process can only access its local memory
  - The data to be shared must be exchanged with explicit inter-process communications
  - It is in charge to the programmer to design and implement the data exchange between process (taking care of work balance)
- You cannot adopt a strategy of incremental parallelization
  - The communications structure of the entire program has to implemented
- It is difficult to maintain a single version of the code for the serial and MPI program
  - Additional variables are needed
  - Large use of #ifdef can be confusing

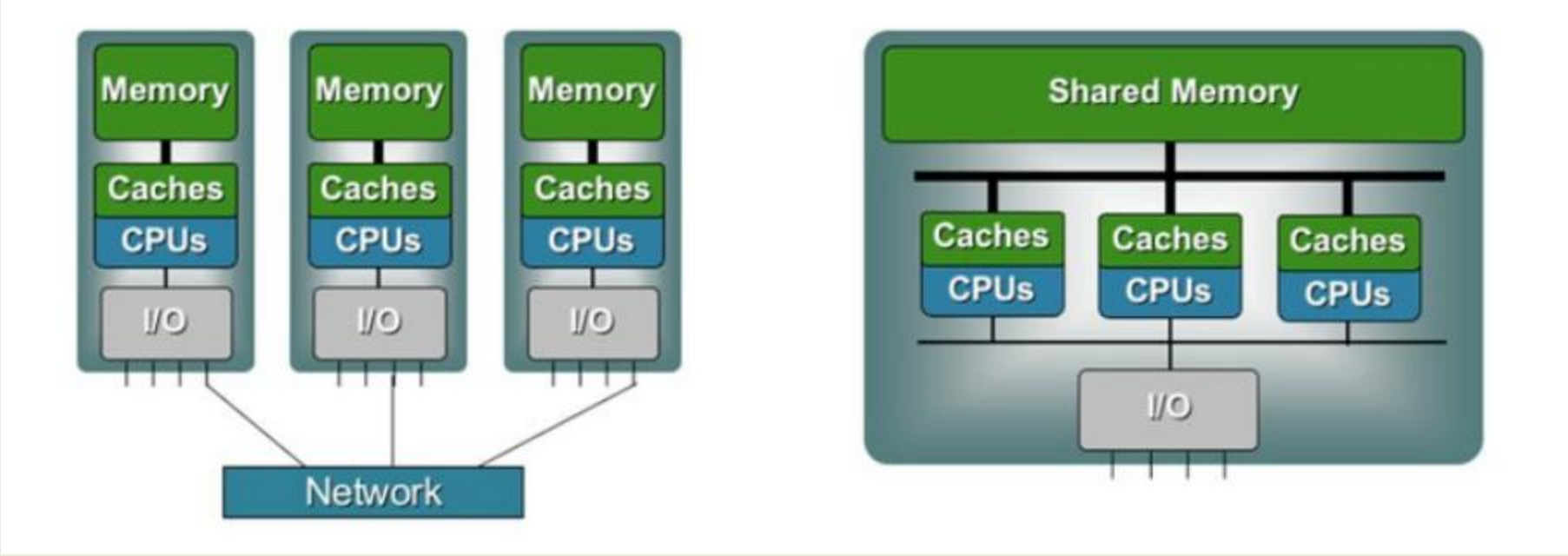
What is OpenMP

- OpenMP is de-facto standart API to write shared memory applications in C, C++ and Fortran
- It consists of compiler directives, run-time routines and environment variables
- It is mainained by the OpenMP Architecture Review Board (ARB) (http://www.openmp.org)
- The “workers” who do the work in parallel (threads) “cooperate” through shared memory
  - Memory accesses instead of explicit messages
  - “local” model of parallelization of the serial code
- It allows an incremental parallelization

OpenMP history

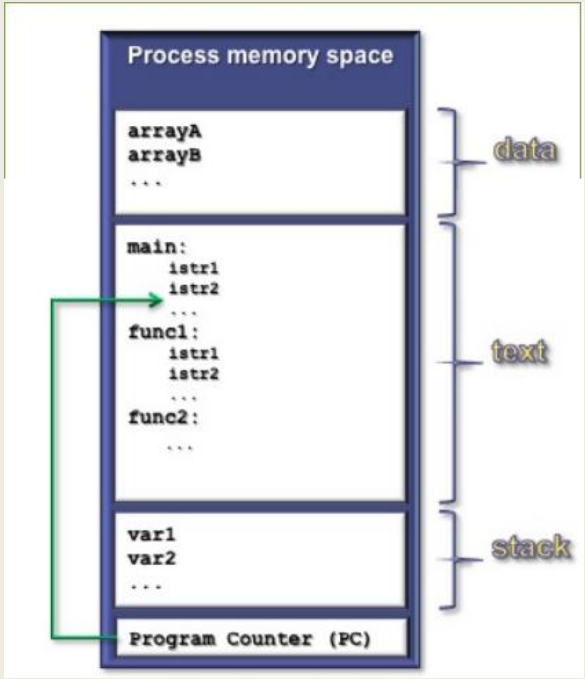
- Born to satisfy the need of unification of different proprietary solutions:
- The past:
  - 1997: Fortran version 1.0
  - 1998: C/C++ version 1.0
  - 2000: Fortran version 2.0
  - 2002: C/C++ version 2.0
  - 2005: combined C7C++ version 2.5
  - 2008: version 3.0 (introduction of tasks)
- The present:
  - 2011 version 3.1
  - 2013 version 4.0 (accelerator, SIMD extensions, affinity, etc.)
  - ? 4.1/5

UMA and NUMA


Memory
Caches
CPUs
I/O
Memory
Caches
CPUs
I/O
Memory
Caches
CPUs
I/O
Network
Shared Memory
Caches
CPUs
Caches
CPUs
Caches
CPUs
I/O

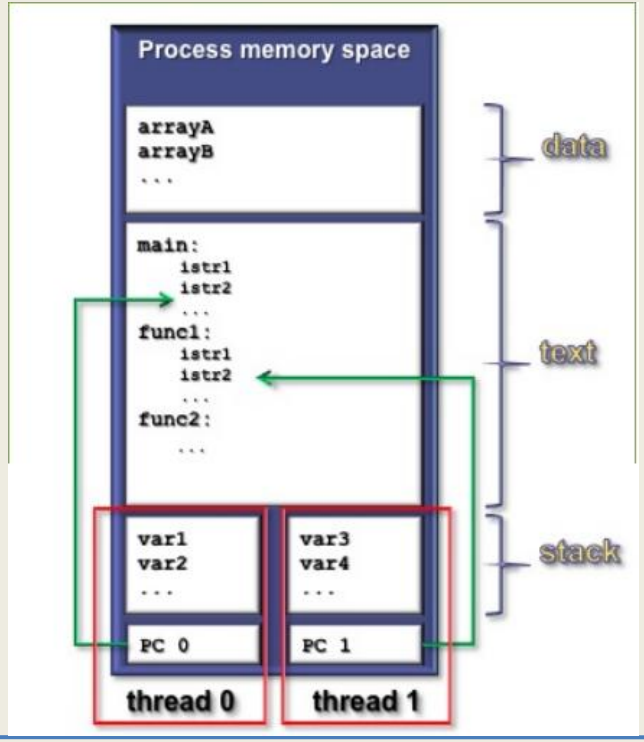
What is a thread?

- A process is an instance of a computer program
- Some information included in a process are:
  - Text (machine code)
  - Data (global variables)
  - Stack (local variables)
  - Program counter (A pointer to the instruction to be executed)

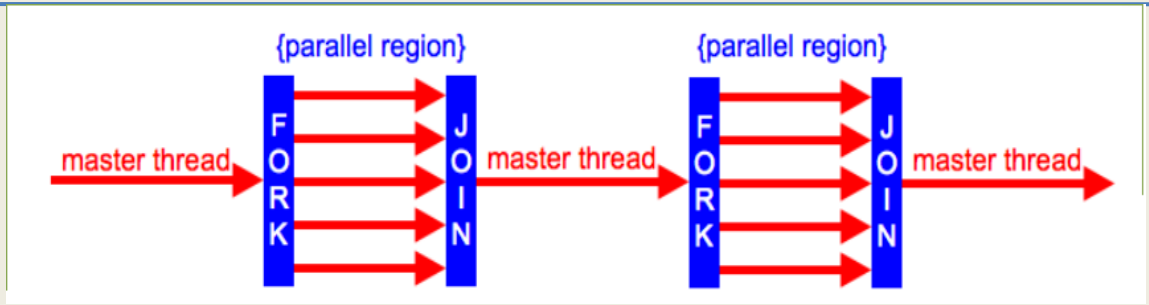


Multithreading

- The process contains several concurrent execution flows (threads)
  - Each thread has its own program counter (PC)
  - Each thread has its own private stack (variables local to the thread)
  - The instructions executed by a thread can access
    - The process global memory
    - The thread local stack



OpenMP execution model



Fork-join model:

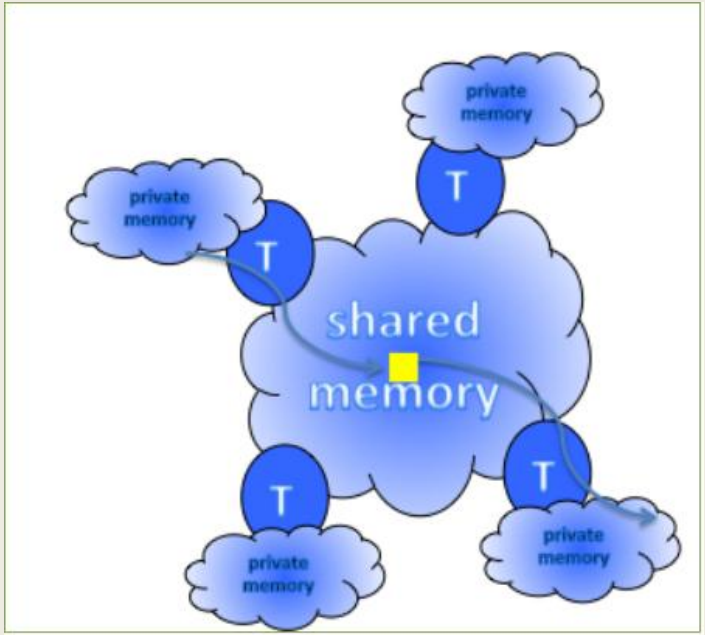
At the beginning of a parallel region the master thread creates a team of threads composed by itself and by a set of other threads

The thread team runs in parallel the code contained in the parallel region (Single Program Multiple Data model)

At the end of the parallel region the thread team ends the execution and only the master thread continues the execution ofthe (serial) program

OpenMP memory model

- All threads have access to the same globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s)
- Data transfer is through shared memory and is completely transparent to the application



Directives

OpenMP is a directive based language. Directives mark the block of code that should be made parallel and are interpreted by the precompiler only when a flag is specified.

Syntax:

```
#pragma omp <directive>
```

```
!$omp <directive>
```

Clauses

- Clauses are used to add information to the directives, i.e.:
  - Variables handling and scoping (shared, private, default)
  - Execution control (how many threads, work distribution...)

```
#pragma omp <directive> [clause [clause] ...]
```

```
!$omp <directive> [clause [clause]...]
```

Environment variables

- OMP_NUM_THREADS set the number of threads
- OMP_STACKSIZE “size [B|K|M|G]” size of the stack for threads
- OMP_DYNAMICS {true|false} bound threads to processors
- OMP_SCHEDULE “schedule[,chunk]” iteration scheduling scheme
- OMP_PROC_BIND {true|false} bound threads to processors
- OMP_NESTED {true|false} nested parallelism
- …

Run-time functions

- Query/specify some specific feature or setting:
  - omp_get_thread_num() : get thread ID
  - omp_get_num_threads() : get number of threads in the team
  - omp_set_num_threads(int n) : set the number of threads
  - …
- Allow you to manage the fine grain access (lock)
  - omp_init_lock(lock var) : initializes the OpenMP lock
  - ..
- Timing functions:
  - omp_get_wtime() : returns elapsed wallclock time
  - omp_get_wtick() : returns timer precision
  - …

Conditional compilation

To avoid dependancy on OpenMP libraries you can use pre-processing libraries and the preprocessor macro _OPENMP predefined by the standard

```
#ifdef _OPENMP
printf("Compiled with OpenMP support: %d",_OPENMP);
#else
printf("Compiled for serial execution");
#endif
```

Compiling and linking

- Compilers that support OpenMP interpret the directives only if they are invoked with a compiler option (switch):
- GNU: -fopenmp
- IBM: -qsmp=omp
- Sun: -xopenmp
- Intel: -openmp
- PGI: -mp

Parallel construct

It creates a parallel region:

- a construct is the lexical extent to which an executable directive applies
- a region is the dynamic extent to which an executable directive applies
- a parallel region is a block of code executed by all the threads in the team

```
#pragma omp parallel
{
//some code to be executed in parallel
} // end of the parallel region
```

C/Fortran syntax differences

```
#pragma omp parallel
{
//some code to be executed in parallel
} // end of the parallel region
```

```
!$omp parallel
! some code to be executed in parallel
!$omp end parallel
```

Hello world

```
#include <stdio.h>

int main(){

#pragma omp parallel
        {
            printf("Hello world!");
        }
        return 0;
}
```

Shared and private variables

Inside a parallel region the scope of a variable can be shared o private.

- shared: there is only one instance of the data
  - data is accessible by all threads in the team
  - threads can read and write the data simultaneously
  - all threads access the same address space
- private: each thread has a copy of the data
  - no other thread can access this data
  - changes are only visible to the thread owning the data
  - values are undefined on entry and exit

Variables are shared by default but you can nullify the default with the clause default (none)

Data races

A data race is when two or more threads access the same (shared) memory location:

- asynchronously
- without holding any common exclusive locks
- at least one of the accesses is a write/store

In this case, the resulting values are undefined!

Critical construct

- One of the possible solutions for data races is to use the critical construct
- The block of code inside a critical construct is executed by only one thread at time.
- It is a synchronization to avoid simultaneous access to a shared data.
- A critical construct locks the associated region
- It is a good practise to associate to each critical region a label

Critical construct

```
sum = 0;
#pragma omp parallel private(i, MyThreadID)
{
  ThreadId = omp_get_thread_num();
  NumThreads = omp_get_num_threads();
  for (i=ThreadId*N/NumThreads; i<(ThreadId+1)*N/NumThreads; i++){
    psum += x[i];
  }
#pragma omp critical label
  sum += psum;
}
```

Worksharing

In principle, for a parallelization the fundamental ingredients are:

- the parallel construct
- the critical construct
- the omp_get_thread_num and omp_get_num_threads functions

But we need to distribute the work among threads and doing it by hand is unefficient. The worksharing construct can automate the process.

Worksharing construct

- A worksharing construct distributes the execution of the associated parallel region over the threads that must encounter it
- A worksharing construct has no barrier on entry, but implicit barrier exists at the end of the worksharing region
- The nowait clause can remove the barrier at the end of the worksharing region

Worksharing construct

The OpenMP defines the following workshare constructs:

- for/do loop construct
- sections construct
- single construct
- workshare construct

Loop construct

- The iterations of a loop are distributed over the threads that already exist in the team
- The iteration variable is made private by default
- The inner loops are executed sequentially by each thread
- Beware the data sharing attribute of the inner loop iteration variables:
  - in Fortran they are private by default
  - in C/C++ they aren't
- Requirement for (loop) parallelization:
  - no dependencies between loop indexes

Loop construct

```
#pragma omp for [clauses]
  for (i=0; i<n; i++)
  { ... }
```

```
!$omp do
do i = 1, n
  ...
end do
!$omp end do
```

Loop construct

```
int main()
{
 int i, n=10;
 int a[n], b[n], c[n];
...
#pragma omp parallel
{
 #pragma omp for
 for (i=0; i<n; i++){
   a[i] = b[i] = i;
   c[i] = 0;
 }
 #pragma omp for
 for (i=0; i<n; i++){
   c[i] = a[i] + b[i];
 }
}
```

Loop collapse

- allows the parallelization of perfect nested loops
- the collapse clause on for/do loops indicates how many loops should be collapsed
- compiler then makes a single loop and then makes it parallel

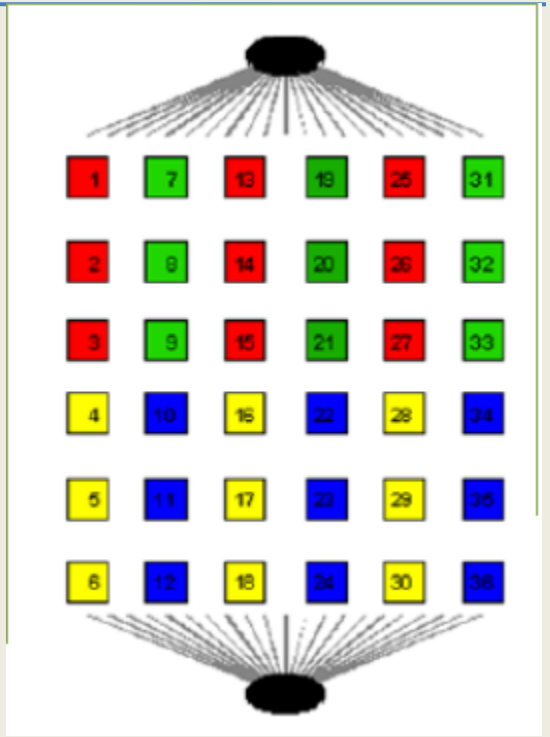
```
#pragma omp for collapse(2) private(j)
for (i=0; i<n; i++){
  for (j=0; j<m; j++){
    ..
  }
}
```

Schedule clause

- the schedule clause specifies how the iterations of the associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team.
- schedule presets are:
  - static
  - dynamic
  - guided
  - auto
- the chunksize can be assigned too

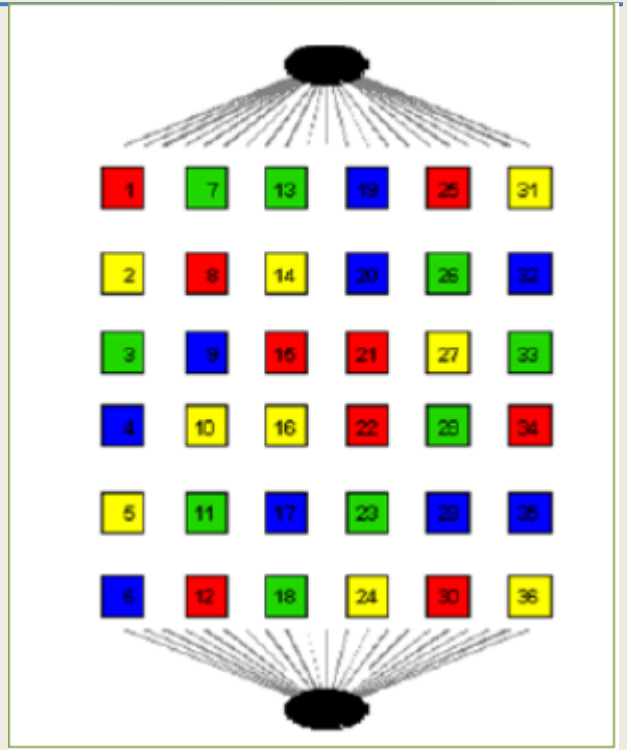
Static scheduling

- Iterations are divided into chunks of size chunk.
- The chunks are assigned to the threads in the team in a round-robin way in the order of the thread number
- It is the default schedule and the default chunk is number of iterations / number of threads
- Figure: schedule (static,3)



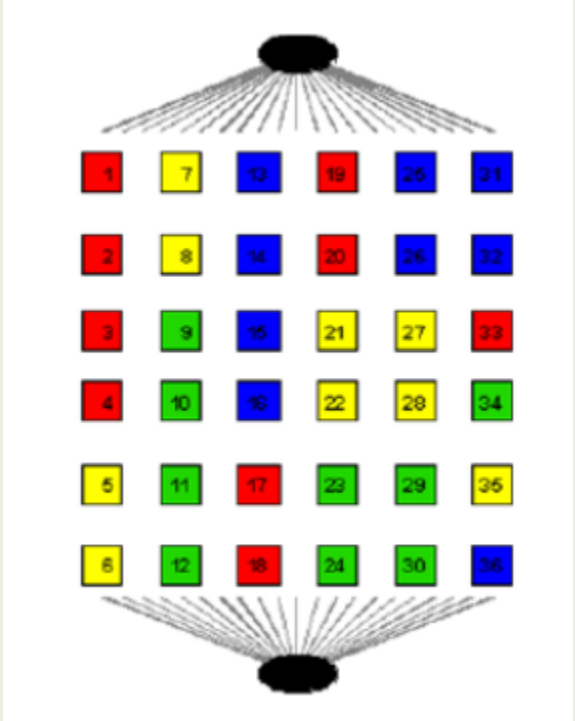
Dynamic scheduling

- Iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.
- The default chunk is 1



Guided scheduling

- Iterations are assigned to threads in the team in chunks as the executing threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. The chunk decreases to chunk
- The default value of chunk is 1

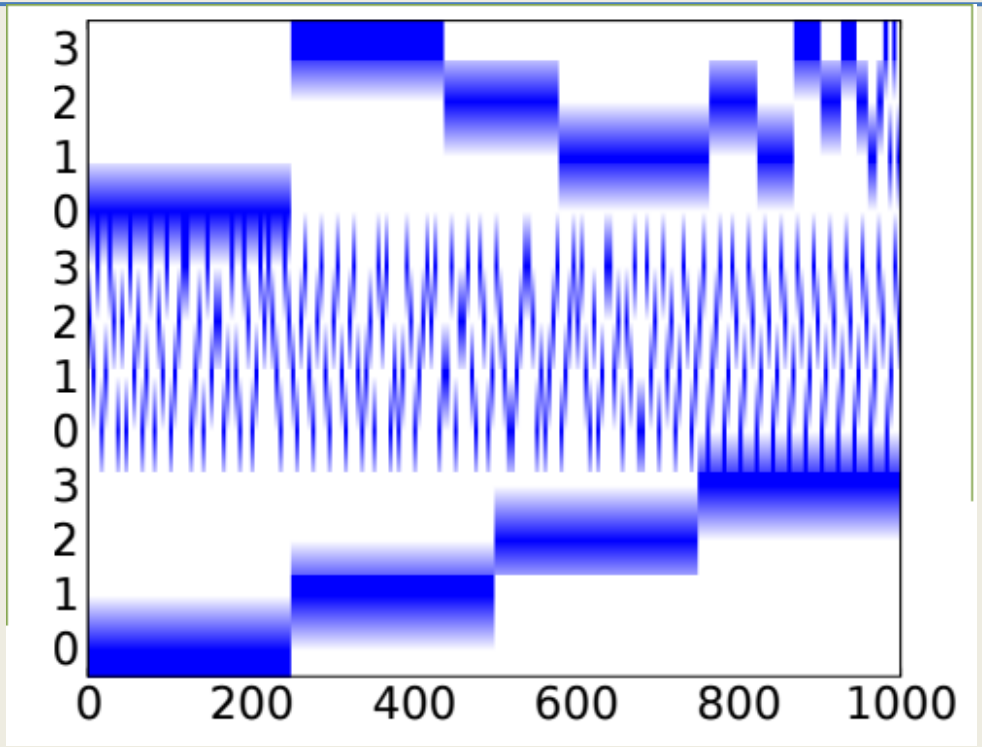


Runtime and auto scheduling

- runtime: iteration scheduling scheme is set at runtime using the environment variable OMP_SCHEDULE
  - the schedule scheme can be modified without recompiling
  - useful only when doing experiments during the development of a code
- auto: the decision is delegated to the compiler and/or runtime system

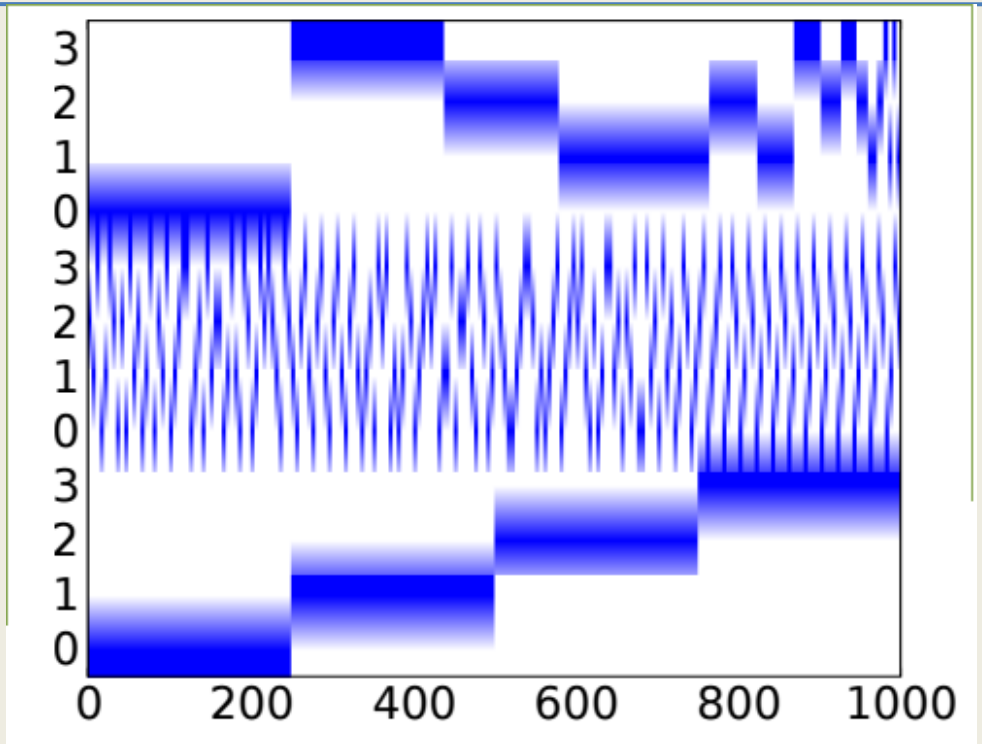
Scheduling

???



Scheduling

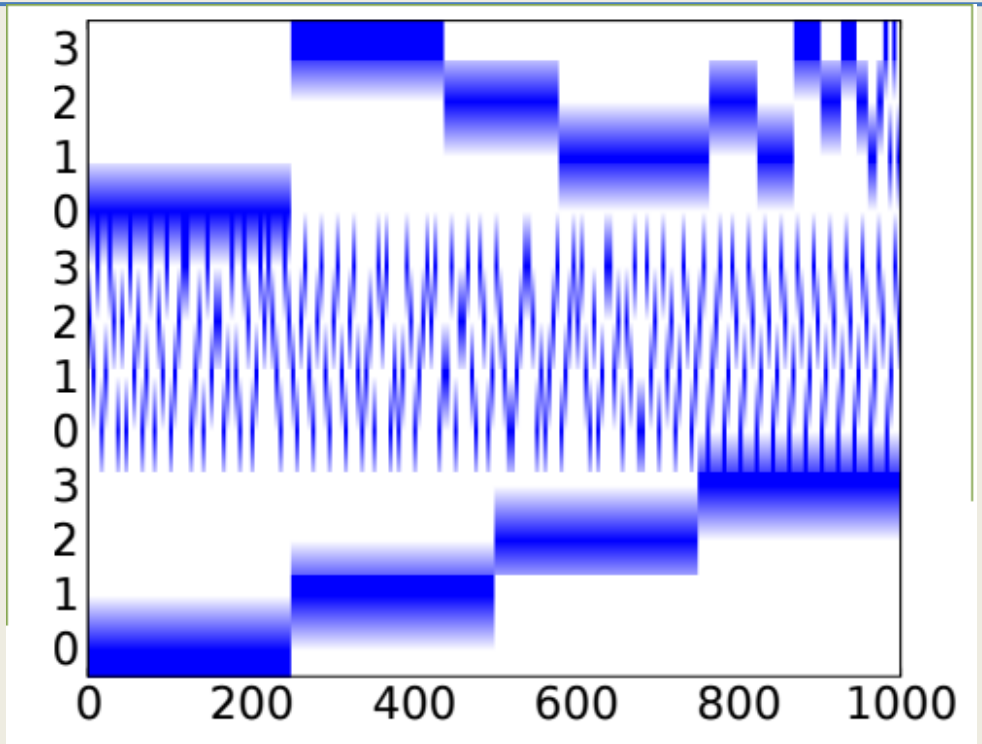
static



Scheduling

dynamic

static

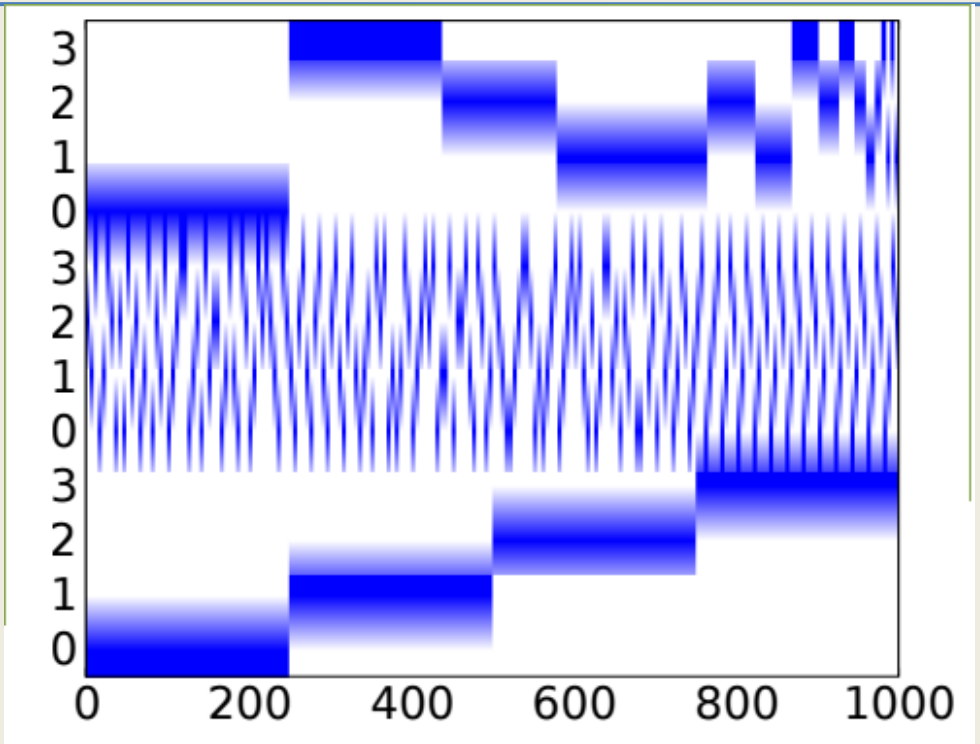


Scheduling

guided

dynamic

static



Sections construct

- it is a worksharing construct to distribute structured blocks of code among threads in a team
- each thread receives a section
- when a thread has finished to execute its section it receives another section
- if there are not other sections to execute, threads wait for others to end up

Sections construct

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

Single construct

- the first thread that reach the single construct executes the associated block
- the other threads in the team wait at the implicit barrier at the end of the construct unless a nowait clause is specified

```
#pragma omp single [private] [firstprivate] [copyprivate] [nowait]
{
....
}
```

Workshare construct

- only supported in Fortran
- the structured block enclosed in the workshare construct is divided into units of work that are assigned to the threads.

```
!$omp workshare
! structured block
!$omp end workshare [nowait]
```

Data-sharing attributes

- in a parallel construct the data-sharing attributes are implicitly determined by the default clause, if present
  - if no default clause is present, they are shared
- there are pre-determined data sharing attributes, but it always safer to specify default(none) and to assign the right value for each variable

Data-sharing attributes

- shared: there is only one instance of the objects accessible by all threads in the team
- private: each thread has a copy of the variables
- firstprivate: same as private but all variables are initialized with the value that the original object had before entering the parallel construct
- lastprivate: same as private but the thread that executes the sequentially last iteration or section updates the value of the objects

Attention: if a variable is defined with the wrong attribute, it can produce meaningless results!

Reduction

- if a variable is used to accumulate a value from the different threads, this should be declared as a reduction variable
- an implicit private copy is created for each thread and properly initialized
- at the end of the parallel region the variable is updated according to the specified operator
- reduction variables must be shared variables
- the reduction clause is valid on parallel, for/do loops and sections constructs

Reduction

```
#pragma omp parallel for reduction (+:sum)
for (i=0; i<n; i++){
  sum+=x[i]
}
```

Supported operators for a reduction clause are:

- C: +, *, -, &, |, ^, &&, || max e min from 3.1)
- Fortran: +, *, -, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor

Barrier construct

- in a parallel region threads proceed asynchronously
- the synchronization can be enforced using an implicit barrier with #pragma omp barrier
- implicit barriers are assumed at the end of a worksharing construct
- when not necessary, a barrier can cause slowdowns
- implicit barriers can be removed using the nowait clause

Atomic construct

- this construct applies only to statements that update the value of a variable
- it ensures that no other thread updates the variable between reading and writing
- it is similar to the critical construct (that applies to a region rather than to a single instruction)

```
#pragma omp atomic
<instruction>
```

Master construct

- only the master thread executes the associated code block
- there is not an implicit barrier at the entry neither at the end

```
#pragma omp master
{structured block}
```

Task parallelism

- Main addition to OpenMP 3.0, then enhanced in OpenMP 3.1 and 4.
- Allows to parallelize irregular problems (unbounded, recursive algorithms, multiblocks)
- Allows oversubcription of the resources

Pointer chasing in OpenMP 2.5

```
#pragma omp parallel private (p)
 p=head;
 while (p){
 #pragma omp single nowait
  process(p);
 p = p->next;
 }
```

- using the omp single does not permit to obtain a good parallelization;
- transform this code to a “canonical” loop can be very difficult

Tree traversal in OpenMP 2.5

```
void preorder (node *p){
 process (p->data);
 #pragma omp parallel sections num_threads(2)
 {
  #pragma omp section
  if (p->left)
    preorder (p->left);
  #pragma omp sectio
  if (p->right)
    preorder (p->right);
 }
}
```

First tasking construct...

```
#pragma omp parallel
{
....
}
```

- creates both tasks and threads
- tasks are implicit
- each task is executed by one thread
- each task is tied to one thread

New tasking construct

```
#pragma omp task [clauses]
{
....
}
```

- it creates a new task but not a new thread
- this task is explicit
- it will be executed by a thread in the current team
- it can be deferred until a thread is available to execute
- the data environment is built at creation time (variables inherit thei data sharing attributes, but private become firstprivate)

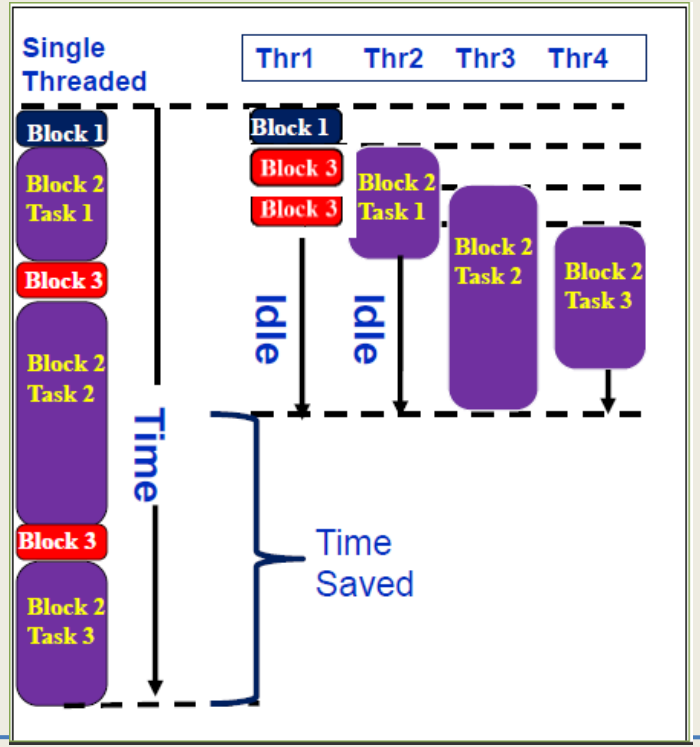
Pointer chasing with tasks

```
#pragma omp parallel private (p){
 #pragma omp single
 {
 p=head;
 while (p){
 #pragma omp task
  process(p);
 p = p->next;
 }
}
```

1. create a team of threads
2. one thread executes the single construct; other threads wait at the barrier at the end of the single construct
3. the single thread creates a task with its own value for the pointer p
4. threads waiting at the barrier execute tasks; executions move beyond the barrier after all the tasks are completed

Pointer chasing with tasks

```
#pragma omp parallel private (p){
 #pragma omp single
 {
 p=head;
 while (p){
 #pragma omp task
   process(p);
 p = p->next;
 }
}
```



Load balancing with tasks

```
#pragma omp parallel {
 #pragma omp for private(p)
 for (i=0; i<n; i++){
  p = head[i];
  while(p){
   #pragma omp task
     process(p);
   p = p->next;
  }
 }
}
```

- The assignement of one thread per task can lead to work unbalance (mitigated by the scheduling...)
- Multiple threads create tasks
- The whole team cooperates to execute them

Tree traversal with task

```
void preorder (node *p){
  process (p->data);
  if (p->left)
  #pragma omp task
    preorder(p->left);
  if (p->right)
  #pragma omp task
    preorder(p->right);
}
```

- tasks are composable
- tasks are not worksharing construct

False sharing

Is this world so wonderful?

Not really...

```
static long num_steps =100000;
#define NUM_THREADS=2
void main(){
  int i, nthreads; double pi, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
    int i, id, nthrds; double x;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id == 0) nthreads = nthrds;
    for (i=id, sum[id]=0.0; i<num_steps; i=i+nthrds){
      x = (i+0.5)*step;
      sum[id] += 4.0/(1.0+x*x);
    }
  }
  for (i=0, pi=0.0; i<nthreads; i++)pi +=sum[i]*step;
}
```

False sharing

Is this world so wonderful?

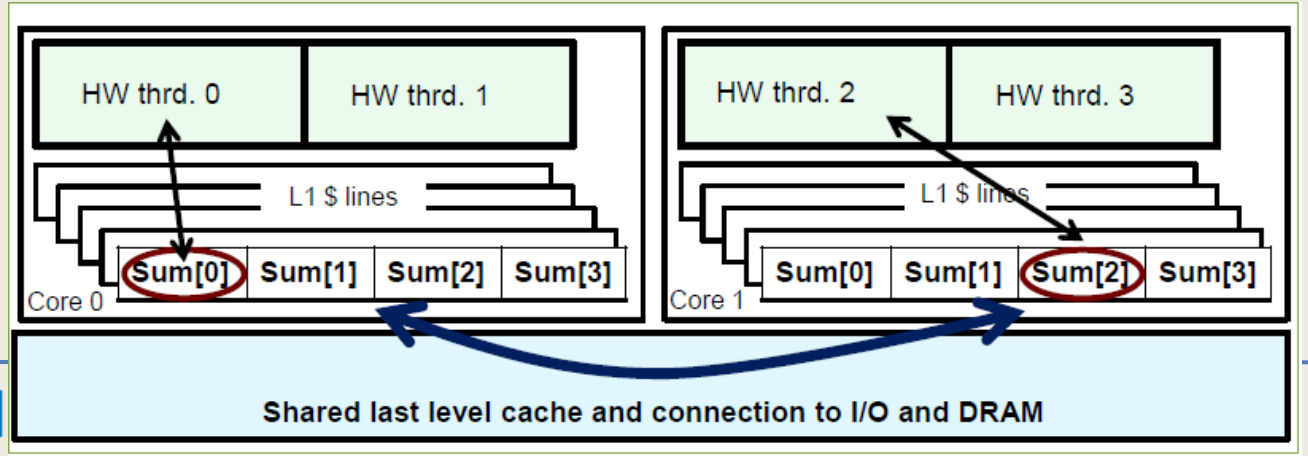
Not really...

Threads	time
1	1.86
2	1.03
3	1.08
4	0.97

```
static long num_steps =100000;
#define NUM_THREADS=2
void main(){
  int i, nthreads; double pi, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
    int i, id, nthrds; double x;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id == 0) nthreads = nthrds;
    for (i=id, sum[id]=0.0; i<num_steps; i=i+nthrds){
      x = (i+0.5)*step;
      sum[id] += 4.0/(1.0+x*x);
    }
  }
  for (i=0, pi=0.0; i<nthreads; i++)pi +=sum[i]*step;
}
```

False sharing

- if independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads: this is called false sharing
- solution: pad arrays in order to place elements on distinct cache lines



False sharing

assume in this example that
L1 cache line is 64 byte
pad the array so each sum
value is in a different cache
line

```
static long num_steps =100000;
#define NUM_THREADS=2
#define PAD 8
void main(){
 int i, nthreads; double pi, sum[NUM_THREADS][PAD];
 step = 1.0/(double) num_steps;
 omp_set_num_threads(NUM_THREADS);
 #pragma omp parallel
 {
  int i, id, nthrds; double x;
  id = omp_get_thread_num();
  nthrds = omp_get_num_threads();
  if (id == 0) nthreads = nthrds;
  for (i=id, sum[id][0]=0.0; i<num_steps; i=i+nthrds){
   x = (i+0.5)*step;
   sum[id][0] += 4.0/(1.0+x*x);
  }
 }
 for (i=0, pi=0.0; i<nthreads; i++)pi +=sum[i][0]*step;
}
```

False sharing

assume in this example that
L1 cache line is 64 byte
pad the array so each sum
value is in a different cache
line

```
static long num_steps =100000;
#define NUM_THREADS=2
#define PAD 8
void main(){
 int i, nthreads; double pi, sum[NUM_THREADS][PAD];
 step = 1.0/(double) num_steps;
 omp_set_num_threads(NUM_THREADS);
 #pragma omp parallel
 {
  int i, id, nthrds; double x
  id = omp_get_thread_n
  nthrds = omp_get_num
  if (id == 0) nthreads = nt
  for (i=id, sum[id][0]=0.0
    x = (i+0.5)*step;
    sum[id][0] += 4.0/(1.0+
  }
 }
 for (i=0, pi=0.0; i<nthreads; i++)pi += sum[i][0]*step;
}
```

Threads	time
1	1.86
2	1.01
3	0.69
4	0.53

Coming soon...

- OpenMP 4 is implemented in almost every compiler..
- ... but always check carefully which features are implemented
- But already some new specifications are available for OpenMP 4.1
  - taskloops: task associated to loops
  - offload
  - other...

Is this enough?

- This lecture didn’t cover some important features of OpenMP:
  - flush
  - locks
  - simd
  - offload
  - ordered construct
- If you are still hungry -> www.openmp.org

Credits

To all the people who contributed more or less synchronously and more or less consciently to these slides:

Gian Franco Marras, Marco Comparato, Massimiliano Culpo, Massimiliano Guarrasi, Giorgio Amati, Cristiano Padrin, Federico Massaioli, Marco Rorro, Vittorio Ruggiero, Francesco Salvadore, Claudia Truini, etc.