Μεταγλωττιστές για Ενσωματωμένα Συστήματα

Χειμερινό Εξάμηνο 2023-24
«OpenMP - I»

Παναγιώτης Χατζηδούκας

Outline

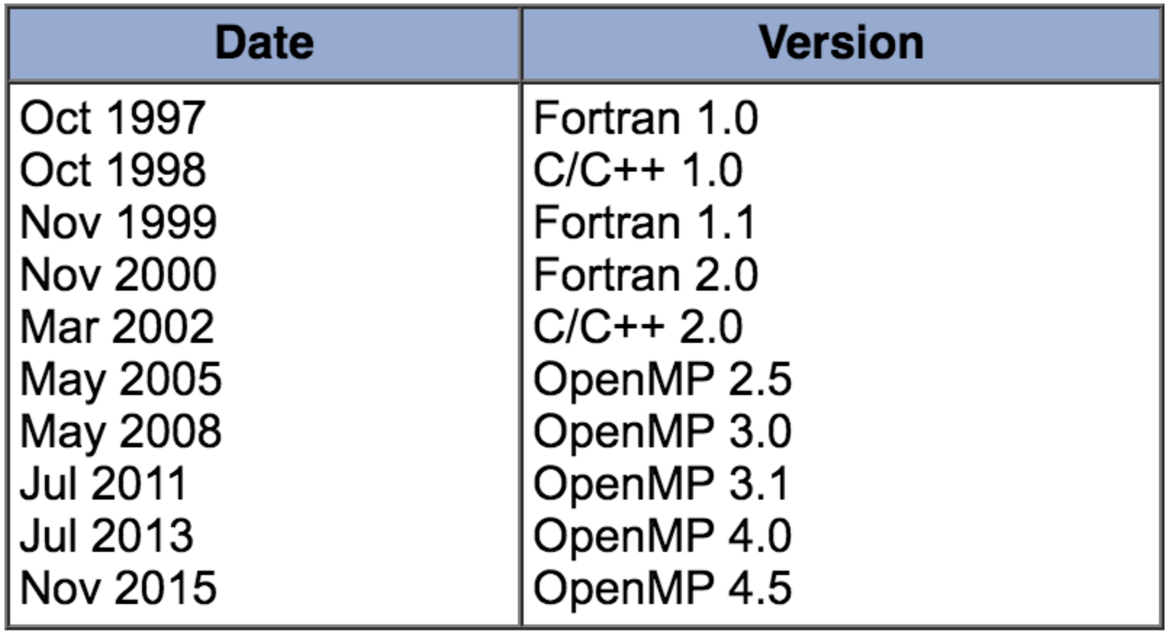
- Introduction to OpenMP
- Parallel regions
- Worksharing constructs
    - loops, sections. single
- Combined parallel worksharing
- Data environment
- Synchronization
    - critical, atomic, barrier, master
- Library routines
- Environment variables
- Examples

OpenMP

- OpenMP: An Application Program Interface (API) for writing multithreaded applications
  - simple, portable, widely supported standard
  - facilitates the development of multithreaded code in Fortran, C and C++
  - suitable for shared memory platforms

- Three primary components
  - compiler directives - instruct the compiler to generate multithreaded code
  - library calls
  - environment variables

Evolution of OpenMP



Date	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0
Nov 2015	OpenMP 4.5

http://computing.llnl.gov/tutorials/openMP/

- OpenMP specifications at www.openmp.org
  - OpenMP 3.1 (2011): C/C++, Fortran and Examples
  - OpenMP 4.0 (2013): Examples in a separate PDF file

Syntax Format

- Compiler directives
  - C/C++
    - `#pragma omp construct [clause [clause] …]`
  - Fortran
    - `C$OMP construct [clause [clause] … ]`
    - `!$OMP construct [clause [clause] … ]`
    - `*$OMP construct [clause [clause] … ]`

- Since we use directives, **no changes** need to be made to a program for a compiler that does not support OpenMP

OpenMP Directive

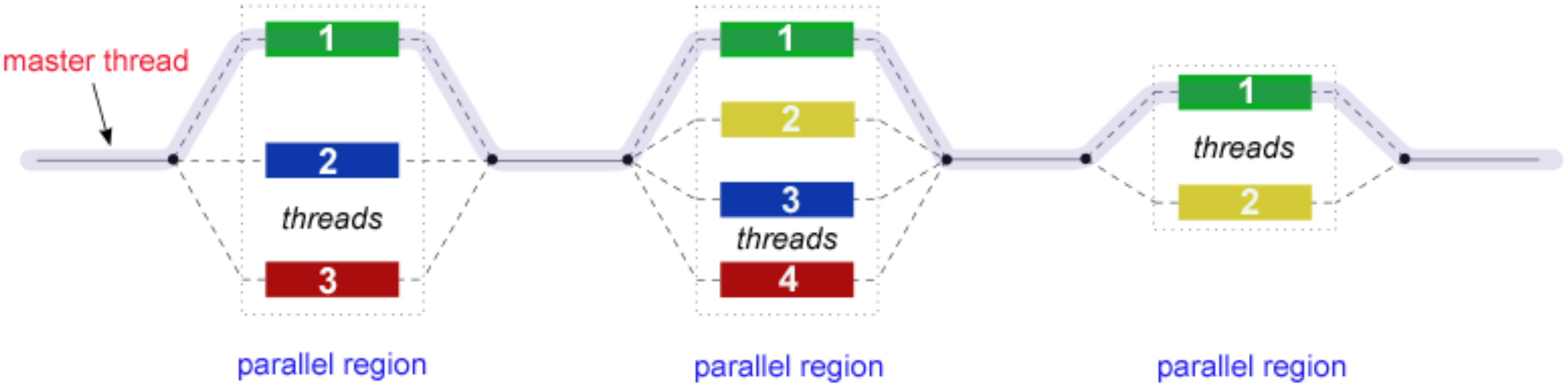
- Program executes serially until it encounters a parallel directive

```
#pragma omp parallel [clause list]
/* structured block of code */
```

- Clause list is used to specify conditions
  - Conditional parallelism: **`if (cond)`**
  - Degree of concurrency: **`num_threads(int)`**
  - Data handling: **`private(vlist)`**, **`firstprivate(vlist)`**, **`shared(vlist)`**

Programming Model

- Fork-join type of parallelism:
  - The master thread spawns teams of threads according to the user / application requirements
  - Parallelism is added incrementally
    - the sequential code is transformed to parallel



http://computing.llnl.gov/tutorials/openMP/

Typical Usage

- OpenMP is generally used for loop parallelization
  - Find the most time-consuming loops
  - Distribute the loop iterations to the threads

Assign this loop to different threads

```
void main()
{
   double Res[1000];

   for (int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
  }
}
```

Sequential code

```
void main()
{
    double Res[1000];
   #pragma omp parallel for
   for (int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

Parallel code

But OpenMP is not just that!

Using OpenMP

- Some compilers can automatically place directives with option
  - `-qsmp=auto` (IBM xlc)
  - some loops may speed up, some may slow down

- Compiler option required when you use directives
  - `-fopenmp` (GNU compilers)
  - `-openmp` (Intel compilers)
  - `-qsmp=omp` (IBM)

- Scoping variables can be sometimes the hard part!
  - shared variables, thread private variables

Hello World!

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

int main() {
 #pragma omp parallel
 {
   int me = omp_get_thread_num();
   int nthr = omp_get_num_threads();
   printf("Hello world from thread %d of %d\n", me, nthr);
 }
 return 0;
}
```

OpenMP include file

Parallel region with default number of threads

Library calls

End of parallel region

- Compilation with the GNU GCC and Intel compilers

```
$ gcc -fopenmp -o hello hello.c
$ icc -openmp -o hello hello.c
```

MacOS: brew install gcc

Usage

- Execution

```
$ export OMP_NUM_THREADS=4
$ ./hello
Hello world from thread 0 of 4
Hello world from thread 2 of 4
Hello world from thread 1 of 4
Hello world from thread 3 of 4
$ export OMP_NUM_THREADS=1
$ ./hello
Hello world from thread 0 of 1
```

Environment variable

Thread Interaction

- OpenMP is a shared-memory programming model
  - Threads communicate through shared variables
- Data sharing can lead to **race conditions**
  - the output of some code can change due to thread scheduling, e.g. their order of execution
- Synchronization at the right places can eliminate race conditions
  - However, **synchronization is expensive**
  - the way data is stored might need to change to minimize the need for synchronization

OpenMP Directives

- 5 categories
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions & environment variables

- Basically the same between C/C++ and Fortran

Parallel Regions

- Create threads with **`omp parallel`**
- The following code will create a parallel region of 4 threads:

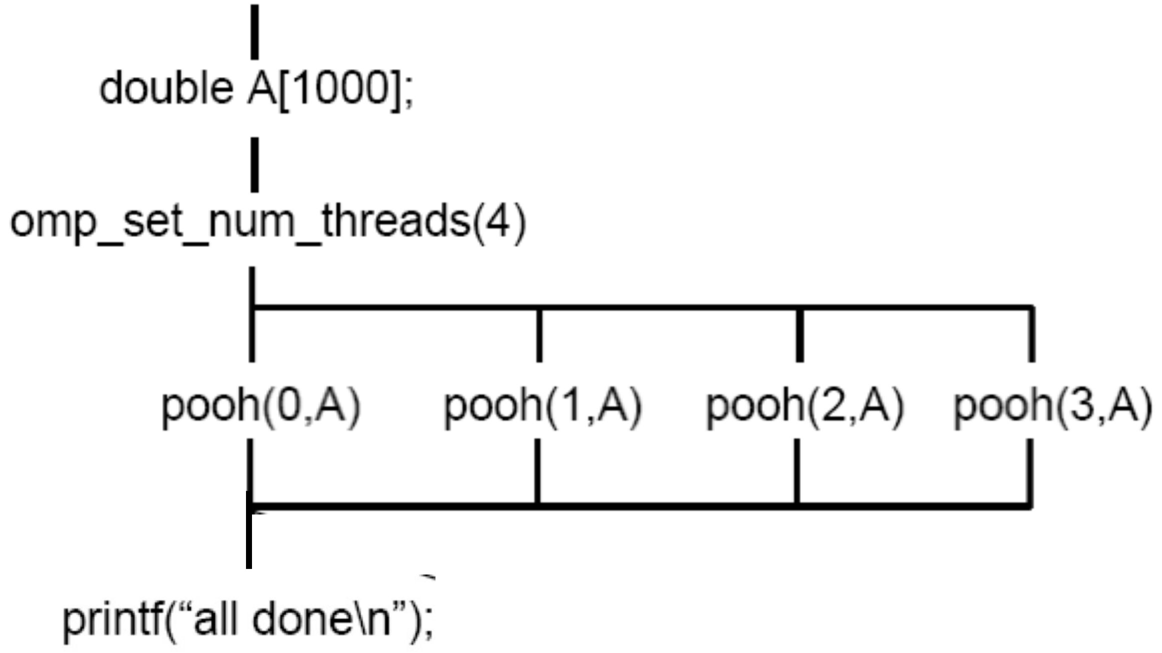
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

- Threads share A (default behavior)
- Master thread creates the threads
- Threads all start at same time then synchronize at a barrier at the end to continue with code
- Each threads calls pooh for its own ID (0 to 3)

Parallel Regions

- Each threads runs the same code
- All threads share A
- Execution continues when all threads have finished their work (barrier)

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID,A);
}
printf("all done\n");
```



Parallel Regions - Syntax

```
#pragma omp parallel [clause ...]  newline

   structured_block
```

Clauses

```
      if (scalar_expression)
      num_threads (integer-expression)
      private (list)
      shared (list)
      firstprivate (list)
      default (shared | none)
      reduction (operator: list)
      copyin (list)
```

Structured Blocks

- Most OpenMP directives are applied to structured blocks of code
  - Structured block: piece of code with a single entry point at the beginning and a single exit point at the end.

```
#pragma omp parallel
{
  int id =omp_get_thread_num();
  res[id] = work(id);

}
printf("after parallel\n");
```

Structured block

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  res[id] = work(id);
  if (res[id] == 0) goto out;
}
out: printf("after parallel\n");
```

Unstructured block

Clauses for omp parallel

`if (scalar_expression)`	Only parallelize if the expression is true. Can be used to stop parallelization if the work is too little
`num_threads (integer-expression)`	Set the number of threads
`private (list)`	The specified variables are thread-private
`shared (list)`	The specified variables are shared among all threads
`firstprivate (list)`	The specified variables are thread-private and initialized from the master thread
`reduction (operator: list)`	Perform a reduction on the thread-local variables and assign it to the master thread
`default (shared | none)`	Unspecified variables are shared or not

```
#pragma omp parallel private(i) shared(n) if(n > 10)
{
    //…
}
```

Actual Number of Threads

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
    1. Evaluation of the **`if`** clause
    2. Setting of the **`num_threads`** clause
    3. Use of the **`omp_set_num_threads()`** library function
    4. Setting of the **`OMP_NUM_THREADS`** environment variable
    5. Implementation default - usually the number of CPUs on a node, though it could be dynamic.

- Reminder: threads are numbered from 0 (master thread) to N-1

Static and Dynamic modes

- Dynamic mode (default):
  - The number of threads can differ between parallel regions of the same program
  - The specified number of threads actually defines the maximum number - the actual number of threads can be smaller
- Static mode:
  - The number of threads is fixed and exactly equal to the number specified by the programmer
- OpenMP supports nested parallel regions but…
  - The compiler is allowed to serialize all the inner levels
  - This means that it uses a single OpenMP thread for those parallel regions

Worksharing Constructs

- the for construct splits up loop iterations

```
#pragma omp parallel
{
#pragma omp for
for (i=0; i<N; ++i){
        do_work(i);
}
}
```

can be omitted

parallel region

worksharing

end of **omp for**

- By default, there is a barrier at the end of the **`omp for`**.
- Use the **`nowait`** clause to turn off the barrier.

Rule

- In order to be made parallel, a loop must have canonical “shape”

```
                                                    index++;
                                                    ++index;
                          <                         index--;
                          <=                        --index;
for (index=start; index           end;              index += inc;             )
                          >=                        index -= inc;
                          >                         index = index + inc;
                                                    index = inc + index;
                                                    index = index – inc;
```

Sections construct

- The sections construct gives a different structured block to each thread

```
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
     x_calculation();
#pragma omp section
     y_calculation();
#pragma omp section
     z_calculation();
}
```

parallel region

worksharing

each section gets assigned to a different thread

end of **omp sections**

- By default there is a barrier at the end. The **`nowait`** clause turns it off

Single construct

- The structured block is executed only by one of the threads
- An implicit barrier exists at the end of **`single`**
- Can be considered as a synchronization construct

```
#pragma omp parallel
{
        do_many_things();
        #pragma omp single
        {
                exchange_boundaries();
        }
        do_many_other_things();
}
```

implicit barrier here

and here, end of parallel region

Combined Directives

- Parallel regions can be combined with the `for` and `sections` worksharing constructs

- **`omp parallel`** + **`omp for`** → **`omp parallel for`**

```
#pragma omp parallel for
for (i=0; i<N; i++){
        do_work(i);
}
```

Combined Directives

- **`omp parallel`** + **`omp sections`** → **`omp parallel sections`**

```
#pragma omp parallel sections
{
#pragma omp section
    x_calculation();
#pragma omp section
    y_calculation();
#pragma omp section
    z_calculation();
}
```

Directive Scoping

- OpenMP directives can be extended in multiple files
- Orphan directives: appear outside a parallel region

```
//foo.c

#pragma omp parallel
{
  whoami();
}
```

```
//bar.c
void whoami()
{
  int iam = omp_get_thread_num();
  #pragma omp critical
  {
    printf("Hello from %d"\n, iam);
  }
  return;
}
```

synchronization

- foo.c: Static (lexical) extent of parallel region
- bar.c: Dynamic extent of parallel region

Data Scoping

- OpenMP is a shared memory programming model
  - most variables are shared by default
- Global variables are shared
- But not everything is shared
  - loop index variables
  - stack variables in called functions from parallel region

Storage Attributes

- The programmer can change the storage attributes of variables with the following clauses
  - **`shared`**
  - **`private`**
  - **`firstprivate`**
  - **`threadprivate`**
- The value of a private variable used in a parallel loop can be exported as global value with the clause:
  - **`lastprivate`**
- The default behavior can be changed using:
  - **`default(private | shared | none)`**
- The data clauses are applied to the parallel region and worksharing constructs - however, shared is only valid for parallel regions
- Data scoping clauses are valid only in the lexical extent of the OpenMP directive

Data Environment

- Example of **`private`** and **`firstprivate`**

```
int A, B, C;
A = B = C = 1;
#pragma omp parallel private(B) firstprivate(C)
{
   // ...
}
```

- Within the parallel region :
  - "A" is shared between threads and equal to 1
  - Both "B" and "C" are private for each thread
    - B has undefined initial value
    - C has initial value equal to 1
- After the parallel region:
  - Both B and C have the same value as before the parallel region

private

- **`private`**(var) creates a private copy of var in each thread
  - The value of the copy is not initialized
  - The private copy is not related to the original variable with respect to the memory location

```
int is = 0;
#pragma omp parallel for private(is)
for (int j=1; j<=1000; j++)
    is = is + j;

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

- IS has not been initialized inside the loop

firstprivate

- **`firstprivate`**: special case of **`private`**
  - The private copy of each thread is initialized with the value of the original variable, which belongs to the master thread

```
int is = 0;
#pragma omp parallel for firstprivate(is)
for (int j=1; j<=1000; j++)
    is = is + j;

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

- Each thread has a private copy of IS with initial value 0

lastprivate

- Copies the value of the private variable, as assigned by the last loop iteration, to the original (global) variable

continue to the next line

```
         int is = 0;
        #pragma omp parallel for firstprivate(is) \
 lastprivate(is)
        for (int j=1; j<=1000; j++)
            is = is + j;

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

- Each thread has a private copy of IS with initial value 0
- IS has the value it was assigned by the last loop iteration (i.e. for j=1000)

Synchronization

- OpenMP supports several synchronization constructs:
  - `critical` section
  - `atomic`
  - `barrier`
  - `master` (in fact, not a synchronization construction)
  - *`ordered`* not studied
  - *`flush`* not studied

Synchronization – critical

- No two threads will simultaneously be in the critical section
- Critical sections can be named
    - `omp critical (name)`

```
#pragma omp parallel for private(b) shared(res)
for (i=0; i<niters; i++) {
    b = doit(i);
    #pragma omp critical
    {                                        lock mutex
        update(b, &res);
    }                                        unlock mutex
}
```

res: initialized before the parallel region

Synchronization – atomic

- Special case of critical section that can be used only for simple instructions.
- Can be applied only when a single memory location (variable) is updated

```
#pragma omp parallel private(b)
{
    int i = omp_get_thread_num();
    b = doit(i);
    #pragma omp atomic
        res = res + b;
}
```

use of some hardware-supported atomic operation

res: initialized before the parallel region

Synchronization – barrier

- Barrier: all threads wait until each thread has reached the barrier

```
#pragma omp parallel shared (A, B) private(id)
{
      id=omp_get_thread_num();
      A[id] = big_calc1(id);
      #pragma omp barrier

      #pragma omp for
      for(int i=0; i<N; i++){
            B[i]=big_calc2(i,A);
      }
}
```

initialization of A

necessary synchronization

these computations depend on A

Synchronization – master

- The structured block is executed only by the **master thread** - the other threads of the team ignore it
- There is no barrier at the end of `master`

```
#pragma omp parallel
{
        do_many_things();
        #pragma omp master
        {
                exchange_boundaries();
        }
        #pragma barrier
        do_many_other_things();
}
```

nothing more than
if (omp_get_thread_num()==0)

Synchronization – ordered

- Enforces the sequential order of execution of some code

```
#pragma omp parallel private (res)
{
      #pragma omp for ordered
      for (i=0; i<N; i++){
            res = do_work(i);

            #pragma ordered
            printf(“res = %d\n”, res);
      }
}
```

Synchronization - Implicit Barriers

- A barrier is implicitly called at the end of the following constructs:
  - **`parallel`**
  - **`for`** (except when nowait is used)
  - **`sections`** (except when nowait is used)
  - **`single`** (except when nowait is used)

– **`for`**, **`sections`** and **`single`** accept the **`nowait`** clause

```
int nthreads;

#pragma omp parallel
#pragma omp single nowait
nthreads = omp_get_num_threads();
```

Reductions

- The reduction clause modifies the way variables are "shared":
  - `reduction (op : list)`
- Variables included in `list` must be shared in the parallel region where the reduction clause exists
- Allowed reduction operations: `+`,`-`,`*`,`&`,`^`,`|`,`&&`,`||`,`min`, `max`
- Within a parallel region or a worksharing construct:
  - A local copy for each variable in the list is created and initialized accordingly to the reduction operation
    - 0 for "+"
  - The values of the local copies are combined (reduced) to a single value that is stored to the original variable after the end of the construct

Reduction - Example

```
#include <omp.h>
#define NUM_THREADS 2

double func(int i);

void main ()
{
   int i;
   double ZZ, res=0.0;
   omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:res) private(ZZ)
   for (i=0; i< 1000; i++){
       ZZ = func(i);
       res = res + ZZ;
   }
}
```

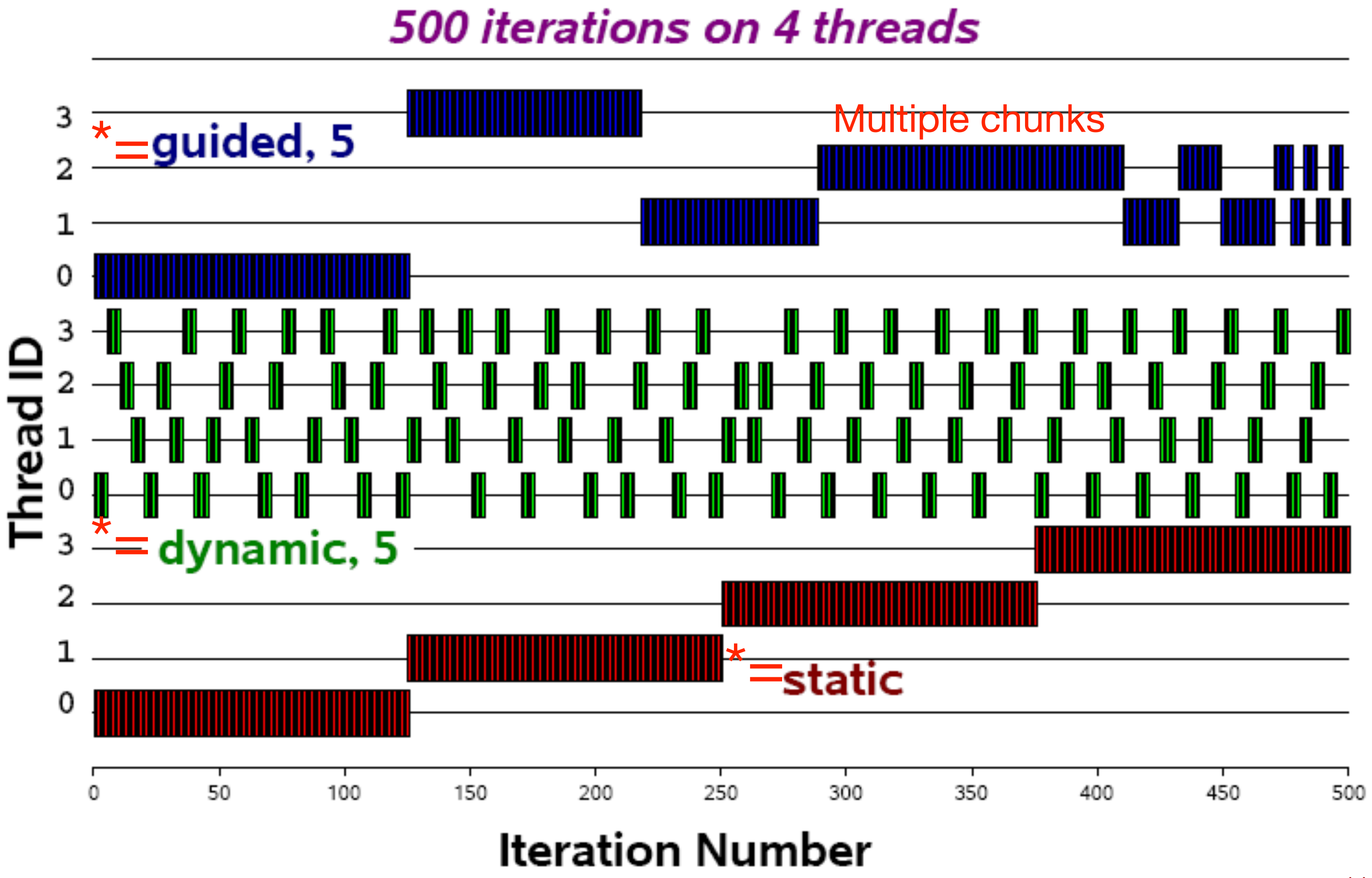
Loop Scheduling

- Usage: **`#pragma omp parallel for <schedule clause>`**
  - `schedule ( static | dynamic | guided [, chunk] )`
  - `schedule (runtime)`

- **`static [,chunk]`**
  - Loop iterations are divided into segments of size **`chunk`** and distributed cyclically to the threads of the parallel region
  - If **`chunk`** is not specified, it is equal to N/P and each thread executes a single chunk of iterations
- **`dynamic [,chunk]`**
  - Loop iterations are divided into segments of size **`chunk`**
  - An idle thread gets dynamically the next available chunk of iterations
- **`guided [,chunk]`**
  - Similar to dynamic but the chunk size decreases exponentially.
  - **`chunk`** <u>specifies the minimum segment size</u>
- **`runtime`**
  - decide at runtime depending on the OMP_SCHEDULE environment variable
- **`auto`**
  - decided by the compiler and/or the underlying OpenMP runtime library

Example

```
#pragma omp parallel for num_threads(4) schedule(*)
for (int i = 0; i < 500; i++) do_work(i);
```



More details in the next lecture and the exercises

Library Calls

- OpenMP locks
  - `omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock()`
- Functions that control the runtime environment:
  - Number of threads
    - `omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()`
  - Dynamic mode and nested parallelism
    - `omp_set_dynamic(), omp_set_nested(),`
    - `omp_get_dynamic(), omp_get_nested()`
  - Check if code is in a parallel region
    - `omp_in_parallel()`
  - Number of processors / cores
    - `omp_get_num_procs()`
- Wall-clock time measurement (in seconds)
  - `omp_get_wtime()`

ex01: get_wtime()

OpenMP Locks

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int tmp = do_lots_of_work(id);

    omp_set_lock(&lck);
    printf("%d %d\n", id, tmp);
    omp_unset_lock(&lck);
}

omp_destroy_lock(&lck);
```

lock variable
initialization

destruction

Libraries Calls

- Dynamic mode is disabled and then the number of threads is specified. This ensures that the parallel region will have 4 threads.

```
#include <omp.h>
void main()
{
    omp_set_dynamic(0);
    omp_set_num_threads(4);
    #pragma omp parallel
    {
          int id=omp_get_thread_num();
          do_lots_of_stuff(id);
    }
}
```

Environment Variables

- Default number of threads
  - **`OMP_NUM_THREADS int_literal`**
- Control of dynamic mode
  - **`OMP_DYNAMIC TRUE || FALSE`**
- Control of nested parallelism
  - **`OMP_NESTED TRUE || FALSE`**
- Control of loop scheduling if the programmer has used **`omp for schedule(RUNTIME)`**
  - **`OMP_SCHEDULE “schedule[, chunk_size]”`**
- Control of threads binding
  - **`OMP_PROC_BIND TRUE || FALSE`**

"Use Cases"

Case 1: Loop & Parallel Region

- Parallelize the following sequential code with
  - parallel regions
  - worksharing

```
#define N 1024
for(int i=0; i<N; i++) { a[i] = a[i] + b[i];}
```

- OpenMP parallel region

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int Nthrds = omp_get_num_threads();
  int istart = id * N / Nthrds;
  int iend = (id+1) * N / Nthrds;
  if (id == omp_get_num_threads()-1) iend = N;
  for(int i=istart; i<iend; i++) {a[i] = a[i] + b[i];}
}
```

adjustment for the last thread

Loop & Worksharing

- Sequential code

```
#define N 1024
for(int i=0; i<N; i++) { a[i] = a[i] + b[i];}
```

- OpenMP parallel region with worksharing

```
#pragma omp parallel
{
    #pragma omp for schedule(static)          default scheduling
    for(int i=0; i<N; i++) { a[i] = a[i] + b[i];}
}
```

or simply:

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

Case 2: Functional parallelism

- Parallelize the following sequential code
  - what is the total execution time if each function takes one second?

```
V = alpha();
W = beta();
X = gamma(V, W);
Y = delta();
printf("%f\n", epsilon(X,Y));
```

total time = 5s

Functional parallelism - Solution 1

```
#pragma omp parallel num_threads(3)
#pragma omg sections
{
     #pragma omp section
     V = alpha();

     #pragma omp section
     W = beta();

     #pragma omp section
     Y = delta();
}
X = gamma(V, W);
printf("%f\n", epsilon(X,Y));
```

no sense to use more threads

total time = 3s

Functional parallelism - Solution 2

```
#pragma omp parallel num_threads(2)
{
   #pragma omp sections
   {
        #pragma omp section
        V = alpha();

        #pragma omp section
        W = beta();
   }
   #pragma omp sections
   {
        #pragma omp section
        X = gamma(V, W);

        #pragma omp section
        Y = delta();
   }
}
printf(“%f\n”, epsilon(X,Y));
```

no sense to use more threads

implicit barrier

total time = 3s
but with fewer threads

Case 3 - Reductions

- Parallelize the following sequential code

```
long num_steps = 100000;
double step;

void main ()
{
  double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;


  for (int i=0; i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;

  printf("Pi is %lf\n", pi);
}
```

Using the reduction clause

```
long num_steps = 100000;
double step;

void main ()
{
  double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;

  #pragma omp parallel for reduction(+:sum) private(x)
  for (long i=0; i<num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;

  printf("Pi is %lf\n", pi);
}
```

References

- OpenMP Specifications & Quick Reference Card
  - www.openmp.org
- OpenMP tutorial at LLNL, Blaise Barney
  - https://computing.llnl.gov/tutorials/openMP/
- An Overview of OpenMP, Ruud van der Pas – Sun Microsystems
  - http://www.openmp.org/wp-content/uploads/ntu-vanderpas.pdf