Shared-memory programming : OpenMP (II)

ING2-GSI-MI Architecture et Programmation Parallèle

Juan Angel Lorenzo del Castillo

juan-angel.lorenzo-del-castillo@cyu.fr

CY Cergy Paris Université

2023-2024

Table of Contents

Table of Contents

OpenMP directives

Most relevant OpenMP directives

- Parallel regions construction
  - `parallel`
- Work sharing
  - `for`, `sections`, `single`.
- Synchronisation
  - `master`, `critical`, `atomic`, `barrier`.
- Task management
  - `task`, `taskwait`.

There are more...

OpenMP directives

Most relevant OpenMP directives

- Parallel regions construction
  - `parallel`
- Work sharing
  - **`for`**, **`sections`**, **`single`**.
- Synchronisation
  - `master`, `critical`, `atomic`, `barrier`.
- Task management
  - `task`, `taskwait`.

There are more...

Today's class

Work sharing

#pragma omp for

Syntax:

```
#pragma omp for [clauses]
    for loop
```

- A parallel *for* loop divides up the iterations of the loop between threads.
- With no additional clauses, the *for* directive will usually partition the iterations as equally as possible between the threads.
- The iterations of the loop will be executed in parallel by the thread team.
- The parallel loop index variable is private.
- No new threads are created: the iterations of the loop are divided up between the existing threads from the thread team.
- **Implicit barrier** at the end of the loop unless the **`nowait`** clause was used.

Work sharing

#pragma omp for

Syntaxe :

```
#pragma omp for [clauses]
   for loop
```

Allowed clauses:

- `private(list of variables)`
- `firstprivate(list of variables)`
- `lastprivate(list of variables)`
- `reduction(operator: list of variables)`
- `schedule(type[,block size])`
- `ordered`
- `collapse(positive value)`
- `nowait`

Work sharing

#pragma omp for

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

int main () {

 int i,id;
 const int SIZE = 12;
 int A[SIZE];

 omp_set_num_threads(4);

 #pragma omp parallel
 {
  #pragma omp for private(id)
  for (i=0; i<SIZE;i++)
  {
   id=omp_get_thread_num();
   A[i] = id;
  }
 }

 for (i=0; i<SIZE; i++) {
   printf(" | %d",A[i]);
 }
   printf(" | \n");

 return 0;
}
```

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

int main () {

 int i,id;
 const int SIZE = 12;
 int A[SIZE];

 omp_set_num_threads(4);

 #pragma omp parallel for private(id)
  for (i=0; i<SIZE;i++)
  {
   id=omp_get_thread_num();
   A[i] = id;
  }


 for (i=0; i<SIZE; i++) {
   printf(" | %d",A[i]);
 }
   printf(" | \n");

 return 0;
}
```

Work sharing

#pragma omp for

What is the difference between these two programs?

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

int main ()
{
 int tid;
 omp_set_num_threads(4);

 #pragma omp parallel private(tid)
 {
   for (int i=0;i<10;i++)
   {
    tid = omp_get_thread_num();
    printf("Thread %d  i = %d\n",tid,i);
   }
 }
 return 0;
}
```

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

int main ()
{
 int tid;
 omp_set_num_threads(4);

 #pragma omp parallel for private(tid)
  for (int i=0;i<10;i++)
  {
   tid = omp_get_thread_num();
   printf("Thread %d  i = %d\n",tid,i);
  }

 return 0;
}
```

Work sharing

#pragma omp for private(list of variables)
#pragma omp for firstprivate(list of variables)
#pragma omp for reduction(operator: list of variables)

- Already seen in the *Parallel region* section.

Work sharing

#pragma omp for lastprivate(list of variables)

- The variables of the list are declared **private**.
  - Reminder: private variables are undefined before and after the parallel *for* loop.
- The variables of the list are updated with their last value at the end of the parallel region.
- This last value is the value given by the thread that has executed the last iteration of the loop.

Work sharing

#pragma omp for lastprivate(list of variables)

Example:

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

int show(int, int);

int main () {
  int a = 2;
  int tid;

  omp_set_num_threads(4);

  # pragma omp parallel for
        firstprivate(a) lastprivate(a)
        private(tid)
  for (int i = 0; i < 5; i++)
  {
    tid = omp_get_thread_num();
    a++;
    show(tid,a);
  }

  printf("Master. a: %d\n",a);
  return 0;
}
```

```c
int show(int id, int a)
{
  // ignore this pragma for now
  #pragma omp critical
  {
    printf("Thread %d. a: %d\n",id,a);
  }
  return 0;
}
```

Work sharing

#pragma omp for schedule(type[,block size])

- Gives a variety of options for specifying which loops iterations are executed by which thread.
- Important for load balancing purposes (i.e. performance).
- Types:
  - schedule(static[,chunk])
  - schedule(dynamic[,chunk])
  - schedule(guided[,chunk])
  - schedule(runtime)
  - schedule(auto)

Work sharing

#pragma omp for schedule(static[,chunk])

- With no `chunk` specified, the iteration space is divided into (approximately) equal chunks (num_it/num_threads)), and one `chunk` is assigned to each thread (***block** schedule*).
- If `chunk` is specified, the iteration space is divided into chunks, each of `chunk` iterations, and the chunks are assigned cyclically to each thread (**block cyclic** schedule)

#pragma omp for schedule(dynamic[,chunk])

- It divides the iteration space up into chunks of size `chunk` (like *static*), but chunks are assigned to threads on a *first-come-first-served* basis.
- i.e. as a thread finishes a chunk, it is assigned the next chunk in the list.
- When no `chunk` is specified, it defaults to 1.

Work sharing

#pragma omp for schedule(guided[,chunk])

- similar to *dynamic*, but the chunks start off large (num_it/num_threads) and get smaller exponentially.
- The size of the next chunk is (roughly) the number of remaining iterations divided by the number of threads.
- The `chunk` specifies the minimum size of the chunks.
- When no `chunk` is specified, it defaults to 1.

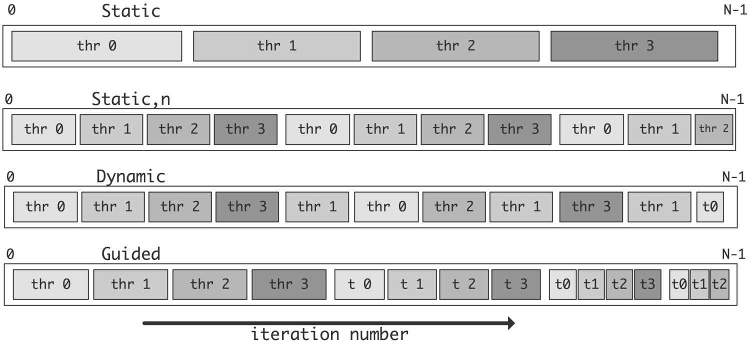
#pragma omp for schedule(runtime)

- It defers the choice of schedule to run time, when it is determined by the value of the environment variable `OMP_SCHEDULE`:
  - `export OMP_SCHEDULE = "guided,4"`

#pragma omp for schedule(auto)

- The choice of scheduling is left to the compiler.

Work sharing



Source : utexas.edu

Work sharing

When to choose which schedule:

- `Static` is the best for already-load balanced loops (least overhead).
- `Static,n` is good for loops with mild or smooth load imbalance, but can induce *false sharing* (to be seen in the exercises).
- `Dynamic` is useful if iterations have widely varying loads, but ruins data locality.
- `Guided` is often less expensive than `Dynamic`, but beware of loops where the first iterations are the most expensive.
- `Runtime` can be used for convenient experimentation of the different scheduling techniques.

Work sharing

#pragma omp for collapse(positive value)

When we have perfectly nested loops we can collapse inner loops. Collapsing a loop:

- The loop needs to be perfectly nested.
- The loop needs to have rectangular iteration space.
- Makes iteration space larger: iterations from all loops are grouped to make a single iteration space that will be shared among the threads.
- Less synchronisation needed than nested parallel loops.

```
#pragma omp for collapse(2)
for(i=0; i<10; i++)
  for(j=0; j<10; j++)
    myfunction(i,j);
```

⇔ (similar to)

```
#pragma omp for
for(i=0; i<100; i++)
   myfunction(i/10,i%10);
```

Work sharing

#pragma omp for nowait

- Synchronisation directive to suppress the implicit barriers at the end of `for`, `sections` and `single` directives (barriers are expensive).

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

int main () {

  int i;
  const int SIZE = 12;
  int A[SIZE];
  int B[SIZE];
  int C[SIZE];
  int D[SIZE];

  omp_set_num_threads(4);

  // Initialisation
  for (i = 0; i < SIZE; i++)
  {
   A[i] = 100;
   B[i] = 100;
   C[i] = 100;
   D[i] = 100;
  }

  #pragma omp parallel private(i) shared(A,B,C,D)
  {
    #pragma omp for nowait
    for (i = 0; i < SIZE-1; i++)
    {
     B[i] = (A[i] + A[i+1])/2;
    }

    #pragma omp for
    for (i = 0; i < SIZE;i++)
    {
     D[i] = 1/C[i];
    }
  }

  return 0;
}
```

Work sharing

#pragma omp sections

Syntax:

```
#pragma omp sections [clauses]
{
  #pragma omp section
     structured block
  #pragma omp section
     structured block
  ...
}
```

- Allows separate blocks of code to be executed in parallel (e.g. several independent subroutines)
- **Implicit barrier** at the end, unless **`nowait`** is specified.
- Rarely used, except with nested parallelism.

Work sharing

#pragma omp sections

Syntax (shorthand form):

```
#pragma omp [parallel] sections [clauses]
{
  #pragma omp section
     structured block
  #pragma omp section
     structured block
   ...
}
```

Allowed clauses:

- `private(liste de variables)`
- `firstprivate(liste de variables)`
- `lastprivate(liste de variables)`
- `reduction(operateur : liste de variables)`
- `nowait`

Work sharing

#pragma omp single

Syntax:

```
#pragma omp single [clauses]
{
   structured block
}
```

- The block of code will be executed by a single thread only.
- The first thread to reach the `single` directive will execute the block.
- Other threads wait until the block has been executed.
- **Implicit barrier** at the end unless **`nowait`** is specified.

Work sharing

#pragma omp single

Syntax:

```
#pragma omp single [clauses]
{
    structured block
}
```

Allowed clauses:

- `private(list of variables)`
- `firstprivate(list of variables)`
- `copyprivate(list of variables)`
  - At the end of the `single` directive, the value of the private variable(s) is copied to the private variable(s) of the other threads.
  - Incompatible with `nowait`.
- `nowait`