OpenMP Examples (1A)

-
-

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice and Octave.

Installation

STEP 1: Check the GCC version of the compiler
gcc –version

STEP 2: Configuring OpenMP
echo | cpp -fopenmp -dM |grep -i open
sudo apt install libomp-dev

STEP 3: Setting the number of threads
export OMP_NUM_THREADS=8

https://www.geeksforgeeks.org/openmp-introduction-with-installation-guide/

Parallel regions

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    int nthreads, tid;

    // Begin of parallel region
    #pragma omp parallel private(nthreads, tid)
    {
        // Getting thread number
        tid = omp_get_thread_num();
        printf("Welcome to GFG from thread = %d\n", tid);
        if (tid == 0) {
            // Only master thread does this
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
}
```

https://www.geeksforgeeks.org/openmp-introduction-with-installation-guide/

Private variables

```
#include <omp.h>

main(int argc, char *argv[]) {

    int nthreads, tid;

    /* Fork a team of threads with each thread having a private tid variable */
    #pragma omp parallel private(tid)
    {

        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

    }  /* All threads join master thread and terminate */

}
```

https://computing.llnl.gov/tutorials/openMP/#Compiling

OpenMP Code Structure

```
#include <omp.h>

main ()  {
   int var1, var2, var3;
   Serial code
   …

   Beginning of parallel region. Fork a team of threads.
   Specify variable scoping

   #pragma omp parallel private(var1, var2) shared(var3)
   {
      Parallel region executed by all threads
      Other OpenMP directives
      Run-time Library calls
      All threads join master thread and disband
   }

   Resume serial code
   ...
}
```

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

OpenMP Directives

#pragma omp parallel [**clause** ...] newline
if (scalar_expression)
private (list)
shared (list)
default (shared | none)
firstprivate (list)
reduction (operator: list)
copyin (list)
num_threads (integer-expression)

structured_block

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

OpenMP Directives

Directive name
A valid OpenMP directive.
Must appear after the pragma and before any clauses.

[clause, …]
Optional.
Clauses can be in any order, and repeated as necessary
unless otherwise restricted.

Newline
Required.
Precedes the structured block
which is enclosed by this directive.

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

Installation

Compile:
gcc -fopenmp test.c

Execute:
./a.out

https://www.geeksforgeeks.org/openmp-introduction-with-installation-guide/

Number of cores

grep processor /proc/cpuinfo | wc -l

sysconf(_SC_NPROCESSORS_CONF)
sysconf(_SC_NPROCESSORS_ONLN)

grep -c ^processor /proc/cpuinfo

grep -c ^cpu /proc/stat # subtract 1 from the result

https://stackoverflow.com/questions/150355/programmatically-find-the-number-of-cores-on-a-machine

OpenMP API Overview

The OpenMP 3.1 API is comprised of three distinct components:

- **Compiler Directives**
- **Runtime Library Routines**
- **Environment Variables**

https://computing.llnl.gov/tutorials/openMP/#API

Compiler Directives

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

https://computing.llnl.gov/tutorials/openMP/#API

Runtime Library Routines

- Setting and querying the number of threads
- Querying a thread's unique identifier (thread ID),
  a thread's ancestor's identifier, the thread team size
- Setting and querying the dynamic threads feature
- Querying if in a parallel region, and at what level
- Setting and querying nested parallelism
- Setting, initializing and terminating locks and nested locks
- Querying wall clock time and resolution

https://computing.llnl.gov/tutorials/openMP/#API

Environment Variables

- Setting the number of threads
- Specifying how loop iterations are divided
- Binding threads to processors
- Enabling/disabling nested parallelism;
  setting the maximum levels of nested parallelism
- Enabling/disabling dynamic threads
- Setting thread stack size
- Setting thread wait policy

https://computing.llnl.gov/tutorials/openMP/#API

Examples

Compiler Directive Examples

```
#pragma omp parallel
#pragma omp parallel private(partial_Sum) shared(total_Sum)
#pragma omp parallel private(thread_id)
#pragma omp barrier
#pragma omp for
#pragma omp critical
```

Runtime Library Routine Examples

```
omp_get_thread_num();
omp_get_max_threads();
```

https://stackoverflow.com/questions/150355/programmatically-find-the-number-of-cores-on-a-machine

Hello

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

int main(int argc, char** argv) {

    printf("Hello from process: %d\n", omp_get_thread_num());

    return 0;
}


// only one thread giving us a Hello statement
// must use the #pragma omp parallel { … } directive
// for multiple threads
```

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Hello

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

int main(int argc, char** argv){
    int thread_id;

    #pragma omp parallel
    {
        printf("Hello from process: %d\n", omp_get_thread_num());
    }
    return 0;
}
```

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Private clauses

The PRIVATE clause declares variables in its list
to be private to each thread.

- A new object of the same type is declared once
  for each thread in the team
- All references to the original object are replaced with
  references to the new object
- Should be assumed to be uninitialized for each thread

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Shared clauses

The SHARED clause declares variables in its list t
o be shared among all threads in the team.

A shared variable exists in only one memory location and
all threads can read or write to that address

It is the programmer's responsibility to ensure that
multiple threads properly access SHARED variables
(such as via CRITICAL sections)

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Shared clauses

Variables that are created and assigned inside of a parallel section of code will be inherently be **private**

variables created outside of parallel sections will be inherently **public**.

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Data Sharing Rules – Implicit Rules

```
int n = 10;                    // shared
int a = 7;                     // shared

#pragma omp parallel for
for (int i = 0; i < n; i++)    // i private
{
   int b = a + i;              // b private
   ...
}
```

http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html

Data Sharing Rules – Explicit Rules

```
#pragma omp parallel for shared(n, a)
for (int i = 0; i < n; i++)
{
    int b = a+ i;
    ...
}
```

```
#pragma omp parallel for shared(n, a) private(b)
for (int i = 0; i < n; i++)
{
    b = a + i;
    ...
}
```

http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html

Data Sharing Rules – Explicit Rules

```
int p = 0;
// the value of p is 0

#pragma omp parallel private(p)
{
    // the value of p is undefined
    p = omp_get_thread_num();
    // the value of p is defined
    ...
}
// the value of p is undefined
```

```
#pragma omp parallel
{
    int p = omp_get_thread_num();
    ...
}
```

http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html

Data Sharing Rules – Default(Shared)

int **a**, **b**, **c**, **n**;

...

```
#pragma omp parallel for default(shared)
for (int i = 0; i < n; i++)
{
    // using a, b, c
}
```

http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html

Data Sharing Rules – Default(none)

```
int n = 10;
std::vector<int> vector(n);
int a = 10;

#pragma omp parallel for default(none) shared(n, vector)
for (int i = 0; i < n; i++)
{
    vector[i] = i * a;
}
```

```
error: 'a' not specified in enclosing parallel
        vector[i] = i * a;
                    ^
error: enclosing parallel
    #pragma omp parallel for default(none) shared(n, vector)
            ^
```

http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html

Data Sharing Rules – Default(none)

```
int n = 10;
std::vector<int> vector(n);
int a = 10;
```

```
#pragma omp parallel for default(none) shared(n, vector, a)
for (int i = 0; i < n; i++)
{
    vector[i] = i * a;
}
```

http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html

Data Sharing Rules – Default(none)

The default context of a variable is determined by the following rules:

- **static** variables – **shared**.
- **auto** variables in a **parallel** region – **private**
- **dynamically allocated** objects – **shared**.
- **heap allocated** variables – **shared**.
  there can be only one **shared heap**.
- all variables defined outside a **parallel** construct
- **– shared** in a **parallel** region
- **loop iteration** variables are **private** within their loops.
  the value of the iteration variable after the **loop**
  is the same as if the **loop** were run sequentially.
- memory allocated within a **parallel** loop
  by the **alloca** function
  persists only for the duration of one iteration,
  and is **private** for each thread.

https://www.ibm.com/support/knowledgecenter/SSLTBW_2.4.0/com.ibm.zos.v2r4.cbcpx01/cuppvars.htm

alloca()

NAME

alloca - allocate memory that is automatically freed

SYNOPSIS

```
#include <alloca.h>

void *alloca(size_t size);
```

DESCRIPTION

The alloca() function allocates size bytes of space in the stack frame of the caller. This temporary space is automatically freed when the function that called **alloca**() returns to its caller.

RETURN VALUE

The **alloca**() function returns a pointer to the beginning of the allocated space. If the allocation causes stack overflow, program behavior is undefined.

https://man7.org/linux/man-pages/man3/alloca.3.html

Data Sharing Rules – Default(none)

```
int E1;                          /* shared static                    */

void main (argvc,...) {          /* argvc is shared                  */
   int i;                        /* shared automatic                 */

void *p = malloc(...);           /* memory allocated by malloc       */
                                 /* is accessible by all threads (shared) */
                                 /* and cannot be privatized         */
```

https://www.ibm.com/support/knowledgecenter/SSLTBW_2.4.0/com.ibm.zos.v2r4.cbcpx01/cuppvars.htm

Data Sharing Rules – Default(none)

```
void main (argvc,...) {              // argvc is shared
   int i;      void *p = malloc(...);

   #pragma omp parallel firstprivate (p)
   {
      int b;                         // private automatic
      static int s;                  // shared static

      #pragma omp for
      for (i =0;...) {
         b = 1;                      // b is still private here !
         foo (i);                    // i is private here because it is an iteration variable
      }
      #pragma omp parallel
      {
         b = 1;                      // b is shared here because it
      }                              // is another parallel region
   }
}
```

https://www.ibm.com/support/knowledgecenter/SSLTBW_2.4.0/com.ibm.zos.v2r4.cbcpx01/cuppvars.htm

Data Sharing Rules – Default(none)

```
int E2;                    /*shared static */

void foo (int x) {           /* x is private for the parallel */
                          /* region it was called from     */

int c;                    /* the same */
 ... }
```

https://www.ibm.com/support/knowledgecenter/SSLTBW_2.4.0/com.ibm.zos.v2r4.cbcpx01/cuppvars.htm

Data Sharing Rules – Default(none)

The **private** clause declares the variables in the list to be
private to each thread in a team.

The **firstprivate** clause provides a superset of the functionality
provided by the private clause.
The private variable is initialized by the original value of the variable
when the parallel construct is encountered.

The **lastprivate** clause provides a superset of the functionality
provided by the private clause.
The private variable is updated after the end of the parallel construct.

The **shared** clause declares the variables in the list to be
shared among all the threads in a team.
All threads within a team access the same storage area for shared variables.

The **reduction** clause performs a reduction on the scalar variables
that appear in the list, with a specified operator.

The **default** clause allows the user
to affect the data-sharing attribute of the variables appeared in the parallel construct.

https://www.ibm.com/support/knowledgecenter/SSLTBW_2.4.0/com.ibm.zos.v2r4.cbcpx01/cuppvars.htm

Nested Parallelism (1)

```
void fun1()
{
   for (int i=0; i<80; i++)
      ...
}

main()
{
  #pragma omp parallel
  {
     #pragma omp for
     for (int i=0; i<100; i++)
        ...

     #pragma omp for
     for (int i=0; i<10; i++)
        fun1();
   }
}
```

the 2nd loop in **main**
can only be distributed to 10 threads

80 loop iterations in **fun1**
which will be called 10 times in **main** loop.

total **800** iterations in **fun1** and the **main** loop

This gives much more parallelism potential
if parallelism can be added in both levels.

https://software.intel.com/content/www/us/en/develop/articles/exploit-nested-parallelism-with-openmp-tasking-model.html

Nested Parallelism (2)

```
void fun1()
{
   #pragma omp parallel for
   for (int i=0; i<80; i++)
      ...
}

main
{
   #Pragma omp parallel
   {
      #pragma omp for
      for (int i=0; i<100; i++)
         …

      #pragma omp for
      for (int i=0; i<10; i++)
         fun1();
   }
}
```

The problem with this implementation is you may either have insufficient threads for the 1st main loop as it has larger loop count, or create exploded number of threads for the 2nd main loop when OMP_NESTED=TRUE. The simple solution is to split the parallel region in main and create separate ones for each loop with a distinct thread number specified.

https://software.intel.com/content/www/us/en/develop/articles/exploit-nested-parallelism-with-openmp-tasking-model.html

Nested Parallelism (3)

```
void fun1()
{
    #pragma omp taskloop
    for (int I = 0; i<80; i++)
        ...
}

main
{
   #pragma omp parallel
   {
      #pragma omp for
      for (int i=0; i<100; i++)
          ...

      #pragma omp for
      for (int i=0; i<10; i++)
          fun1();
   }
}
```

As you can see, you don't have to worry about the thread number changes in 1st and 2nd main loops. Even though you still have a small amount of (10) threads allocated for 2nd main loop, the rest available threads will be able to be distributed through omp taskloop in fun1.

https://software.intel.com/content/www/us/en/develop/articles/exploit-nested-parallelism-with-openmp-tasking-model.html

Hello

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

int main(int argc, char** argv){
    int thread_id;

    #pragma omp parallel private(thread_id)
    {
        thread_id = omp_get_thread_num();
        printf("Hello from process: %d\n", thread_id );
    }

    return 0;
}
```

// create a separate instance of thread_id for each task.

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Barrier and critical directives

#pragma omp **barrier**

The barrier directive stops all processes
for proceeding to the next line of code
until all processes have reached the barrier.
This allows a programmer
to **synchronize** sequences in the parallel process.

#pragma omp **critical** { … }

A critical directive ensures that
a line of code is only run by one process at a time,
ensuring **thread safety** in the body of code.

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Barrier (1)

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

int main(int argc, char** argv){
    //define loop iterator variable outside parallel region
    int i;
    int thread_id;

    #pragma omp parallel
    {
        thread_id = omp_get_thread_num();

        //create the loop to have each thread print hello.
        for(i = 0; i < omp_get_max_threads(); i++){
            printf("Hello from process: %d\n", thread_id);
        }
    }
    return 0;
}
```

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Barrier (2)

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

int main(int argc, char** argv){
    int i;
    int thread_id;

    #pragma omp parallel
    {
        thread_id = omp_get_thread_num();

        for(i = 0; i < omp_get_max_threads(); i++){
            if(i == thread_ID){
                printf("Hello from process: %d\n", thread_id);
            }
        }
    }
    return 0;
}
```

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Barrier (3)

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

int main(int argc, char** argv){
    int i;
    int thread_id;

    #pragma omp parallel
    {
        thread_id = omp_get_thread_num();

        for( int i = 0; i < omp_get_max_threads(); i++){
            if(i == omp_get_thread_num()){
                printf("Hello from process: %d\n", thread_id);
            }
            #pragma omp barrier
        }
    }
    return 0;
}
```

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

OMP for

OpenMP's power comes from
easily splitting a larger task into multiple smaller tasks.
Work-sharing directives allow for simple and effective **splitting**
of normally serial tasks into fast parallel sections of code.

The directive omp for divides a normally serial for loop into a parallel task.

#pragma omp for { … }

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

OMP for

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

int main(int argc, char** argv){
   int partial_Sum, total_Sum;

#pragma omp parallel private(partial_Sum) shared(total_Sum)
{
   partial_Sum = 0;
   total_Sum = 0;

   #pragma omp for
   {
      for(int i = 1; i <= 1000; i++){
         partial_Sum += i;
      }
   }

   //Create thread safe region.
   #pragma omp critical
   {
         //add each threads partial sum to the total sum
         total_Sum += partial_Sum;
   }
}

   printf("Total Sum: %d\n", total_Sum);
   return 0;
}
```

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

References

[1] en.wikipedia.org
[2] M Harris, http://beowulf.lcs.mit.edu/18.337-2008/lectslides/scan.pdf