OpenMP

Open Multi-Processing

OpenMP (Open Multi-Processing)

Open standard for parallelization of programs in C, C++, Fortran languages

Interface standard for parallel programming on shared memory

Brief history:

- Version 1.0 1997
- Version 4.0 July 2013
- Version 4.1 in progress

Compilers: GCC, Intel C++ compiler, Visual C++

Advantages Of OpenMP

- Ease of use
- Cross-platform for shared memory systems
- Hiding low-level operations
- Support for parallel and serial versions of programs

Core components

- Environment variable
- Compiler directive
- Functions

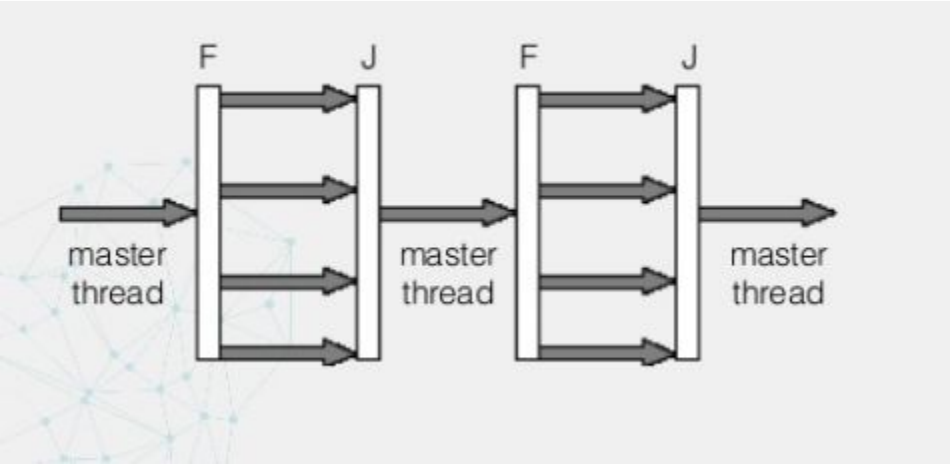
Steps of development

1. Write and debug a sequential program
2. Add OpenMP directives to the program
3. Use compiler with support for OpenMP
4. Set the environment variables
5. Run the program

Compilation

gcc source.c -o source.x **-fopenmp**

Model Fork-Join



- Explicit indication of parallel sections
- Support for nested parallelism
- Dynamic threads support

Simple program

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

void main() {
    #pragma omp parallel
    printf("Hellow word\n");
}
```

Environment variable

OMP_NUM_THREADS - Sets the number of threads in the parallel block. By default, the number of threads is equal to the number of virtual processors.

OMP_DYNAMIC - Allows or disables dynamic changes in the number of threads that are actually used for calculations (depending on system load). The default value depends on the implementation.

OMP_NESTED - Allows or disables nested parallelism (parallelization of nested loops). The default is disabled.

Functions of OpenMP

- **omp_set_num_threads** To set the number of threads
- **omp_get_num_threads** Return the number of threads in the group
- **omp_get_max_threads** Maximum number of threads
- **omp_get_thread_num** ID of thread
- **omp_get_num_procs** Maximum number of processors
- **omp_in_parallel** Check for location in a parallel region

Directives (Directive Format (C / C ++))

#pragma omp < name> [<attributes> {[,] <attributes>}]

name - the name of the **Directive**;

Attributes - a construction specifying additional information and depending on the **Directive**;

Directive parallel

#pragma omp parallel [<attributes>]

<structural block>

- **A thread that encounters a parallel construct creates a thread group, becoming the master.**
- **Threads are assigned unique integer numbers starting with 0 (the main thread).**
- **Each thread executes code defined by a building block, at the end of which a barrier is implicitly set**

Declaring a parallel section

```
#include <omp.h>
int main()
{
   // Sequential code
   #pragma omp parallel
   {
      // Parallel code
   }
   // Sequential code

   return 0;
}
```

Условное объявление параллельной секции

```
#include <omp.h>
int main()
{
   //  Sequential code
   #pragma omp parallel if (expr)
   {
      //  Parallel code
   }
   //  Sequential code

   return 0;
}
```

The parallel Directive. Simple example

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

void main() {
    #pragma omp parallel
    printf("Hellow word\n");
}
```

Examples

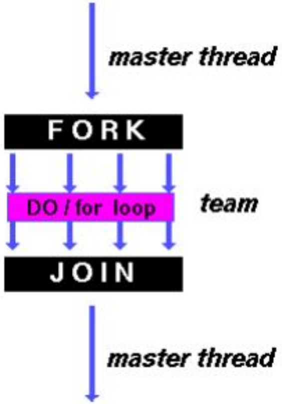
omp_hello.c

omp_hello_functions.c

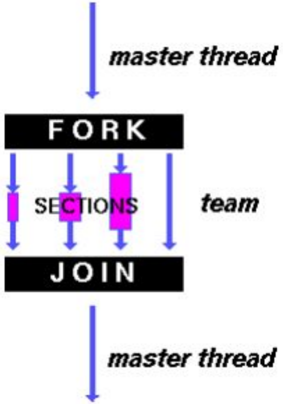
omp_hello_env.c

Ways to divide work between threads

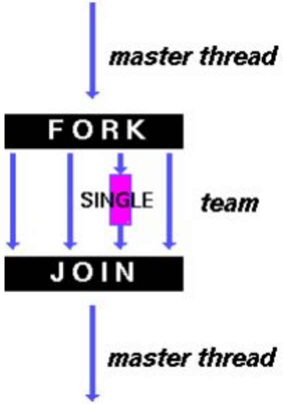
```
#pragma omp for
  for (i=0;i<N;i++)
  {
    // code
  }
```




```
#pragma omp sections
{
 #pragma omp section
 // code 1
 #pragma omp section
 // code 2
}
```




```
#pragma omp single
{
  // code
}
```



Directive **for**

#pragma omp for [<attributes>]

<loop body>

Restrictions:

- **The only counter is an integer, pointer, or random access iterator. Should only change in the loop header.**
- **Loop condition: comparing a variable with a loop-invariant expression using <, <=, >, >=.**
- **Changing the counter: using ++,--, i += d, i -= d, i = i + d, i = d + i, i = i - d (d — cycle-invariant integer expression)**

Directive **for**

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel
  {
    #pragma omp for
    for (i=0;i<1000;i++)
      printf("%d ",i);
  }
  return 0;
}
```

Directive **for**

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;

  #pragma omp parallel for
    for (i=0;i<1000;i++)
      printf("%d ",i);

  return 0;
}
```

Directive **sections**

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel sections private(i)
  {
    #pragma omp section
    { printf("1st half\n");
      for (i=0;i<500;i++) printf("%d ",i);
    }
    #pragma omp section
    { printf("2nd half\n");
      for (i=501;i<1000;i++) printf("%d ",i);
    }
  }
  return 0;
}
```

In this case, you need to keep in mind the need to support the serial version of the program.

Directive **omp single**

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel private(i)
  {
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
    #pragma omp single
      printf("I'm thread %d!\n",get_thread_num());
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
  }
  return 0;
}
```

Directive **omp single**

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel private(i)
  {
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
    #pragma omp single
      printf("I'm thread %d!\n",get_thread_num());
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
  }
  return 0;
}
```

barrier

Directive **omp single**

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel private(i)
  {
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
    #pragma omp single nowait
      printf("I'm thread %d!\n",get_thread_num());
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
  }
  return 0;
}
```

No barrier

Directive **omp master**

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
  #pragma omp parallel private(i)
  {
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
    #pragma omp master
      printf("I'm Master!\n")
    #pragma omp for
      for (i=0;i<1000;i++) printf("%d ",i);
  }
  return 0;
}
```

No barrier

The scope of variables

Variables declared inside a parallel block are local to the thread:

```
#pragma omp parallel
{
  int num = omp_get_thread_num();
  printf("Thread %d\n",num);
}
```

The scope of variables

Variables declared outside the parallel block **are shared by default** across all threads:

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

```
int num;
#pragma omp parallel
{
  num = omp_get_thread_num();
  printf("Thread %d\n",num);
}
```

The scope of variables

The scope of variables declared outside the parallel block is determined by the Directive parameters:

- private
- firstprivate
- lastprivate
- shared
- default
- reduction
- threadprivate
- copyin

The scope of variables

The scope of variables declared outside the parallel block is determined by the Directive parameters:

- **private - Own local variable in each thread**
- firstprivate
- lastprivate
- shared
- reduction

```
int num;
#pragma omp parallel private(num)
{
  num = omp_get_thread_num();
  printf("%d\n",num);
}
```

The scope of variables

The scope of variables declared outside the parallel block is determined by the Directive parameters:

- private
- **firstprivate - Local variable with initialization**
- lastprivate
- shared
- reduction

```c
int num = 5;
#pragma omp parallel \
                firstprivate(num)
{
  printf("%d\n",num);
}
```

The scope of variables

The scope of variables declared outside the parallel block is determined by the Directive parameters:

- private
- firstprivate
- **lastprivate**
- shared
- reduction

Local variable with the last value saved (in sequential execution)

```
int i,j;
#pragma omp parallel for \
            lastprivate(j)
for (i=0;i<100;i++) j = i;

printf("Последний j = %d\n",j);
```

The scope of variables

The scope of variables declared outside the parallel block is determined by the Directive parameters:

- private
- firstprivate
- lastprivate
- **shared**
- reduction

Shared (shared) variable

```
int i,j;

#pragma omp parallel for \
            shared(j)
for (i=0;i<100;i++) j = i;

printf("j = %d\n",j);
```

The scope of variables

The scope of variables declared outside the parallel block is determined by the Directive parameters:

- private
- firstprivate
- lastprivate
- shared
- **reduction**

The variable to perform the reducing operation

```
int i,s = 0;
#pragma omp parallel for \
              reduction(+:s)
  for (i=0;i<100;i++)
    s += i;

printf("Sum: %d\n",s);
```

Synchronizing threads

Thread synchronization directives:
- master
- barrier
- critical
- atomic

Locks:
- omp_lock_t

Synchronizing threads

Thread synchronization directives:

- **master**
- barrier
- critical
- atomic

Выполнение кода только главным потоком

```
#pragma omp parallel
{
  //code
  #pragma omp master
  {
    // critical code
  }
  // code
}
```

Synchronizing threads

Thread synchronization directives:

- master
- **barrier**
- critical
- atomic

```
#pragma omp parallel
{
  printf("Hello!\n");

  #pragma omp barrier
  printf("I am thread %d\n",
           omp_get_thread_num());
}
```

Synchronizing threads

Thread synchronization directives:

- master
- barrier
- **critical**
- atomic

```
int i,idx[N],x[M];

#pragma omp parallel for
for (i=0;i<N;i++)
{
  #pragma omp critical
  {
    x[idx[i]] += count(i);
  }
}
```

Synchronizing threads

Thread synchronization directives:

- master
- barrier
- critical
- **atomic**

```
int i,idx[N],x[M];

#pragma omp parallel for
for (i=0;i<N;i++)
{
  #pragma omp atomic
    x[idx[i]] += count(i);
}
```

Locks

- Lock-a special object common to threads
- Threads can capture (lock) and release (unlock) a lock
- Only one thread at a time can capture a lock
- When attempting to capture a lock, threads wait for the lock to be released
- Using locks, you can control access to shared resources

An example of using the lock

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int x[1000];
int main()
{ int i,max;
  omp_lock_t lock;
  omp_init_lock(&lock);
  for (i=0;i<1000;i++) x[i]=rand();
  max = x[0];
  #pragma omp parallel for
    for(i=0;i<1000;i++)
    { omp_set_lock(&lock);
      if (x[i]>max) max = x[i];
      omp_set_unlock(&lock);
    }
  omp_destroy_lock(&lock);
  return 0;
}
```

Example: single vs critical

```c
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
    #pragma omp single
    a++;
    #pragma omp critical;
    b++;
}

printf("single: %d -- critical: %d\n", a, b);
```

Example: single vs critical

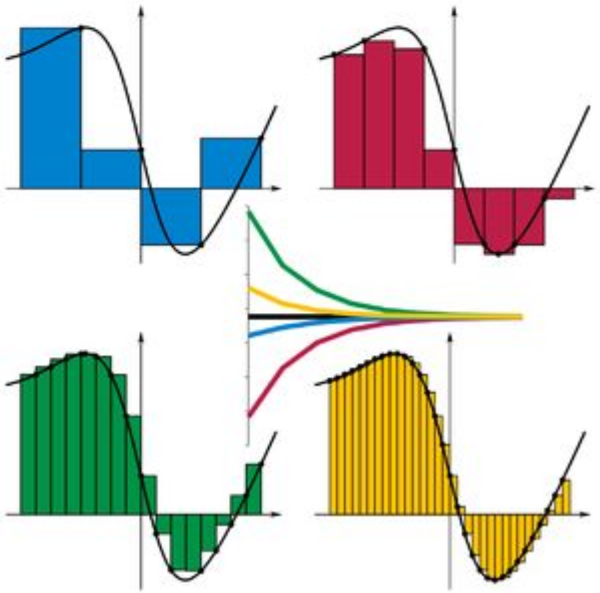
Output:

single: 1 -- critical: 4

Example

Numerical integration (Riemann sum)

Numerical integration is the approximate computation of an integral using numerical techniques



Source: omp_integral_4.c

Exercise 5a

Create program for Pi calculation based on OpenMP. Implement two versions:

1. Without reduction
2. Use reduction attribute

Questions:

Compare execution time of two versions.