A Short Introduction to OpenMP

OpenMP

- **`Open Multi-Processing`**
  - API for shared memory multi-processing in Fortran, C and C++
- **`Simpler than MPI, and simpler than pthread`**
  - Coarse and fine grain parallelisation
  - Fork-join model
  - Since version 3.0, supports arbitrary tasks
- **`Compiler directives and environment variables`**
  - Requires compiler support
  - Intel compiler: OpenMP v2.5, GCC 4.2: v2.5, GCC 4.4: v3.0, GOMP extension for earlier GCC versions, Microsoft Visual Studio 2005: v2.0, Sun Studio: v2.5, Pathscale: v2.5

hello.c

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

int main(int argc, char *argv[])
{
#pragma omp parallel
   printf("Hello from thread %d out of %d",
          omp_get_thread_num(), omp_get_num_threads());
   printf("Hello again from thread %d",
          omp_get_thread_num());
   return 0;
}
```

Compiling and executing

```
%> gcc –fopenmp –Wall hello.c
%> icc –openmp –Wall hello.c
%> export OMP_NUM_THREADS=4
%> ./a.out
Hello from thread 0 out of 4
Hello from thread 1 out of 4
Hello from thread 2 out of 4
Hello from thread 3 out of 4
Hello again from thread 0
```

- **The number of threads can also be set within the source code using the *omp_set_num_threads* function**

All the examples assume that 4 threads are available

Execution model

- **`Synchronization at beginning and end of parallel regions`**
  - Can be refined via synchronization directives
- **`By default, threads in parallel regions share all variables`**
  - Visibility can be refined via data placement directives

Parallel sections

```
int main(int argc, char *argv[])
{
#pragma omp parallel sections
  {
#pragma omp section
    printf("Hello from thread %d out of %d",
               omp_get_thread_num(),omp_get_num_threads());
#pragma omp section
    printf("Hello again from thread %d",
               omp_get_thread_num());
  }
  return 0;
}
```

Output:

```
Hello from thread 0 out of 4
Hello again from thread 1
```

Parallel for loops

```
#pragma omp parallel
{
#pragma omp for
  for (int i=0; i<size; ++i)
  {
    do stuff in parallel
  }
}
```

- **`Loop index is private within each thread`**
- **`Fortran`**
  - Parallel DO loops
  - Parallel workshare (exists for C/C++ but not supported by current compilers)

Parallel for loops: scheduling

- **`schedule` directive describes distribution of iterations onto threads**
- **`RUNTIME`**
  - Default strategy, uses strategy defined in OMP_SCHEDULE environment variable, or static by default
- **`STATIC`**
  - Iterations are pre-assigned to threads before loop execution, iterations are grouped into chunks
  - Chunk size parameter specifies iterations count per block
- **`DYNAMIC`**
  - Chunks are assigned dynamically to threads
  - Chunk size parameter specifies iterations count per block
- **`GUIDED`**
  - Block sizes are decreasing according to exponential law
  - Chunk parameter specifies minimum chunk size to be used

Load balancing

- **Good load balancing depends on**
  - The algorithm
  - The scheduling strategy
  - The specified chunk size

```
#pragma omp parallel for schedule(static, 200)
  for (int i=0; i<size; ++i)
  {
    if (i<20) do stuff else do nothing
  }
```

- **Here: bad load balance for chunk size of 200, much better if chunk size is 5**
- **For dynamic strategy: small chunk → more overheads, large chunks → less chunks to distribute**

Data placement directives

- **`Except for loop indices, variables are shared by default`**

- **`Variables on the stack can be made private to be replicated by each thread`**
  - Only variables on the stack can be made private

- **`private`**
  - Variable is undefined at beginning of parallel region
- **`firstprivate`**
  - Variables takes last value outside parallel region
- **`lastprivate`**
  - Only for parallel for loops, after exiting the loop, the variable takes the value reached in the last iteration
- **`shared`**
  - Default behavior

Data placement: shared

```
int i; int mynum = 10; int a = 10;

printf("a=%d before parallel region\n", a);
#pragma omp parallel for
for (i=0; i<4; ++i) {
  mynum = omp_get_thread_num();
  a += mynum;
  printf("a=%d in thread %d\n", a, mynum);
}
printf("a=%d after parallel region\n", a);
```

Output:

```
a=10 before parallel region
a=10 in thread 0
a=11 in thread 1
a=13 in thread 2
a=16 in thread 3
a=16 after parallel region
```

Data placement: private

```
int i; int mynum = 10; int a = 10;
printf("a=%d before parallel region\n", a);
#pragma omp parallel for private(a)
for (i=0; i<4; ++i) {
  mynum = omp_get_thread_num();
  a += mynum;
  if (mynum==0) sleep(1); /* thread 0 sleeps 1 second */
  printf("a=%d in thread %d\n", a, mynum);
}
printf("a=%d after parallel region\n", a);
```

Output:

```
a=10 before parallel region
a=10 in thread 3
a=8 in thread 1
a=9 in thread 2
a=6322816 in thread 3
a=10 after parallel region
```

mynum is indeed shared

Data placement: firstprivate

```
int i; int mynum = 10; int a = 10;
printf("a=%d before parallel region\n", a);
#pragma omp parallel for firstprivate(a, mynum)
for (i=0; i<4; ++i) {
  mynum = omp_get_thread_num();
  a += mynum;
  printf("a=%d in thread %d\n", a, mynum);
}
printf("a=%d after parallel region\n", a);
```

Output:

```
a=10 before parallel region
a=13 in thread 3
a=12 in thread 2
a=11 in thread 1
a=10 in thread 0
a=10 after parallel region
```

Data placement: lastprivate

```
int i; int mynum = 10; int a = 10;
printf("a=%d before parallel region\n", a);
#pragma omp parallel for lastprivate(a) private(mynum)
for (i=0; i<4; ++i) {
  mynum = omp_get_thread_num();
  a += mynum;
  printf("a=%d in thread %d\n", a, mynum);
}
printf("a=%d after parallel region\n", a);
```

Output:

```
a=10 before parallel region
a=6363522 in thread 2
a=6380803 in thread 3
a=6317952 in thread 0
a=6346113 in thread 1
a=6380803 after parallel region
```

Synchronization directives

- **Operations**
  - ATOMIC
  - BARRIER
  - REDUCTION

- **Blocs**
  - CRITICAL
  - SINGLE

- **Buffer flush**
  - FLUSH

- **Remove synchronization**
  - NOWAIT
- **Remove synchronization**

Atomic execution

- **Accesses to a shared variable must be protected**

```
#pragma omp parallel for
for (int i=0; i<N; ++i) {
  for (int j=0; j<N; ++j) {
#pragma omp atomic
    sum += A[i][j] ;
  }
}
```

- **Equivalent to using *pthread* locks, but much simpler**

Reduction

- **All-to-one operation in a parallel for**

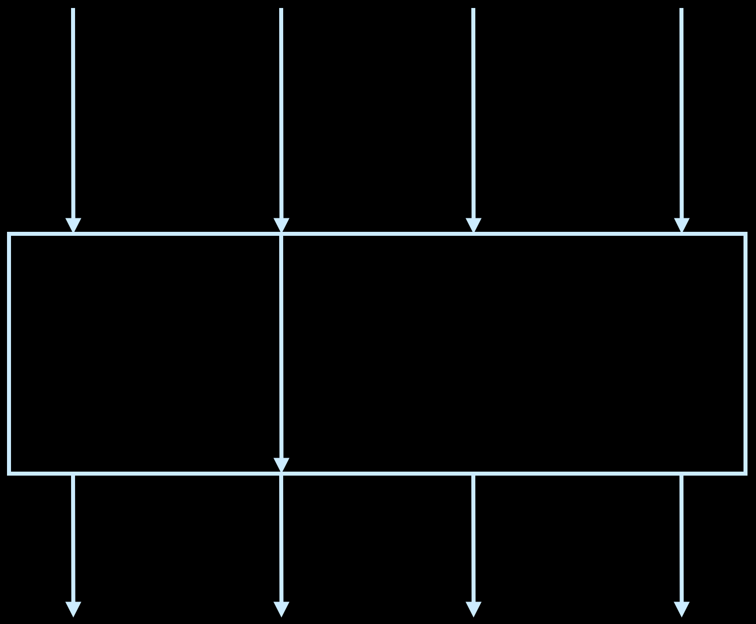
```
#pragma omp parallel for reduction(+:_sum)
for (int i=0; i<N; ++i) {
  for (int j=0; j<N; ++j) {
    _sum += A[i][j]
  }
}
```

- **The OpenMP runtime sums the values of each thread at the end of the loop**

Single directive

- **Only first incoming thread executes a single region**
- **The other threads wait, unless nowait is specified**

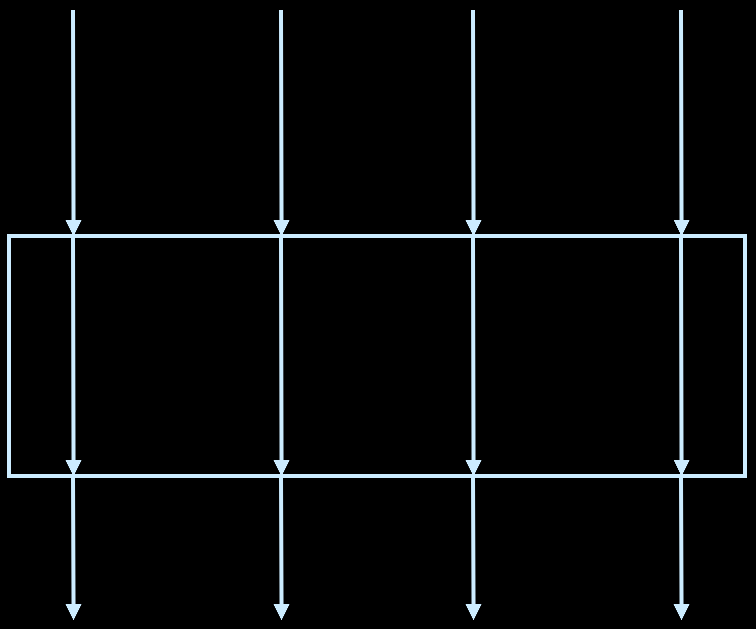
```
#pragma omp parallel
{
  ...
#pragma omp single
  {
    ...
  }
  ...
}
```



Critical directive

- **At a given time, only one thread can be in a critical region**

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



References

- **`OpenMP 2.0`**

  http://www.openmp.org/mp-documents/cspec20.pdf

- **`OpenMP 2.5`**

  http://www.openmp.org/mp-documents/spec25.pdf

- **`OpenMP 3.0`**

  http://www.openmp.org/mp-documents/spec30.pdf