An Introduction To OpenMP

Matthew Willyard

Financial Mathematics
Florida State University

Graduate Student Seminar
February 5

Outline

Why Bother To Use Parallel Code

- Why go to the extra effort of writing parallel code?

Why Bother To Use Parallel Code

- Why go to the extra effort of writing parallel code?
  - Most new computers have multiple cores

Why Bother To Use Parallel Code

- Why go to the extra effort of writing parallel code?
  - Most new computers have multiple cores
  - In addition to multiple cores many processors also have hyperthreading enabled

Why Bother To Use Parallel Code

- Why go to the extra effort of writing parallel code?
  - Most new computers have multiple cores
  - In addition to multiple cores many processors also have hyperthreading enabled
  - Most importantly it is easy to convert sequential code to parallel code using

Amdahl's Law

$$\text{Absolute Speedup} = \frac{\text{Sequential Algorithm Time}}{\text{Parallel Algorithm Time}} = \frac{1}{(1-P) + \frac{P}{N}}$$

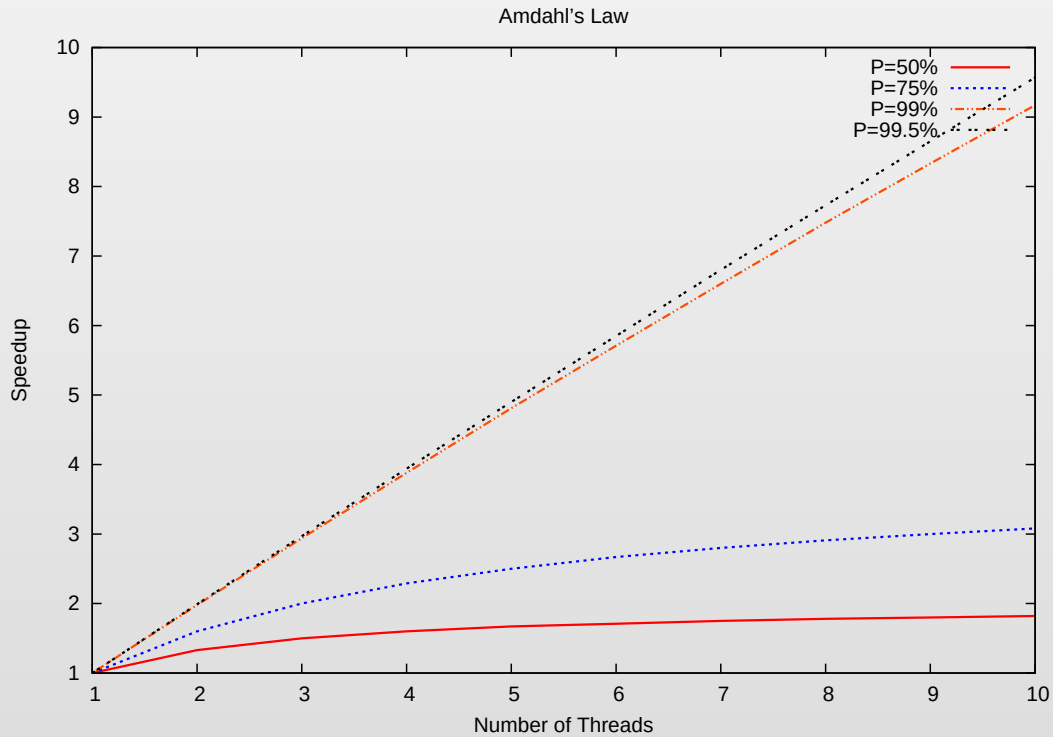


Figure: Speedup vs. Number of Threads

Open Multi-Processing(OpenMP)

API for explicitly direct multi-threaded, shared memory parallelism

Open Multi-Processing(OpenMP)

API for explicitly direct multi-threaded, shared memory parallelism

- Advantages
  - Portable
    - Works with C/C++ and Fortran
    - Works on Unix/Linux and Windows
  - Easy to implement

Open Multi-Processing(OpenMP)

API for explicitly direct multi-threaded, shared memory parallelism

- Advantages
  - Portable
    - Works with C/C++ and Fortran
    - Works on Unix/Linux and Windows
  - Easy to implement
- Disadvantages
  - Shared memory only
  - Large overhead
  - Threads executed in non-deterministic order

How OpenMP works

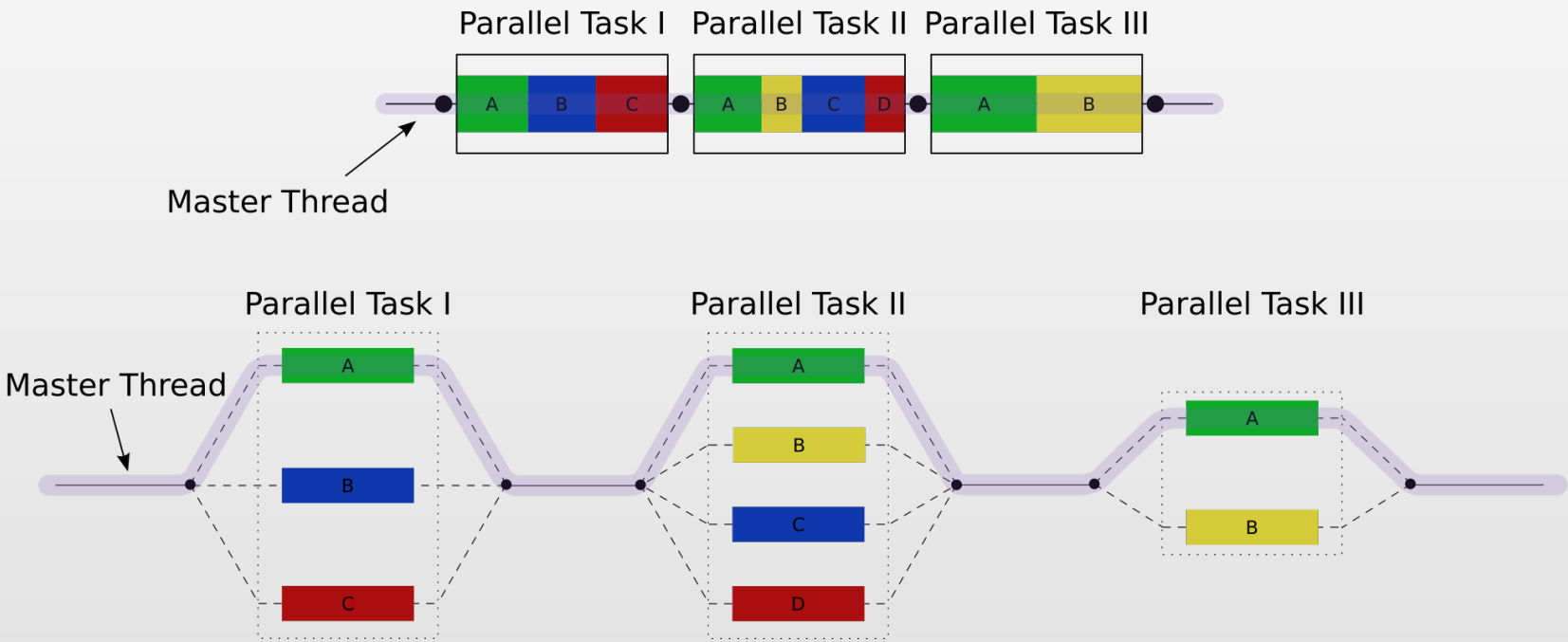


Figure: Fork-Join Model

How to use OpenMP

Steps to Parallize Code

1. Add compiler directives
   - !$omp DIRECTIVE in Fortran
   - #pragma omp DIRECTIVE in C/C++
2. Compile compiler option -openmp
3. Set the number or threads with
   - export OMP_NUM_THREADS=4
   - setenv OMP_NUM_THREADS 4
4. Run program like normal ./OpenMPProgram

Problem Description

Goals

1. Fork multiple threads
2. Have each thread output a statement to the screen
3. Join the threads together and exit

Hello World

Code

```
#include <omp.h>
#include <iostream>
int main (int argc, char *argv[]) {
 int th_id, nthreads;

/* Fork a team of threads with
each thread having a private th_id variable */
#pragma omp parallel private(th_id)
 {
/* Obtain and print thread id */
 th_id = omp_get_thread_num();
  std::cout << "Hello World from thread" << th_id << "\n";

/* Wait for all threads to get to this point */
#pragma omp barrier

/* Only master thread does the command below */
 if ( th_id == 0 ) {
   nthreads = omp_get_num_threads();
   std::cout << "There are " << nthreads << " threads\n";
  }
 }
 return 0;
}
```

Output

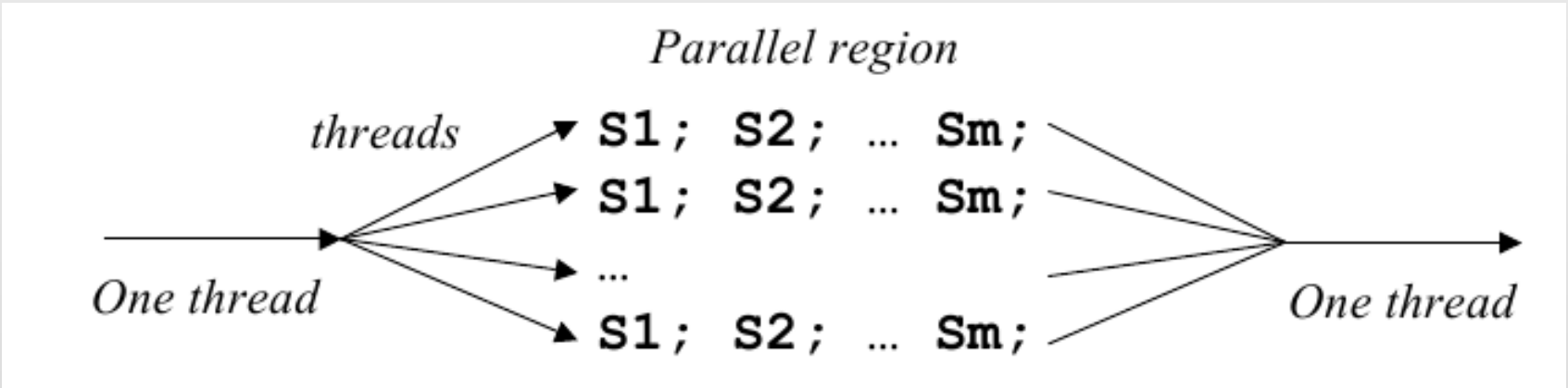
```
Hello World from thread4
Hello World from thread1
Hello World from thread2
Hello World from thread5
Hello World from thread0
Hello World from thread3
Hello World from thread6
Hello World from thread7
There are 8 threads
```

Parallel Directive

Parallel

- Starts a group of threads and executes the body statements and joins the threads when done.

```
#pragma omp parallel default(none) shared(x,y) private(z)
{
S1;
S2:
...
Sm
}
```

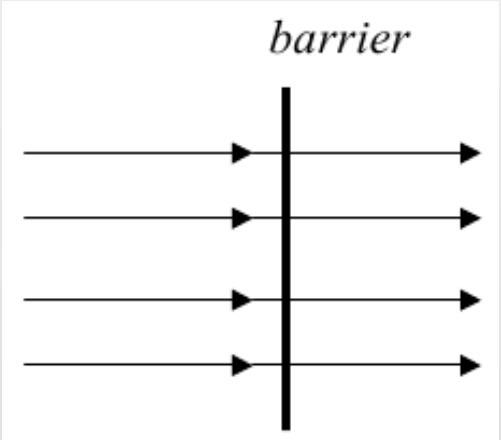


Barrier Directive

Barrier

- The barrier directive synchronizes the threads.

```
#pragma omp barrier
```



Problem Description

Goals

$$\text{solve } Ax = b$$

we will use Jacobi Iteration $x_i^k = \frac{1}{a_{i,i}} \left[b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1} \right]$

not Gauss-Sidel Iteration $x_i^k = \frac{1}{a_{i,i}} \left[b_i - \sum_{j<i} a_{i,j} x_j^k - \sum_{j>i} a_{i,j} x_j^{k-1} \right]$

Jacobi Iteration

Sequential Code

```
x=b
do iteration=1,MaxIteration
  do i=1, N
    tempsum=-A(i,i)*x(i)
    do j=1,N
      tempsum=tempsum+A(i,j)*x(j)
    enddo
      newx(i)=(b(i)-tempsum)/A(i,i)
  enddo
  x=newx
enddo
```

Parallel Code

```
!$omp parallel shared(A,b,newx,N) private(iteration,i,j,tempsum)
!$omp do
do i=1, N
  x(i)=b(i)
enddo
!$omp end do
do iteration=1,MaxIteration
!$omp do
  do i=1, N
    tempsum=-A(i,i)*x(i)
    do j=1,N
      tempsum=tempsum+A(i,j)*x(j)
    enddo
    newx(i)=(b(i)-tempsum)/A(i,i)
  enddo
!$omp end do
!$omp do
  do i=1, N
    x(i)=newx(i)
  enddo
!$omp end do
enddo
!$omp end parallel
```

for/do Loops Directives

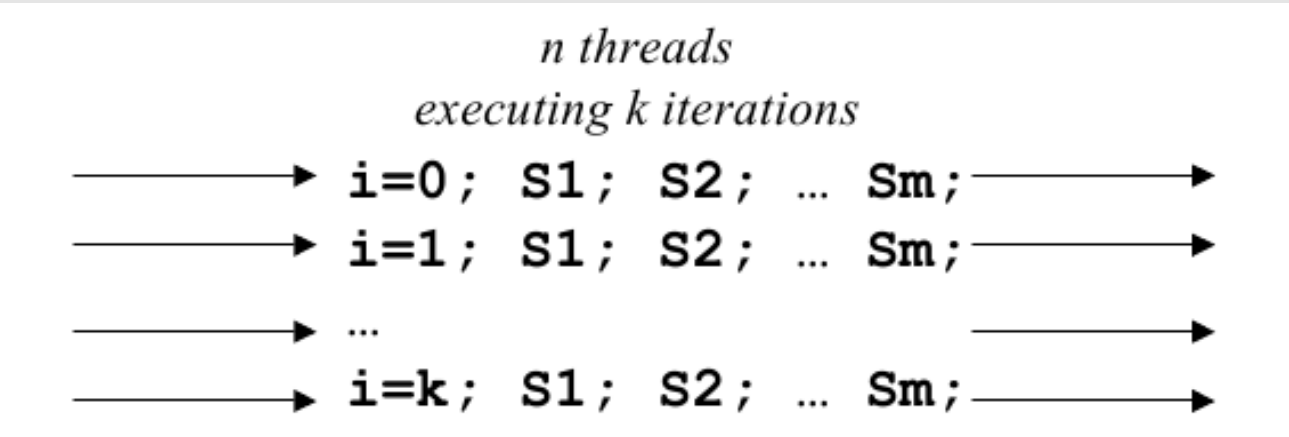
for/Do

- The for directive in c/c++ and the do directive in Fortran are work-sharing directives where a group of threads execute a loop concurrently.

```
#pragma omp for
for(i=0;i<=k;i++)
{
  S1;
  S2;
  ...
  Sm;
}
```

```
!$omp end do
do i=0,k

  S1
  S2
  ...
  Sm
enddo
!$omp end do
```

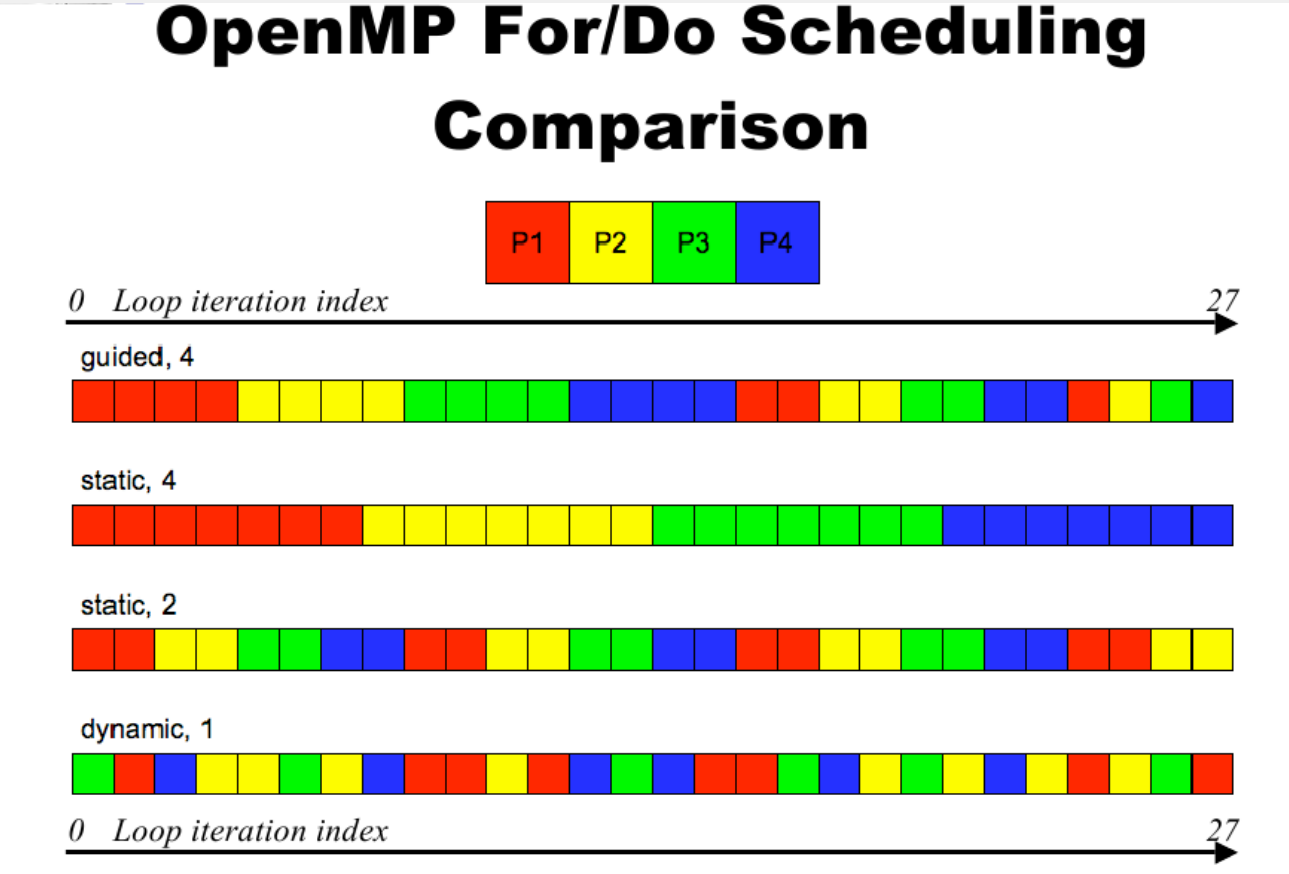


for/DO Loops Scheduling

for/Do

- To change the loop scheduling you could use

```
#pragma omp for schedule(guided,4) or !$omp end do schedule(static 4)
```



Problem Description

Goals

Solve the European option problem with Monte Carlo

1. Simulate N price paths
2. Find the exercise price of each path
3. Find the average value of all paths

Monte Carlo

European Option

```
!$omp parallel default(private) shared(Value)
Value=0
Call Init_Random_Number_Generator()
t1=OMP_GET_WTIME()
!$omp do schedule(static) REDUCTION(+:Value)
do j=1,Number_of_Simulations
      V=0
      S = S_0
      Call Normal_Random_Number_Generator( Z⃗ )
      do i=1,Number_of_Time_Steps
            S = S * e^((r − 1/2 σ^2)Δt+σ√(Δt)Z(i))
      enddo
      V=max(K−S,0)
      Value=Value+V
enddo
!$omp end do
t2=OMP_GET_WTIME()
time=t2−t1
!$omp end parallel
Option_Price= Value/Number_of_Simulations
```

for/DO Loops Directives

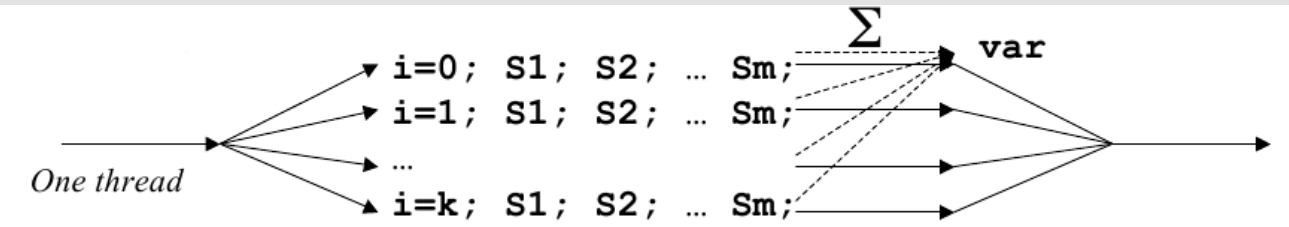
REDUCTION(operator:Value)

- The reduction option performs a global reduction over all threads
- REDUCTION(operator:Value) can take on operators +, *, -, .AND., .OR., .EQV., or .NEQV

```
#pragma omp for reduction(+:Value)
for(i=0;i<=k;i++)
{
  S1;
  S2;
  ...
  Sm;
  Value=Value+...
}
```

```
!$omp do schedule(static) REDUCTION(+:Value)
do i=0,k

  S1
  S2
  ...
  Sm
  Value=Value+...
enddo
!$omp end do
```



Problem Description

Goals

Compute the matrix matrix multiply $AB = C$

Sequential Matrix Matrix Multiply

Sequential Code

```
#define NRA 62  /* number of rows in matrix A */
#define NCA 15  /* number of columns in matrix A */
#define NCB 7   /* number of columns in matrix B */
double A[NRA][NCA],B[NCA][NCB],C[NRA][NCB];
int i, j, k;
  /*** Initialize matrices ***/
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      A[i][j]= i+j;
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      B[i][j]= i*j+1;
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
      C[i][j]= 0;
/*** Perform Multiplication ***/
for(i=0;i<NRA;i++)
    for(j=0;j<NCB;j++)
for(k=0;k<NCA;k++)
    C[i][j]+=A[i][k]*B[k][j];
/*** Print Results ***/
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)
{
  for (j=0; j<NCB; j++)
    printf("%6.2f   ", C[i][j]);
  printf("\n");
}
```

Parallel Matrix Matrix Multiply Version 1

Parallel Code

```
int chunk;
chunk = 10;
#pragma omp parallel shared(A,B,C,chunk) private(i,j,k)
{
  /*** Initialize matrices ***/
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      A[i][j]= i+j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      B[i][j]= i*j+1;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
      C[i][j]= 0;
  /*** Perform Multiplication ***/
  #pragma omp for schedule (static, chunk)
  for(i=0;i<NRA;i++)
    for(j=0;j<NCB;j++)
      for(k=0;k<NCA;k++)
        C[i][j]+=A[i][k]*B[k][j];
}
/*** Print Results ***/
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)
{
  for (j=0; j<NCB; j++)
    printf("%6.2f   ", C[i][j]);
  printf("\n");
}
```

Parallel Matrix Matrix Multiply Version 2

Parallel Code

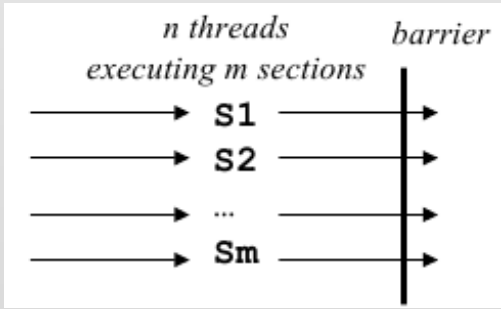
```
#pragma omp parallel shared(A,B,C,chunk) private(i,j,k)
{ /*** Initialize matrices ***/
  #pragma omp sections
  {
    #pragma omp section
    for (i=0; i<NRA; i++)
      for (j=0; j<NCA; j++)
        A[i][j]= i+j;
    #pragma omp section
    for (i=0; i<NCA; i++)
      for (j=0; j<NCB; j++)
        B[i][j]= i*j+1;
    #pragma omp section
    for (i=0; i<NRA; i++)
      for (j=0; j<NCB; j++)
        C[i][j]= 0;
  } /*** Perform Multiplication ***/
  #pragma omp for schedule (static, chunk)
  for(i=0;i<NRA;i++)
    for(j=0;j<NCB;j++)
      for(k=0;k<NCA;k++)
        C[i][j]+=A[i][k]*B[k][j];
  #pragma omp single
  { /*** Print Results ***/
    printf("Result Matrix:\n");
    for (i=0; i<NRA; i++)
    {
      for (j=0; j<NCB; j++)
        printf("%6.2f   ", C[i][j]);
      printf("\n");
    }
  }
}
```

Sections Directive

Sections

- The sections directive splits up the code into sections where each section is run concurrently on different threads.
- There is an implied boundary at the end of the sections unless the nowait option is stated

```
#pragma omp sections nowait
{
  #pragma omp section
    S1;
  #pragma omp section
    S2;
  ...
  #pragma omp section
    Sm;
}
```

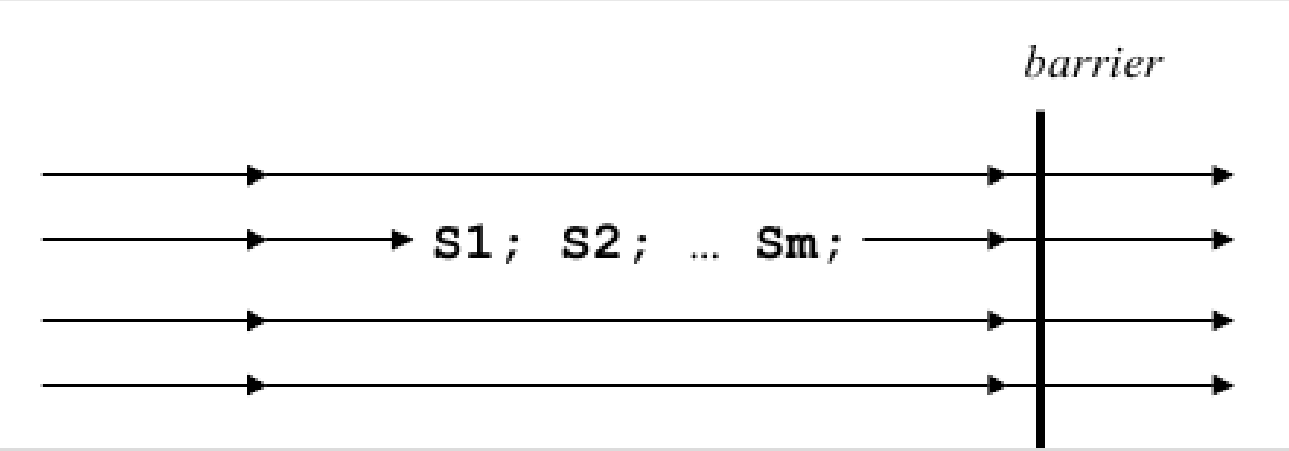


Single Directive

Single

- The single directive causes a single thread to execute the code with a barrier at the end.

```
#pragma omp single
{
  S1;
  S2;
  ...
  Sm;
}
```

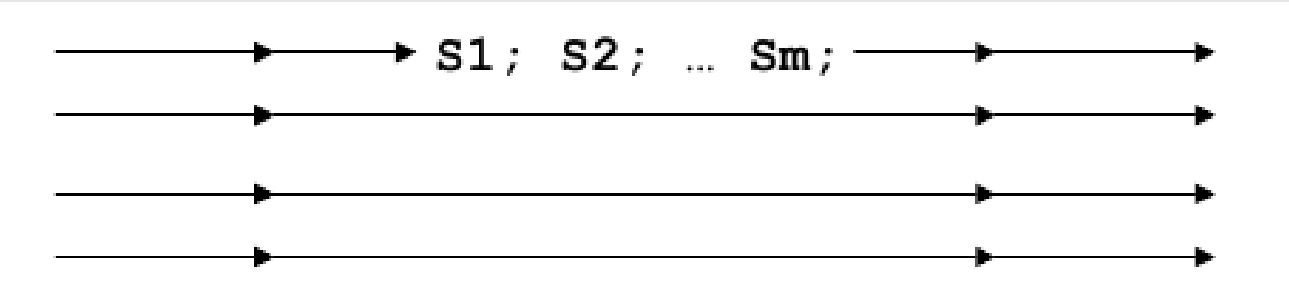


Master Directive

Master

- The master directive causes the master thread to execute the code with no barrier at the end.

```
#pragma omp master
{
  S1;
  S2;
  ...
  Sm;
}
```



Further Reading

Many of the pictures and descriptions are adapted from

- OpenMP webpage
  - http://openmp.org/
- OpenMP tutorial at Lawrence Livermore by Blaise Barney
  - http://www.llnl.gov/computing/tutorials/openMP/
- Robert van Engelen HPC Lectures
  - http://www.cs.fsu.edu/ engelen/courses/HPC/