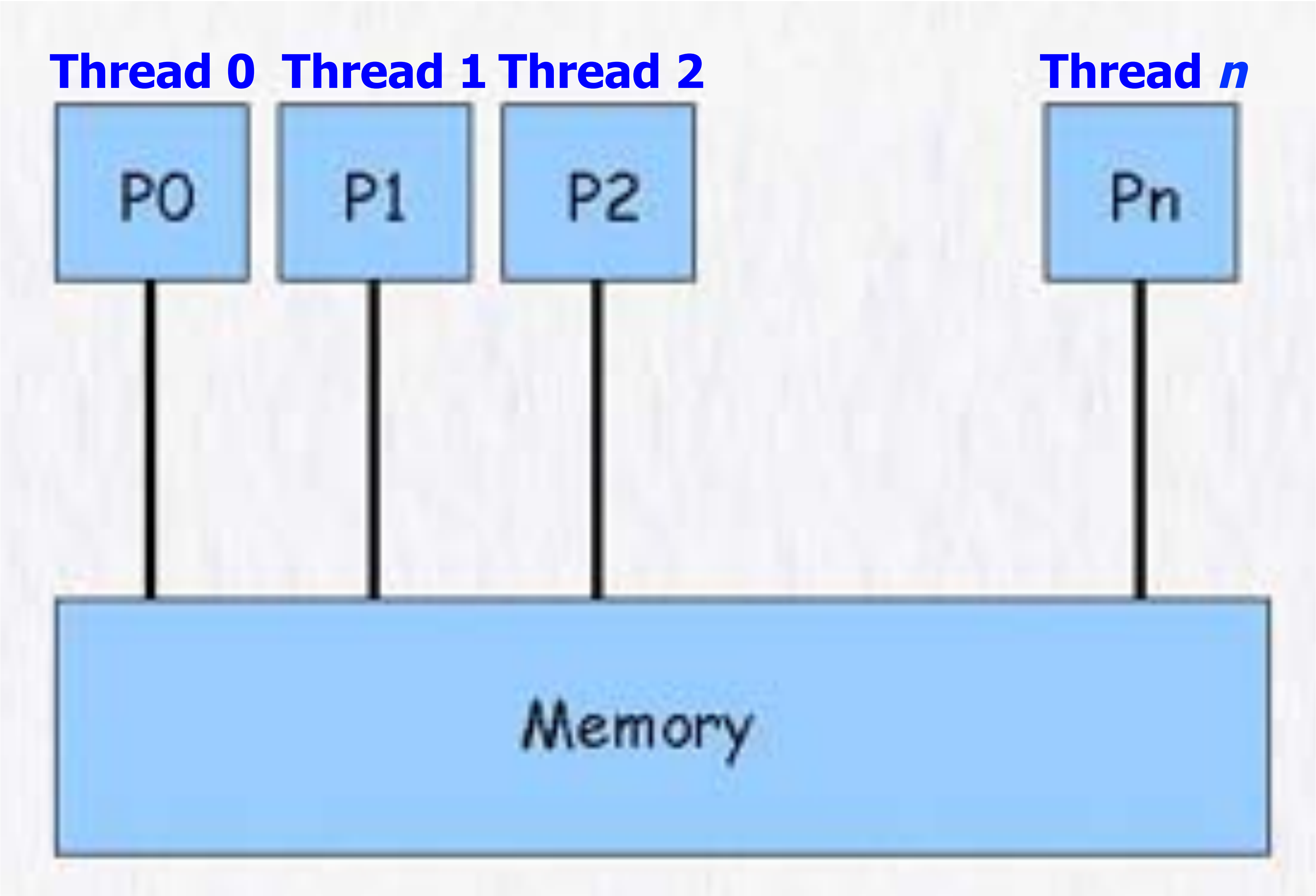
Introduction to OpenMP

Introduction

- OpenMP is designed for shared memory systems.
- OpenMP is easy to use
  - achieve parallelism through compiler directives
  - or the occasional function call
- OpenMP is a “quick and dirty” way of parallelizing a program.
- OpenMP is usually used on existing serial programs to achieve moderate parallelism with relatively little effort

Computational Threads

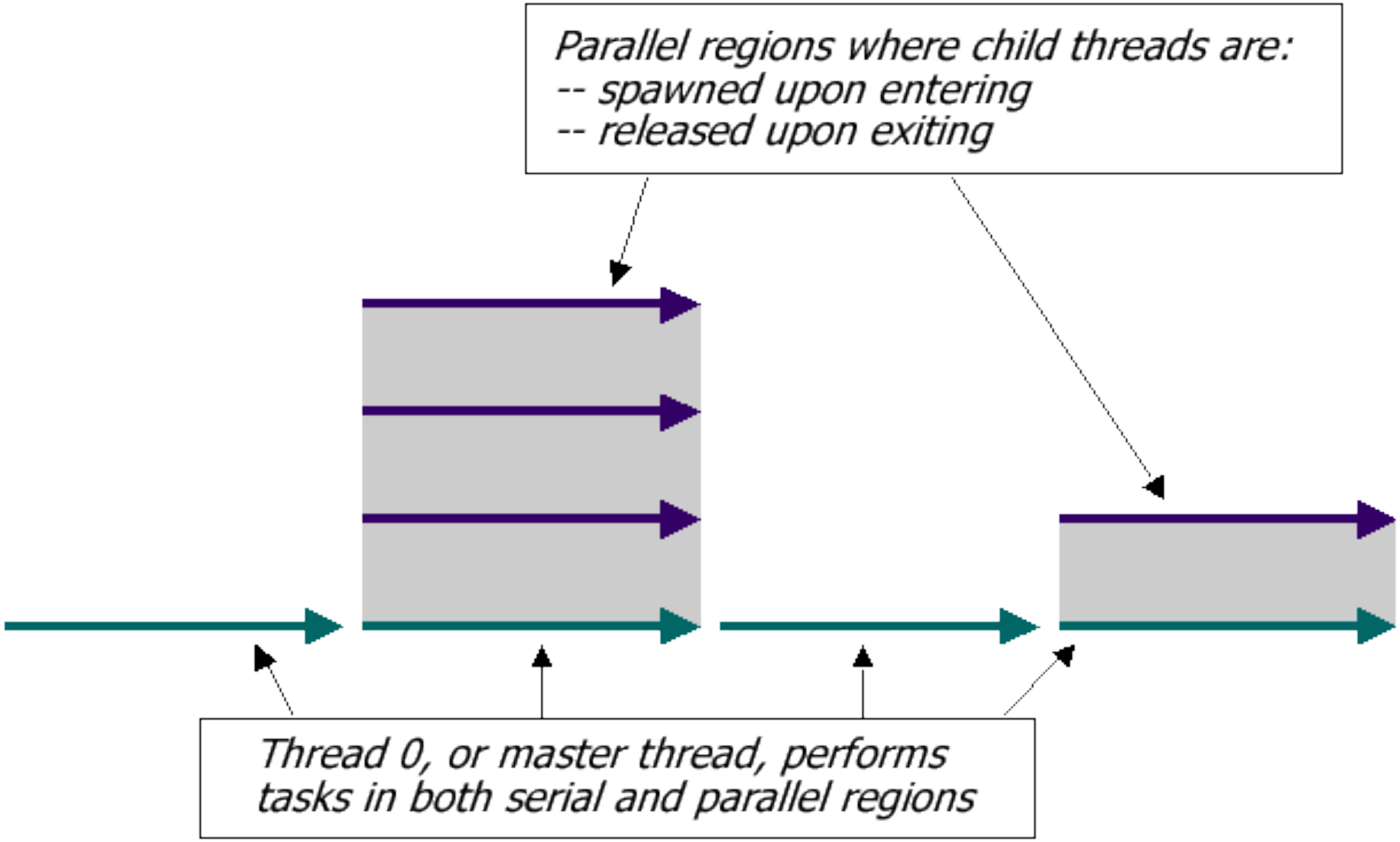
- Each processor has one thread assigned to it
- Each thread runs one copy of your program



OpenMP Execution Model

- In OpenMP, execution begins only on the master thread. Child threads are spawned and released as needed.
  - Threads are spawned when program enters a parallel region.
  - Threads are released when program exits a parallel region

OpenMP Execution Model



Parallel Region Example: For loop

Fortran:

```
!$omp parallel do
do i = 1, n
 a(i) = b(i) + c(i)
enddo
```

C/C++:

```
#pragma omp parallel for
for(i=1; i<=n; i++)
    a[i] = b[i] + c[i];
```

This comment or pragma tells openmp compiler to spawn threads *and* distribute work among those threads

These actions are combined here but they can be specified separately between the threads

Pros of OpenMP

- Because it takes advantage of shared memory, the programmer does not need to worry (that much) about data placement
- Programming model is "serial-like" and thus conceptually simpler than message passing
- Compiler directives are generally simple and easy to use
- Legacy serial code does not need to be rewritten

Cons of OpenMP

- Codes can only be run in shared memory environments!
  - In general, shared memory machines beyond ~8 CPUs are much more expensive than distributed memory ones, so finding a shared memory system to run on may be difficult
- Compiler must support OpenMP
  - whereas MPI can be installed anywhere
  - However, gcc 4.2 now supports OpenMP

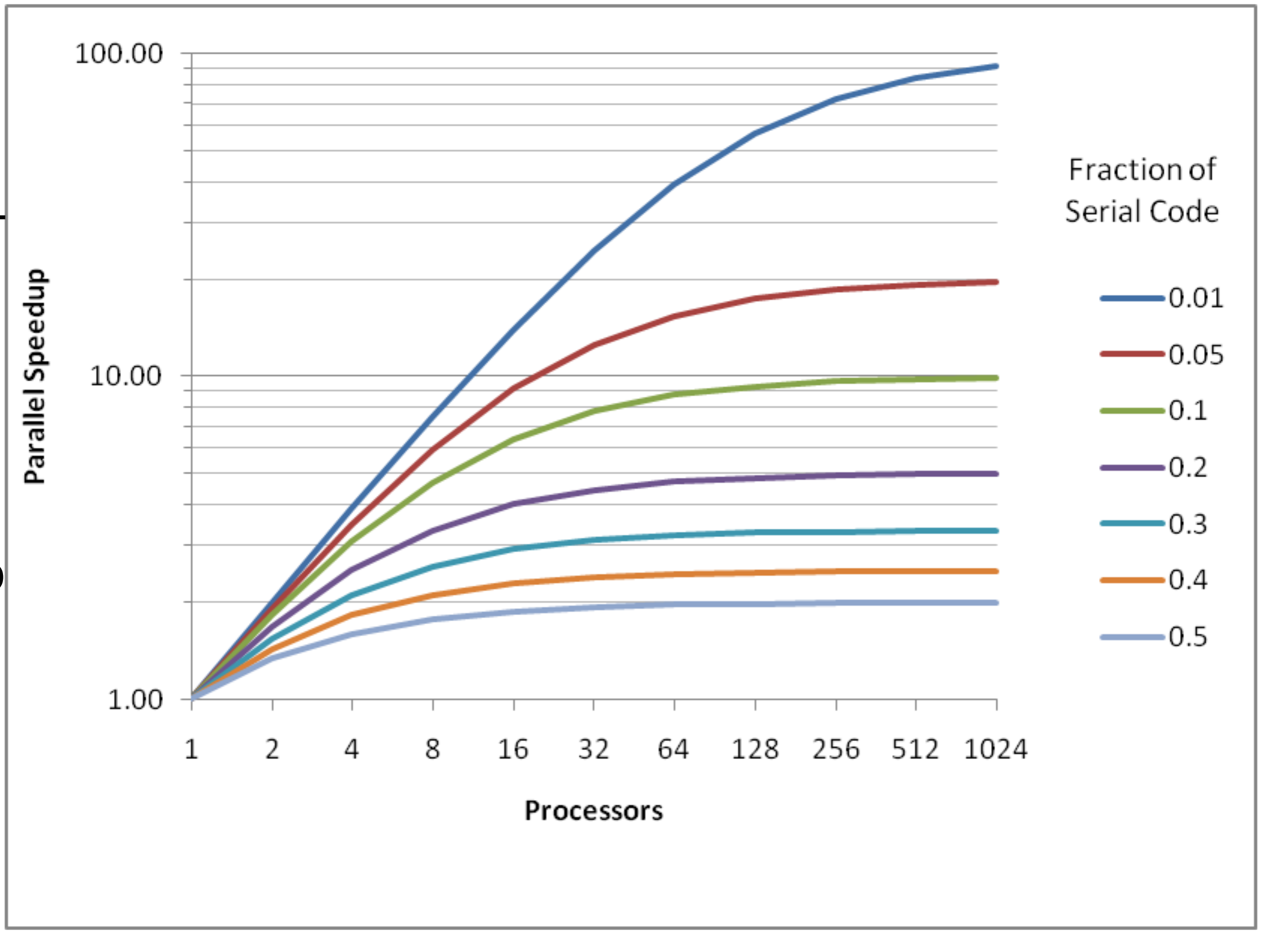
Cons of OpenMP

- In general, only moderate speedups can be achieved.
  - Because OpenMP codes tend to have serial-only portions, Amdahl's Law prohibits substantial speedups

- Amdahl's Law:

  F = Fraction of serial execution time that canno be
  parallelized

  N = Number of processors



Execution time = $$\frac{1}{F + (1 - F)/N}$$

If you have big loops that dominate execution time, these are ideal targets for OpenMP

Compiling and Running OpenMP

- True64: **-mp**
- SGI IRIX: **-mp**
- IBM AIX: **-qsmp=omp**
- Portland Group: **-mp**
- Intel: **-openmp**
- gcc (4.2) **-fopenmp**

Compiling and Running OpenMP

- OMP_NUM_THREADS environment variable sets the number of processors the OpenMP program will have at its disposal.
- Example script

```
#!/bin/tcsh
setenv OMP_NUM_THREADS 4
mycode < my.in > my.out
```

Sections: Functional parallelism

```
#pragma omp parallel
{
    #pragma omp sections
    {
      #pragma omp section
          block1
      #pragma omp section
          block2
    }
}
```

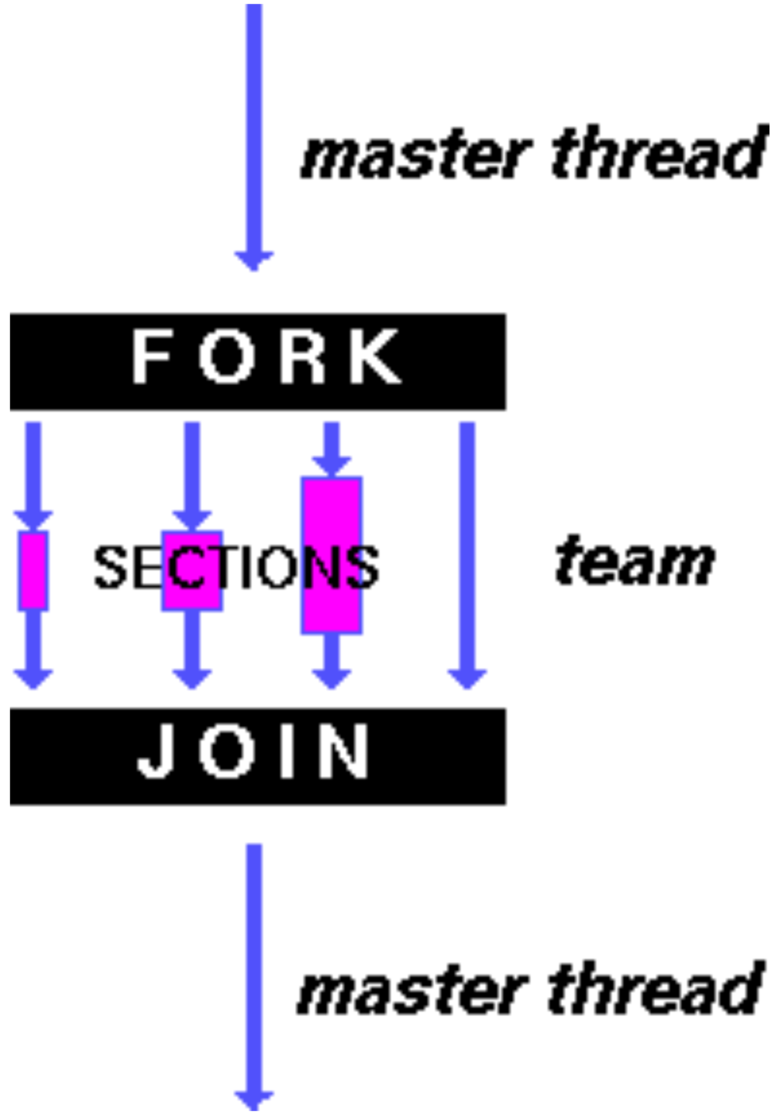


Image from: https://computing.llnl.gov/tutorials/openMP

Parallel DO/for: Loop level parallelism

Fortran:

```
!$omp parallel do
do i = 1, n
 a(i) = b(i) + c(i)
enddo
```

C/C++:

```
#pragma omp parallel for
for(i=1; i<=n; i++)
    a[i] = b[i] + c[i];
```

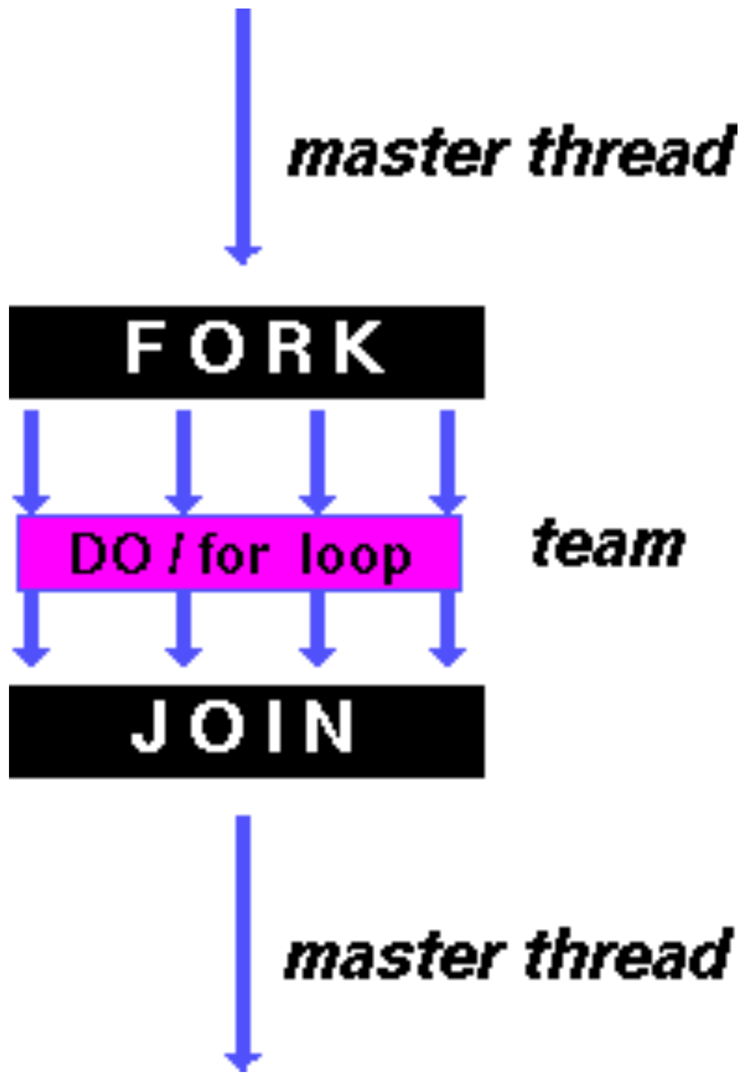


Image from: https://computing.llnl.gov/tutorials/openMP

OpenMP Functions

- **omp_set_num_threads(n)**
  - sets number of openmp threads to n
- **omp_get_num_threads()**
  - returns integer value of total number of threads
- **omp_get_thread_num()**
  - returns individual thread id number.

OpenMP ***single*** clause

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

int main()
{
	int num_thds, myid;

	omp_set_num_threads(4);

	#pragma omp parallel
	{
		num_thds = omp_get_num_threads();

		myid = omp_get_thread_num();

		#pragma omp single
		printf("\nHello World from thd num %d out of %d thds!", myid,
num_thds);
	}
	printf("\nProgram Exit!\n");
}
```

OpenMP ***master*** clause

```
int main()
{
   int num_thds, myid;

   //omp_set_num_threads(4);

   #pragma omp parallel private(num_thds, myid)
   {
      num_thds = omp_get_num_threads();

      myid = omp_get_thread_num();

      #pragma omp master
      {
         printf("\nI am Master: %d out of %d thds!", myid, num_thds);
      }

      printf("\nAll: %d out of %d thds!", myid, num_thds);

   }
   printf("\nProgram Exit!\n");
}
```

OpenMP ***private*** clause

```
int main()
{
   int num_thds, myid;
   int data = 10;

   omp_set_num_threads(4);

   #pragma omp parallel private(num_thds, myid, data)
   {
      num_thds = omp_get_num_threads();

      myid = omp_get_thread_num();

      data = data + myid;

      printf("\nSection 1: From thd num %d out of %d thds : data = %d", myid,
num_thds, data);
   }

   printf("\n\ndata = %d \n", data);
}
```

OpenMP ***firstprivate*** clause

```
int main()
{
   int num_thds, myid;
   int data = 10;

   omp_set_num_threads(4);

   #pragma omp parallel private(num_thds, myid) firstprivate(data)
   {
      num_thds = omp_get_num_threads();

      myid = omp_get_thread_num();

      data = data + myid;

      printf("\nSection 2: From thd num %d out of %d thds : data = %d", myid,
num_thds, data);
   }

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

OpenMP ***threadprivate*** clause

```
omp_set_num_threads(4);

        #pragma omp threadprivate(val)

        #pragma omp parallel
        {
                num_thds = omp_get_num_threads();

                myid = omp_get_thread_num();

                val = 50;

                printf("\nSection 1: from thd num %d out of %d thds : val = %d", myid, num_thds, val);
        }

        printf("\n");

        #pragma omp parallel
        {
                myid = omp_get_thread_num();

                val = val + myid;
        }

        printf("\n");

        #pragma omp parallel
        {
                num_thds = omp_get_num_threads();

                myid = omp_get_thread_num();

                printf("\nSection 2: from thd num %d out of %d thds : val = %d", myid, num_thds, val);
        }
```

OpenMP ***barrier*** clause

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

int main()
{
        int num_thds, myid;

        omp_set_num_threads(4);

        #pragma omp parallel
        {
                num_thds = omp_get_num_threads();

                myid = omp_get_thread_num();

                printf("\nFirst printf: %d out of %d thds!", myid, num_thds);

                #pragma omp barrier

                printf("\nSecond printf: %d out of %d thds!", myid, num_thds);

        }
        printf("\nProgram Exit!\n");
```

OpenMP ***shared, critical*** clause

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

//Gets the total of all the myid's

int main()
{
        int myid, total;

        omp_set_num_threads(4);

        #pragma omp parallel private(myid) shared(total)
        {
                myid = omp_get_thread_num();

                #pragma omp atomic update
                total += myid;
        }
        printf("\n total = %d", total);
        printf("\nProgram Exit!\n");
}
```

OpenMP ***reduction*** clause

Addition

```
 #pragma omp parallel for default(none) shared(A) reduction(+:sum)
        for(i=0;i<ARRSIZE;i++)
        {
                sum = sum + A[i];
        }
```

Multiplication

```
  #pragma omp parallel for default(none) shared(A) reduction(*:val)
        for(i=0;i<ARRSIZE;i++)
        {
                val = val * A[i];
        }
```

OpenMP ***reduction*** clause

Min

```
#pragma omp parallel for default(none) shared(A) reduction(min:val)
        for(i=0;i<ARRSIZE;i++)
        {
                if(val > A[i])
                {
                        val = A[i];
                }
        }
```

Max

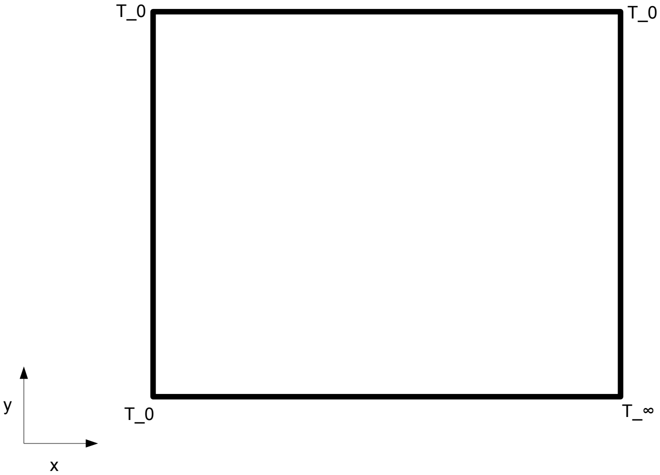
```
 #pragma omp parallel for default(none) shared(A) reduction(max:val)
        for(i=0;i<ARRSIZE;i++)
        {
                if(val > A[i])
                {
                        val = A[i];
                }
        }
```

OpenMP ***section*** clause

```
omp_set_num_threads(2);
#pragma omp parallel sections
{
        #pragma omp section
        {
                num_thds = omp_get_num_threads();
                myid = omp_get_thread_num();
                printf("\nSection 1: thd num %d out of %d thds!", myid, num_thds);
        }
        #pragma omp section
        {
                num_thds = omp_get_num_threads();
                myid = omp_get_thread_num();
                printf("\nSection 2: thd num %d out of %d thds!", myid, num_thds);
        }

        #pragma omp section
        {
                num_thds = omp_get_num_threads();
                myid = omp_get_thread_num();
                printf("\nSection 3: thd num %d out of %d thds!", myid, num_thds);
        }
}
printf("\nProgram Exit!\n");
```

Steady State Heat Conduction

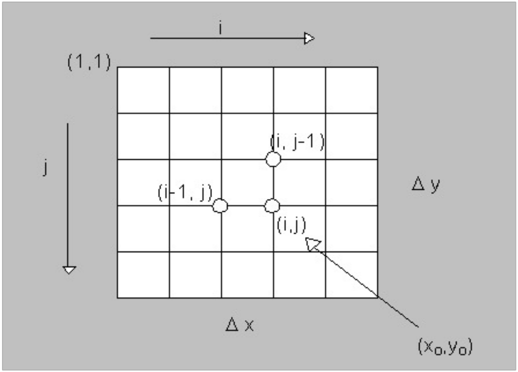


Steady State Heat Conduction

Phenomenon is modelled using Laplace's equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \nabla^2 T = 0$$

which can be discretised on a grid as,



Steady State Heat Conduction

The discretised equation at a single point (i,j) is

$$\frac{T_{i-1,j} - T_{i,j} + T_{i+1,j}}{(\Delta x)^2} + \frac{T_{i,j-1} - T_{i,j} + T_{i,j+1}}{(\Delta y)^2} = 0$$

Assemble all the equations for all unknown points in the Matrix form and then solve

$$Ax = B$$

You can choose any Linear Algebra Solver (iterative or Direct). Iterative is more efficient.