PARALLEL PROGRAMMING...

Parallel Programming: Overview

SESSION 3/6

GOAL

Programming Interface for parallel computing

OpenMP (Open Multi-Processing)

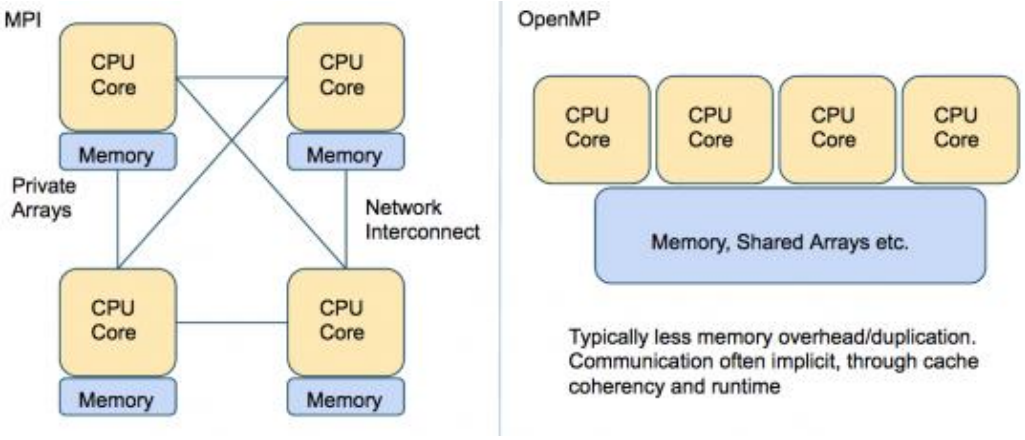
Programming interface for parallel computing

병렬 컴퓨팅을 위한 프로그래밍 인터페이스

Programming interface…

Remember

.



.

MPI (Message Passing Interface) is a multi-process model whose mode of communication between the processes is **explicit.**

==> communication management is the responsibility of the user.

.

OpenMP (Open Multi-Processing) is a multitasking model whose mode of communication between tasks is **implicit**

==> communications is the responsibility of the compiler.

OpenMP (Open Multi-Processing)

OpenMP (Open Multi-Processing)

OpenMP is a programming interface for parallel computing on shared memory architecture.

The OpenMP API consists of

- compiler directives (for insertion ***into sequential*** Fortran/C/C++$code)
- a few library routines
- some environment variables

Advantages:

- User-friendly
- Incremental parallelization of a serial code
- Possible to have a single source code for both serial and parallelized versions

Disadvantages:

- Relatively limited user control
- Most suitable for parallelizing loops (data parallelism)
- Performance?

OpenMP (Open Multi-Processing)

OpenMP is a programming interface for parallel computing on shared memory architecture.

- **It allows you to manage:**

  - the creation of light processes
  - the sharing of work between these lightweight processes
  - synchronizations (explicit or implicit) between all light processes
  - the status of the variables (private or shared).

OpenMP (Open Multi-Processing)

- **Shared memory model**
  - Threads communicate by accessing shared variables

    - **The sharing is defined syntactically**
      - Any variable that is seen by two or more threads is shared
      - Any variable that is seen by one thread only is private

        - **Race conditions possible**
          - Use synchronization to protect from conflicts
          - Change how data is stored to minimize the synchronization

OpenMP (Open Multi-Processing)

- **Multicore CPUs are everywhere:**
  - Servers with over 100 cores today and more
  - Even smartphone CPUs have 8 cores

- **Multithreading, natural programming model**
  - All processors share the same memory
  - Threads in a process see same address space
  - Many shared-memory algorithms developed

- **Multithreading is hard**
  - Lots of expertise necessary
  - Deadlocks and race conditions
  - Non-deterministic behavior makes it hard to debug

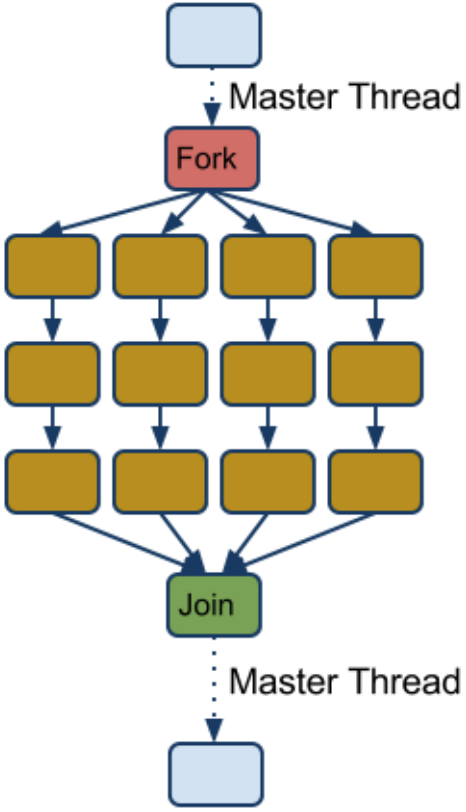
OpenMP: Process and thread: whatis the difference ?

- You need an existing process to create a thread.
- Each process has at least one thread of execution.
- A process has its own virtual memory space that cannot be accessed by other processes running on the same or on a different processor.
- All threads created by a process share the virtual address space of that process. They read and write to the same address space in memory. They also share the same process and user ids, file descriptors, and signal handlers. However, they have their own program counter value and stack pointer, and can run independently on several processors.

OpenMP: Terminology and behavior

- **OpenMP Team** = Master + Worker

- **Parallel Region** is a block of code executed by all threads simultaneously (has implicit barrier)
  - The master thread always has thread id 0
  - Parallel regions can be nested
  - If clause can be used to guard the parallel region



OpenMP: General Code Structure

```
#include <omp.h>

main ()
{
      int var1, var2, var3;
      //serial code

      //start of a parallel region
      #pragma omp parallel private(var1, var2) shared(var3)
      {...}

      //more serial code
      {...}

      //another parallel region
      #pragma omp parallel
      {…}
      }
```

OpenMP: Constructs

Parallel region

Thread creates team, and becomes master (id 0)
All threads run code after
Barrier at end of parallel section

```
#pragma omp parallel [clause ...]
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     num_threads (integer)

   structured_block                                  (not a complete list)
```

OpenMP: Parallel Clauses

```
#pragma omp parallel if (scalar_expression)
```

Only execute in parallel.
Otherwise serial.

```
#pragma omp parallel private (list)
```

Data local to thread.
Value are not guaranted to be defined on exit (even if defined before)
No storage associated with original object
Use firstprivate and/or lastprivate clause to override

```
#pragma omp parallel firstprivate (list)
```

Variables in list are private.
Initialized with the value the variable had before entering the construct.

```
#pragma omp parallel for lastprivate (list)
```

Only in for loops
Variables in list are private.
The thread that executes the sequentially last iteration updates the value of the variables in the list.

OpenMP: Parallel Clauses

```
#pragma omp shared (list)
```

Data is accessible by all threads in team.
All threads access same address space.

Improperly scoped variables are big source of OMP bugs
- Shared when should be private
- Race condition

```
#pragma omp default (shared | none)
```

Tip: Safest is to use default (none) and declare by hand.

OpenMP Barrier

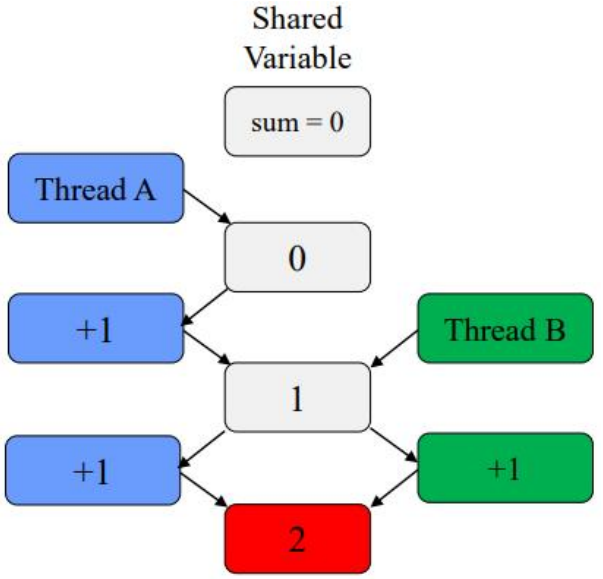
- When a thread reaches a barrier it only continues after all the threads in the same thread team have reached it
- Each barrier must be encountered by all threads in a team, or none at all
- The sequence of work-sharing regions and barrier regions encountered must be same for all threads in team
- Implicit barrier at the end of: do, parallel, single, workshare

OpenMP: Caution Race Condition

When multiple threads simultaneously read/write
Multiple OMP solutions

- Reduction
- Atomic
- Critical

```
#pragma omp parallel for private(i) shared(sum)
  for (i=0; i<N; i++) {
    sum += i;
  }
```



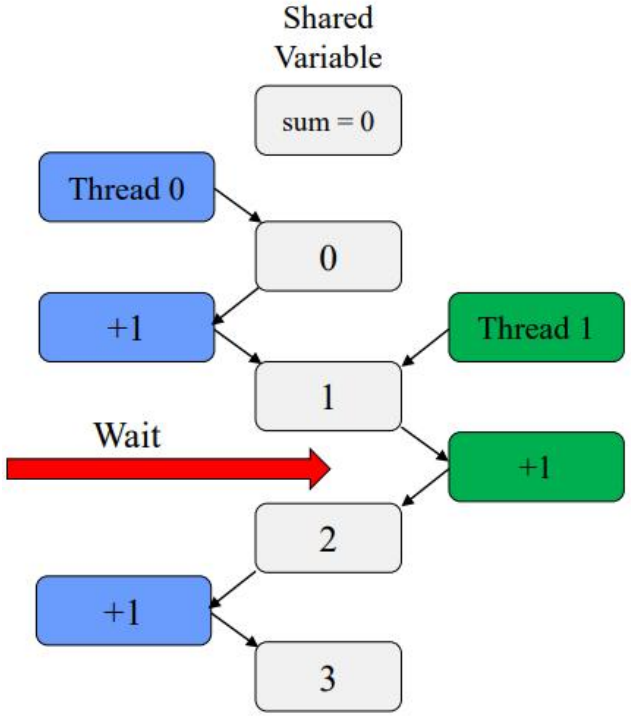
Should be 3!

OpenMP: Critical Section

One solution: use critical
Only one tread at a time can execute a critical section

```
#pragma omp critical
  {
    sum += i;
  }
```

Downside ?
SLOOOOWWW
Overhead and serialization

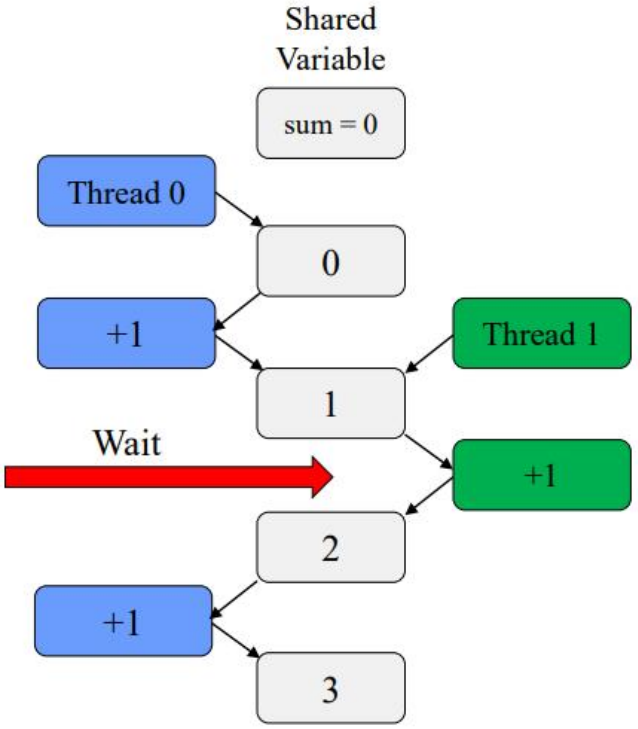


OpenMP: Atomic

Atomics like "mini" critical
Only one line
Certain limitations

```
#pragma omp atomic
    sum += i;
```

Hardware controlled
Less overhead the critical



OpenMP: Reduction

```
#pragma omp reduction (operator:variable)
```

Avoids race condition
Reduce variable must be shared
Makes variable private, then performs operator at end of loop
operator cannot be overloaded (c++)
One of: +,*,-,/ (and &,^,|,&&,||)
OpenMP 3.1: added min and max for c/c++

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

int main() {

  int i;
  const int N = 1000;
  int sum = 0;

#pragma omp parallel for private(i) reduction(+: sum)
  for (i=0; i<N; i++) {
    sum += i;
  }

  printf("reduction sum=%d (expected %d)\n", sum, ((N-1)*N)/2);
```

OpenMP: Scheduling omp for

How does a loop get split up ?

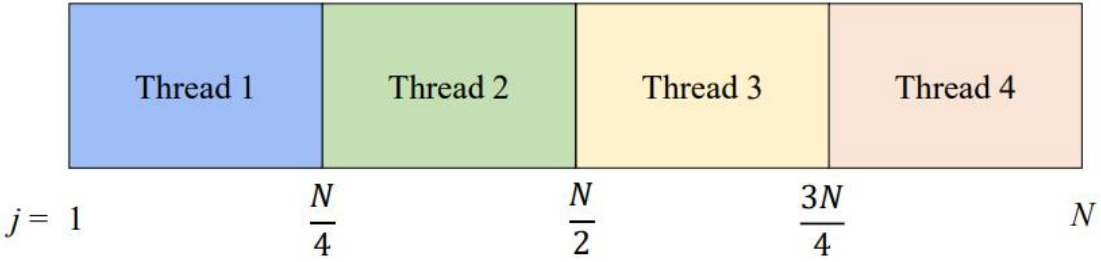
In MPI, we have to do it manually

If you do not tell what to do, the compiler decides

Usually compiler chooses "static" - chunks of N/p

```
#pragma omp parallel for default(shared) private(j)
  for (j=0; j<N; j++) {
        ... // some work here
}
```

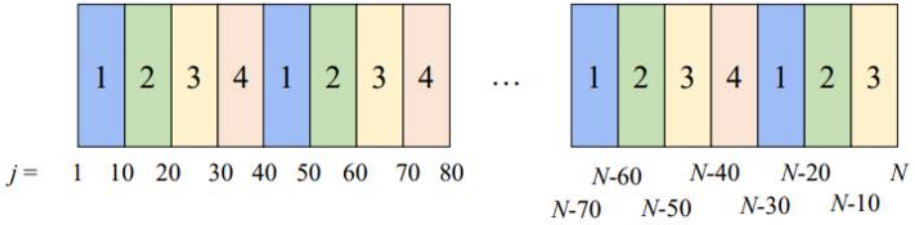
Unspecified schedule



OpenMP: Static Scheduling

You can tell the compiler what size chunks to take.

```
#pragma omp parallel for default(shared) private(j) schedule(static,10)
  for (j=0; j<N; j++) {
        ... // some work here
}
```

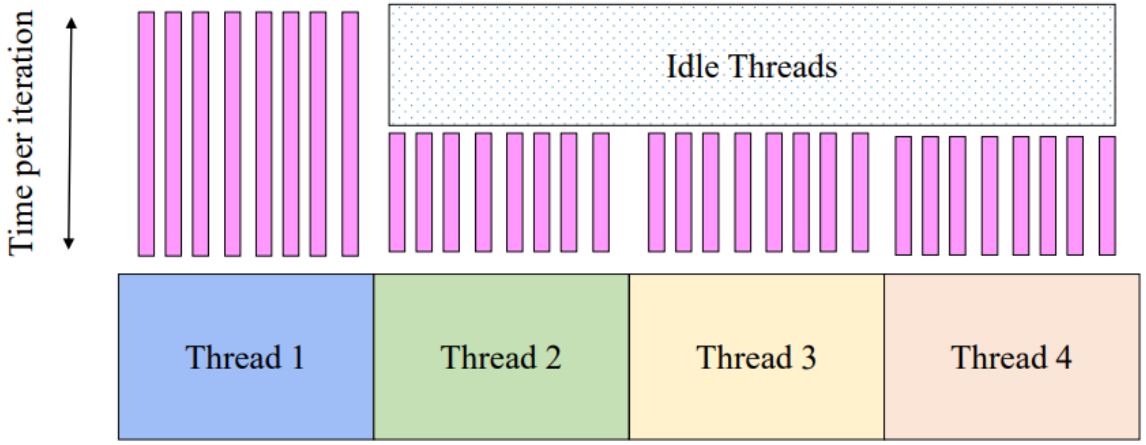


Keeps assigning chunks until done.
Chunk size that is not a multiple of the loop will results in thread with uneven numbers.

OpenMP: Problem with Static Scheduling

What happens if loop iterations do not take the same amount of time ?

Load imbalance



OpenMP: Dynamic Scheduling

Chunks are assigned on the fly, as threads become available
When a thread finishes on chunk, it is assigned another

```
#pragma omp parallel for default(shared) private(j) schedule(dynamic,10)
  for (j=0; j<N; j++) {
        ... // some work here
}
```

Caveat: higher overhead than static!

OpenMP: For Scheduling Recap

```
#pragma omp parallel for schedule(type [,size])
```

Scheduling types

Static

- Chunks of specified size assigned round-robin

Dynamic

- Chunks of specified size are assigned when thread finishes previous chunk

Guided

- Like dynamic, but chunks are exponentially decreasing
- Chunk will not be smaller than specified size

Runtime

- Type and chunk determined at runtime via environment variables

OpenMP: API

- API for library calls that perform useful functions
- Must include "omp:h"
- Will not compile without OpenMP compiler support

```
#include <omp.h>  //<-- necessary header file for OpenMP API
#include <stdio.h>

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

  printf("OpenMP running with %d threads\n", omp_get_max_threads());

#pragma omp parallel
  {
    //Code here will be executed by all threads
    printf("Hello World from thread %d\n", omp_get_thread_num());
  }

  return 0;
}
```

OpenMP: API

```
void omp_set_num_threads(int num_threads)
```

Sets number of treads used in next parallel section
Overrides OMP_NUM_THREADS environment variable
Positive integer

```
int omp_get_max_threads()
```

Returns max possible (generally set by OMP_NUM_THREADS)

```
int omp_get_num_threads()
```

Returns number of threads currently in team

```
int omp_get_thread_num()
```

Returns thread id of calling thread
Between 0 and omp_get_num_threads-1

```
double omp_get_wtime()
```

Returns number of seconds since some point
Use inpais time=(t2-t1)

OpenMP: Example Fibonacci

```
int fib ( int n )
{
  int x,y;
  if ( n < 2 ) return n;
  #pragma omp task shared(x,n)
  x = fib(n-1);

  #pragma omp task shared(y,n)
  y = fib(n-2);

#pragma omp taskwait
return x+y;
}
```

```
int main()
{
#pragma omp parallel
  #pragma omp single nowait
  result = comp_fib_numbers(10);
return EXIT_SUCCESS;
}
```

OpenMP: Example Quicksort

```
void quick_sort (int p, int r, float *data)
{
  if (p < r) {
    int q = partition (p, r, data);
    #pragma omp task
    quick_sort (p, q-1, data, low_limit);
    #pragma omp task
    quick_sort (q+1, r, data, low_limit);}
  }
}
```

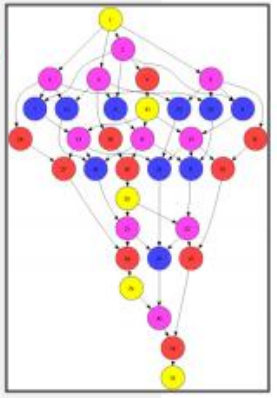
```
void par_quick_sort (int n, float *data)
{
  #pragma omp parallel
  {
    #pragma omp single nowait
    quick_sort (0, n, data);
  }
```

OpenMP: Example Cholesky Factorization

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
  printf("Hello world! -main\n");
#pragma omp parallel
  {
    printf(".. worker reporting for duty.\n");
  }
  printf("Over and out! -main\n");
}
```

```
> gcc -fopenmp omp_hello.c -o omp
> OMP_NUM_THREADS=3 ./omp
Hello world! -main
.. worker reporting for duty.
.. worker reporting for duty.
.. worker reporting for duty.
Over and out! -main
```

```c
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    #pragma omp task depend(inout: a[k][k])
    potrf(a[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      #pragma omp task depend(in: a[k][k])
                       depend(inout: a[k][i])
      trsm(a[k][k], a[k][i], ts, ts);
    }

    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        #pragma omp task depend(inout: a[j][i])
                         depend(in: a[k][i], a[k][j])
        dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
      }
      #pragma omp task depend(inout: a[i][i])
                       depend(in: a[k][i])
      syrk(a[k][i], a[i][i], ts, ts);
    }
  }
}
```

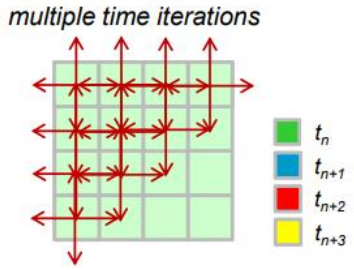


OpenMP 4.0

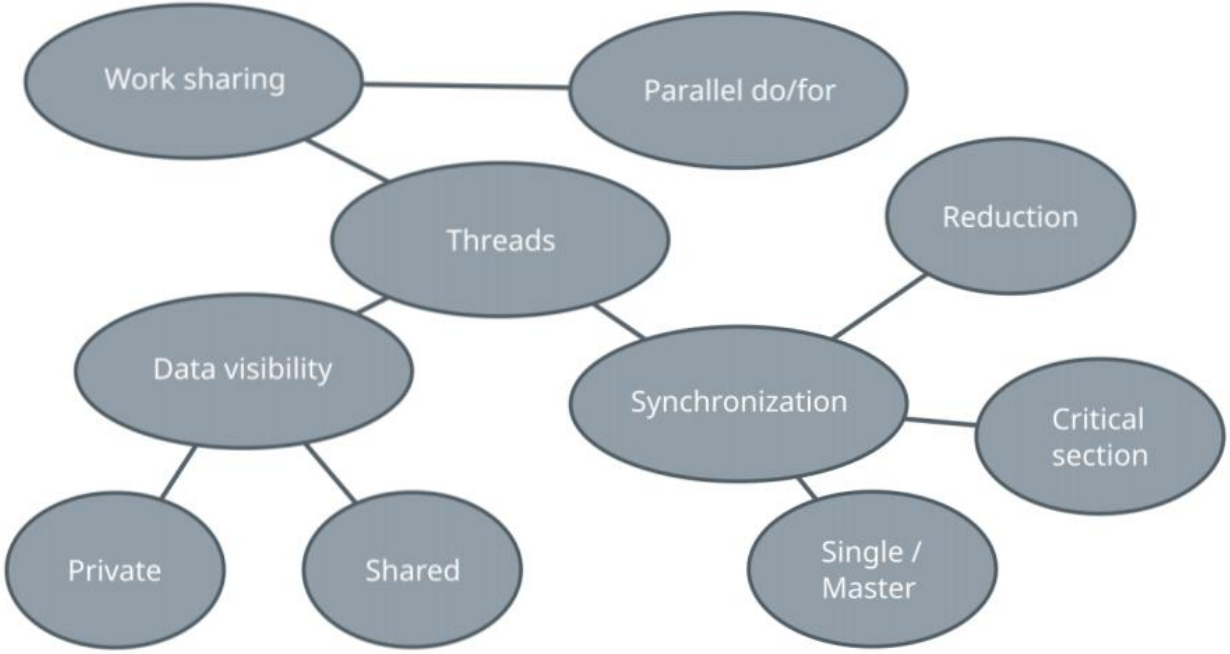
OpenMP: Example Gauss-Seidel

```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;

  #pragma omp parallel
  #pragma omp single
  for (int t = 0; t < tsteps; ++t)
    for (int ii=1; ii < size-1; ii+=TS)
      for (int jj=1; jj < size-1; jj+=TS) {
        #pragma omp task depend(inout: p[ii:TS][jj:TS])
            depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                       p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
        {
          for (int i=ii; i<(1+ii)*TS; ++i)
            for (int j=jj; j<(1+jj)*TS; ++j)
              p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                                p[i-1][j] * p[i+1][j]);
        }
      }
}
```

Gauss-Seidel Method is used to solve the linear system Equations. It is a method of iteration for solving n linear equation Ax=b with the unknown variables.



OpenMP: Summary



OpenMP: Summary Overview

OpenMP language extensions

parallel control structures	work sharing	data environment	synchronization	runtime functions, env. variables

governs flow of control in the program
 	
`parallel` directive
	distributes work among threads
	
	`do/parallel do` and `section` directives
		scopes variables
		
		`shared` and `private` clauses
			coordinates thread execution
			
			`critical` and `atomic` directives
			`barrier` directive
				runtime environment
				
				`omp_set_num_threads()`
				`omp_get_thread_num()`
				`OMP_NUM_THREADS`
				`OMP_SCHEDULE`

OpenMP: Performance Tips

- Avoid serialization!
- Avoid using `#pragma omp parallel for` before each loop
  - Can have significant overhead
    - Thread creation and scheduling is NOT free!!
  - Try for broader parallelism
    - One `#pragma omp parallel`, multiple `#pragma omp for`
  - Always try to parallelize the outer most loop
- Use reduction whenever possible
- Minimize I/O
- Minimize `critical`
  - Use `atomic` instead of `critical` where possible

Thank you for your attention !