PARALLEL PROGRAMMING...

By Patrick Lemoine 2023.

Parallel Programming: Overview

SESSION 3/6

GOAL

Programming Interface for parallel computing

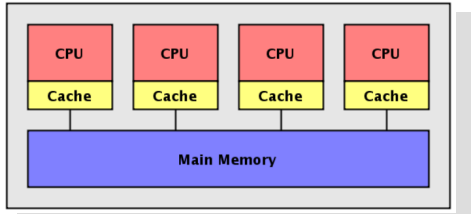
OpenMP (Open Multi-Processing)

병렬 컴퓨팅을 위한 프로그래밍
인터페이스

OpenMP (Open Multi-Processing)

OpenMP (Open Multi-Processing)

Open Specifications for Multi Processing (OpenMP) is a programming interface for parallel computing on **shared memory architecture**.

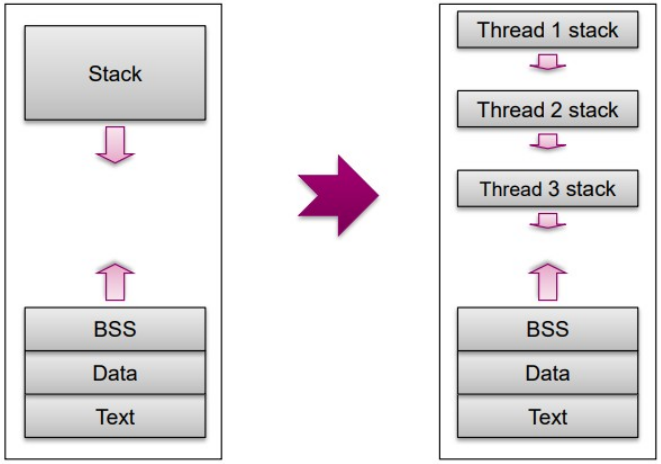


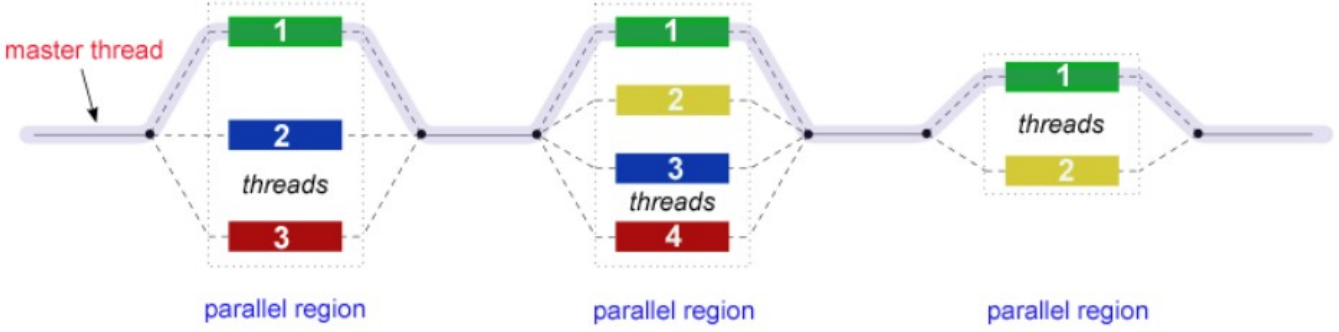
- **It allows you to manage:**
  - the creation of light processes.
  - the sharing of work between these lightweight processes.
  - synchronizations (explicit or implicit) between all light processes.
  - the status of the variables (private or shared).

OpenMP (Open Multi-Processing)

OpenMP is based on **Fork/Join model**

1. When program starts, one Master thread is created
2. Master thread executes sequential portions of the program
3. At the beginning of parallel region, master thread forks new threads
4. All the threads together now forms a “team”
5. At the end of the parallel region, **the forked threads die !**





OpenMP (Open Multi-Processing)

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? ~

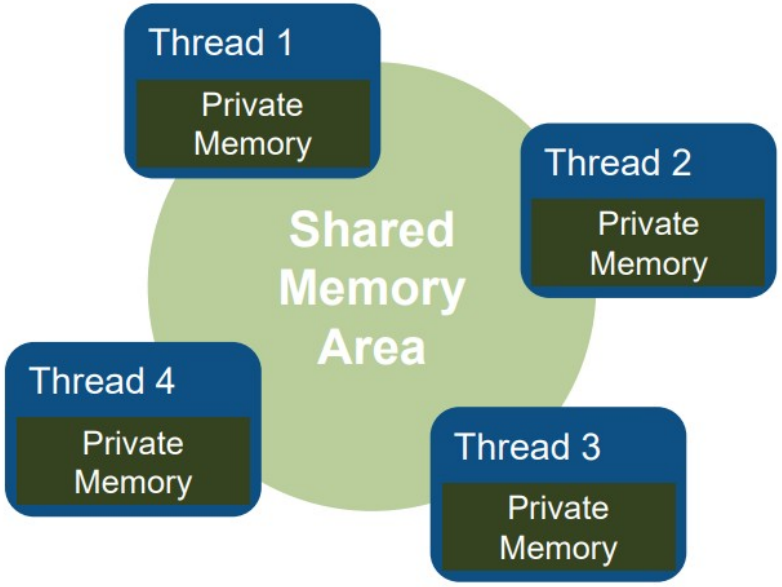
OpenMP (Open Multi-Processing)

What is a **Shared-Memory Program**?

- One process that spawns multiple threads

- Threads can communicate via shared memory
  - ➢ Read/Write to shared variables
  - ➢ Synchronization can be required!

- OS decides how to schedule threads



OpenMP: Shared Memory

- **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: Multithreading

- **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

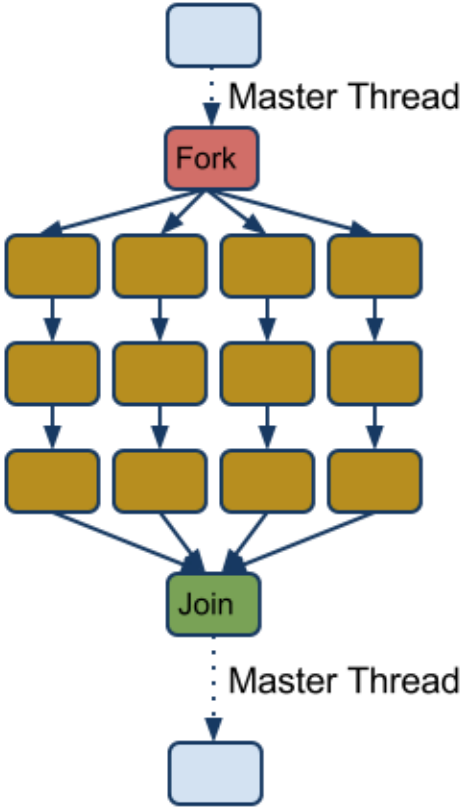
What is the difference ?

- 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 share the same process and user ids, file descriptors, and signal handlers.
  - 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**

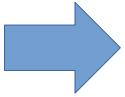
- **Parallel Region** is a block of code executed by all threads simultaneously (*has implicit barrier*)
  - The master thread always has thread id **0 !**
  - Parallel regions can be nested.
  - If **clause** can be used **to guard the parallel region.**



OpenMP: Example Code Structure

Make “Hello World” multi-threaded..

```
int main(){
    int ID=0;
    printf("hello(%d) ", ID);;
    printf("world(%d)\n", ID);;
}
```

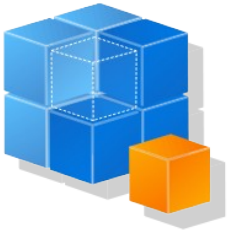


Include OpenMP header

```
#include "omp.h"

int main(){
 #pragma omp parallel
 {
    int ID = omp_get_thread_num();
    printf("hello(%d) ", ID);;
    printf("world(%d)\n", ID);;
 }
}
```

Start parallel region with “default” number of threads

Who am I?



OpenMP: Parallel Region

A **parallel region** identifies a portion of code that can be executed by different threads

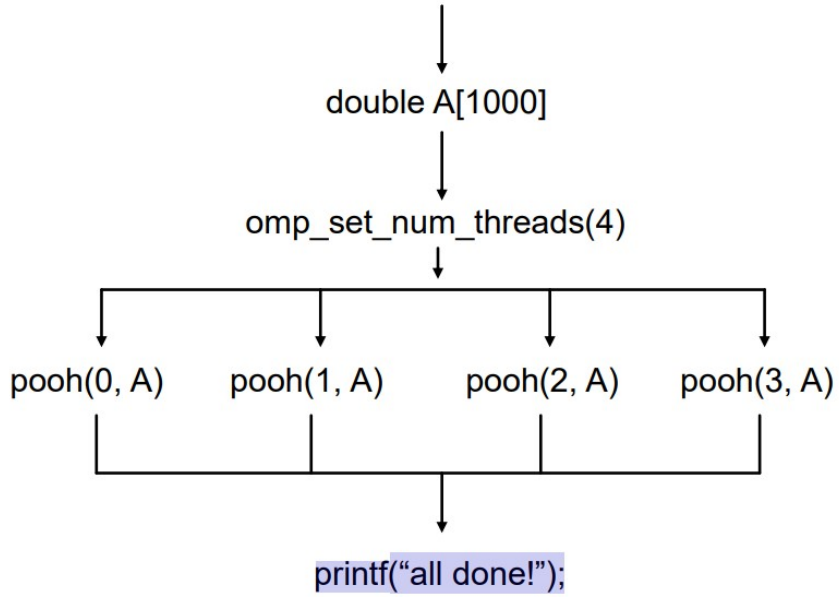
- You can create a parallel region with the "parallel" directive
- You can request a specific number of threads with **omp_set_num_threads**(N)

```
double A[1000];
omp_set_num_threads(4);
 #pragma omp parallel
 {
    int ID = omp_get_thread_num();
    pooh(ID,A);
 }
printf("all done!");
```

```
double A[1000];

 #pragma omp parallel num_threads(4)
 {
    int ID = omp_get_thread_num();
    pooh(ID,A);
 }
printf("all done!");
```

Each thread will call *pooh(ID,A)* function with a different value of ID

OpenMP: Parallel Region




```
double A[1000];
omp_set_num_threads(4);
 #pragma omp parallel
 {
    int ID = omp_get_thread_num();
    pooh(ID,A);
 }
printf("all done!");
```

- All the threads execute the same code
- The **[A]** array is shared
- Implicit synchronization at the end of the parallel region

OpenMP: Behind the scenes...

- The **OpenMP compile**r generates code logically analogous to that on the right.

- All known OpenMP implementations **use a thread pool** so full cost **of threads creation**
- and destruction is not incurred for each parallel region.

- **Only three threads** are created because **the last parallel section** will be invoked from the parent thread.

```
#pragma omp parallel num_threads(4)
{
   foobar();
}
```

```
void thunk(){
    foobar();
}

pthread_t tid[4];

for (int i= 1; i< 4; ++i)
    pthread_create(&tid[i],0,thunk, 0);

thunk();

for (int i = 1; i< 4; ++i)
    pthread_join(tid[i]);
```

OpenMP: Constructs Parallel Region

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)

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

OpenMP: Data Sharing Attributes

```
int x=1;
#pragma omp parallel shared(x) num_threads(2)
{
   x++;
   printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 2 or 3 (three printfs in total)

Shared

The variable inside the construct is the same as the one outside the construct.

- In a parallel construct this means all threads see the same variable but not necessarily the same value.
- Usually need some kind of synchronization to update them correctly.

```
int x=1;
#pragma omp parallel private(x) num_threads(2)
{
   x++;
   printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 1

Private

The variable inside the construct is a **new** variable of the same type with an **undefined** value

- In a parallel construct this means all threads have a different variable
- Can be accessed without any kind of synchronization

OpenMP: Data Sharing Attributes

```
int x=1;
#pragma omp parallel firstprivate(x) num_threads(2)
{
   x++;
   printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 1

Firstprivate

The variable inside the construct is a **new** variable of the same type but it is initialized to the original value of the variable

- In a parallel construct this means all threads have a different variable with the same initial value
- Can be accessed without any kind of synchronization

And default. What is the default?

- If there is a **default clause**, what the clause says
  - **none** means that the compiler will issue an error if the attribute is not explicitly set by the programmer.
- Otherwise, depends on the construct
  - For the parallel region the default is shared.

OpenMP: Synchronization

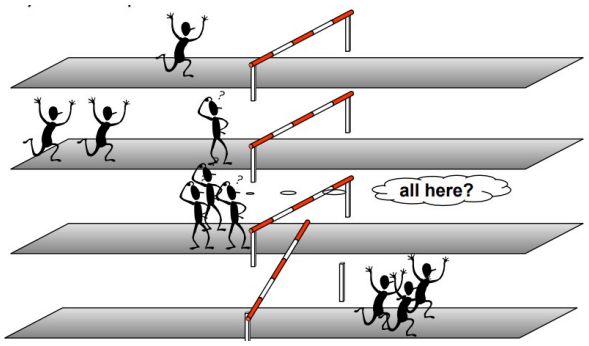
Directives to synchronize thread team or control thread access to code fragments

$OMP MASTER	Execute section only with master thread (no implied barrier).
$OMP CRITICAL	Restrict access to one thread at a time (otherwise block).
$OMP BARRIER	Synchronize all threads.
$OMP ATOMIC	Special case of CRITICAL, the statement following allows a specific memory location to be updated atomically (no multiple writes, can take advantage of specific hardware instructions for atomic writes).
$OMP FLUSH [(list)]	Ensure threads have consistent view of shared variables (else just the named list).
$OMP ORDERED	Execute code in same order as under sequential execution.
$OMP SINGLE	Block executed by only one thread (implied BARRIER and FLUSH at the end)

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*



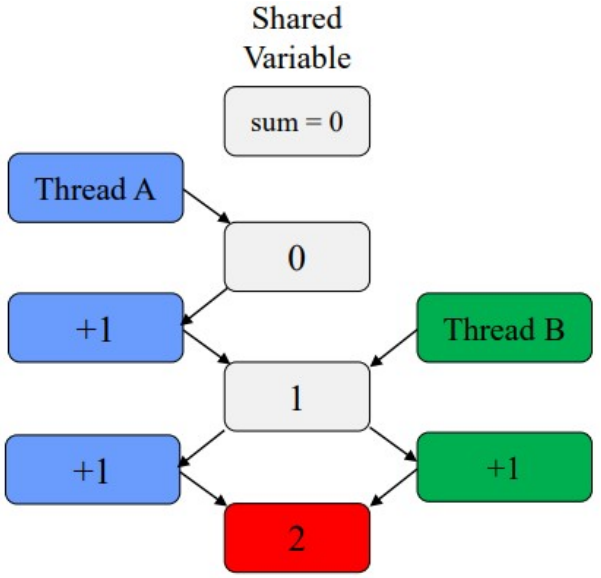
OpenMP: Caution Race Condition

When multiple threads simultaneously read/write

Multiple OMP solutions :

- **R**eduction
- **A**tomic
- **C**ritical

```
#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**

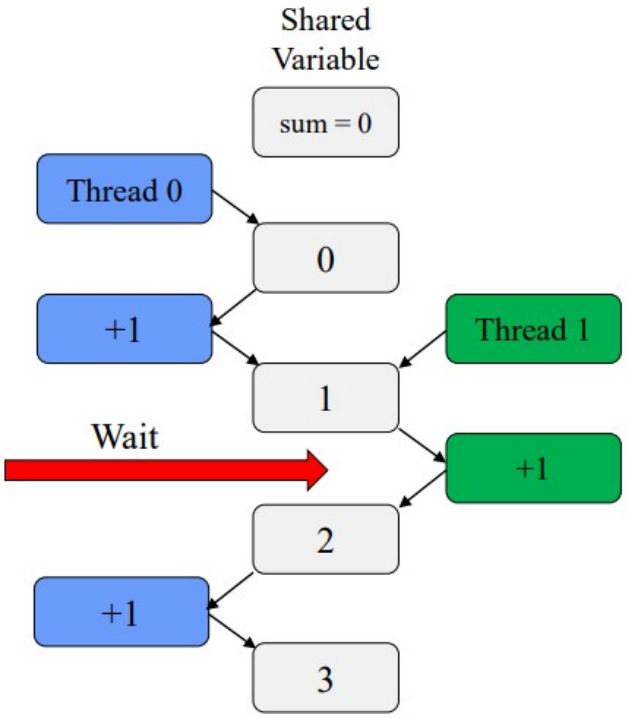
Only one tread at a time can execute a critical section

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

Downside ?

YES SLOOOOWWW

Overhead and serialization



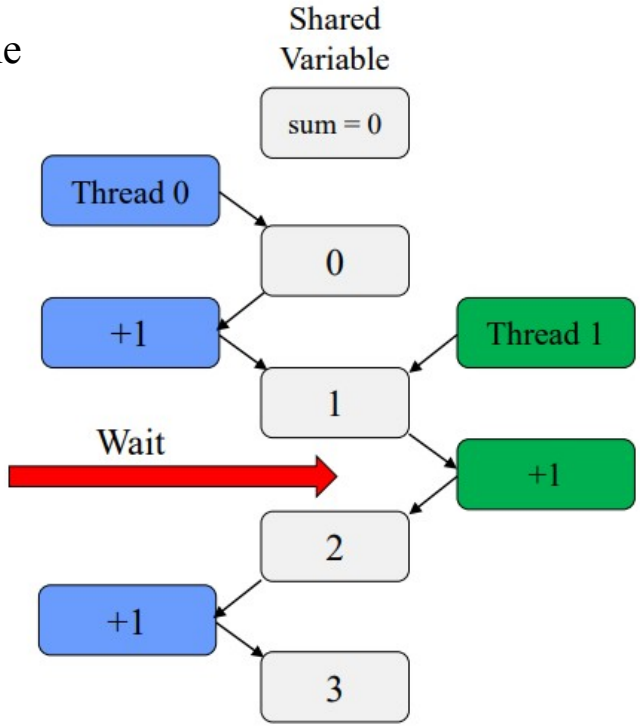
OpenMP: Atomic

Atomic provides mutual exclusion but only applies to the update of a memory location.

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++

```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

```
#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: Scheduling

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

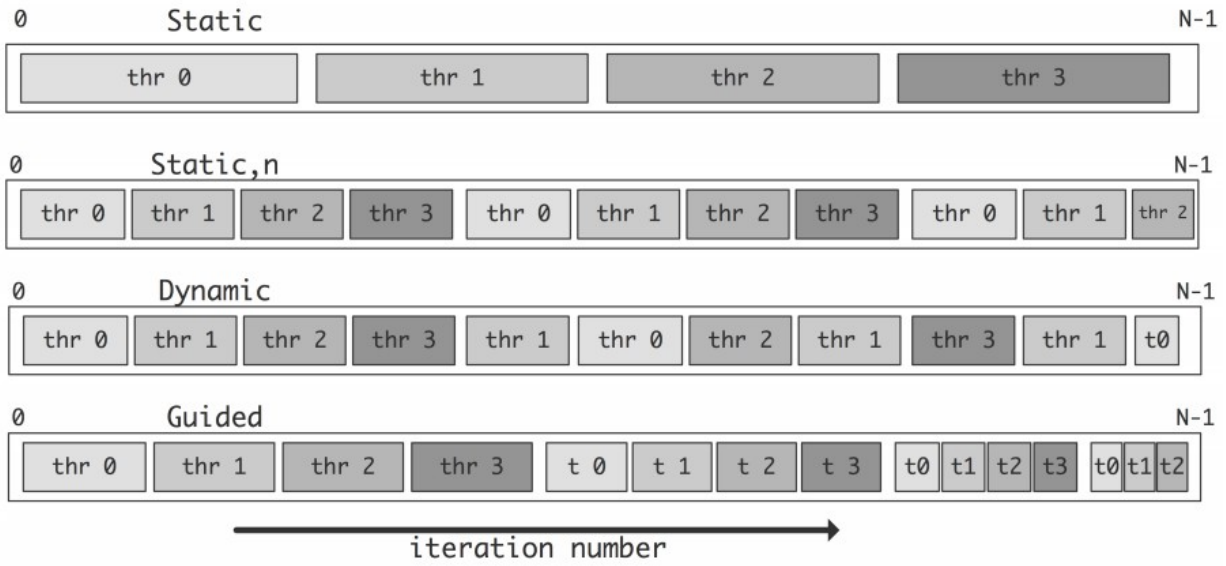


Illustration of the scheduling strategies of loop iterations.

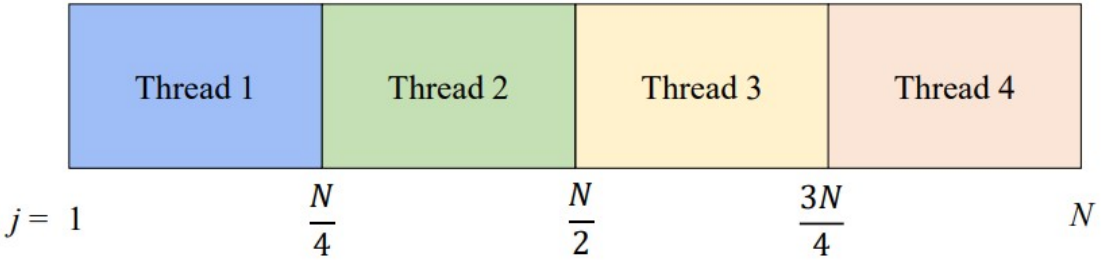
OpenMP: Scheduling

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
}
```

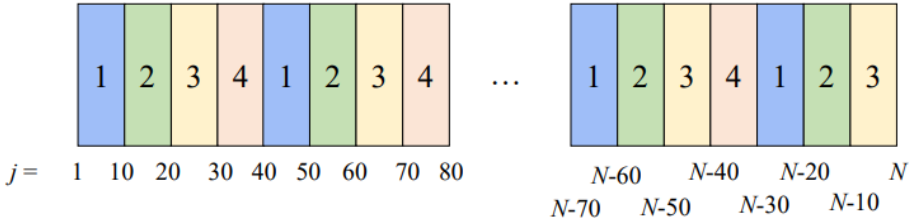
Unspecified schedule



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
}
```



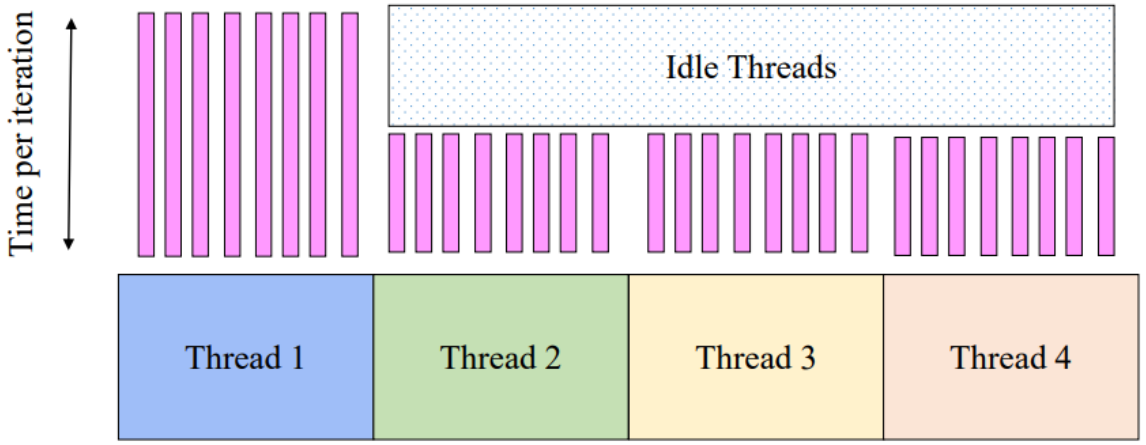
Keeps assigning chunks until done.

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

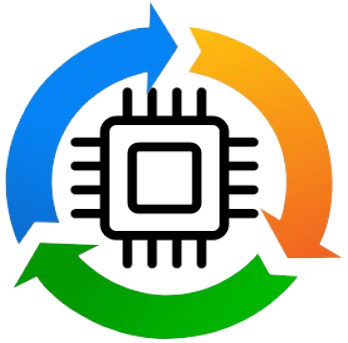
Chunks are assigned on the fly, as threads become available.

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 Examples



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;
}
```

COMPILING an OpenMP Program

- **Compiling a program** for MPI is almost just like compiling a regular C or C++ program
  - For example, to compile **MyProg.c** you would use a command like
  - **gcc** -fopenmp -o **MyProg** **MyProg.c**
  - **gcc** -fopenmp -o **MyProg** **MyProg.cpp**
- **g++** -c **MyProg. cpp** -o **MyProg.o** -fopenmp
- **g++** **MyProg.o** -o **MyProg** -fopenmp -lpthread

OpenMP: Compute PI

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{          int i, nthreads;  double pi, sum[NUM_THREADS];
           step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
        int i, id,nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0)   nthreads = nthrds;
        for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                x = (i+0.5)*step;
                sum[id] += 4.0/(1.0+x*x);
        }
  }
        for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

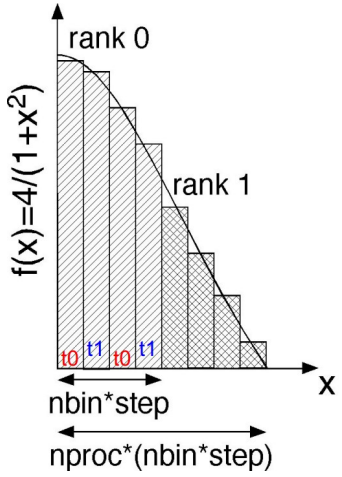
Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

GOAL : The following code computes the π number by using a numerical evaluation of an integral by a rectangle method.

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \cong \Delta \sum_{i=0}^{N-1} \frac{4}{1+x_i^2}$$



OpenMP: Compute PI with padding

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define    PAD      8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{          int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
           step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {       int i, id,nthrds;
           double x;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
           for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                      x = (i+0.5)*step;
                      sum[id][0] += 4.0/(1.0+x*x);
           }
   }
           for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

Pad the array so each sum value is in a different cache line

Remark about false sharing : If independent data elements happen to sit on the same cache line, each update will cause the cache lines to **"slosh back and forth" between threads.**

HotFix with PAD, elements you use are on distinct cache lines.

threads	1st SPMD	1st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

Results

Padding arrays requires deep knowledge of the cache architecture, also be careful...

OpenMP: Compute PI with omp for reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{    int i;          double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel
     {
        double x;
       #pragma omp for reduction(+:sum)
          for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
          }
      }
          pi = step * sum;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop indix is local to a thread by default.

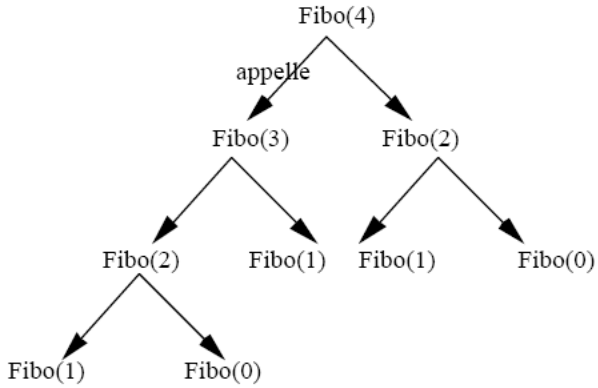
OpenMP: Fibonacci

```
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;
}
```

$$
\begin{cases}
fib(0) = 1 \\
fib(1) = 1 \\
fib(n) = fib(n-1) + fib(n-2)
\end{cases}
$$

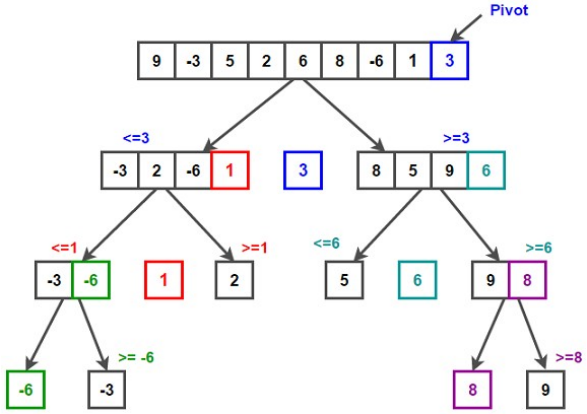
avec $n \in \mathbb{N}$



OpenMP: 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);}
     }
   }
}
```

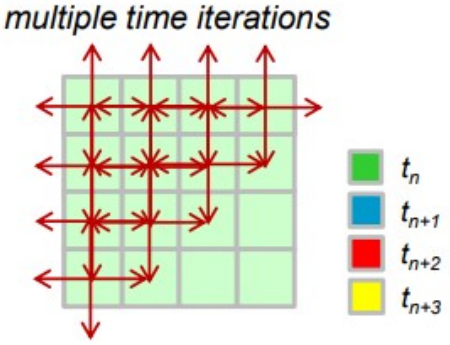
```
void par_quick_sort (int n, float *data)
{
   #pragma omp parallel
   {
      #pragma omp single nowait
      quick_sort (0, n, data);
   }
```



OpenMP: 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.

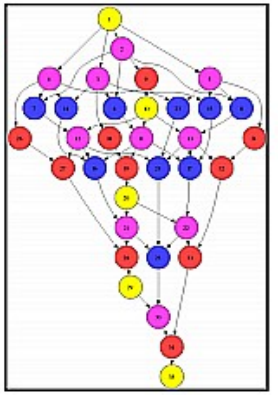


OpenMP: Cholesky Factorization

The **Cholesky factorization**, also known as Cholesky decomposition, is a process of breaking down of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose, which is important for quick numerical solutions in linear algebra.

1: Input: Dictionary $\mathbf{D}$, signal $\underline{x}$, target sparsity K *or* target error ϵ
2: Output: Sparse representation $\underline{\gamma}$ such that $\underline{x} \approx \mathbf{D}\underline{\gamma}$
3: Init: Set $I := (\,)$, $\mathbf{L} := [1]$, $\underline{r} := \underline{x}$, $\underline{\gamma} := \underline{0}$, $\underline{\alpha} := \mathbf{D}^T\underline{x}$, $n := 1$
4: **while** (*stopping criterion not met*) **do**
5: $\hat{k} := \underset{k}{\text{Argmax}}\ |\underline{d}_k^T\,\underline{r}\,|$
6: **if** $n > 1$ **then**
7: $\underline{w} :=$ Solve for $\underline{w}$ $\{\ \mathbf{L}\underline{w} = \mathbf{D}_I^T\underline{d}_{\hat{k}}\ \}$
8: $\mathbf{L} := \begin{bmatrix} \mathbf{L} & \underline{0} \\ \underline{w}^T & \sqrt{1-\underline{w}^T\underline{w}} \end{bmatrix}$
9: **end if**
10: $I := (I, \hat{k})$
11: $\underline{\gamma}_I :=$ Solve for $\underline{c}$ $\{\ \mathbf{L}\mathbf{L}^T\underline{c} = \underline{\alpha}_I\ \}$
12: $\underline{r} := \underline{x} - \mathbf{D}_I\underline{\gamma}_I$
13: $n := n + 1$
14: **end while**

```
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: Performance Tips...

- Avoid serialization !
- Avoid using **#pragma omp parallel** for before loop.
- Use **reduction** whenever possible.
- Minimize I/O
- Minimize critical
  - Use **atomic** instead of **critical** where possible.

Thank you for your attention !