

Lecture 12: Introduction to OpenMP (Part 1)

What is OpenMP

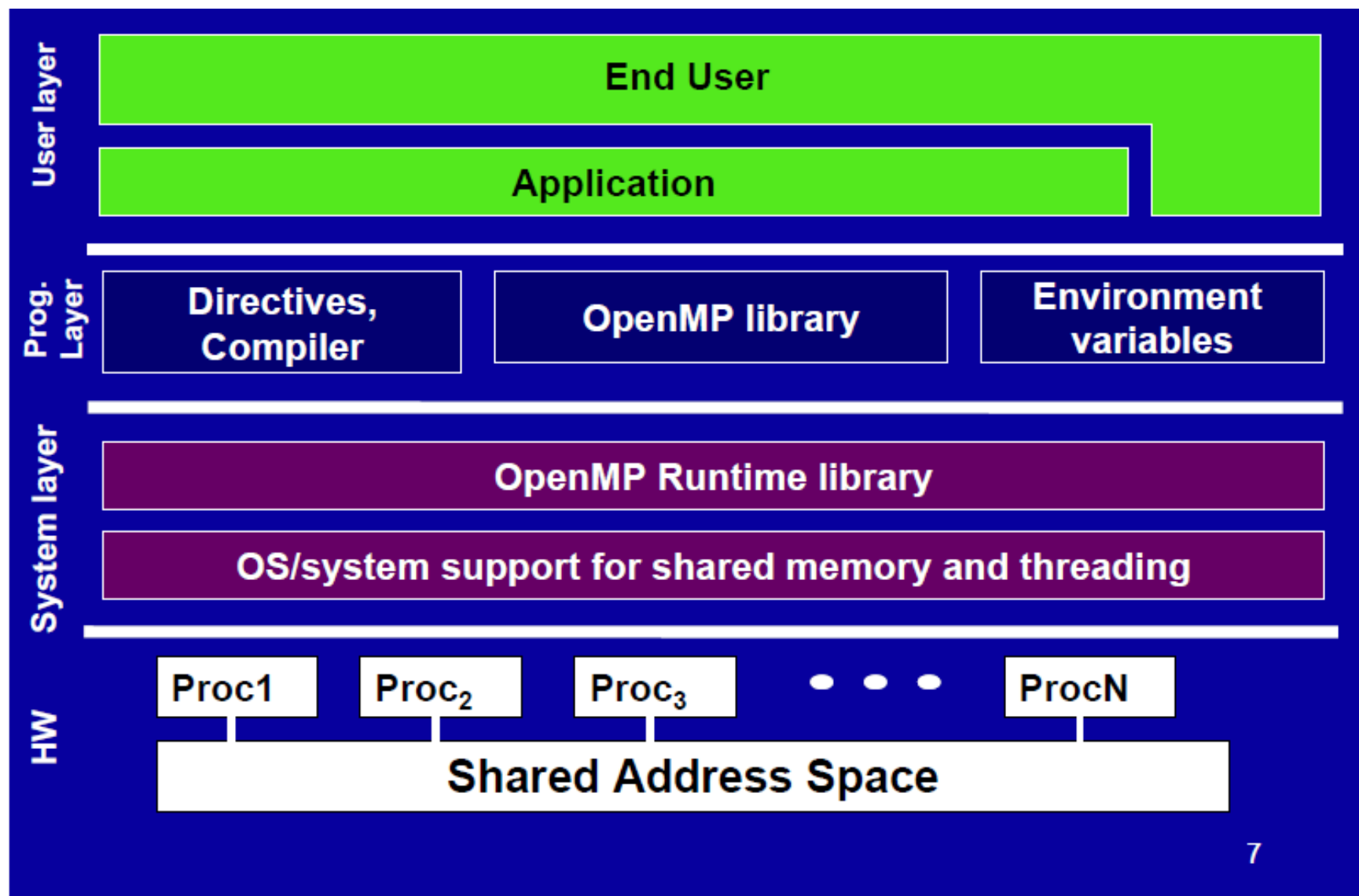
Open specifications for Multi Processing

Long version: Open specifications for MultiProcessing via collaborative work between interested parties from the hardware and software industry, government and academia.

- An Application Program Interface (API) that is used to explicitly direct multi-threaded, shared memory parallelism.
- API components:
 - Compiler directives
 - Runtime library routines
 - Environment variables
- Portability
 - API is specified for C/C++ and Fortran
 - Implementations on almost all platforms including Unix/Linux and Windows
- Standardization
 - Jointly defined and endorsed by major computer hardware and software vendors
 - Possibility to become ANSI standard

Brief History of OpenMP

- In 1991, Parallel Computing Forum (PCF) group invented a set of directives for specifying loop parallelism in Fortran programs.
- X3H5, an ANSI subcommittee developed an ANSI standard based on PCF.
- In 1997, the first version of OpenMP for Fortran was defined by OpenMP Architecture Review Board.
- Binding for C/C++ was introduced later.
- Version 3.1 is available since 2011.



Thread

- A **process** is an instance of a computer program that is being executed. It contains the program code and its current activity.
- A **thread of execution** is the smallest unit of processing that can be scheduled by an operating system.
- Differences between threads and processes:
 - A thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory. The threads of a process share the latter's instructions (code) and its context (values that its variables reference at any given moment).
 - Different processes do not share these resources.

[http://en.wikipedia.org/wiki/Process \(computing\)](http://en.wikipedia.org/wiki/Process_(computing))

Process

- A process contains all the information needed to execute the program
 - Process ID
 - Program code
 - Data on run time stack
 - Global data
 - Data on heap

Each process has its own address space.

- In multitasking, processes are given time slices in a round robin fashion.
 - If computer resources are assigned to another process, the status of the present process has to be saved, in order that the execution of the suspended process can be resumed at a later time.

Threads

- Thread model is an extension of the process model.
- Each process consists of multiple independent instruction streams (or threads) that are assigned computer resources by some scheduling procedure.
- Threads of a process share the address space of this process.
 - Global variables and all dynamically allocated data objects are accessible by all threads of a process
- Each thread has its own run-time stack, register, program counter.
- Threads can communicate by reading/writing variables in the common address space.

OpenMP Programming Model

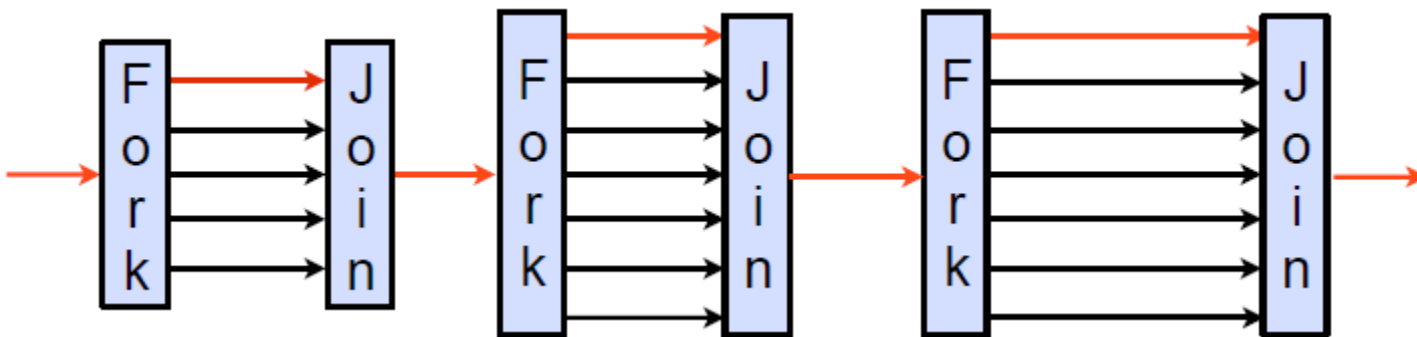
- Shared memory, thread-based parallelism
 - OpenMP is based on the existence of multiple threads in the shared memory programming paradigm.
 - A shared memory process consists of multiple threads.
- Explicit Parallelism
 - Programmer has full control over parallelization. OpenMP is not an automatic parallel programming model.
- Compiler directive based
 - Most OpenMP parallelism is specified through the use of compiler directives which are embedded in the source code.

OpenMP is not

- Necessarily implemented identically by all vendors
- Meant for distributed-memory parallel systems (it is designed for shared address spaced machines)
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- Required to check for code sequences
- Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
- Designed to guarantee that input or output to the same file is synchronous when executed in parallel.

Fork-Join Parallelism

- OpenMP program begin as a single process: the *master thread*. The master thread executes sequentially until the first *parallel region* construct is encountered.
- When a parallel region is encountered, master thread
 - Create a group of threads by **FORK**.
 - Becomes the master of this group of threads, and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the *parallel region* construct are then executed in parallel among these threads.
- **JOIN**: When the threads complete executing the statement in the *parallel region* construct, they synchronize and terminate, leaving only the master thread.



Master thread is shown in red.

I/O

- OpenMP does not specify parallel I/O.
- It is up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

Memory Model

- Threads can “cache” their data and are not required to maintain exact consistency with real memory all of the time.
- When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is updated by all threads as needed.

OpenMP Code Structure

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

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

Set # of threads for OpenMP

In csh

setenv OMP_NUM_THREAD 8

Compile: g++ -fopenmp hello.c

Run: ./a.out

See: <http://wiki.crc.nd.edu/wiki/index.php/OpenMP>

- “Pragma”: stands for “pragmatic information. A pragma is a way to communicate the information to the compiler.
- The information is non-essential in the sense that the compiler may ignore the information and still produce correct object program.

OpenMP Core Syntax

```
#include "omp.h"
int main ()
{
    int var1, var2, var3;
    // Serial code
    ...
    // Beginning of parallel section.
    // Fork a team of threads. Specify variable scoping
    #pragma omp parallel private(var1, var2) shared(var3)
    {
        // Parallel section executed by all threads
        ...
        // All threads join master thread and disband
    }

    // Resume serial code ...
}
```

OpenMP C/C++ Directive Format

OpenMP directive forms

- C/C++ use compiler directives
 - Prefix: `#pragma omp ...`
- A directive consists of a directive name followed by *clauses*

Example: `#pragma omp parallel default (shared) private (var1, var2)`

OpenMP Directive Format (2)

General Rules:

- Case sensitive
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be “continued” on succeeding lines by escaping the newline character with a backslash “\” at the end of a directive line.

OpenMP parallel Region Directive

#pragma omp parallel [*clause list*]

Typical clauses in [*clause list*]

- Conditional parallelization
 - **if** (scalar expression)
 - Determine whether the parallel construct creates threads
- Degree of concurrency
 - **num_threads** (integer expression)
 - number of threads to create
- Data Scoping
 - **private** (variable list)
 - Specifies variables local to each thread
 - **firstprivate** (variable list)
 - Similar to the private
 - Private variables are initialized to variable value before the parallel directive
 - **shared** (variable list)
 - Specifies variables that are shared among all the threads
 - **default** (data scoping specifier)
 - Default data scoping specifier may be **shared** or **none**

Example:

```
#pragma omp parallel if (is_parallel == 1) num_threads(8) shared (var_b)
private (var_a) firstprivate (var_c) default (none)
{
/* structured block */
}
```

- **if (is_parallel == 1) num_threads(8)**
 - If the value of the variable is_parallel is one, create 8 threads
- **shared (var_b)**
 - Each thread shares a single copy of variable b
- **private (var_a) firstprivate (var_c)**
 - Each thread gets private copies of variable var_a and var_c
 - Each private copy of var_c is initialized with the value of var_c in main thread when the parallel directive is encountered
- **default (none)**
 - Default state of a variable is specified as none (rather than shared)
 - Signals error if not all variables are specified as **shared** or **private**

Number of Threads

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
 1. Evaluation of the `if` clause
 2. Setting of the `num_threads()` clause
 3. Use of the `omp_set_num_threads()` library function
 4. Setting of the `OMP_NUM_THREAD` environment variable
 5. Implementation default – usually the number of cores on a node
- Threads are numbered from 0 (master thread) to N-1

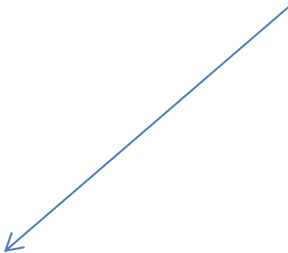
Thread Creation: Parallel Region Example

- Create threads with the parallel construct

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

```
int main()
{
    int nthreads, tid;
    #pragma omp parallel num_threads(4) private(tid)
    {
        tid = omp_get_thread_num();
        printf("Hello world from (%d)\n", tid);
        if(tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("number of threads = %d\n", nthreads);
        }
    } // all threads join master thread and terminates
}
```

Clause to request
threads



Each thread executes a
copy of the code
within the structured
block

Thread Creation: Parallel Region Example

```
#include <stdlib.h>
```

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

```
#include "omp.h"
```

```
int main(){
```

```
    int nthreads, A[100] , tid;
```

```
    // fork a group of threads with each thread having a private tid variable
```

```
    omp_set_num_threads(4);
```

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

```
{
```

```
    tid = omp_get_thread_num();
```

```
    foo(tid, A);
```

```
} // all threads join master thread and terminates
```

```
}
```

A single copy of A[] is shared
between all threads

SPMD vs. Work-Sharing

- A parallel construct by itself creates a “single program multiple data” program, i.e., each thread executes the same code.
- Work-sharing is to split up pathways through the code between threads within a team.
 - Loop construct
 - Sections/section constructs
 - Single construct
 - ...

Work-Sharing Construct

- Within the scope of a parallel directive, work-sharing directives allow concurrency between iterations or tasks
- Work-sharing constructs do not create new threads
- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all.
- Two directives to be studied
 - **Do/for**: concurrent loop iterations
 - **sections**: concurrent tasks

Work-Sharing Do/for Directive

Do/for

- Shares iterations of a loop across the group
- Represents a “data parallelism”.

for directive partitions parallel iterations across threads

Do is the analogous directive in Fortran

Usage:

```
#pragma omp for [clause list]
```

```
/* for loop */
```

- Implicit barrier at end of **for** loop

Example Using *for*

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

int main()
{
    int nthreads, tid;

    omp_set_num_threads(3);
    #pragma omp parallel private(tid)
    {
        int i;
        tid = omp_get_thread_num();
        printf("Hello world from (%d)\n", tid);
        #pragma omp for
        for(i = 0; i <=4; i++)
        {
            printf("Iteration %d by %d\n", i, tid);
        }
    } // all threads join master thread and terminates
}
```

Another Example Using *for*

- Sequential code to add two vectors
`for(i=0;i<N;i++) {c[i] = b[i] + a[i];}`

- OpenMP implementation 1 (not desired)**

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id*N/Nthrds;
    iend = (id+1)*N/Nthrds;
    if(id == Nthrds-1) iend = N;
    for(i = istart; i<iend; i++) {c[i] = b[i]+a[i];}
}
```

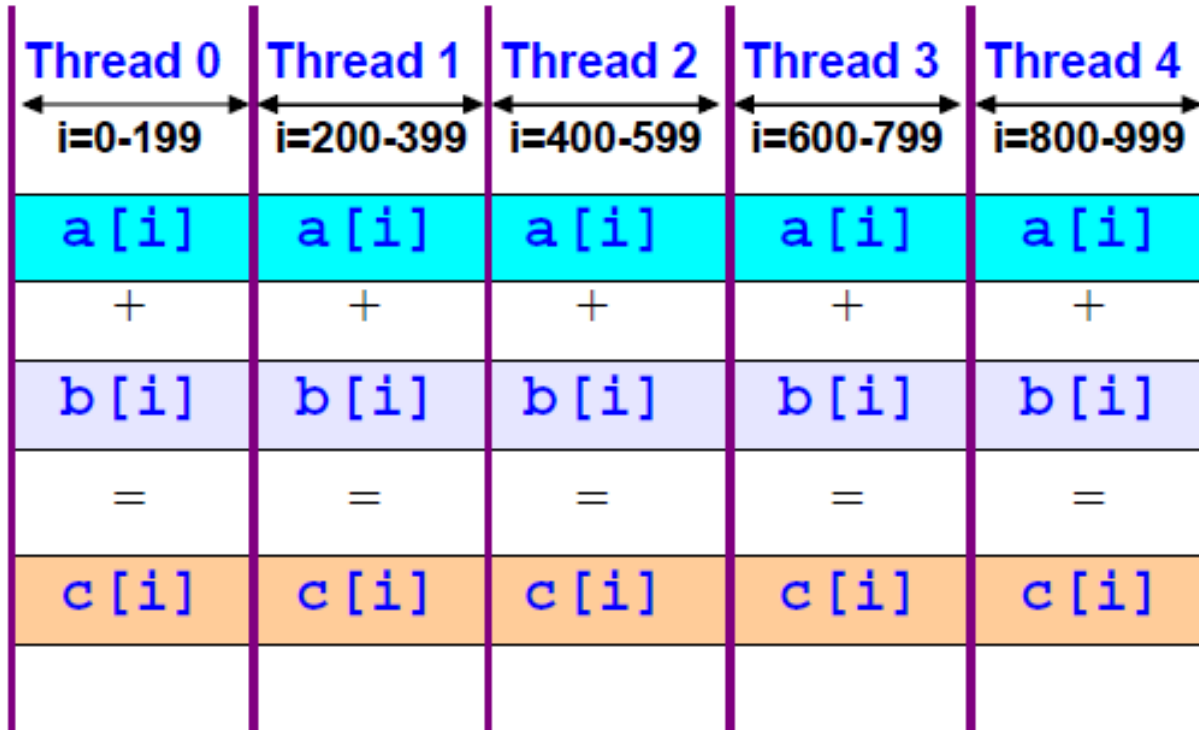
- A worksharing for construct to add vectors**

```
#pragma omp parallel
{
    #pragma omp for
    {
        for(i=0; i<N; i++) {c[i]=b[i]+a[i];}
    }
}
```

or

```
#pragma omp parallel for
{
    for(i=0; i<N; i++) {c[i]=b[i]+a[i];}
}
```

Execution for loop in parallel



```
int main()
```

```
{
```

```
    int b[3];
```

```
    char *cptr;
```

```
    int i;
```

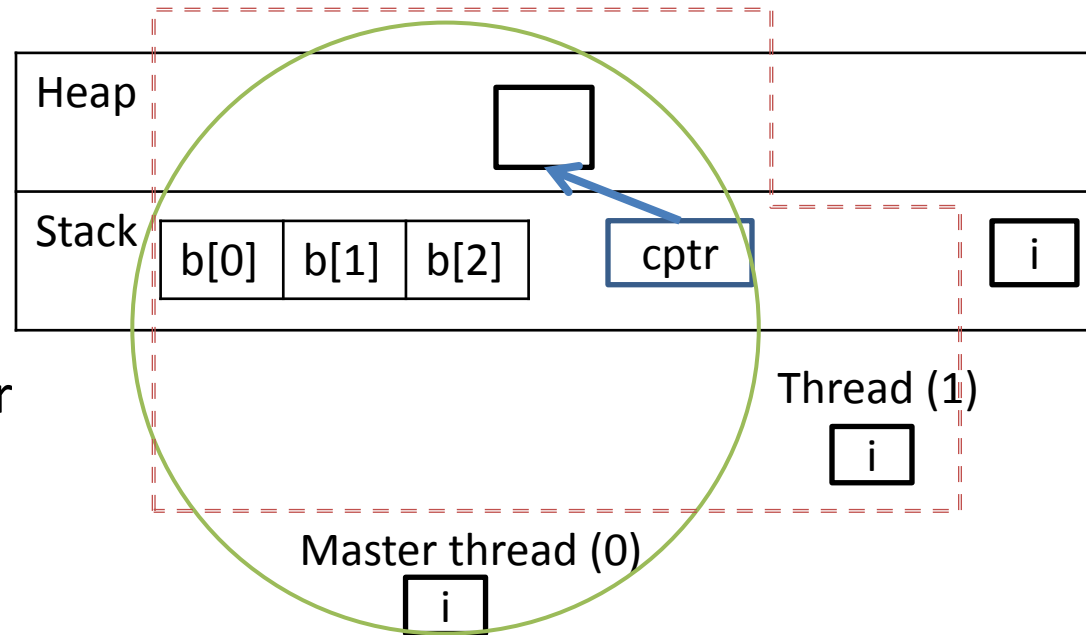
```
    cptr = malloc(1);
```

```
    #pragma omp parallel for
```

```
        for(i=0; i<3; i++)
```

```
            b[i]=i;
```

```
}
```



Every thread has its own **execution context**: an address space containing all of the variables the thread may access. The execution context includes static variables, dynamically allocated data structures in the heap, and variables on the run-time stack. The execution context includes its own additional run-time stack. A **shared variable** has the same address in the execution context of every thread. All threads have access to shared variables. A **private variable** has a different address in the execution context of every thread.

Example. During parallel execution of the for loop, index “i” is a private variable, while “b”, “cptr” and heap data are shared.

- Canonical shape of “for” loop

for(index = start; index {<, or ≤ or ≥ or >} end; $\begin{matrix} index++ \\ index-- \\ index += inc \\ index -= inc \end{matrix}$)

- “for” loop must not contain statements that allow the loop to be exited prematurely.
 - Examples include: “break” statement, “return” statement, “exit” statement and “goto” statement.
- The “continue” statement is allowed.

C/C++ **for** Directive Syntax

```
#pragma omp for [clause list]
    schedule (type [,chunk])
    ordered
    private (variable list)
    firstprivate (variable list)
    lastprivate (variable list)
    shared (variable list)
    reduction (operator: variable list)
    collapse (n)
    nowait

/* for_loop */
```

Private Clause

- Direct the compiler to make one or more variables private.

```
#pragma omp parallel for private (j)
  for(i = 0; i < M; i++)
    for(j=0; j < N; j++)
      a[i][j] = min(a[i][j], a[i][k]+tmp[j]);
```

- We need every thread to work through N values of “j” for each iteration of the “i” loop.
- If we do not make “j” private, all of threads try to initialize and increment the same shared variable “j” – meaning the data race.
- The private copies of variable “j” will be accessible only inside the for loop. The values are undefined on loop entry and exit.

firstprivate Clause

```
x[0] = 1.0;
for(i=0; i < n; i++){
    for(j=1; j<4; j++)
        x[j]=g(i, x[j-1]);
    answer[i]=x[1]-x[3];
}
```

- We want each thread's private copy of array element x[0] to inherit the value that the shared variable was assigned in the master thread.

```
x[0] = 1.0;
#pragma omp parallel for private (j) firstprivate (x)
for(i=0; i < n; i++){
    for(j=1; j<4; j++)
        x[j]=g(i, x[j-1]);
    answer[i]=x[1]-x[3];
}
```


lastprivate Clause

- **Sequentially last iteration:** the iteration that occurs last when the loop is executed sequentially.
- The lastprivate clause directs the compiler to generate code at the end of the parallel for loop that copies back to the master thread's copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration of the loop.

```
for(i=0; i < n; i++){  
    x[0] = 1.0;  
    for(j=1; j<4; j++)  
        x[j]= x[j-1]*(i+1);  
    answer[i]=x[0]+x[1]+x[2]+x[3];  
}  
n_cubed = x[3];
```

- In the sequentially last iteration of the loop, $x[3]$ gets assigned the value n^3 .
- To have this value accessible outside the parallel for loop, we declare x to be a lastprivate variable.

```
#pragma omp parallel for private(j) lastprivate(x)  
for(i=0; i < n; i++){  
    x[0] = 1.0;  
    for(j=1; j<4; j++)  
        x[j]= x[j-1]*(i+1);  
    answer[i]=x[0]+x[1]+x[2]+x[3];  
}  
n_cubed = x[3];
```

Reduction

- **Serial code**

```
{  
    double avg = 0.0, a[MAX];  
    int i;  
    ...  
    for(i = 0; i < MAX; i++) {avg += a[i];}  
    avg /= MAX;  
}
```

- How to combine values into a single accumulation variable (avg)?

Reduction Clause

- *Reduction (operator: variable list)*: specifies how to combine local copies of a variable in different threads into a single copy at the master when threads exit. Variables in *variable list* are implicitly private to threads.

- Operators: +, *, -, &, |, ^, &&, and ||

- Usage

```
#pragma omp parallel reduction(+: sums) num_threads(4)
```

```
{
```

```
    /* compute local sums in each thread
```

```
}
```

```
/* sums here contains sum of all local instances of sum */
```

Reduction in OpenMP **for**

- Inside a parallel or a work-sharing construct:
 - A local copy of each list variable is made and initialized depending on *operator* (e.g. 0 for “+”)
 - Compiler finds standard reduction expressions containing *operator* and uses it to update the local copy.
 - Local copies are reduced into a single value and combined with the original global value when returns to the master thread.

```
{  
    double avg = 0.0, a[MAX];  
    int i;  
    ...  
    #pragma omp parallel for reduction (+:avg)  
    for(i =0; i<MAX; i++) {avg += a[i];}  
    avg /= MAX;  
}
```

Reduction Operators/Initial-Values

C/C++:

Operator	Initial Value
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0

Monte Carlo to estimate Pi

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

int main(int argc, char *argv[])
{
    long int  i, count;          // count points inside unit circle
    long int  samples;          // number of samples
    double pi;
    unsigned short xi[3] = {1, 5, 177}; // random number seed
    double x, y;
    samples = atoi(argv[1]);
    count = 0;
    for(i = 0; i < samples; i++)
    {
        x = erand48(xi);
        y = erand48(xi);
        if(x*x + y*y <= 1.0) count++;
    }

    pi = 4.0*count/samples;
    printf("Estimate of pi: %7.5f\n", pi);
}
```

OpenMP version of Monte Carlo to Estimate PI

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

main(int argc, char *argv[])
{
    int i, count; /* points inside the unit quarter circle */
    unsigned short xi[3]; /* random number seed */
    int samples; /* samples Number of points to generate */
    double x,y; /* Coordinates of points */
    double pi; /* Estimate of pi */

    samples = atoi(argv[1]);

    #pragma omp parallel
    {
        xi[0] = 1; /* These statements set up the random seed */
        xi[1] = 1;
        xi[2] = omp_get_thread_num();
        count = 0;
        printf("I am thread %d\n", xi[2]);
        #pragma omp for firstprivate(xi) private(x,y) reduction(+:count)
        for (i = 0; i < samples; i++)
        {
            x = erand48(xi);
            y = erand48(xi);
            if (x*x + y*y <= 1.0) count++;
        }
    }
    pi = 4.0 * (double)count / (double)samples;
    printf("Count = %d, Samples = %d, Estimate of pi: %7.5f\n", count, samples, pi);
}
```

- A local copy of “count” for each thread
- All local copies of “count” added together and stored in master thread
- Each thread needs different random number seeds.

Matrix-Vector Multiplication

```
#pragma omp parallel default (none) \  
shared (a, b, c, m,n) private (i,j,sum)  
num_threads(4)  
for(i=0; i < m; i++){  
    sum = 0.0;  
    for(j=0; j < n; j++)  
        sum += b[i][j]*c[j];  
    a[i] =sum;  
}
```

```
for (i=0,1,2,3,4)
```

```
i = 0
```

```
sum = b[i=0][j]*c[j]  
a[0] = sum
```

```
i = 1
```

```
sum = b[i=1][j]*c[j]  
a[1] = sum
```

Thread 0,

```
for (i=5,6,7,8,9)
```

```
i = 5
```

```
sum = b[i=5][j]*c[j]  
a[5] = sum
```

```
i = 6
```

```
sum = b[i=6][j]*c[j]  
a[6] = sum
```

Thread 1,

...etc...

schedule clause

- Describe how iterations of the loop are divided among the threads in the group. The default schedule is implementation dependent.
- Usage: `schedule (scheduling_class[, parameter])`.
 - *static*

Loop iterations are divided into pieces of size `chunk` and then statically assigned to threads. If `chunk` is not specified, the iteration are evenly (if possible) divided contiguously among the threads.
 - *dynamic*

Loop iterations are divided into pieces of size `chunk` and then dynamically assigned to threads. When a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.
 - *guided*

For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size with value $k (k > 1)$, the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). The default chunk size is 1.
 - *runtime*

The scheduling decision is deferred until runtime by the environment variable `OMP_SCHEDULE`. It is illegal to specify a chunk size for this clause
 - *auto*

The scheduling decision is made by the compiler and/or runtime system.

- Static scheduling
- 16 iterations, 4 threads:

Thread	0	1	2	3
<i>no chunk*</i>	1-4	5-8	9-12	13-16
<i>chunk = 2</i>	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

Static Scheduling

```
// static scheduling of matrix multiplication loops
#pragma omp parallel default (private) \
shared (a, b, c, dim) num_threads(4)
#pragma omp for schedule(static)
for(i=0; i < dim; i++)
{
    for(j=0; j < dim; j++)
    {
        c[i][j] = 0.0;
        for(k=0; k < dim; k++)
            c[i][j] += a[i][k]*b[k][j];
    }
}
```

Static schedule maps iterations to threads
at compile time

Dynamic Scheduling

- The time needed to execute different loop iterations may vary considerably.

```
for(i=0; i<n; i++)  
{  
    for(j=i; j < n; j++)  
        a[i][j] = rand();  
}
```

- The first iteration of the outermost loop ($i=0$) requires n times more work than the last iteration ($i=n-1$). Inverting the two loops will not remedy the imbalance.

```
#pragma omp parallel default (private) \  
shared (a, n) private(j) num_threads(4)  
#pragma omp for schedule(dynamic)  
for(i=0; i<n; i++)  
{  
    for(j=i; j < n; j++)  
        a[i][j] = rand();  
}
```

Environment Variables

- `OMP_SCHEDULE` “schedule[, chunk_size]”
 - Control how “omp for schedule (RUNTIME)” loop iterations are scheduled.
- `OMP_NUM_THREADS` integer
 - Set the default number of threads to use
- `OMP_DYNAMIC` TRUE | FALSE
 - Can the program use a different number of threads in each parallel region?
- `OMP_NESTED` TRUE | FALSE
 - Will nested parallel regions create new teams of threads, or will they be serialized?

By default, worksharing **for** loops end with an implicit barrier

- *nowait*: If specified, threads do not synchronize at the end of the parallel loop
- *ordered*: specifies that the iteration of the loop must be executed as they would be in serial program.
- *collapse*: specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iteration in all associated loops determines the order of the iterations in the collapsed iteration space.

Avoiding Synchronization with nowait

```
#pragma omp parallel shared(A,B,C) private(id)
{
    id = omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
    for(i = 0; i < N; i++) { C[i] = big_calc3(i,A); }
    #pragma omp for nowait
    for(i = 0; i < N; i++) { B[i] = big_calc2(C,i); }
    A[id] = big_calc4(id);
}
```

Barrier: each threads waits till all threads arrive.

No implicit barrier due to **nowait**. Any thread can begin **big_calc4()** immediately without waiting for other threads to finish the loop

Implicit barrier at the end of the parallel region

- By default: worksharing **for** loops end with an implicit barrier
- **nowait** clause:
 - Modifies a **for** directive
 - Avoids implicit barrier at end of **for**

Loop Collapse

- Allows parallelization of perfectly nested loops without using nested parallelism
- Compiler forms a single loop and then parallelizes this

```
{  
  ...  
  #pragma omp parallel for collapse (2)  
  for(i=0;i< N; i++)  
  {  
    for(j=0;j< M; j++)  
    {  
      foo(A,i,j);  
    }  
  }  
}
```

For Directive Restrictions

For the “*for* loop” that follows the *for* directive:

- It must not have a break statement
- The loop control variable must be an integer
- The initialization expression of the “*for* loop” must be an integer assignment.
- The logical expression must be one of $<$, \leq , $>$, \geq
- The increment expression must have integer increments or decrements only.

Lecture 12: Introduction to OpenMP (Part 2)

Performance Issues I

- C/C++ stores matrices in row-major fashion.
- Loop interchanges may increase cache locality

```
{  
  ...  
  #pragma omp parallel for  
  for(i=0;i< N; i++)  
  {  
    for(j=0;j< M; j++)  
    {  
      A[i][j] =B[i][j] + C[i][j];  
    }  
  }  
}
```

- Parallelize outer-most loop

Performance Issues II

- Move synchronization points outwards. The inner loop is parallelized.
- In each iteration step of the outer loop, a parallel region is created. This causes parallelization overhead.

```
{  
    ...  
    for(i=0;i< N; i++)  
    {  
        #pragma omp parallel for  
        for(j=0;j< M; j++)  
        {  
            A[i][j] =B[i][j] + C[i][j];  
        }  
    }  
}
```

Performance Issues III

- Avoid parallel overhead at low iteration counts

```
{  
  ...  
  
  #pragma omp parallel for if(M > 800)  
  for(j=0;j< M; j++)  
  {  
    aa[j] =alpha*bb[j] + cc[j];  
  }  
}
```

C++: Random Access Iterators Loops

- Parallelization of random access iterator loops is supported

```
void iterator_example(){
    std::vector vec(23);
    std::vector::iterator it;

    #pragma omp parallel for default(none) shared(vec)
    for(it=vec.begin(); it< vec.end(); it++)
    {
        // do work with it //
    }
}
```

Conditional Compilation

- Keep sequential and parallel programs as a single source code

```
#if def _OPENMP
#include "omp.h"
#endif

Main()
{
#ifdef _OPENMP
    omp_set_num_threads(3);
#endif
    for(i=0;i< N; i++)
    {
        #pragma omp parallel for
        for(j=0;j< M; j++)
        {
            A[i][j] =B[i][j] + C[i][j];
        }
    }
}
```


Be Careful with Data Dependences

- Whenever a statement in a program reads or writes a memory location and another statement reads or writes the same memory location, and at least one of the two statements writes the location, then there is a data dependence on that memory location between the two statements. The loop may not be executed in parallel.

```
for(i=1;i< N; i++)  
{  
    a[i] = a[i] + a[i-1];  
}
```

$a[i]$ is written in loop iteration i and read in loop iteration $i+1$. This loop can not be executed in parallel. The results may not be correct.

Classification of Data Dependences

- A data dependence is called **loop-carried** if the two statements involved in the dependence occur in different iterations of the loop.
- Let the statement executed earlier in the sequential execution be loop S1 and let the later statement be S2.
 - Flow dependence: the memory location is written in S1 and read in S2. S1 executes before S2 to produce the value that is consumed in S2.
 - Anti-dependence: The memory location is read in S1 and written in S2.
 - Output dependence: The memory location is written in both statements S1 and S2.

- Anti-dependence

```
for(i=0;i< N-1; i++)  
{  
    x = b[i] + c[i];  
    a[i] = a[i+1] + x;  
}
```

- Parallel version with dependence removed

```
#pragma omp parallel for shared (a, a2)  
for(i=0; i < N-1; i++)  
    a2[i] = a[i+1];  
#pragma omp parallel for shared (a, a2) lastprivate(x)  
for(i=0;i< N-1; i++)  
{  
    x = b[i] + c[i];  
    a[i] = a2[i] + x;  
}
```

```
for(i=1;i< m; i++)  
    for(j=0;j<n;j++)  
    {  
        a[i][j] = 2.0*a[i-1][j];  
    }
```

```
for(i=1;i< m; i++)  
    #pragma omp parallel for  
    for(j=0;j<n;j++)  
    {  
        a[i][j] = 2.0*a[i-1][j];  
    }
```

```
#pragma omp parallel for private (i)  
for(j=0;j< n; j++)  
    for(i=1;i<m;i++)  
    {  
        a[i][j] = 2.0*a[i-1][j];  
    }
```

Poor performance, it requires $m-1$ fork/join steps.

- Invert loop to yield better performance(?).
- With this inverting, only a single fork/join step is needed. The data dependences have not changed.
- However, this change affect the cache hit rate.

- Flow dependence is in general difficult to be removed.

```
X = 0.0;  
for(i=0;i< N; i++)  
{  
    X = X + a[i];  
}
```

```
X = 0.0;  
#pragma omp parallel for reduction(+:x)  
for(i=0;i< N; i++)  
{  
    x = x + a[i];  
}
```

- Elimination of induction variables.

```
idx = N/2+1; isum = 0; pow2 = 1;
for(i=0;i< N/2; i++)
{
    a[i] = a[i] + a[idx];
    b[i] = isum;
    c[i] = pow2;
    idx++; isum += i; pow2 *=2;
}
```

- Parallel version

```
#pragma omp parallel for shared (a,b)
for(i=0;i< N/2; i++)
{
    a[i] = a[i] + a[i+N/2];
    b[i] = i*(i-1)/2;
    c[i] = pow(2,i);
}
```

- Remove flow dependence using loop skewing

```
for(i=1;i< N; i++)  
{  
    b[i] = b[i] + a[i-1];  
    a[i] = a[i]+c[i];  
}
```

- Parallel version

```
b[1]=b[1]+a[0];  
#pragma omp parallel for shared (a,b,c)  
for(i=1;i< N-1; i++)  
{  
    a[i] = a[i] + c[i];  
    b[i+1] = b[i+1]+a[i];  
}  
a[N-1] = a[N-1]+c[N-1];
```

- A flow dependence that can in general not be remedied is a **recurrence**:

```
for(i=1;i< N; i++)  
{  
    z[i] = z[i] + l[i]*z[i-1];  
}
```


Recurrence: LU Factorization of Tridiagonal Matrix

$$\begin{pmatrix} a_0 & c_0 & & & & \\ b_1 & a_1 & c_1 & & & \\ & b_2 & a_2 & c_2 & & \\ & & b_3 & a_3 & c_3 & \\ & & & b_4 & a_4 & c_4 \\ & & & & b_5 & a_5 \end{pmatrix} = \begin{pmatrix} 1 & & & & & \\ \ell_1 & 1 & & & & \\ & \ell_2 & 1 & & & \\ & & \ell_3 & 1 & & \\ & & & \ell_4 & 1 & \\ & & & & \ell_5 & 1 \end{pmatrix} \begin{pmatrix} d_0 & c_0 & & & & \\ & d_1 & c_1 & & & \\ & & d_2 & c_2 & & \\ & & & d_3 & c_3 & \\ & & & & d_4 & c_4 \\ & & & & & d_5 \end{pmatrix}$$

$T = LU$

- $\mathbf{Tx}=\mathbf{LUx}=\mathbf{Lz}=\mathbf{b}$, $\mathbf{z}=\mathbf{Ux}$.
- Proceed as follows:
- $\mathbf{Lz}=\mathbf{b}$, $\mathbf{Ux}=\mathbf{z}$
- $\mathbf{Lz}=\mathbf{b}$ is solved by:

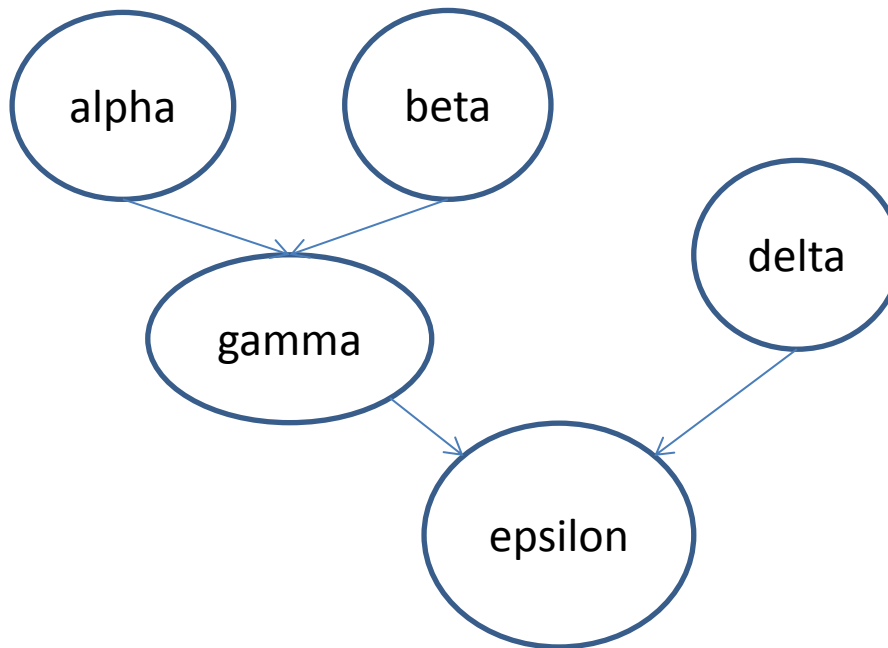
```

z[0] = b[0];
for(i=1;i< n; i++)
{
    z[i] = b[i] - l[i]*z[i-1];
}

```

- Cyclic reduction probably is the best method to solve tridiagonal systems
- Z. Liu, B. Chapman, Y. Wen and L. Huang. *Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization*. OpenMP shared memory parallel programming: International Workshop on OpenMP
- C. Addison, Y. Ren and M. van Waveren. *OpenMP Issues Arising in the Development of Parallel BLAS and LAPACK libraries*. J. Sci. Programming – OpenMP, 11(2), 2003.
- S.F. McGinn and R.E. Shaw. Parallel Gaussian Elimination Using OpenMP and MPI

```
V=alpha();  
W=beta();  
X=gamma(v,w);  
Y=delta();  
printf("%g\n", epsilon(x,y));
```



Data dependence diagram

Functions alpha, beta, delta may be executed in parallel

Worksharing **sections** Directive

sections directive enables specification of task parallelism

- Sections construct gives a different structured block to each thread.

```
#pragma omp sections [clause list]
    private (list)
    firstprivate (list)
    lastprivate (list)
    reduction (operator: list)
    nowait
```

```
{
#pragma omp section
    structured_block
#pragma omp section
    structured_block
}
```

```

#include "omp.h"
#define N 1000
int main(){
    int i;
    double a[N], b[N], c[N], d[N];
    for(i=0; i<N; i++){
        a[i] = i*2.0;
        b[i] = i + a[i]*22.5;
    }
    #pragma omp parallel shared(a,b,c,d) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for(i=0; i<N; i++) c[i] = a[i]+b[i];
            #pragma omp section
            for(i=0; i<N; i++) d[i] = a[i]*b[i];
        }
    }
}

```

Two tasks are
computed
concurrently

By default, there is a barrier at the end of the sections. Use the "nowait" clause to turn off the barrier.

```
#include "omp.h"

#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        v=alpha();
        #pragma omp section
        w=beta();
    }
    #pragma omp sections
    {
        #pragma omp section
        x=gamma(v,w);
        #pragma omp section
        y=delta();
    }
    printf("%g\n", epsilon(x,y));
}
```

Synchronization I

- Threads communicate through shared variables. Uncoordinated access of these variables can lead to undesired effects.
 - E.g. two threads update (write) a shared variable in the same step of execution, the result is dependent on the way this variable is accessed. This is called a **race condition**.
- To prevent race condition, the access to shared variables must be synchronized.
- Synchronization can be time consuming.
- The **barrier** directive is set to synchronize all threads. All threads wait at the barrier until all of them have arrived.

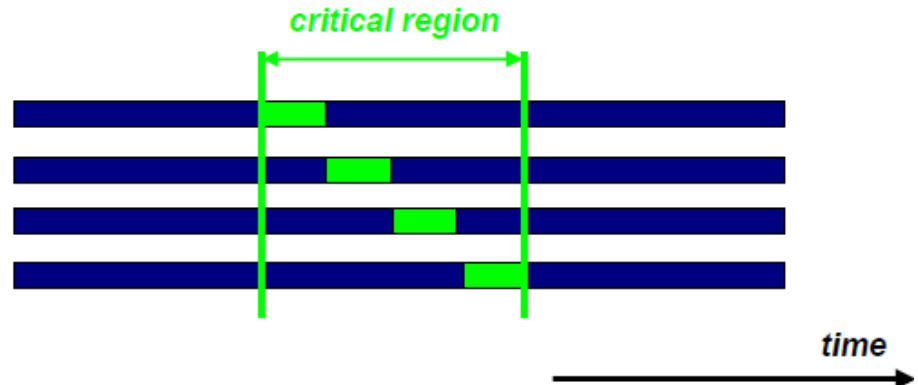
Synchronization II

- Synchronization imposes order constraints and is used to protect access to shared data
- High level synchronization:
 - critical
 - atomic
 - barrier
 - ordered
- Low level synchronization
 - flush
 - locks (both simple and nested)

Synchronization: critical

- Mutual exclusion: only one thread at a time can enter a **critical** region.

```
{
double res;
#pragma omp parallel
{
    double B;
    int i, id, nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for(i=id; i<niters; i+=nthrds){
        B = some_work(i);
        #pragma omp critical
        consume(B,res);
    }
}
```

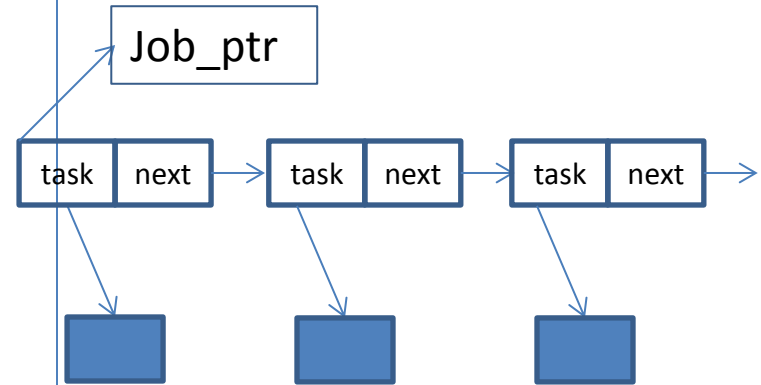


Threads wait here: only one thread at a time calls `consume()`. So this is a piece of sequential code inside the for loop.

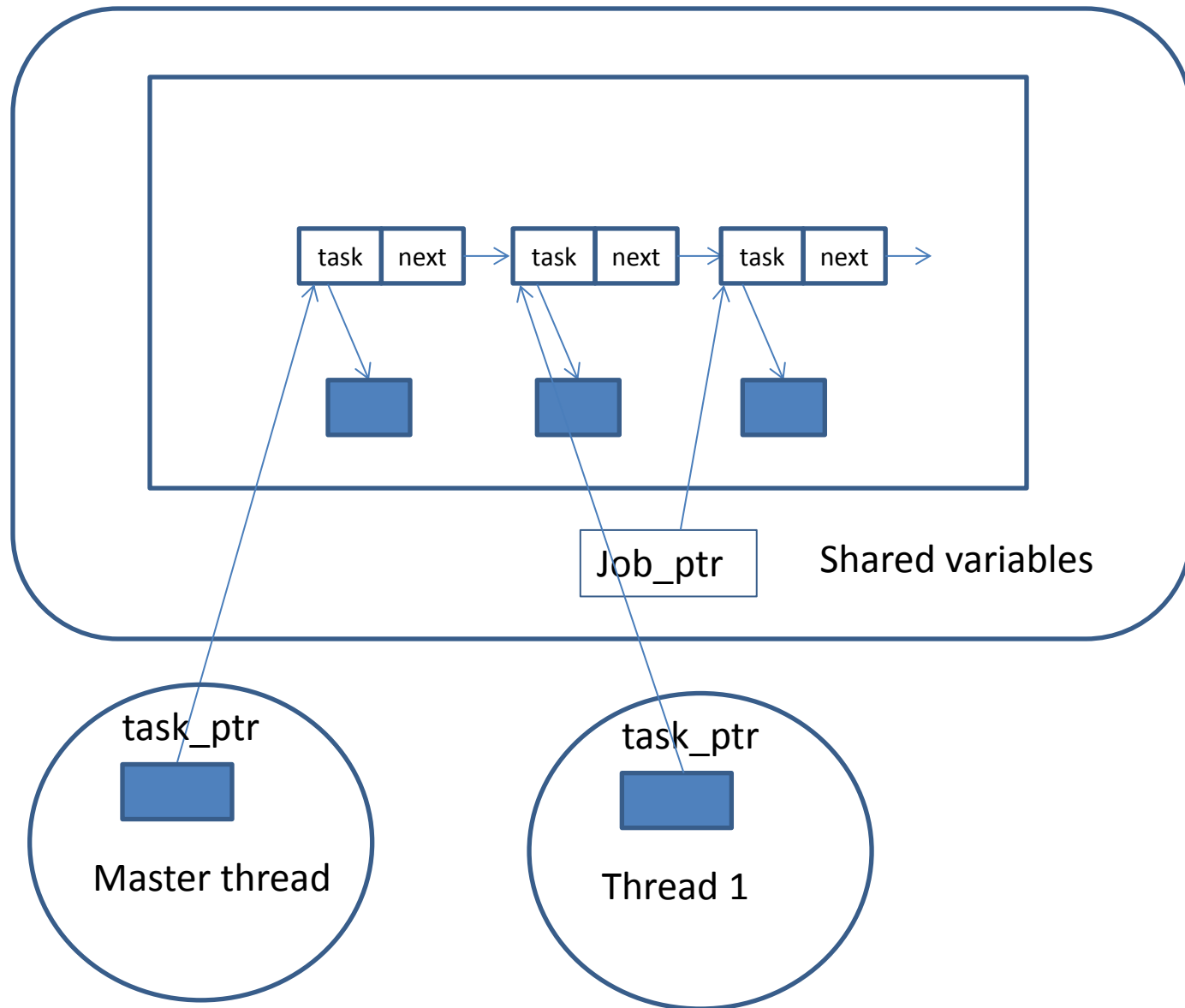
Code Fragment for Manager/Worker Model

```
int main(int argc, char argv[])
{
    struct job_struct job_ptr;
    struct task_struct *task_ptr;
    ...
    task_ptr = get_next_task(&job_ptr);
    while(task_ptr != NULL){
        complete_task(task_ptr);
        task_ptr = get_next_task(&job_ptr);
    }
    ...
}

struct task_struct *get_next_task(struct job_struct *job_ptr)
{
    struct task_struct *answer;
    if(job_ptr == NULL) answer = NULL;
    else
    {
        answer = job_ptr->task;
        job_ptr = job_ptr->next;
    }
    return answer;
}
```



- Two threads complete the work



```

int main(int argc, char argv[])
{
    struct job_struct job_ptr;
    struct task_struct *task_ptr;
    ...
#pragma omp parallel private (task_ptr)
    {
        task_ptr = get_next_task(&job_ptr);
        while(task_ptr != NULL){
            complete_task(task_ptr);
            task_ptr = get_next_task(&job_ptr);
        }
    }
    ...
}

struct task_struct *get_next_task(struct job_struct *job_ptr)
{
    struct task_struct *answer;
#pragma omp critical
    {
        if(job_ptr == NULL) answer = NULL;
        else
        {
            answer = job_ptr->task;
            job_ptr = job_ptr->next;
        }
    }
    return answer;
}

```

The execution of the code block after the parallel program is replicated among the threads

Ensure function `get_next_task()` executes atomically.

```

sum = 0;
#pragma omp parallel shared(n,a,sum) private(TID,sumLocal)
{
    TID = omp_get_thread_num();
    sumLocal = 0;
    #pragma omp for
        for (i=0; i<n; i++)
            sumLocal += a[i];
    #pragma omp critical (update_sum)
    {
        sum += sumLocal;
        printf("TID=%d: sumLocal=%d sum = %d\n",TID,sumLocal,sum);
    }
} /*-- End of parallel region --*/

```

```

{
...
#pragma omp parallel
{
    #pragma omp for nowait shared(best_cost)
    for(i=0; i<N; i++){
        int  my_cost;
        my_cost = estimate(i);
        #pragma omp critical
        {
            if(best_cost < my_cost)
                best_cost = my_cost;
        }
    }
}
}

```

Only one thread at a time executes if() statement. This ensures mutual exclusion when accessing shared data. Without critical, this will set up a **race condition**, in which the computation exhibits nondeterministic behavior when performed by multiple threads accessing a shared variable

Synchronization: atomic

- **atomic** provides mutual exclusion but only applies to the load/update of a memory location.
- This is a lightweight, special form of a critical section.
- It is applied only to the (single) assignment statement that immediately follows it.

```
{  
  ...  
  #pragma omp parallel  
  {  
    double tmp, B;  
    ....  
    #pragma omp atomic  
    {  
      X+=tmp;  
    }  
  }  
}
```

Atomic only protects the update of X.

```
int ic, i, n;  
ic = 0;  
#pragma omp parallel shared(n,ic) private(i)  
  for (i=0; i++, i<n)  
  {  
    #pragma omp atomic  
    ic = ic + 1;  
  }
```

“ic” is a counter. The atomic construct ensures that no updates are lost when multiple threads are updating a counter value.

- Atomic construct may only be used together with an expression statement with one of operations: +, *, -, /, &, ^, |, <<, >>.

```
int ic, i, n;  
ic = 0;  
#pragma omp parallel shared(n,ic) private(i)  
  for (i=0; i++, i<n)  
  {  
    #pragma omp atomic  
    ic = ic + bigfunc();  
  }
```

- The atomic construct does not prevent multiple threads from executing the function bigfunc() at the same time.

Synchronization: barrier

Suppose each of the following two loops are run in parallel over i , this may give a wrong answer.

```
for(i= 0; i<N; i++)  
    a[i] = b[i] + c[i];  
for(i= 0; i<N; i++)  
    d[i] = a[i] + b[i];
```

There could be a data race in $a[]$.

```
for(i= 0; i<N; i++)  
    a[i] = b[i] + c[i];
```

wait

```
for(i= 0; i<N; i++)  
    d[i] = a[i] + b[i];
```

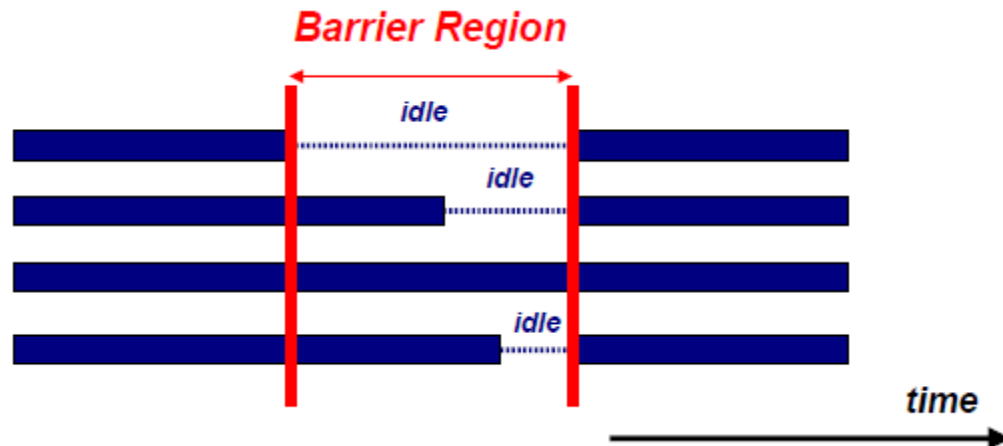
barrier

To avoid race condition:

- NEED: All threads wait at the barrier point and only continue when all threads have reached the barrier point.

Barrier syntax:

- `#pragma omp barrier`

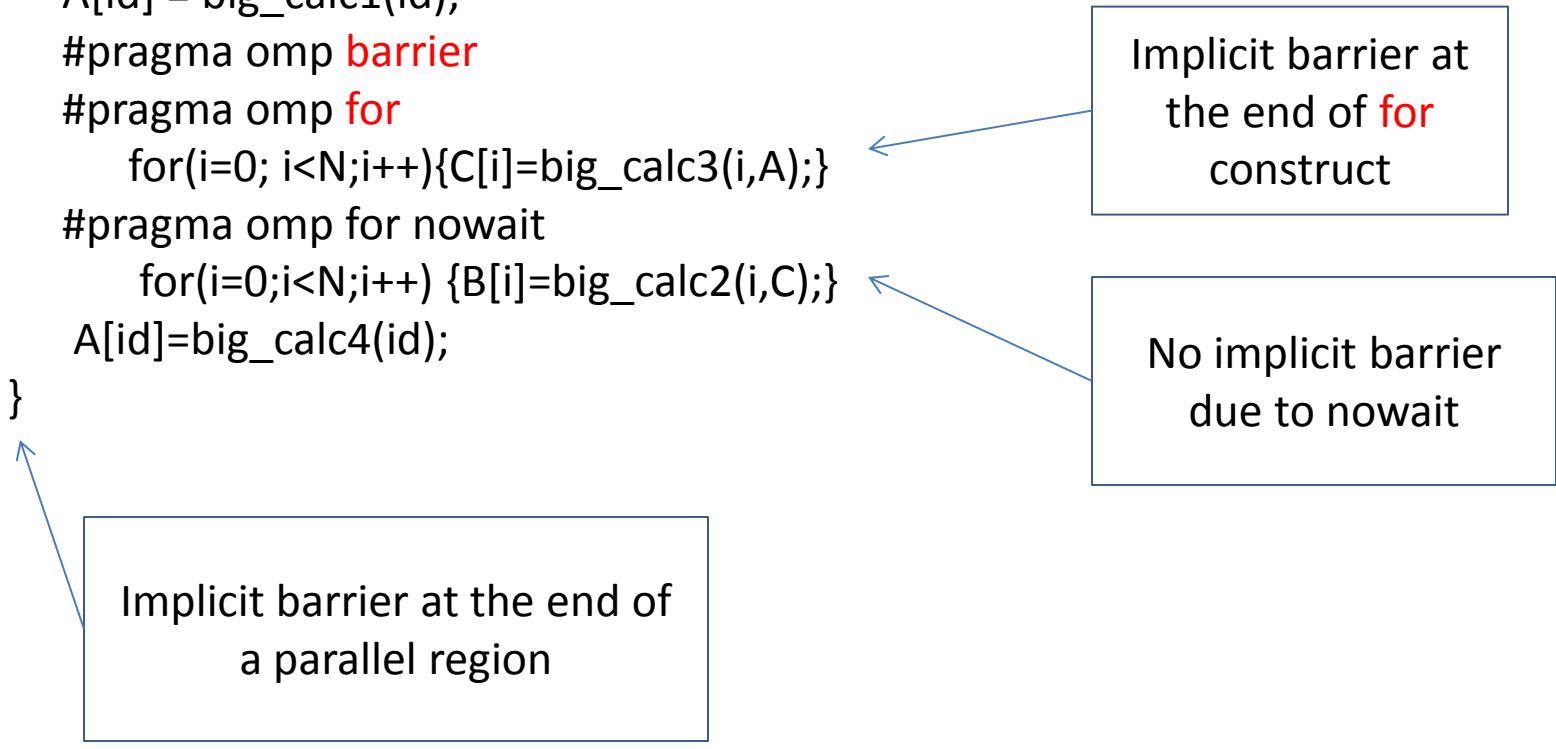


Synchronization: barrier

barrier: each threads waits until all threads arrive

```
#pragma omp parallel shared (A,B,C) private (id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
        for(i=0; i<N;i++){C[i]=big_calc3(i,A);}
    #pragma omp for nowait
        for(i=0;i<N;i++) {B[i]=big_calc2(i,C);}
    A[id]=big_calc4(id);
}
```

Implicit barrier at
the end of **for**
construct



No implicit barrier
due to nowait

Implicit barrier at the end of
a parallel region

When to Use Barriers

- If data is updated asynchronously and data integrity is at risk
- Examples:
 - Between parts in the code that read and write the same section of memory
 - After one timestep/iteration in a numerical solver
- Barriers are expensive and also may not scale to a large number of processors

“master” Construct

- The “master” construct defines a structured block that is only executed by the master thread.
- The other threads skip the “master” construct. No synchronization is implied.
- It does not have an implied barrier on entry or exit.
- The lack of a barrier may lead to problems.

```
#pragma omp parallel
{
    ...
    #pragma omp master
    {
        exchange_information();
    }
    #pragma omp barrier
    ...
}
```

```

#pragma omp parallel shared(a,b) private(i)
{
    #pragma omp master
    {
        a = 10;
        printf("Master construct is executed by thread %d\n",
               omp_get_thread_num());
    }

    #pragma omp barrier

    #pragma omp for
    for (i=0; i<n; i++)
        b[i] = a;

} /*-- End of parallel region --*/

printf("After the parallel region:\n");
for (i=0; i<n; i++)
    printf("b[%d] = %d\n",i,b[i]);

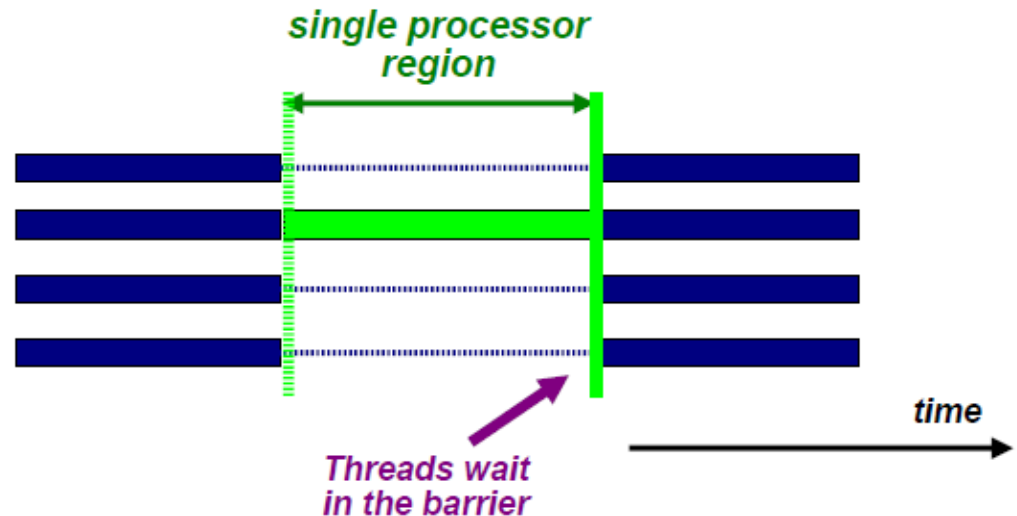
```

- Master construct to initialize the data

“single” Construct

- The “**single**” construct builds a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implicitly set at the end of the single block (the barrier can be removed by the **nowait** clause)

```
#pragma omp parallel
{
    ...
    #pragma omp single
    {
        exchange_information();
    }
    do_other_things();
    ...
}
```




```

#pragma omp parallel shared(a,b) private(i)
{
    #pragma omp single
    {
        a = 10;
        printf("Single construct executed by thread %d\n",
               omp_get_thread_num());
    }
    /* A barrier is automatically inserted here */

    #pragma omp for
    for (i=0; i<n; i++)
        b[i] = a;

} /*-- End of parallel region --*/

printf("After the parallel region:\n");
for (i=0; i<n; i++)
    printf("b[%d] = %d\n",i,b[i]);

```

- Single construct to initialize a shared variable

Synchronization: ordered

- The “ordered” region executes in the sequential order

```
#pragma omp parallel private (tmp)
{
    ...
    #pragma omp for ordered reduction(+:res)
    for(i=0;i<N;i++)
    {
        tmp = compute(i);
        #pragma ordered
        res += consum(tmp);
    }
    do_other_things();
    ...
}
```

Synchronization: Lock routines

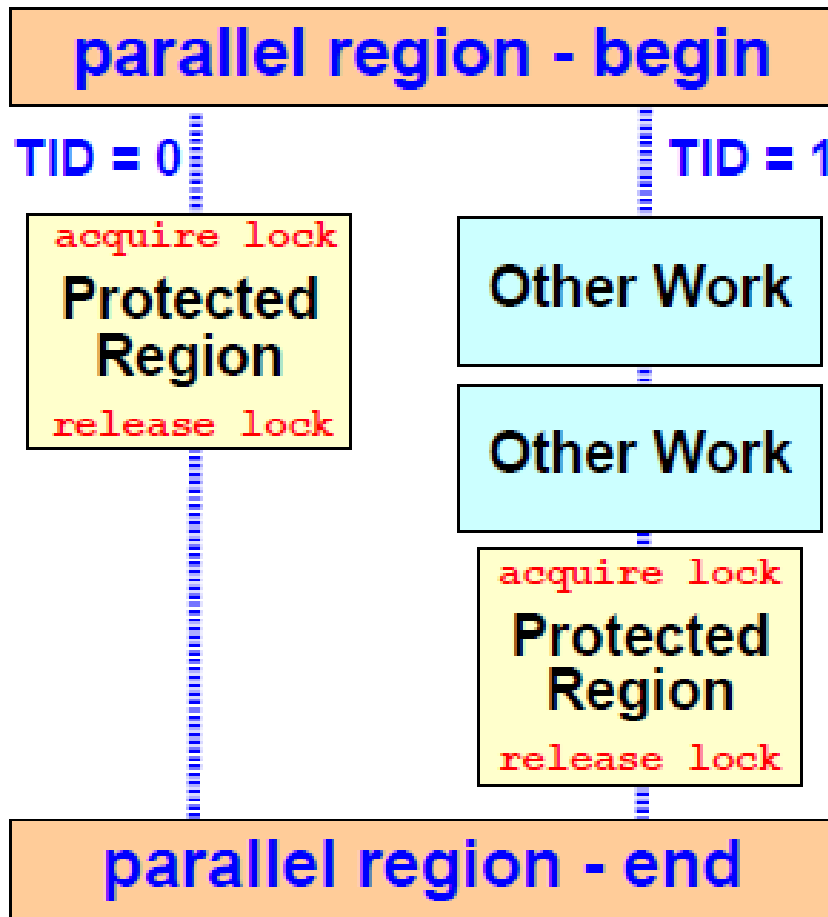
- A lock implies a memory fence of all thread visible variables.
- These routines are used to guarantee that only one thread accesses a variable at a time to avoid race conditions.
- C/C++ lock variables must have type “omp_lock_t” or “omp_nest_lock_t”.
- All lock functions require an argument that has a pointer to omp_lock_t or omp_nest_lock_t.
- Simple Lock routines:
 - `omp_init_lock(omp_lock_t*); omp_set_lock(omp_lock_t*);`
`omp_unset_lock(omp_lock_t*);`
`omp_test_lock(omp_lock_t*); omp_destroy_lock(omp_lock_t*);`

<http://gcc.gnu.org/onlinedocs/libgomp/index.html#Top>

General Procedure to Use Locks

1. Define the lock variables
2. Initialize the lock via a call to `omp_init_lock`
3. Set the lock using `omp_set_lock` or `omp_test_lock`.
The latter checks whether the lock is actually available before attempting to set it. It is useful to achieve asynchronous thread execution.
4. Unset a lock after the work is done via a call to `omp_unset_lock`.
5. Remove the lock association via a call to `omp_destroy_lock`.

Locking Example



- The protected region contains the update of a shared variable
- One thread acquires the lock and performs the update
- Meanwhile, other threads perform some other work
- When the lock is released again, the other threads perform the update

```
omp_lock_t lck;  
omp_init_lock(&lck);  
#pragma omp parallel shared(lck) private (tmp, id)  
{  
    id = omp_get_thread_num();  
    tmp = do_some_work(id);  
    omp_set_lock(&lck);  
    printf("%d %d\n", id, tmp);  
    omp_unset_lock(&lck);  
}  
omp_destroy_lock(&lck);
```

Initialize a lock associated with lock variables "lck" for use in subsequent calls.

Thread waits here for its turn.

Release the lock so that the next thread gets a turn

Dissociate the given lock variable from any locks.

Runtime Library Routines

- Routines for modifying/checking number of threads
 - `omp_set_num_threads(int n);`
 - `int omp_get_num_threads(void);`
 - `int omp_get_thread_num(void);`
 - `int omp_get_max_threads(void);`
- Test whether in active parallel region
 - `int omp_in_parallel(void);`
- Allow system to dynamically vary the number of threads from one parallel construct to another
 - `omp_set_dynamic(int set)`
 - `set = true`: enables dynamic adjustment of team sizes
 - `set = false`: disable dynamic adjustment
 - `int omp_get_dynamic(void)`
- Get number of processors in the system
 - `int omp_num_procs(void);` returns the number of processors online

<http://gcc.gnu.org/onlinedocs/libgomp/index.html#Top>

Default Data Storage Attributes

- A **shared** variable has a single storage location in memory for the whole duration of the parallel construct. All threads that reference such a variable accesses the same memory. Thus, reading/writing a shared variable provides an easy mechanism for communicating between threads.
 - In C/C++, by default, all program variables except the loop index become shared variables in a parallel region.
 - Global variables are shared among threads
 - C: File scope variables, static variables, dynamically allocated memory (by malloc(), or by new).
- A **private** variable has multiple storage locations, one within the execution context of each thread.
 - Not shared variables
 - Stack variables in functions called from parallel regions are private.
 - Automatic variables within a statement block are private.
 - This holds for pointer as well. Therefore, do not assign a private pointer the address of a private variable of another thread. The result is not defined.


```
/** main file **/  
#include <stdio.h>  
#include <stdlib.h>  
  
double A[100];  
int main(){  
    int index[50];  
    #pragma omp parallel  
        work(index);  
    printf("%d\n", index[0]);  
}
```

```
/** file 1 **/  
#include <stdio.h>  
#include <stdlib.h>  
  
extern double A[100];  
void work(int *index){  
    double temp[50];  
    static int count;  
}
```

- Variables “A”, “index” and “count” are shared by all threads.
- Variable “temp” is local (or private) to each thread.

Changing Data Storage Attributes

- Clauses for changing storage attributes
 - “shared”, “private”, “firstprivate”
- The final value of a private inside a parallel “for” loop can be transmitted to the shared variable outside the loop with:
 - “lastprivate”
- The default attributes can be overridden with:
 - Default(private|shared|none)
- All data clauses listed here apply to the parallel construct region and worksharing construct region except “shared”, which only applies to parallel constructs.

Private Clause

- “private (variable list)” clause creates a new local copy of variables for each thread.
 - Values of these variables are not initialized on entry of the parallel region.
 - Values of the data specified in the private clause can no longer be accessed after the corresponding region terminates (values are not defined on exit of the parallel region).

```
/** wrong implementation */  
int main(){  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
    for (int j=0; j<1000;j++)  
        tmp += j;  
    printf(“%d\n”, tmp);  
}
```

“tmp” is not initialized

“tmp” is 0 in version 3.0; unspecified in version 2.5.

Firstprivate Clause

- **firstprivate** initializes each private copy with the corresponding value from the master thread.

```
/** still wrong implementation */  
int main(){  
    int tmp = 0;  
    #pragma omp parallel for firstprivate(tmp)  
    for (int j=0; j<1000;j++){  
        tmp += j;  
        printf("%d\n", tmp);  
    }  
}
```

Each thread get its own
“tmp” with an initial
value of 0.

“tmp” is 0 in version 3.0; unspecified in
version 2.5.

Lastprivate Clause

- Lastprivate clause passes the value of a private variable from the last iteration to a global variable.
 - It is supported on the work-sharing loop and sections constructs.
 - It ensures that the last value of a data object listed is accessible after the corresponding construct has completed execution.
 - In case use with a work-shared loop, the object has the value from the iteration of the loop that would be last in a “sequential” execution.

```
/** useless implementation */  
int main(){  
    int tmp = 0;  
    #pragma omp parallel for firstprivate(tmp) lastprivate(tmp)  
    for (int j=0; j<5;j++)  
        tmp += j;  
    printf(“%d\n”, tmp);  
}
```

“tmp” is defined as its value at the “last sequential” iteration, i.e, $j = 5$.

Correct Usage of Lastprivate

```
/** correct usage of lastprivate */  
int main(){  
    int a, j;  
    #pragma omp parallel for private(j) lastprivate(a)  
    for (j=0; j<5;j++)  
    {  
        a = j + 2;  
        printf("Thread %d has a value of a = %d for j = %d\n",  
            omp_get_thread_num(), a, j);  
    }  
    printf("value of a after parallel = %d\n", a);  
}
```

Tread 0 has a value of a = 2 for j = 0
Tread 2 has a value of a = 4 for j = 2
Tread 1 has a value of a = 3 for j = 1
Tread 3 has a value of a = 5 for j = 3
Tread 4 has a value of a = 6 for j = 4
value of a after parallel = 6

Default Clause

- C/C++ only has default(shared) or default(none)
- Only Fortran supports default(private)
- Default data attribute is default(shared)
 - Exception: #pragma omp task
- Default(none): no default attribute for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice.

Lexical (static) and Dynamic Extent I

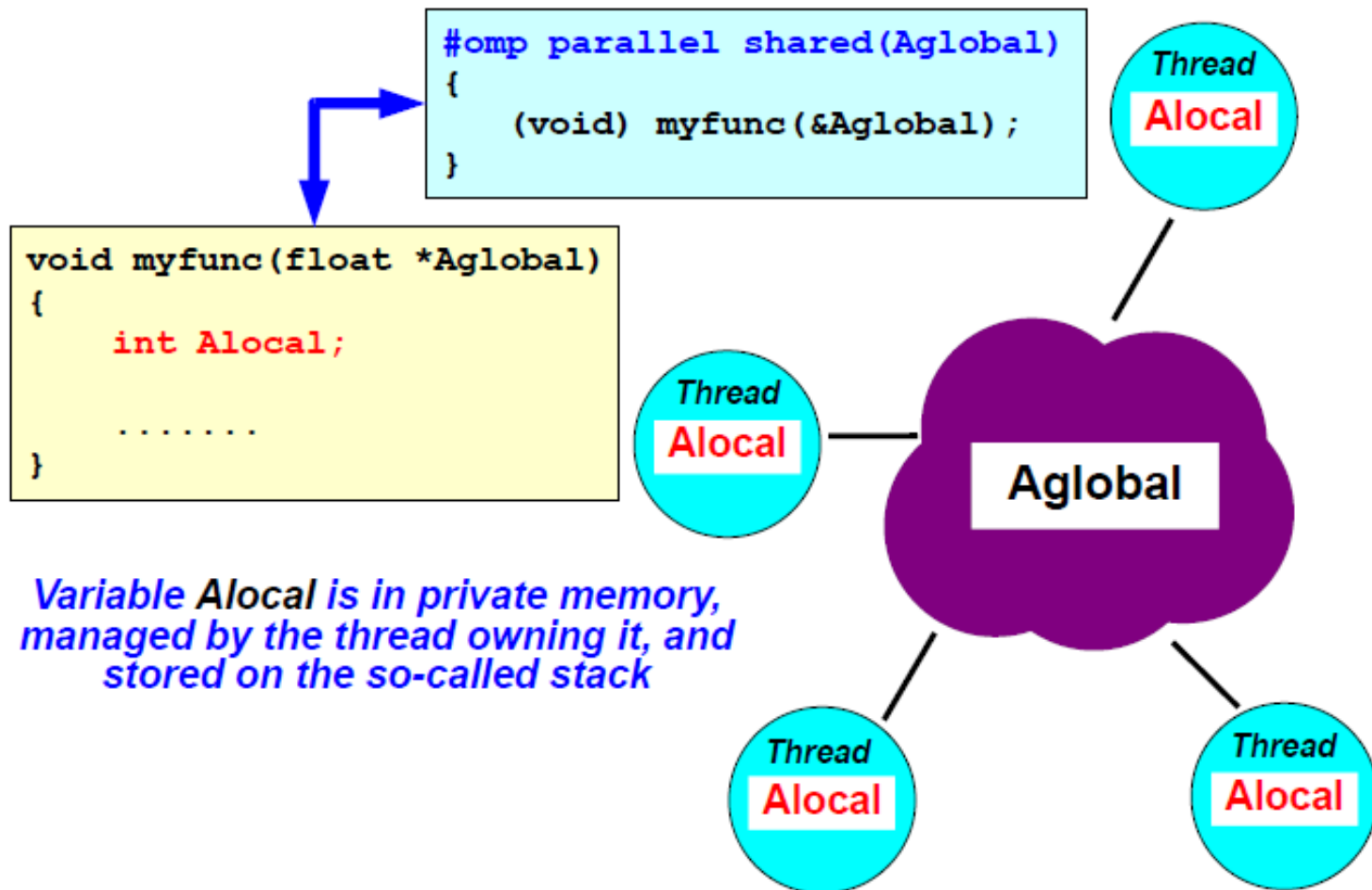
- Parallel regions enclose an arbitrary block of code, sometimes including calls to another function.
- The **lexical or static extent** of a parallel region is the block of code to which the parallel directive applies.
- The **dynamic extent** of a parallel region extends the lexical extent by the code of functions that are called (directly or indirectly) from within the parallel region.
- The dynamic extent is determined only at runtime.

Lexical and Dynamic Extent II

```
int main(){  
#pragma omp parallel  
{  
    print_thread_id();  
}  
}  
  
void print_thread_id()  
{  
    int id = omp_get_thread_num();  
    printf("Hello world from thread %d\n", id);  
}
```

Static extent

Dynamic
extent



```

void caller(int *a, int n) {
    int i,j,m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
    void callee(int *x, int *y, int
        z) {
        int ii;
        static int cnt;
        cnt++;
        for (ii=1; ii<z; ii++) {
            *x = *y + z;
        }
    }
}

```

Var	Scope	Comment
a	shared	Declared outside parallel construct
n	shared	same
i	private	Parallel loop index
j	shared	Sequential loop index
m	shared	Declared outside parallel construct
k	private	Automatic variable/parallel region
x	private	Passed by value
*x	shared	(actually a)
y	private	Passed by value
*y	private	(actually k)
z	private	(actually j)
ii	private	Local stack variable in called function
cnt	shared	Declared static (like global)

Threadprivate

- Threadprivate makes global data private to a thread
 - C/C++: file scope and static variables, static class members
 - Each thread gives its own set of global variables, with initial values undefined.
- Different from private
 - With private clause, global variables are masked.
 - Threadprivate preserves global scope within each thread.
 - Parallel regions must be executed by the same number of threads for global data to persist.
- Threadprivate variables can be initialized using copyin clause or at time of definition.

If all of the conditions below hold, and if a threadprivate object is referenced in two consecutive (at run time) parallel regions, then threads with the same thread number in their respective regions reference the same copy of that variable:

- Neither parallel region is nested inside another parallel region.
- The number of threads used to execute both parallel regions is the same.

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

```
int *pglobal;
#pragma omp threadprivate(pglobal)
```

Threadprivate directive is used to give each thread a private copy of the global pointer pglobal.

```
int main(){
    ...
#pragma omp parallel for private(i,j,sum,TID) shared(n,length,check)
    for (i=0; i<n;i++)
    {
        TID = omp_get_thread_num();
        if((pglobal = (int*) malloc(length[i]*sizeof(int))) != NULL) {
            for(j=sum=0; j < length[i];j++) pglobal[j] = j+1;
            sum = calculate_sum(length[i]);
            printf("TID %d: value of sum for I = %d is %d\n", TID,i,sum);
            free(pglobal);
        } else {
            printf("TID %d: not enough memory : length[%d] = %d\n", TID,i,length[i]);
        }
    }
}
```

```
/* source of function calculate_sum() */  
extern int *pglobal;  
  
int calculate_sum(int length){  
    int sum = 0;  
    for (j=0; j<length;j++)  
    {  
        sum += pglobal[j];  
    }  
    return (sum);  
}
```

```

#include <omp.h>
static int sum0=0;
#pragma omp threadprivate (sum0)
int main()
{ int sum = 0;
  int i ;
  . . .
  for ( . . . )
#pragma omp parallel
  {
    sum0 = 0;
    #pragma omp for
      for ( i = 0; i <= 1000; i++)
        sum0 = sum0 + . . .
    #pragma omp critical
      sum = sum + sum0 ;
  } /* end of parallel region */

```

- Each thread has its own copy of sum0, updated in a parallel region that is called several times. The values for sum0 from one execution of the parallel region will be available when it is next started.

Copyin Clause

- Copyin allows to copy the master thread's threadprivate variables to corresponding threadprivate variables of the other threads.

```
int global[100];
#pragma omp threadprivate(global)

int main(){
    for(int i= 0; i<100; i++) global[i] = i+2; // initialize data
    #pragma omp parallel copyin(global)
    {
        /// parallel region, each thread gets a copy of global, with initialized value
    }
}
```

Copyprivate Clause

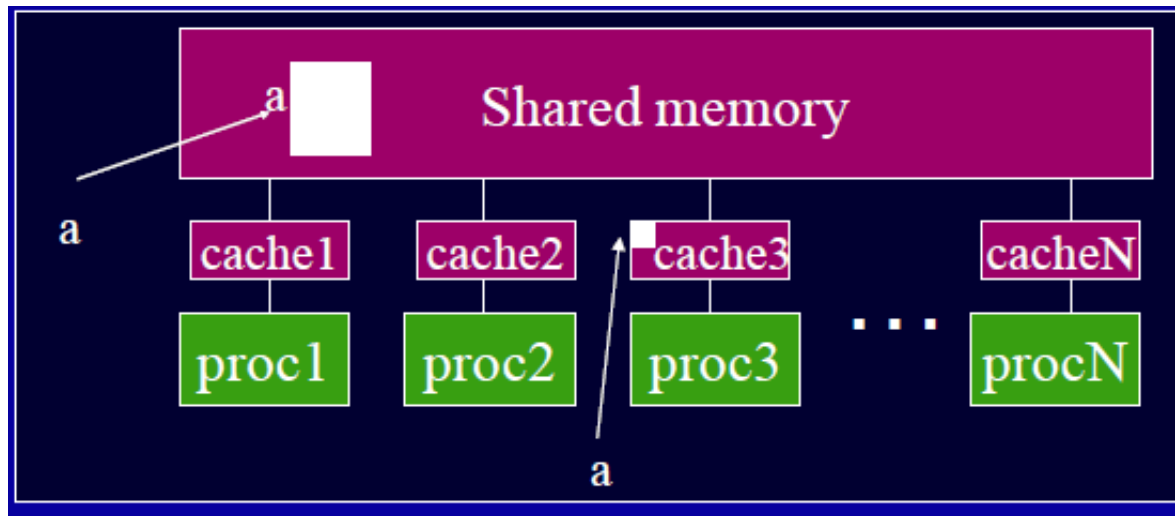
- Copyprivate clause is supported on the single directive to broadcast values of privates from one thread of a team to the other threads in the team.
 - The typical usage is to have one thread read or initialize private data that is subsequently used by the other threads as well.
 - After the single construct has ended, but before the threads have left the associated barrier, the values of variables specified in the associated list are copied to the other threads.
 - Do not use copyprivate in combination with the nowait clause.

```
#include "omp.h"
Void input_parameters(int, int); // fetch values of input parameters

int main(){
    int Nsize, choice;
    #pragma omp parallel private(Nsize, choice)
    {
        #pragma omp single copyprivate (Nsize, choice)
        input_parameters(Nsize, choice);
        do_work(Nsize, choice);
    }
}
```

Flush Directive

- OpenMP supports a shared memory model.
 - However, processors can have their own “local” high speed memory, the registers and cache.
 - If a thread updates shared data, the new value will first be saved in register and then stored back to the local cache.
 - The update are thus not necessarily immediately visible to other threads.



Flush Directive

The flush directive is to make a thread's temporary view of shared data consistent with the value in memory.

- `#pragma omp flush (list)`
- Thread-visible variables are written back to memory at this point.
- For pointers in the list, note that the pointer itself is flushed, not the object it points to.

Why Task Parallelism?

```
#include "omp.h"
/* traverse elements in the list */

Void traverse_list(List *L){
    Element *e;
    #pragma omp parallel private(e)
    {
        for(e = L->first; e != NULL; e = e->next)
            #pragma omp single nowait
            do_work(e);
    }
}
```

- Poor performance

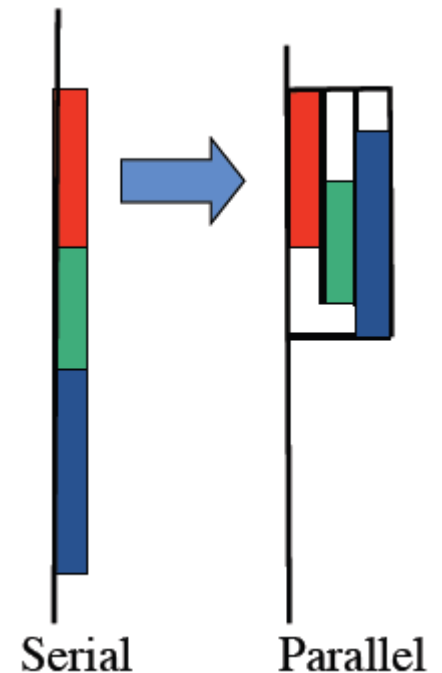
- Improved performance by sections
- Too many parallel regions
 - Extra synchronization
 - Not flexible

```
#include "omp.h"
/* traverse elements in the list */

Void traverse_tree(Tree *T){
    #pragma omp parallel sections
    {
        #pragma omp section
        if(T->left)
            traverse_tree(T->left);
        #pragma omp section
        if(T->right)
            traverse_tree(T->right);
    }
    process(T);
}
```

OpenMP 3.0 and Tasks

- What are tasks?
 - Tasks are independent units of work
 - Threads are assigned to perform the work of each task.
 - Tasks may be deferred
 - Tasks may be executed immediately
 - The runtime system decides which of the above
- Why task?
 - The basic idea is to set up a task queue: when a thread encounters a task directive, it arranges for some thread to execute the associated block – at some time. The first thread can continue.



OpenMP 3.0 and Tasks

Tasks allow to parallelize irregular problems

- Unbounded loops
- Recursive algorithms
- Manager/work schemes
- ...

A task has

- **Code** to execute
- **Data** environment (It owns its data)
- **Internal control variables**
- An assigned thread that executes the code and the data

Two activities: packaging and execution

- Each encountering thread packages a new instance of a task (code and data)
- Some thread in the team executes the task at some later time

- OpenMP has always had tasks, but they were not called “task”.
 - A thread encountering a parallel construct, e.g., “for”, packages up a set of implicit tasks, one per thread.
 - A team of threads is created.
 - Each thread is assigned to one of the tasks.
 - Barrier holds master thread till all implicit tasks are finished.
- OpenMP 3.0 adds a way to create a task explicitly for the team to execute.

Task Directive

```
#pragma omp task [clauses]
    if( logical expression)
    untied
    shared (list)
    private (list)
    firstprivate (list)
    default(shared | none)

structured block
```

- Each encountering thread creates a task
 - Package code and data environment
 - Can be nested
 - Inside parallel regions
 - Inside other tasks
 - Inside worksharing
- An OpenMP barrier (implicit or explicit):
All tasks created by any thread of the current team are guaranteed to be completed at barrier exit.
- Task barrier (taskwait):
Encountering thread suspends until all child tasks it has generated are complete.

Fibonacci series:

$f(1) = 1$

$f(2) = 1$

$f(n) = f(n-1) + f(n-2)$

```
/* serial code to compute Fibonacci */
```

```
int fib(int n)
```

```
{
```

```
    int i, j;
```

```
    if(n < 2) return n;
```

```
    i = fib(n-1);
```

```
    j = fib(n-2);
```

```
    return (i+j);
```

```
}
```

```
int main(){
```

```
    int n = 8;
```

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

```
}
```

```
/* OpenMP code to compute Fibonacci */
```

```
#include <stdlib.h>
```

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

```
#include "omp.h"
```

```
static int fib(int);
```

```
int main(){
```

```
    int nthreads, tid;
```

```
    int n = 8;
```

```
    #pragma omp parallel num_threads(4) private(tid)
```

```
{
```

```
    #pragma omp single
```

```
{
```

```
    tid = omp_get_thread_num();
```

```
    printf("Hello world from (%d)\n", tid);
```

```
    printf("Fib(%d) = %d by %d\n", n, fib(n), tid);
```

```
}
```

```
} // all threads join master thread and terminates
```

```
}
```

```
Static int fib(int n){
```

```
    int i, j, id;
```

```
    if(n < 2)
```

```
        return n;
```

```
    #pragma omp task shared (i) private (id)
```

```
{
```

```
    i = fib(n-1);
```

```
}
```

```
    #pragma omp task shared (j) private (id)
```

```
{
```

```
    j = fib(n-2);
```


```
}
```

```
    return (i+j);
```

```
}
```

```
/* Example of pointer chasing using task*/  
Void process_list(elem_t *elem){  
    #pragma omp parallel  
    {  
        #pragma omp single  
        {  
            while (ele != NULL) {  
                #pragma omp task  
                {  
                    process(elem);  
                }  
                elem = elem->next;  
            }  
        }  
    }  
}
```

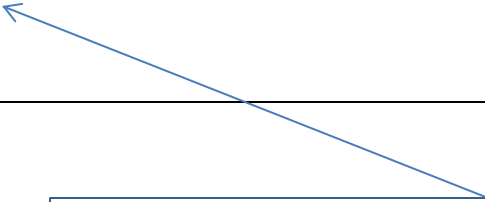
Elem is firstprivate by default



```
#include "omp.h"
/* traverse elements in the list */

Void traverse_list(List *L){
    Element *e;

    for(e = L->first; e != NULL; e = e->next)
        #pragma omp task
        do_work(e);
    #pragma omp taskwait
}
```



All tasks guaranteed to be completed here

```
/* Tree traverse using tasks*/

struct node{
    struct node *left, *right;
};

void traverse(struct node *p, int postorder){
    if(p->left != NULL)
        #pragma omp task
        traverse(p->left, postorder);
    if(p->right != NULL)
        #pragma omp task
        traverse(p->right, postorder);
    if(postorder){
        #pragma omp taskwait
    }
    process(p);
}
```

Task Data Scope

Data Scope Clauses

- shared (list)
- private (list)
- firstprivate (list)
- default (shared | none)

If no clause:

- Implicit rules apply: global variables are shared

Otherwise

- Firstprivate
- Shared attribute is lexically inherited

```
int a;  
void foo(){  
    int b, c;  
    #pragma omp parallel shared (c)  
    {  
        int d;  
        # pragma omp task  
        {  
            int e;  
            /*  
            a = shared  
            b = firstprivate  
            c = shared  
            d = firstprivate  
            e = private  
            */  
        }  
    }  
}
```


Task Synchronization

Barriers (implicit or explicit)

- All tasks created by any thread of the current team are guaranteed to be completed at barrier exit

Task Barrier

`#pragma omp taskwait`

- Encountering task suspends until child tasks complete

Task Execution Model

- Tasks are executed by a thread of the team
 - Can be executed immediately by the same thread that creates it
- Parallel regions in 3.0 create tasks
 - One implicit task is created for each thread
- Threads can suspend the execution of a task and start/resume another

```
#include "omp.h"
/* traverse elements in the list */
List *L;
...
#pragma omp parallel
    traverse_list(L);
```

Multiple traversals of
the same list

```
#include "omp.h"
/* traverse elements in the list */
List *L;
...
#pragma omp parallel
#pragma omp single
    traverse_list(L);
```

Single traversal:

- One thread enters single and creates all tasks
- All the team cooperates executing them

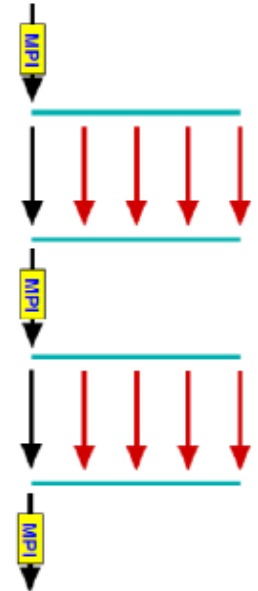
```
#include "omp.h"
/* traverse elements in the list */
List L[N];
...
#pragma omp parallel for
For (i = 0; i < N; i++)
    traverse_list(L[i]);
```

Multiple traversals:

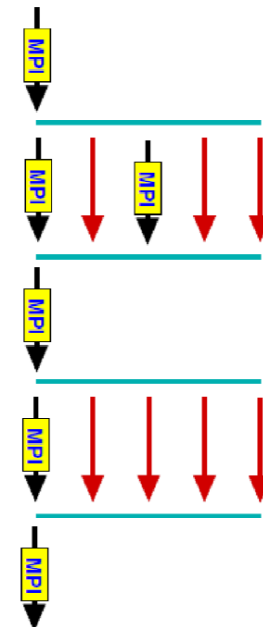
- Multiple threads create tasks
- All the team cooperates executing them

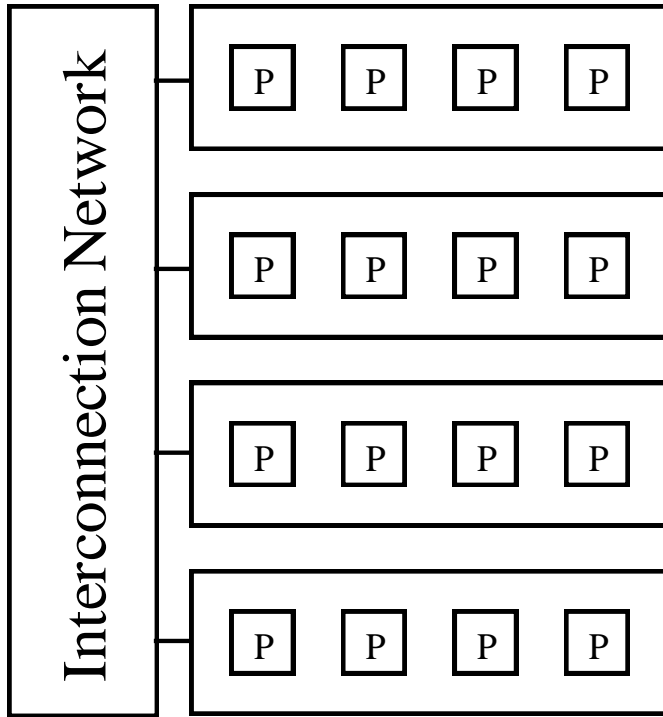
Hybrid MPI/OpenMP

- **Vector mode:** MPI is called only outside OpenMP parallel regions.

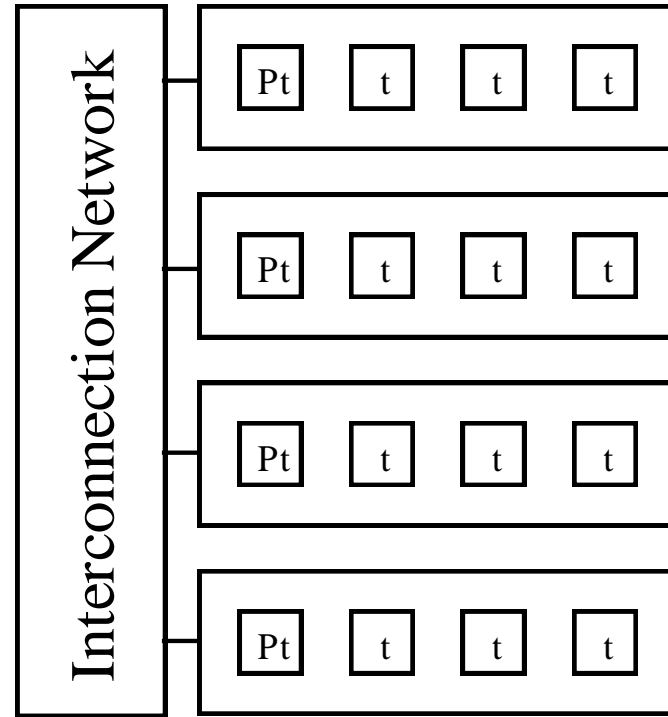


- **Task mode:** One or more threads in the parallel region are dedicated to special tasks, like doing communication in the background.





C+MPI



C+MPI+OpenMP

Basic Hybrid Framework

```
#include <omp.h>
#include "mpi.h"

#define _NUM_THREADS 4

/* Each MPI process spawns a distinct OpenMP
 * master thread; so limit the number of MPI
 * processes to one per node
 */

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

    /* set number of threads to spawn */
    omp_set_num_threads(_NUM_THREADS);

    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

    /* the following is a parallel OpenMP
     * executed by each MPI process
     */
    int c;
    #pragma omp parallel reduction(+:c)
    {
        c = omp_get_num_threads();
    }

    /* expect a number to get printed for each MPI process */
    printf("%d\n",c);
    /* finalize MPI */
    MPI_Finalize();
    return 0;
}
```

Compileing: mpicc -fopenmp test.cc

Concept 1: ROOT MPI Process Controls Communication

- Map one MPI process to one SMP node.
- Each MPI process fork a fixed number of threads.
- Communication among MPI process is handled by main MPI process only.

```
...  
#pragma omp master  
{  
    if(0== my_rank)  
        // some MPI call as root process  
    else  
        // some MPI call as non-root process  
} // end of omp master
```



```

#include <omp.h>
#include "mpi.h"

#define _NUM_THREADS 4

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

    /* set number of threads to spawn */
    omp_set_num_threads(_NUM_THREADS);

    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

    /* the following is a parallel OpenMP
       * executed by each MPI process
       */
    #pragma omp parallel
    {
        #pragma omp master
        {
            if ( 0 == my_rank)
                // some MPI_ call as ROOT process
            else
                // some MPI_ call as non-ROOT process
            }
        }

    /* expect a number to get printed for each MPI process */
    printf("%d\n",c);
    /* finalize MPI */
    MPI_Finalize();
    return 0;
}

```

Concept 2: Master OpenMP Thread Controls Communication

- Each MPI process uses its own OpenMP master thread to communicate.
- Need to take more care to ensure efficient communications.

```
...  
#pragma omp master  
{  
    some MPI call as an MPI process  
} // end of omp master
```

```

#include <omp.h>
#include "mpi.h"

#define _NUM_THREADS 4

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

    /* set number of threads to spawn */
    omp_set_num_threads(_NUM_THREADS);

    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

    /* the following is a parallel OpenMP
       * executed by each MPI process
       */
    #pragma omp parallel
    {
        #pragma omp master
        {
            // some MPI_ call as an MPI process
        }
    }

    /* expect a number to get printed for each MPI process */
    printf("%d\n",c);
    /* finalize MPI */
    MPI_Finalize();
    return 0;
}

```

Concept 3: All OpenMP Threads May Use MPI Calls

- This is by far the most flexible communication scheme.
- Great care must be taken to account for explicitly which thread of which MPI process communicates.
- Requires an addressing scheme that denotes which MPI process participates in communication and which thread of MPI process is involved, e.g., <my_rank, omp_thread_id>.
- Neither MPI nor OpenMP have built-in facilities for tracking communication.
- Critical sections may be used for some level of control.

```
...  
#pragma omp critical  
{  
    some MPI call as an MPI process  
} // end of omp critical
```

```

#include <omp.h>
#include "mpi.h"

#define _NUM_THREADS 4

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

    /* set number of threads to spawn */
    omp_set_num_threads (_NUM_THREADS);

    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

    /* the following is a parallel OpenMP
       * executed by each MPI process
       */
    #pragma omp parallel
    {
        #pragma omp critical /* not required */
        {
            // some MPI_ call as an MPI process
        }
    }

    /* expect a number to get printed for each MPI process */
    printf("%d\n",c);
    /* finalize MPI */
    MPI_Finalize();
    return 0;
}

```

Conjugate Gradient

- Algorithm
 - Start with MPI program
 - MPI_Send/Recv for communication
 - OpenMP “for” directive for matrix-vector multiplication

Init.: $x(0) = 0$, $d(0) = 0$, $g(0) = -b$;

Step 1. Compute the gradient: $g(t) = Ax(t-1) - b$

Step 2. Compute the direction vector:

$$d(t) = -g(t) + (g(t)^T g(t)) / (g(t-1)^T g(t-1)) d(t-1)$$

Step 3. Compute the step size:

$$s(t) = -(d(t)^T d(t)) / (d(t)^T A d(t));$$

Step 4. Compute the new approximation of x :

$$x(t) = x(t-1) + s(t) d(t).$$

```

#include <stdlib.h>
#include <stdio.h>
#include "MyMPI.h"
int main(int argc, char *argv[]){
    double **a, *astorage, *b, *x;
    int p, id, m, n, nl;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    read_block_row_matrix(id,p,argv[1],(void*)&a,(void*)&astorage,MPI_DOUBLE,&m,&n);
    nl = read_replicated_vector(id,p,argv[2],(void**)&b,MPI_DOUBLE);
    if((m!=n) || (n != nl)) {
        printf("Incompatible dimensions %d %d time %d\n", m,n,nl);
    }
    else{
        x = (double*)malloc(n*sizeof(double));
        cg(p,id,a,b,x,n);
        print_replicated_vector(id,p,x,MPI_DOUBLE,n);
    }
    MPI_Finalize();
}

```

```

#define EPSILON 1.0e-10
Double *piece;
cg(int p, int id, double **a, double *b, double *x, int n){
    int i, it;
    double *d, *g, denom1, denom2, num1, num2, s, *tmpvec;
    d = (double*)malloc(n*sizeof(double));
    g = (double*)malloc(n*sizeof(double));
    tmpvec = (double*)malloc(n*sizeof(double));
    piece = (double*)malloc(BLOCK_SIZE(id,p,n)*sizeof(double));
    for(i=0; i<n; i++){
        d[i] = x[i] = 0.0;
        g[i] = -b[i];
    }
    for(it=0; it<n; it++){
        denom1 = dot_product(g,g,n);
        matrix_vector_product(id,p,n,a,x,g);
        for(i=0;i<n;i++) g[i]-=b[i];
        num1 = dot_product(g,g,n);
        if(num1<EPSILON) break;
        for(i=0;i<n;i++) d[i]=-g[i]+(num1/denom1)*d[i];
        num2 = dot_product(d,g,n);
        matrix_vector_product(id,p,n,a,d,tmpvec);
        denom2=dot_product(d,tmpvec,n);
        s=-num2/denom2;
        for(i=0;i<n;i++) x[i] += s*d[i];
    }
}

```



```

double dot_product(double *a, double *b, int n)
{
    int i;
    double answer=0.0;
    for(i=0; i<n;i++)
        answer+=a[i]*b[i];
    return answer;
}

double matrix_vector_product(int id, int p, int n, double **a, double *b, double *c){
    int i, j;
    double tmp;
    #pragma omp parallel for private (i,j,tmp)
    for(i=0; i<BLOCK_SIZE(id,p,n);i++){
        tmp=0.0;
        for(j=0;j<n;j++)
            tmp+=a[i][j]*b[j];
        piece[i] = tmp;
    }
    new_replicate_block_vector(id,p,piece,n, c, MPI_DOUBLE);
}

void new_replicate_block_vector(int id, int p, double *piece, int n, double *c, MPI_Datatype dtype)
{
    int *cnt, *disp;
    create_mixed_xfer_arrays(id,p,n,&cnt,&disp);
    MPI_Allgather(piece,cnt[id], dtype, c, cnt, disp, dtype, MPI_COMM_WORLD);
}

```

Steady-State Heat Distribution

Solve $u_{xx} + u_{yy} = f(x, y)$, $0 \leq x \leq a, 0 \leq y \leq b$

With $u(x, 0) = G_1(x), u(x, b) = G_2(x)$, $0 \leq x \leq a$

$u(0, y) = G_3(y), u(a, y) = G_4(y)$, $0 \leq y \leq b$

- Use row-decomposition.

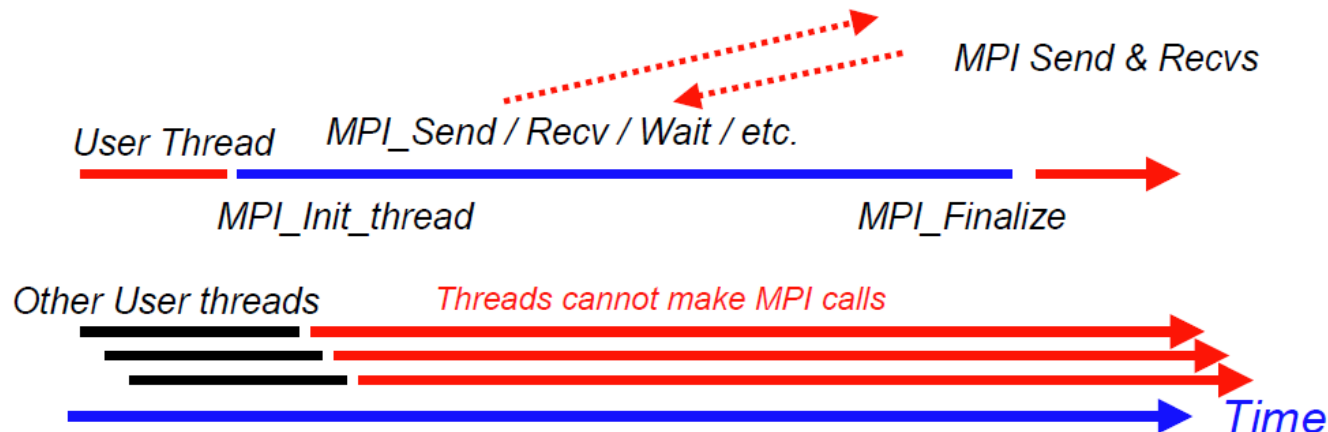
```

int find_steady_state(int p, int id, int my_rows, double **u, double **w)
{
    double diff, global_diff, tdiff; int its;
    MPI_Status status; int i,j;
    its = 0;
    for(;;) {
        if(id>0) MPI_Send(u[1], N, MPI_DOUBLE, id-1,0,MPI_COMM_WORLD);
        if(id < p-1) {
            MPI_Send(u[my_rows-2],N,MPI_DOUBLE,id+1,0,MPI_COMM_WORLD);
            MPI_Recv(u[my_rows-1],N,MPI_DOUBLE,id+1,0,MPI_COMM_WORLD,&status);
        }
        if(id>0) MPI_Recv(u[0],N,MPI_DOUBLE,id-1,0,MPI_COMM_WORLD,&status);
        diff = 0.0;
#pragma omp parallel private (l,j,tdiff)
        {
            tdiff = 0.0;
            #pragma omp for
            for(i=1;i<my_rows-1;i++)
                for(j=1;j<N-1;j++){
                    w[i][j]=(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1])/4.0;
                    if(fabs(w[i][j]-u[i][j]) >tdiff) tdiff = fabs(w[i][j]-u[i][j]);
                }
            #pragma omp for nowait
            for(i=1;i<my_rows-1;i++)
                for(j=1;j<N-1;j++)
                    u[i][j] = w[i][j];
            #pragma omp critical
            if(tdiff > diff) diff = tdiff;
        }
        MPI_Allreduce(&diff,&global_diff,1,MPI_DOUBLE,MPI_MAX,MPI_COMM_WORLD);
        if(global_diff <= EPSILON) break;
        its++;
    }
}

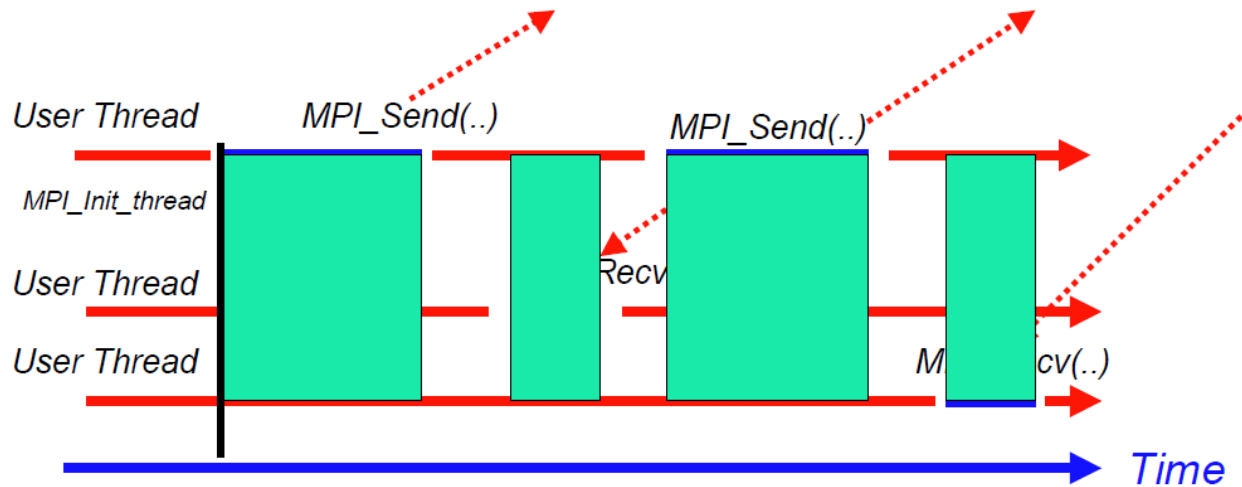
```

OpenMP multithreading in MPI

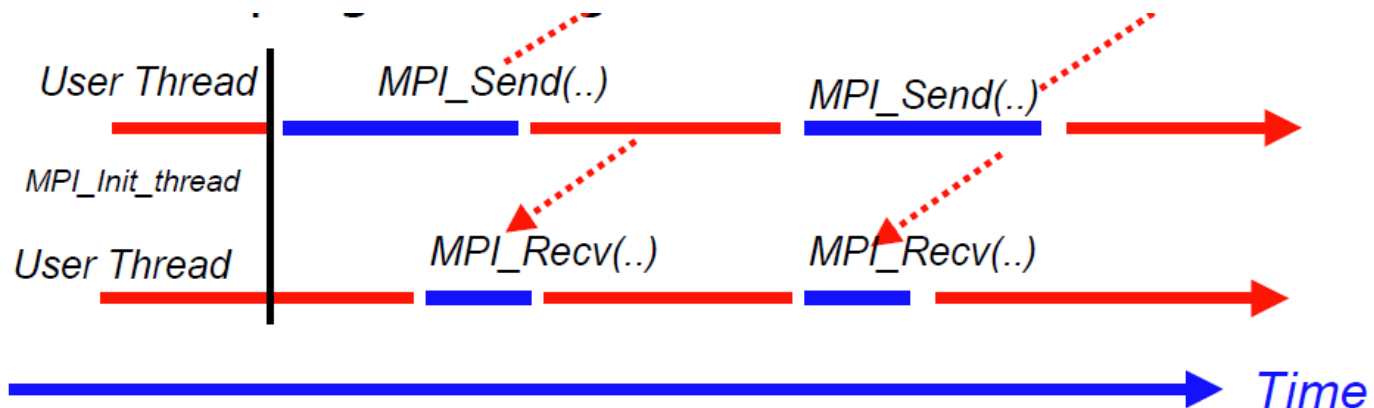
- MPI-2 specification
 - Does not mandate thread support
 - Does define what a “thread compliant MPI” should do
 - 4 levels of thread support
 - MPI_THREAD_SINGLE: There is no OpenMP multithreading in the program.
 - MPI_THREAD_FUNNELED: All of the MPI calls are made by the master thread.
 - This will happen if all MPI calls are outside OpenMP parallel regions or are in master regions.
 - A thread can determine whether it is the master thread by calling `MPI_Is_thread_main`



- **MPI_THREAD_SERIALIZED:** Multiple threads make MPI calls, but only one at a time.



- **MPI_THREAD_MULTIPLE:** Any thread may make MPI calls at any time.



- Threaded MPI Initialization

Instead of starting MPI by MPI_Init,

```
int MPI_Init_thread(int *argc, char ***argv, int  
required, int *provided)
```

required: the desired level of thread support.

provided: the actual level of thread support provided by the system.

Thread support at levels MPI_THREAD_FUNNELED or higher allows potential overlap of communication and computation.

<http://www.mpi-forum.org/docs/mpi-20-html/node165.htm>

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "omp.h"

int main(int argc, char *argv[])
{
    int rank, omp_rank, mpisupport;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &mpisupport);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    #pragma omp parallel private(omp_rank)
    {
        omp_rank = omp_get_thread_num();
        printf("Hello. This is process %d, thread %d\n",
            rank, omp_rank);
    }
    MPI_Finalize();
}
```

References:

- <http://bisqwit.iki.fi/story/howto/openmp/>
- <http://openmp.org/mp-documents/omp-hands-on-SC08.pdf>
- <https://computing.llnl.gov/tutorials/openMP/>
- <http://www.mosaic.ethz.ch/education/Lectures/hpc>
- R. van der Pas. An Overview of OpenMP
- B. Chapman, G. Jost and R. van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, Cambridge, Massachusetts, London, England
- B. Estrade, Hybrid Programming with MPI and OpenMP