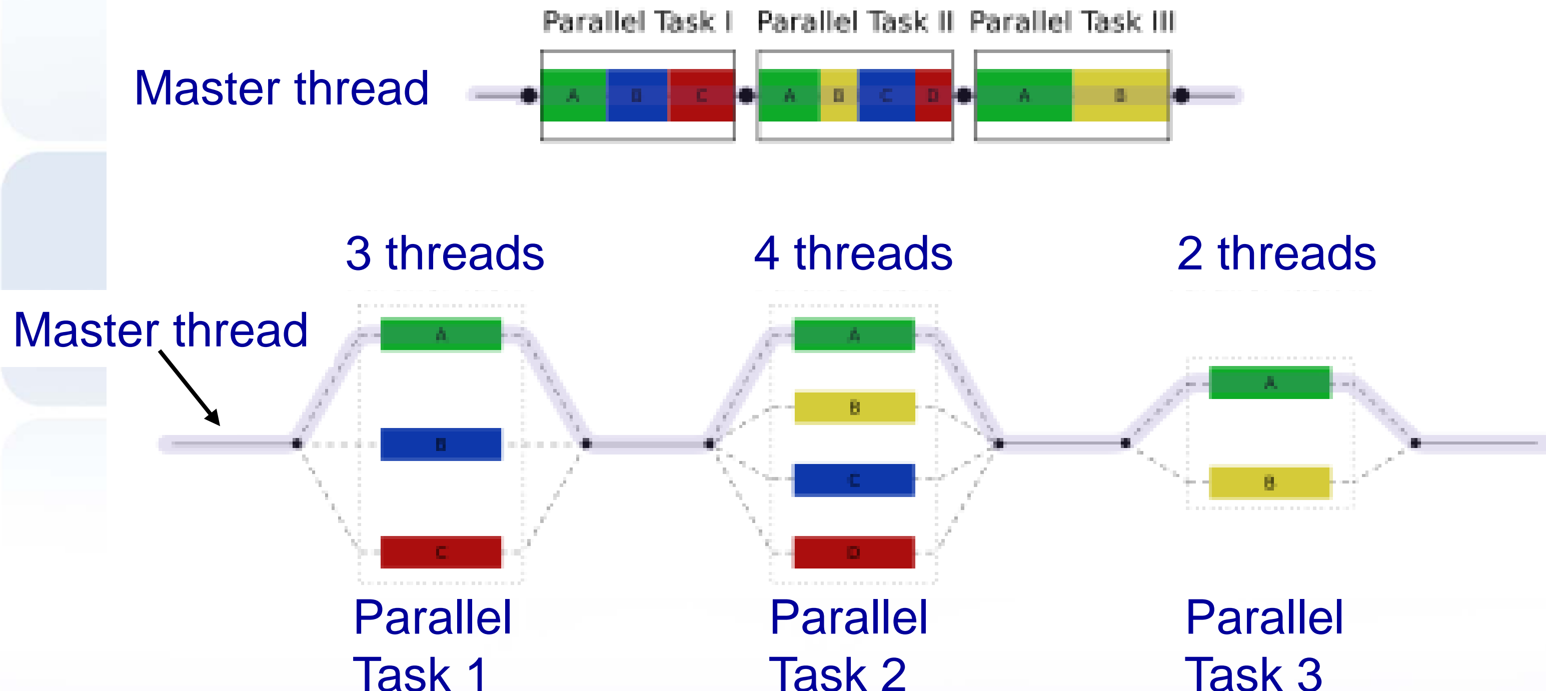
U. de Valparaíso, Escuela Ingeniería Industrial
U. de Chile, Centro de Modelamiento Matemático



Parallel Programming: OpenMP

Gonzalo Hernández

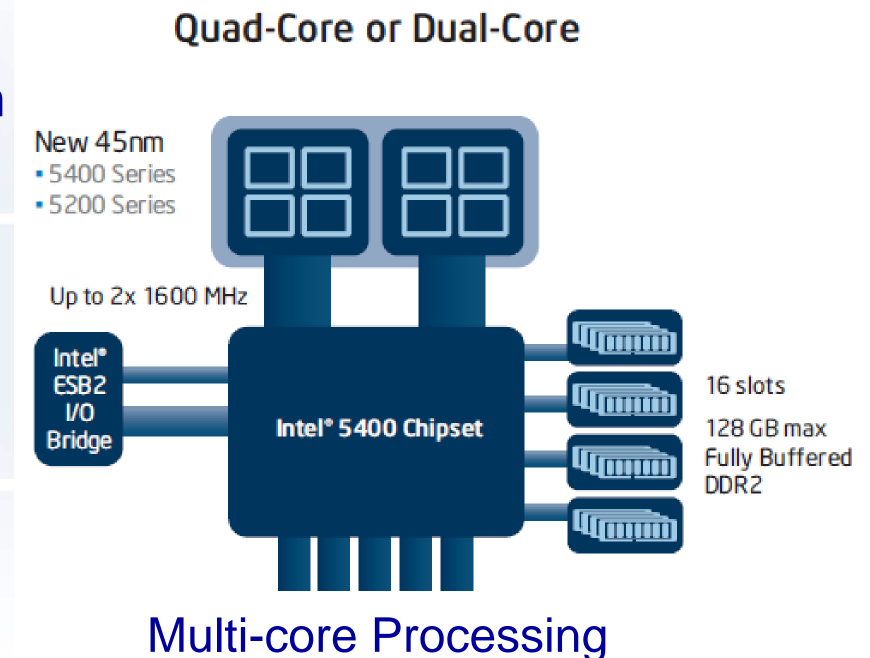
Multicore Programming: Agenda

Shared Memory Programming: OpenMP

1) Multicore Architecture & Shared Memory
2) Parallel Programming Models: Threads
3) Multicore Programming: OpenMP

1) Multicore Architecture

- Quad & Dual Core Intel Xeon 5400: Bandwidth-intensive applications, High Performance Clusters.
- Larger memory support: Up to 128 GB and 20% faster FSB (1600 MHz)
- It delivers up to 34% performance increase over the Intel Xeon 5300.
- 50% larger multi-core optimized L2 cache enable users to increase the probability of data access.

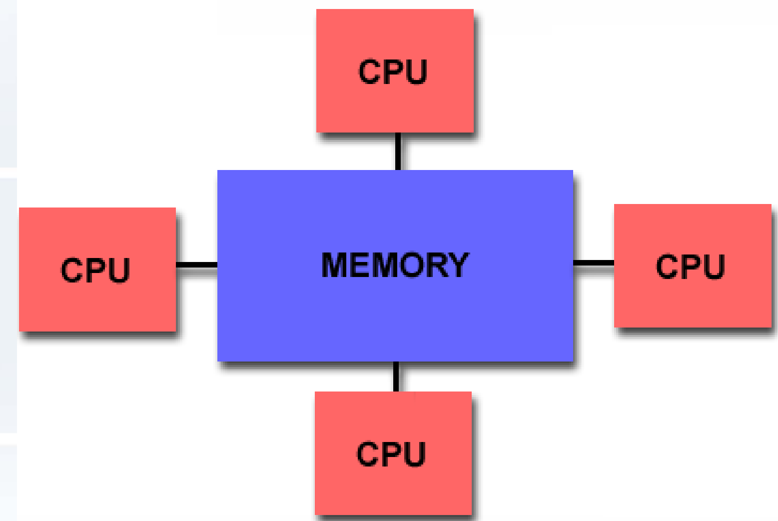


Multi-core Processing

- 2x bandwidth with PCI gen 2.
- Expanded power management capabilities, enhancements designed to reduce virtualization overhead.
- 47 new Intel® SSE4 instructions which can help improve the performance of media and HPC applications.

1) Shared Memory

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.



- Changes in a memory location generated by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: UMA, NUMA.

1) Shared Memory

- Advantages Shared Memory:
  - Global address space provides a user-friendly programming perspective to memory.
  - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.
- Disadvantages Shared Memory:
  - Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
  - Programmer responsibility for synchronization constructs that insure correct access of global memory.
  - It is difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

2) Parallel Programming Models: Threads

- In the threads model of parallel programming, a single process can have multiple, concurrent execution paths.
- Perhaps the most simple analogy that can be used to describe threads is the concept of a single program that includes a number of subroutines:
  - The main program a.out loads and acquires all of the necessary system and user resources to run.
  - a.out performs some serial work, and then creates a number of tasks (called threads) that can be scheduled and run by the operating system concurrently.
  - Each thread has local data, but also, shares the entire resources of a.out. This saves the overhead associated with replicating a program's resources for each thread.
  - Each thread also benefits from a global memory view because it shares the memory space of a.out.

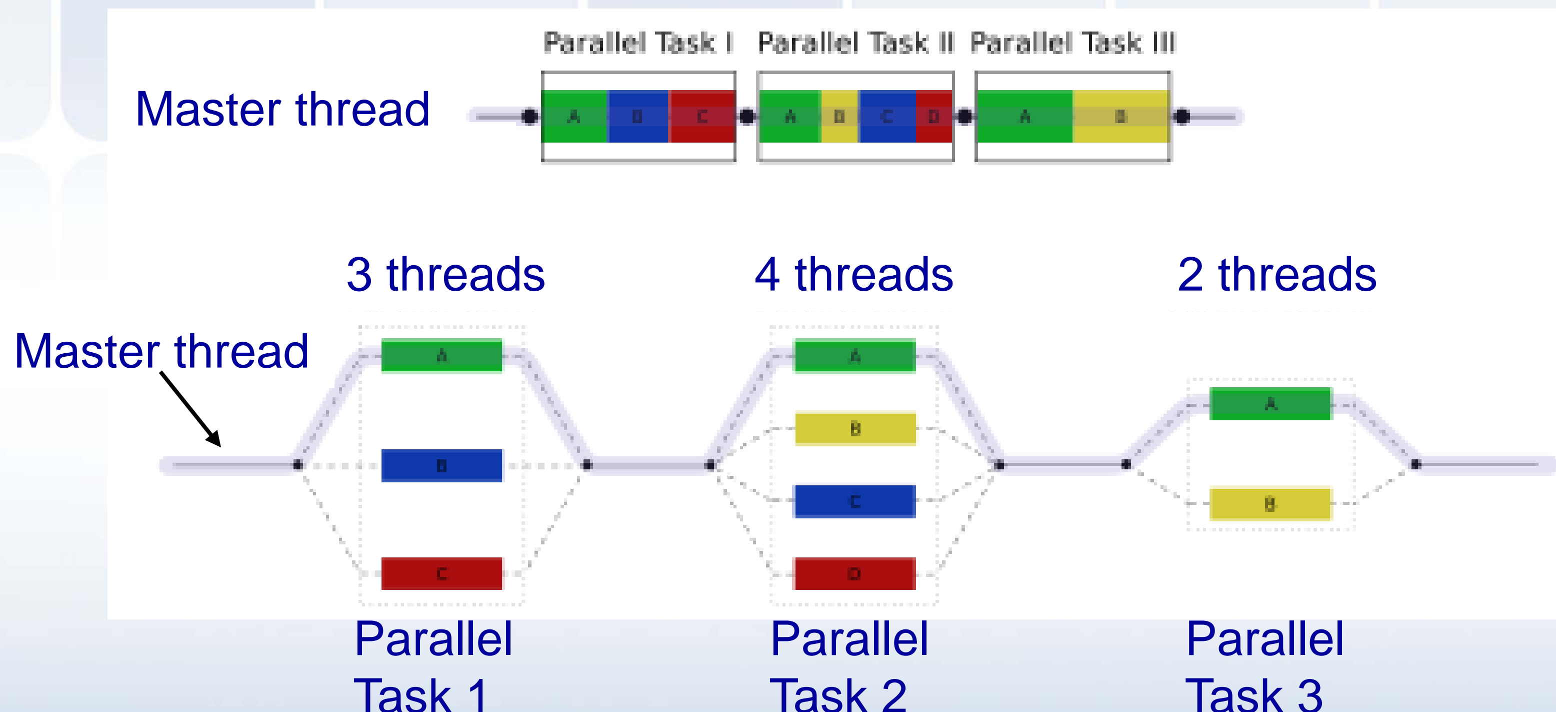
2) Parallel Programming Models: Threads

- A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.
- Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to insure that more than one thread is not updating the same global address at any time.
- Threads can come and go, but a.out remains present to provide the necessary shared resources until the application has completed.

- Threads are commonly associated with shared memory architectures and operating systems.

2) Parallel Programming Models: Threads

- Fork-join parallelism:
  - Master thread spawns a team of threads as needed.
  - Parallelism is added incrementally: the sequential program evolves into a parallel program.



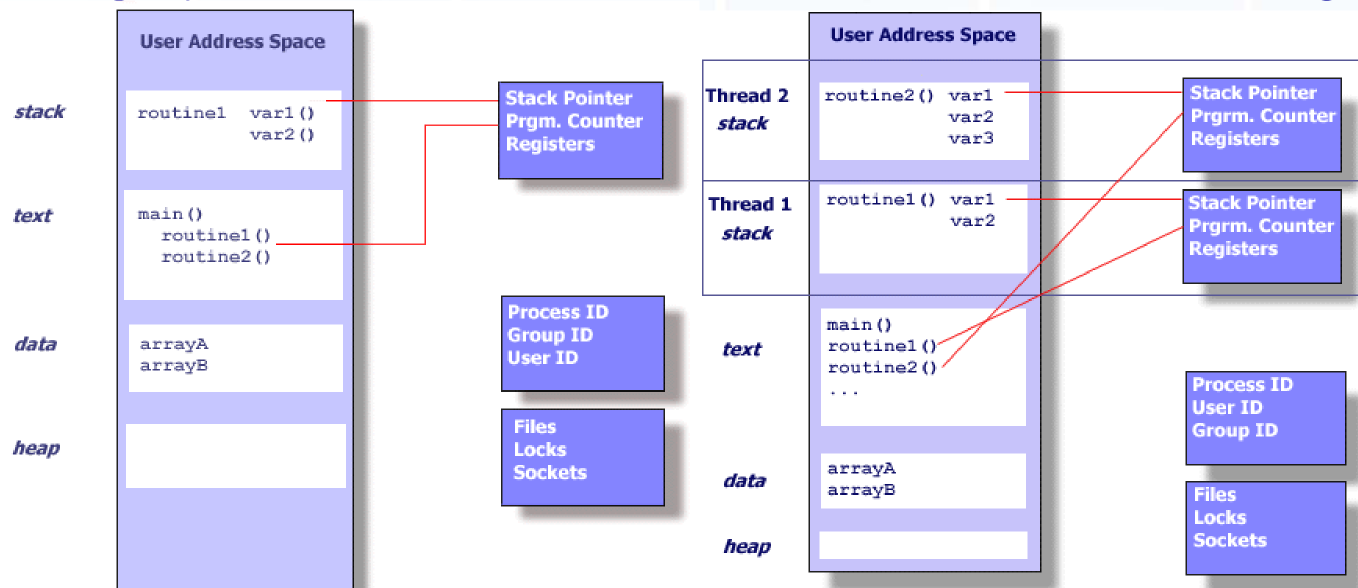
Parallel Task 1

Parallel Task 2

Parallel Task 3

3) Multicore Programming: OpenMP Basics

- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. But what does this mean?
- Imagine a main program (a.out) that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That describe a "multi-threaded" program.



3) Multicore Programming: OpenMP Basics

- Before understanding a thread, one first needs to understand a process.
- A process is created by the operating system, and requires a fair amount of "overhead". Processes contain information about program resources and program execution state, including:
  - Process ID, process group ID, user ID, and group ID
  - Environment
  - Working directory.
  - Program instructions
  - Registers, Stack, Heap
  - File descriptors
  - Signal actions
  - Shared libraries
  - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

3) Multicore Programming: OpenMP Basics

- In summary, a thread:
  - Exists within a process and uses the process resources.
  - Maintains its own: Stack pointer, Registers, Scheduling properties, Set of pending and blocked signals, Thread specific data.
  - Has its own independent flow of control.
  - Duplicates only the essential resources it needs to be independently schedulable
  - May share the process resources with other threads
  - Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.
- Because threads within the same process share resources:
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  - Two pointers having the same value point to the same data.
  - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

3) Multicore Programming: OpenMP Basics

- OpenMP is:
  - Compiler directives and clauses
  - Environment variables
  - Runtime environment

  for multithreaded programming
- Easy to create threaded Fortran and C/C++ codes.
- Supports data parallelism and incremental parallelism.
- Combines serial and parallel code in single source.
- All threads have access to the same globally shared memory.
- Data can be shared or private. Shared data is accessible by all threads.
- Private data can be accessed only by the threads that owns it.
- Data transfer is transparent to the programmer.
- Synchronization takes place, but it is mostly implicit.

3) Multicore Programming: OpenMP Basics
Program Structure

```
#include <stdio.h>
#include <omp.h>
main ()
{
int var1, var2, var3;

// Serial code //

// Beginning of parallel section. Fork a team of threads. //

#pragma omp parallel private(var1, var2) shared(var3) // Specify variable scoping //
{

// Parallel section executed by all threads//
        .
// All threads join master thread and terminate //

}

// Resume serial code //

}
```

3) Multicore Programming: OpenMP Basics
Components of OpenMP

- Directives and clauses
  - Parallel regions
  - Work sharing
  - Synchronization
  - Data scope attributes
    - private
    - firstprivate
    - lastprivate
    - shared
    - reduction
  - Orphaning
- Environment variables
  - Number of threads
  - Scheduling type
  - Dynamic thread adjustment
  - Nested parallelism
- Runtime environment
  - Number of threads
  - Thread ID
  - Dynamic thread adjustment
  - Nested parallelism
  - Timers
  - API for locking

3) Multicore Programming: OpenMP Directives

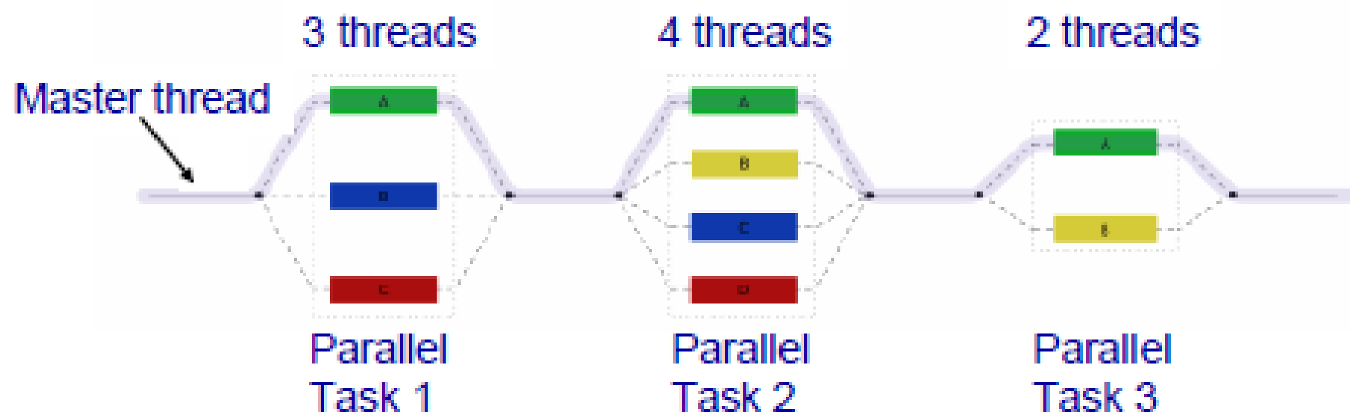
- Format:

#pragma omp	directive-name	[clause, ...]	newline
Required for all OpenMP C/C++ directives.	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Precedes the structured block which is enclosed by this directive.

- Example: #pragma omp parallel default(shared) private(beta,pi)
- 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.

3) Multicore Programming: OpenMP Directives Parallel Region

- When a thread reaches a parallel directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.



3) Multicore Programming: OpenMP Directives Parallel Region

- Format:

```
#pragma omp parallel [clause_1… clause_n]
{
// Instructions //
}
```

- A parallel region supports the following clauses:
  - if (scalar expression)
  - private (list)
  - shared (list)
  - default (none|shared)
  - reduction (operator: list)
  - copyin (list)
  - firstprivate (list)
  - num_threads (scalar_int_expr)

3) Multicore Programming: OpenMP Directives Parallel Region

- How Many Threads?

  The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - Evaluation of the if clause
  - Setting of the num_threads clause
  - Use of the omp_set_num_threads() library function
  - Setting of the omp_ num_threads environment variable
  - Implementation default: Usually the number of CPUs on a node, though it could be dynamic.
- Threads are numbered from 0 to (N – 1).
- Thread 0 is the master thread.

3) Multicore Programming: OpenMP Directives
Parallel Region

- Dynamic Threads: Use the omp_get_dynamic() library function to determine if dynamic threads are enabled.
- If supported, 2 methods are available for enabling dynamic threads:
  - The omp_set_dynamic() library routine
  - Setting of the OMP_DYNAMIC environment variable to TRUE
- Nested Parallel Regions:
  - Use the omp_get_nested() library function to determine if nested parallel regions are enabled.
  - Two methods available for enabling nested parallel regions are:
    - The omp_set_nested() library routine
    - Setting of the OMP_NESTED environment variable to TRUE
- If not supported, a parallel region nested within another parallel region results in the creation of a new team of one thread.

3) Multicore Programming: OpenMP Directives Parallel Region

```
#include <omp.h>
#include <stdio.h>
main ()  {
int nthreads, tid;
#pragma omp parallel private(tid)
{
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    if (tid == 0) /* Only master thread does this */
    {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
    }
}  /* All threads join master thread and terminate */
}
```

3) Multicore Programming: OpenMP Directives Work-Sharing

- Work-Sharing Constructs
  - A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
  - Work-sharing constructs do not launch new threads
  - There is an implied barrier at the end of a work sharing construct.
- Work-Sharing Directives:
  - **for**: Shares iterations of a loop across the team. Represents a type of "data parallelism".
  - **sections / section**: Breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".
  - **single**: Serializes a section of code.
- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.

3) Multicore Programming: OpenMP Directives Work-Sharing: for

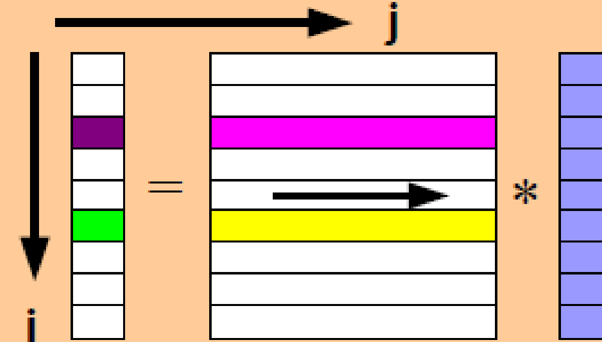
- The iterations of the for loop are distributed over the threads

```
#pragma omp for clause_1 … clause_n
{
// Instructions //
}
```

- Threads are assigned an independent set of iterations.
- Threads must wait at the end of work-sharing construct.
- Clauses supported:
- private, firstprivate, lastprivate
- reduction
- ordered
- schedule
- nowait

3) Multicore Programming: OpenMP Directives
Work-Sharing: parallel region + for

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



TID = 0

```
for (i=0,1,2,3,4)
i = 0
   sum = Σ b[i=0][j]*c[j]
   a[0] = sum
i = 1
   sum = Σ b[i=1][j]*c[j]
   a[1] = sum
```

TID = 1

```
for (i=5,6,7,8,9)
i = 5
   sum = Σ b[i=5][j]*c[j]
   a[5] = sum
i = 6
   sum = Σ b[i=6][j]*c[j]
   a[6] = sum
```

3) Multicore Programming: OpenMP Directives Work-Sharing: sections / section

- The **sections** directive is a non-iterative work-sharing construct.
- It specifies that the enclosed **section(s)** of code are to be divided among the threads in the team.
- Independent **section** directives are nested within a sections directive.
- Each **section** is executed once by a thread in the team.
- Different sections may be executed by different threads.
- It is possible that for a thread to execute more than one section if it is quick enough and the implementation permits such.
- There is an implied barrier at the end of a **sections** directive, unless the **nowait** clause is used.

3) Multicore Programming: OpenMP Directives Work-Sharing: sections / section

- Format:

```
#pragma omp sections clause_1 … clause_n
{
#pragma omp section
// Instructions //
#pragma omp section
// Instructions //
#pragma omp section
}
```

- Clauses supported:
  - private(list)
  - firstprivate(list)
  - lastprivate (list)
  - reduction (operator: list)
  - nowait

3) Multicore Programming: OpenMP Directives Work-Sharing: sections / section

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/b[i];

    } /*-- End of sections --*/

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

3) Multicore Programming: OpenMP Directives Work-Sharing: single

- The **single** directive specifies that the enclosed code is to be executed by only one thread in the team.
- May be useful when dealing with sections of code that are not thread safe (such as I/O).
- Threads in the team that do not execute the **single** directive, wait at the end of the enclosed code, unless a nowait clause is specified.
- Format:

```
#pragma omp single clause_1 … clause_n
{
// Instructions //
}
```

- Clauses supported:
  - private(list)
  - firstprivate(list)
  - nowait

3) Multicore Programming: OpenMP Directives Synchronization: master

- The **master** directive specifies a region must be executed only by the master thread of the team.
- No implicit barrier at end
- All other threads on the team skip this section of code.
- Format:

```
#pragma omp master
{
// Instructions //
}
```

3) Multicore Programming: OpenMP Directives
Synchronization: critical

- If sum is a shared variable, this loop can not be run in parallel:

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

- Within the critical directive all threads execute the instructions, but only one at a time:

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

3) Multicore Programming: OpenMP Directives Critical Example: Dot Product

```
#include <stdio.h>
#include <omp.h>
#define N 1000
#define Num_Threads 8 // Number of threads
int main()
{
int i;
double a[N], b[N], sum=0.0, start_time, elapsed_time;
start_time=omp_get_wtime();
for (i=0; i<N; i++) { a[i]=rand(); b[i]=rand(); }
omp_set_num_threads(Num_Threads);
#pragma omp parallel for shared(sum)
for (i=0; i<N; i++)
{
#pragma omp critical
        sum += a[i] * b[i];
}
elapsed_time = omp_get_wtime() - start_time;
printf("Dot product = %f and elapsed time is: %f\n",sum,elapsed_time);
}
```

3) Multicore Programming: OpenMP Directives Synchronization: barrier, atomic

- The **barrier** directive synchronizes all threads in the team:
  #pragma omp barrier
- Explicit barrier synchronization.
- When a **barrier** directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.
- The **atomic** directive specifies that a memory location must be updated atomically, rather than letting multiple threads attempt to write to it:

  #pragma omp atomic

3) Multicore Programming: OpenMP Directives Synchronization: ordered

- The **ordered** directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.
- Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed yet.
- Used within a **for** loop with an **ordered** clause
- The **ordered** directive provides a way to "fine tune" where ordering is to be applied within a loop.
- Format:

```
#pragma omp for ordered [clauses...]
{
// Instructions //
}
```

3) Multicore Programming: OpenMP Data Shared & Private Clauses

- In a shared memory parallel program variables have a "label" attached to them:
- If label = “private“ the data is visible to one thread only. Change made in local data, is not seen by others

```
void work(float *c, int N)
{
float x, y; int i;
#pragma omp parallel for private(x,y)
        for(i=0; i<N; i++)
        {
            x = a[i]; y = b[i];
            c[i] = x + y;
        }
}
```

- If label = “shared" the data is visible to all threads Change made in global data, is seen by all others

3) Multicore Programming: OpenMP Data Shared & Private Clauses

- shared(var1,var2,…,varn)   private(var1,var2,…,varn)
- Private variables are undefined on entry and exit of the parallel region.
- The value of the original variable (before the parallel region) is undefined after the parallel region.
- A private variable within a parallel region has no storage association with the same variable outside of the region.
- Use the first/last private clause to change this behaviour.
- firstprivate(var1,var2,…,varn)
  - All variables in the list are initialized with the value the original object had before entering the parallel construct

3) Multicore Programming: OpenMP Data Shared & Private Clauses

```
main()
{
  A = 10;

 #pragma omp parallel
 {
  #pragma omp for private(i) firstprivate(A) lastprivate(B)...
  for (i=0; i<n; i++)
  {
      ....
      B = A + i;          /*-- A undefined, unless declared
                               firstprivate --*/
      ....
  }

  C = B;                  /*-- B undefined, unless declared
                               lastprivate --*/

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

}
```

3) Multicore Programming: OpenMP Data Last Private & Default Clauses

- lastprivate(var1,var2,…,varn)
  - The thread that executes the sequentially last iteration or section updates the value of the objects in the list
- default (none | shared)
  - none
    - No implicit defaults
    - Have to scope all variables explicitly
  - shared
    - All variables are shared
    - The default in absence of an explicit "default" clause
- default(private) is not supported in C/C++

3) Multicore Programming: OpenMP Data Reduction Clause

- reduction(operator : list)
- The variables in “list” must be shared in the enclosing parallel region.
- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the “operator”.
  - These copies are updated locally by threads.
  - At end of construct, local copies are combined through “operator” into a single value and combined with the value in the original SHARED variable.

Operand	Initial Value
+	0
*	1
-	0
^	0

Operand	Initial Value
&	~0
|	0
&&	1
||	0

3) Multicore Programming: OpenMP Data Reduction Clause

- Example:

```
#pragma omp parallel for reduction(+:sum)
        for(i=0; i<N; i++)
        {
                a[i]=rand(); b[i]=rand();
                sum += a[i] * b[i];
        }
```

- Local copy of *sum* for each thread
- All local copies of *sum* added together and stored in “global” variable
- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

3) Multicore Programming: OpenMP Data Schedule Clause

- schedule (type [,chunk])
- Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.
- If type = static

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

  Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

3) Multicore Programming: OpenMP Data Schedule Clause

- schedule (type [,chunk])
- Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.
- If type = static

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

  Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

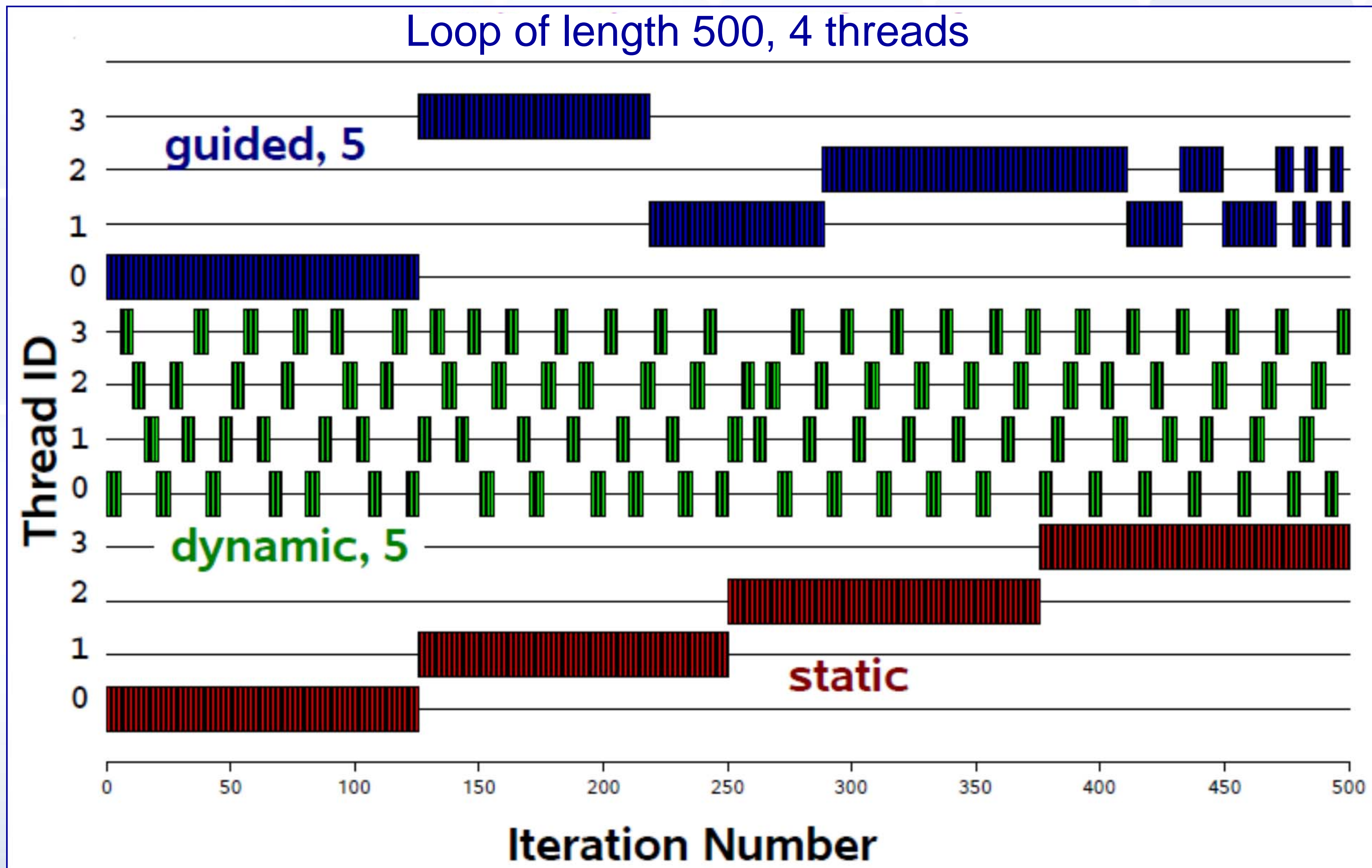
3) Multicore Programming: OpenMP Data Schedule Clause

- schedule (type [,chunk])
- If type = 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 (greater than 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.
- If type = 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.

3) Multicore Programming: OpenMP Data Schedule Clause



3) Multicore Programming: OpenMP Data Nowait Clause

- nowait: Threads do not synchronize at the end of the parallel loop.

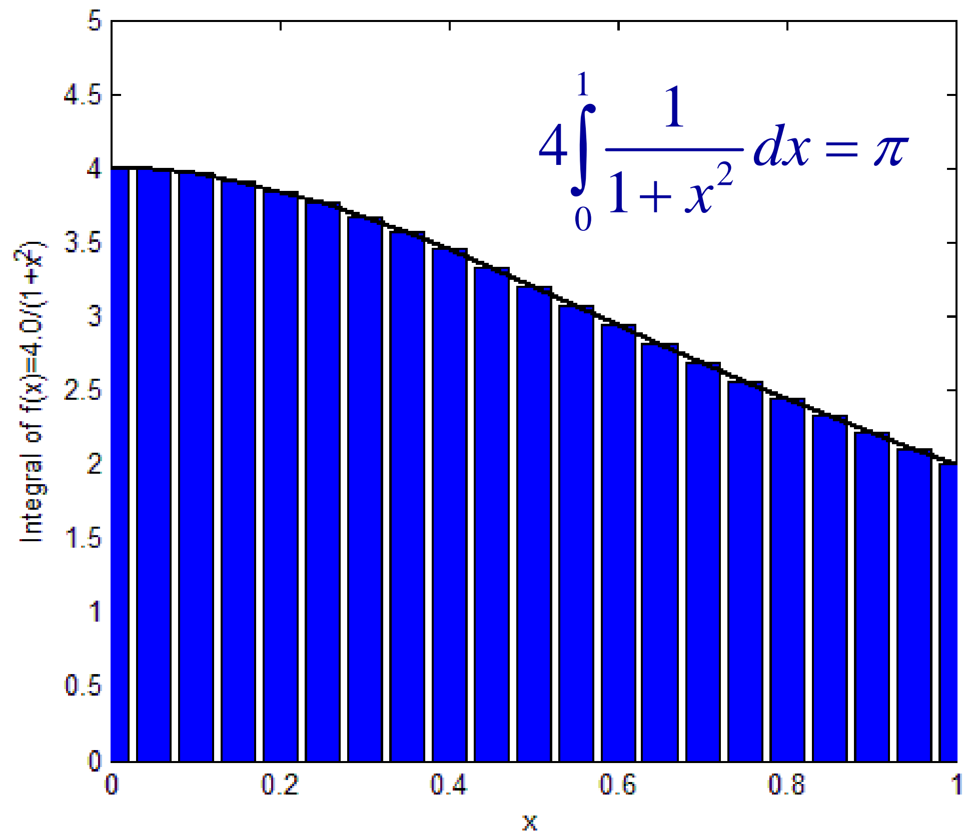
```
#include <omp.h>
#define CHUNKSIZE 100
main ()
{
int i, N=1000; chunk; float a[N], b[N], c[N];
for (i=0; i < N; i++) {a[i] = b[i] = (float)(i);}
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
{
#pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++) c[i] = a[i] + b[i];
}
}
```

3) Multicore Programming: OpenMP Data Runtime Environment

Name	Functionality
omp_set_num_threads	Set number of threads
omp_get_num_threads	Return number of threads in team
omp_get_max_threads	Return maximum number of threads
omp_get_thread_num	Get thread ID
omp_get_num_procs	Return maximum number of processors
omp_in_parallel	Check whether in parallel region
omp_set_dynamic	Activate dynamic thread adjustment
omp_get_dynamic	Check for dynamic thread adjustment
omp_set_nested	Activate nested parallelism
omp_get_nested	Check for nested parallelism
omp_get_wtime	Returns wall clock time
omp_get_wtick	Number of seconds between clock ticks

3) Multicore Programming: OpenMP Data Exercise: Computing Pi

- Parallelize the numerical integration code using OpenMP
- What variables can be shared?
- What variables need to be private?
- What variables should be set up for reductions?



3) Multicore Programming: OpenMP Data Exercise: Computing Pi

```
#include <stdio.h>
#include <omp.h>
static long num_steps=100000;
double step, pi;
void main()
{
int i;
double x, sum = 0.0, start_time;
start_time=omp_get_wtime();
step = 1.0/(double)(num_steps);
for (i=0; i< num_steps; i++)
{
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
}
pi = step*sum;
elapsed_time = omp_get_wtime() - start_time;
printf("Pi = %f\n",pi); printf("Elapsed time = %f\n",elapsed_time);
}
```

References

1) Dongarra, J., I. Foster (Eds.), The Sourcebook of Parallel Computing, Morgan Kaufmann, 2002.
2) Hoffmann, K.H., A. Meyer, Parallel Algorithms and Cluster Computing: Implementations, Algorithms and Applications, Springer, 2006.
3) Quinn, M.J., Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2004.
4) Intel® Software College, Programming with OpenMP, 2008.
5) Tutorial on OpenMP Programming, Sun Microsystems, 2005.
6) Tutorial on Parallel Computing and Programming, Lawrence Livermore National Laboratory, 2007.