Numerical Simulation and Scientific Computing I

Lecture 8:
Shared Memory Parallel Computing

Luiz Felipe Aguinsky, Paul Manstetten, and
Josef Weinbub

Institute for Microelectronics
TU Wien

nssc@iue.tuwien.ac.at

WS 2022

Quiz

- Q1: Consider a 32-bit LCG. How many random numbers can you generate before the sequence repeats?

32-bit → m= 2^{32}:

Periodicity = 2^{32} → ~10^9 random numbers

Quiz

- Q2: What are the consequences of entropy pool depletion? How would you handle it?

The entropy pool is unable to provide more random bits. I could wait until it is replenished.

Quiz

- Q3: What are the requirements to initialize correctly one independent RNG per thread?

Each thread must initialize its own RNG with a different seed

Quiz

- **Q4: What is the difference between a process and a thread? Which resources can be shared among them?**
  → Later.
- **Q5: What does the directive `#pragma omp atomic` do?**
  → Later.

Source

Blaise Barney, Lawrence Livermore National Laboratory

Introduction to Parallel Computing

https://computing.llnl.gov/tutorials/parallel_comp/

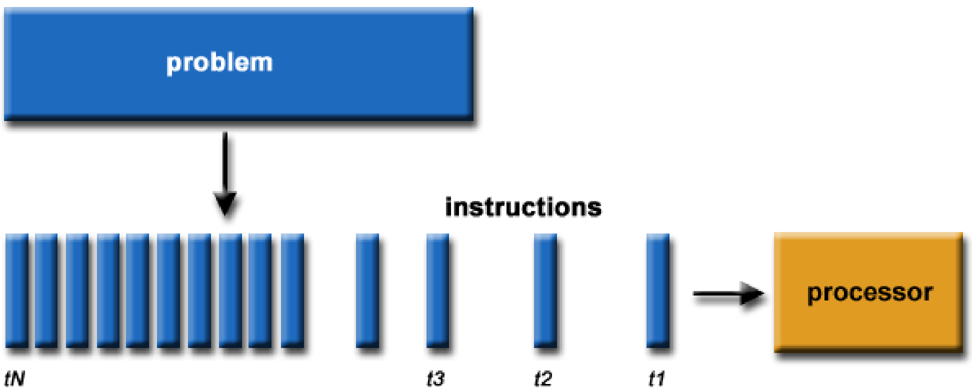
Blaise Barney, Lawrence Livermore National Laboratory

OpenMP Tutorial

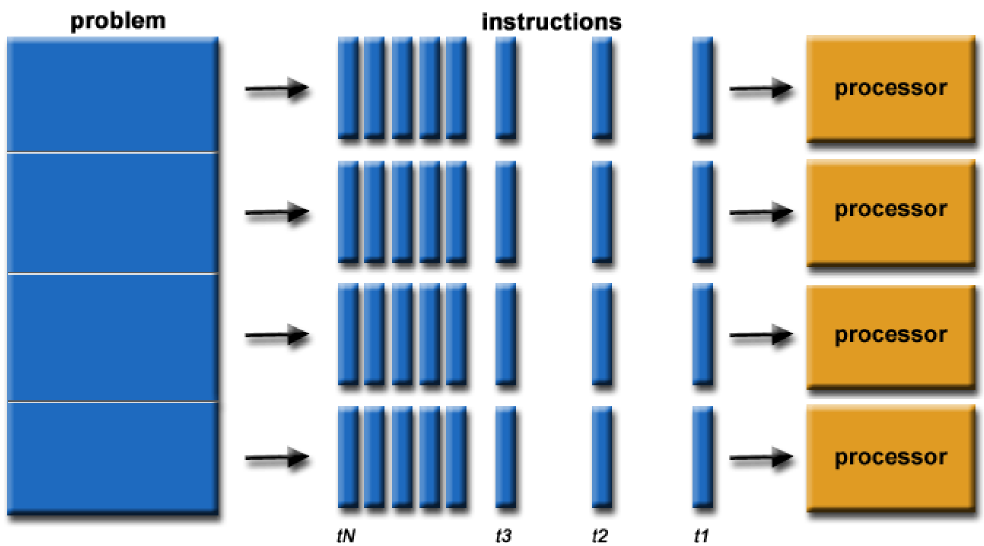
https://computing.llnl.gov/tutorials/openMP/

What is Parallel Computing?

Serial Computing



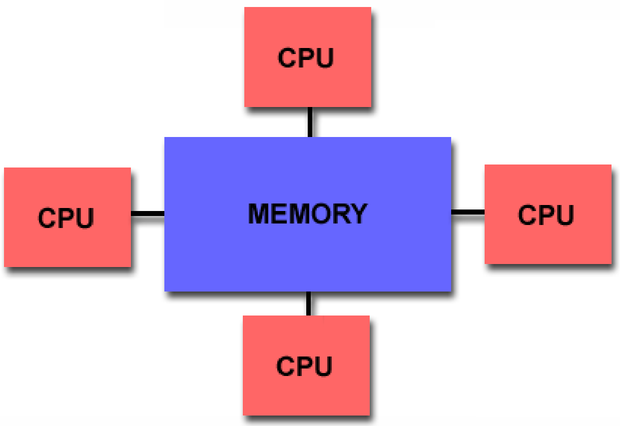
Parallel Computing



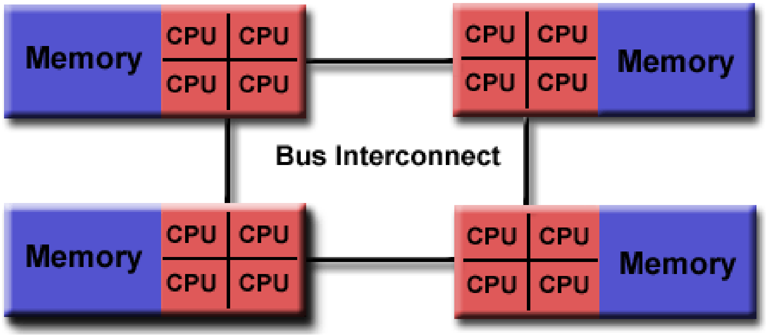
Source: A. Silberschatz *et al.*: Operating System Concepts, Wiley, 2012

Shared-Memory Computer Architecture

Uniform Memory Access (UMA)



Non-Uniform Memory Access (NUMA)



Source: Lyle N. Long, PSU

Terminology

- **Program is an executable file (stored on the disk) containing a set of instructions, e.g., firefox.exe**
- **Process is an executing instance of a program: When a program is executed, a process is generated with a single thread, the program is loaded into memory and executed.**
  - **Note: More than one process can execute the same program.**
- **Thread is the smallest executable unit of a process. When a process is started, the main thread of that process is started.**
  - **Note: A process has at least one thread (single-threaded), but can have more (multi-threaded).**

Terminology

- The threads of a process support:
  - Shared variables
  - Private variables
  - Communication via read/write shared data
  - Coordinate by synchronizing on shared data
- Threads can be dynamically created and destroyed
- Other programming models:
  - Distributed-memory
  - Hybrid

Processes vs Threads

Q4: What is the difference between a process and a thread? Which resources can be shared among them?

State

- Instruction pointer
- Register file
- Stack pointer

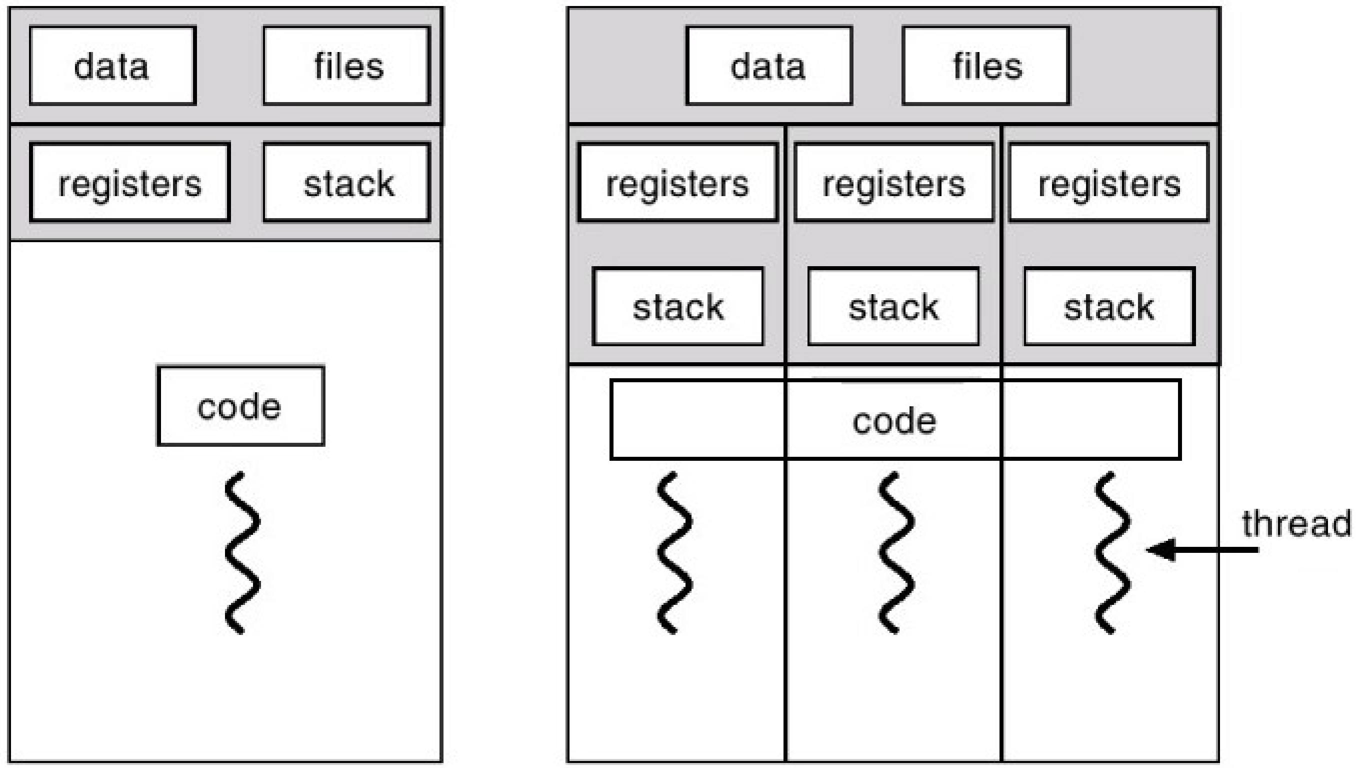
Processes

- Own state and own address space
- Interaction with inter-process communication
- A process may contain several threads

Threads

- All threads within a process share address space
- Own state but global and heap data are shared
- Interaction via shared variables

Processes vs Threads



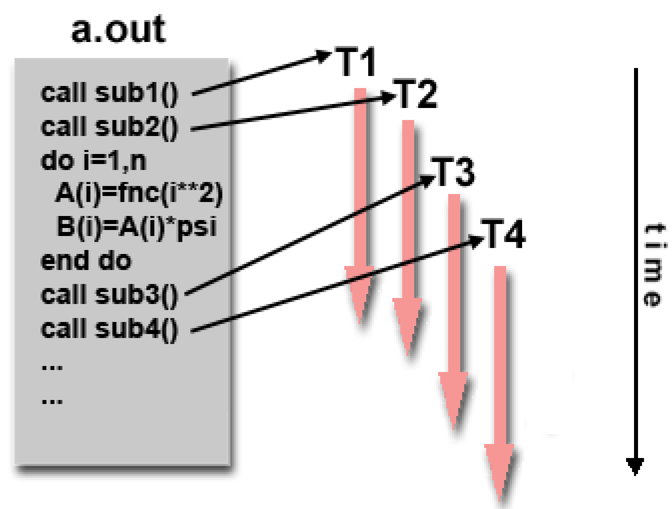
Single-threaded process

Multi-threaded process

Source: U Miami, Course CSC322-09S, UNIX Threads

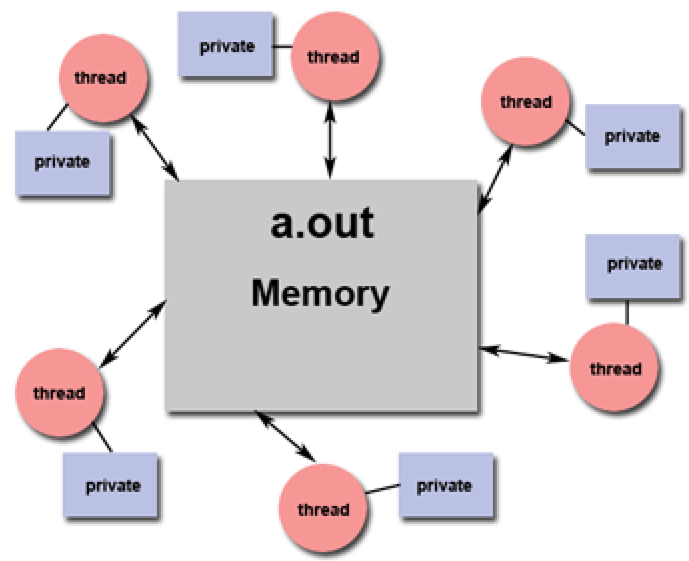
Shared-Memory Programming: Threads Model

One process: >= 1 threads



Example

- a.out program executed by
- process (heavy weight)
- a.out performs serial work →
- creates threads:
- executed concurrently by operating system
- Threads have local data but can also access a.out memory



Thread Model Implementations

- **Programming perspective, thread model implementations are comprised of:**
  - **Library of subroutines: called from within parallel source code**
  - **Set of compiler directives imbedded in either serial or parallel source code**
- **Programmer is responsible for determining the parallelism**
- **Two standardizations:**
  - **POSIX Threads**
  - **OpenMP**

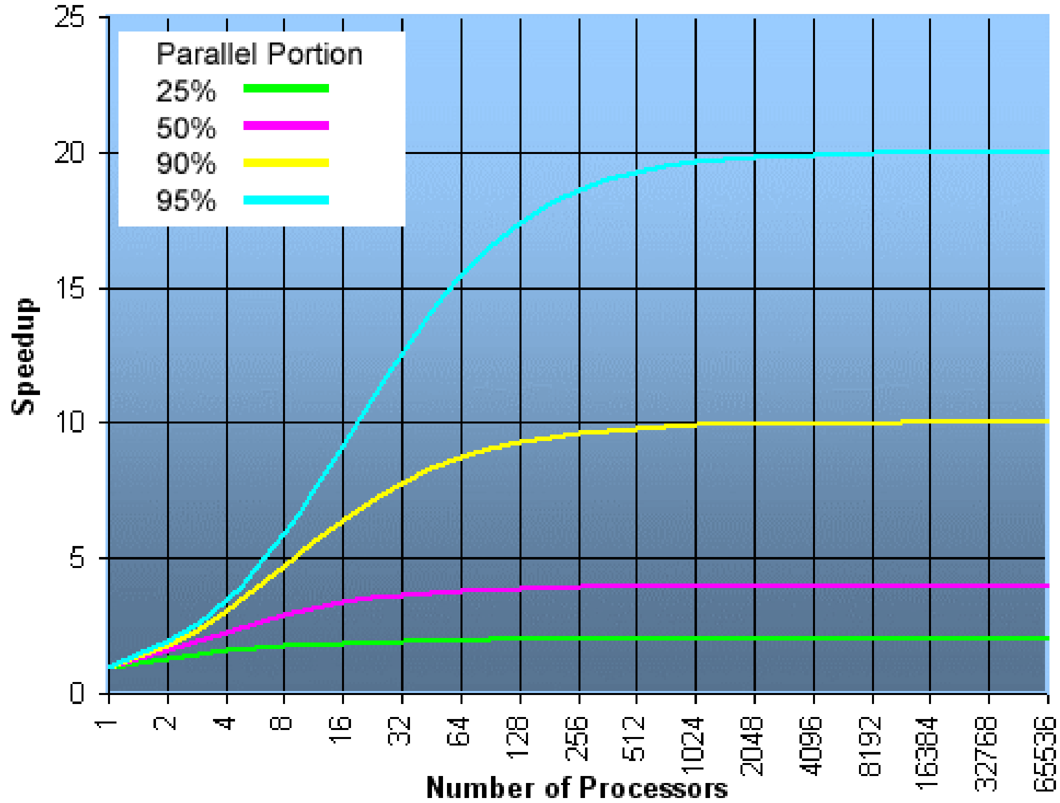
Standards: POSIX Threads and OpenMP

- **POSIX Threads**
  - **Specified by the IEEE POSIX 1003.1c standard (1995)**
  - **C Language**
  - **Part of Unix/Linux operating systems**
  - **Library based**
  - **Commonly referred to as *Pthreads***
  - **Very explicit parallelism; requires significant programmer attention to detail**
- **OpenMP**
  - **Industry standard**
  - **Compiler directive based**
  - **Portable / multi-platform, including Unix and Windows platforms**
  - **Available in C/C++ and Fortran implementations**
  - **Can be very easy and simple to use - provides for "incremental parallelism". Can begin with serial code.**

Amdahl's Law

$$S = \frac{1}{\frac{F_P}{N} + F_S}$$

S … speedup
F_P … parallel fraction
F_S … serial fraction
N … number of processors

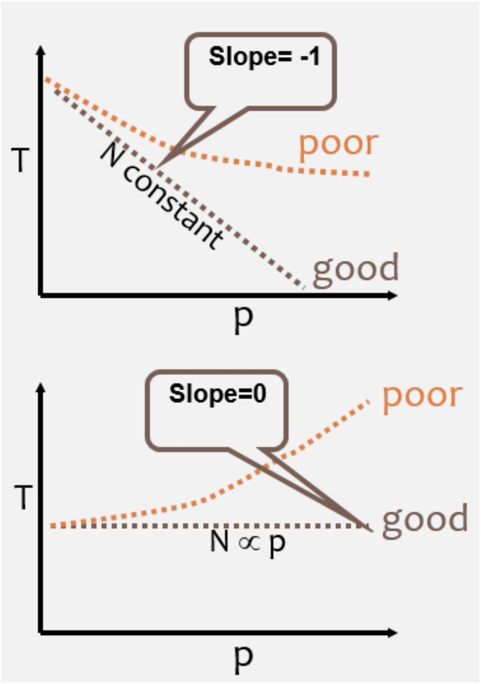


Amdahl's law gives the upper limit of speedup for a problem of fixed size. (see Strong Scaling)

Scalability

Two types of scaling based on time to solution:

- **Strong scaling:**
  - The total problem size stays fixed as more processors are added.
  - Goal is to run the same problem size faster.
  - Perfect scaling means problem is solved in 1/P time (compared to serial).
  - Primarily used for CPU-bound applications.



- **Weak scaling:**
  - The problem size *per processor* stays fixed as more processors are added. The total problem size is proportional to the number of processors used.
  - Goal is to run larger problem in same amount of time.
  - Perfect scaling means problem Px runs in same time as single processor run.
  - Primarily used for memory-bound applications.

https://hpc-wiki.info/hpc/Scaling

Scalability

- **The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more processors is rarely the answer.**
- **The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. This is a common situation with many parallel applications.**
- **Hardware factors play a significant role in scalability:**
  - **Important: Memory-CPU bus bandwidth (latency)**
- **Parallel support libraries and subsystems software can limit scalability independent of your application.**

OpenMP

- OpenMP: Open Multi-Processing
- OpenMP is an Application Program Interface (API)
- Multi-threaded, shared memory parallelism
- Jointly defined by a group of major computer hardware and software vendors
- OpenMP provides a portable, scalable model for developers of shared memory parallel applications.
- API supports C/C++ and Fortran on a wide variety of architectures.

OpenMP – A Look Ahead

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

main ()  {

  int var1, var2, var3;

  /* Serial code

  .

  Beginning of parallel region. Fork a team of threads.

  Specify variable scoping */

  #pragma omp parallel private(var1, var2) shared(var3)

  {

    /* Parallel region executed by all threads

         Other OpenMP directives

         Run-time Library calls

         All threads join master thread and disband

    */

  }

  /*  Resume serial code

  .

  */

}
```

OpenMP Is Not …

- … meant for distributed memory parallel systems (by itself)
- … necessarily implemented identically by all vendors
- … guaranteed to make the most efficient use of shared memory
- … required to check for data dependencies, data conflicts, race conditions, deadlocks, or code sequences that cause a program to be classified as non-conforming
- … designed to handle parallel I/O. The programmer is responsible for synchronizing input and output.

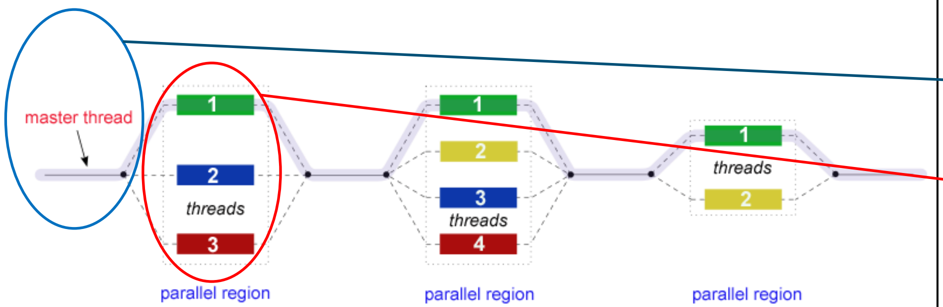
Goals of OpenMP 1/2

- **Standardization:**
  - Provide a standard among a variety of shared memory architectures/platforms
  - Jointly defined and endorsed by a group of major computer hardware and software vendors
- **Lean and Mean:**
  - Establish a simple and limited set of directives for programming shared memory machines.
  - Significant parallelism can be implemented by using just 3 or 4 directives.
  - This goal is becoming less meaningful with each new release, apparently.

Goals of OpenMP 2/2

- **Ease of Use:**
  - **Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach**
  - **Provide the capability to implement both coarse-grain and fine-grain parallelism**
- **Portability:**
  - **The API is specified for C/C++ and Fortran**
  - **Public forum for API and membership**
  - **Most major platforms have been implemented including Unix/Linux platforms and Windows**

Fork – Join Model




```
#include <iostream>
#include <omp.h>
main ()  {
  int var1, var2, var3;
  /* Serial code
  .
  Beginning of parallel region. Fork a team of threads.
  Specify variable scoping */
  #pragma omp parallel private(var1, var2) shared(var3)
  {
    /* Parallel region executed by all threads
               Other OpenMP directives
               Run-time Library calls
               All threads join master thread and disband
    */
  }
  /*  Resume serial code
  .
  */
}
```

- OpenMP programs begin as **single process**: the **master thread** → executes sequentially until parallel region construct
- FORK: master thread creates a team of parallel *threads*.
  - Statements in parallel region are executed in parallel
- JOIN: when parallel region finished, threads synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.

Data Scoping & Nested Parallelism

Data Scoping

- Because OpenMP is a shared memory programming model, most data within a parallel region is shared by default.
- All threads in a parallel region can access this shared data simultaneously.
- OpenMP provides a way for the programmer to explicitly specify how data is "scoped" if the default shared scoping is not desired.

Nested Parallelism

- The API provides for the placement of parallel regions inside other parallel regions.
- Implementations may or may not support this feature.

Dynamic Threads & I/O

Dynamic Threads

- The API provides for the runtime environment to dynamically alter the number of threads used to execute parallel regions. Intended to promote more efficient use of resources, if possible.
- Implementations may or may not support this feature.

I/O

- OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
- If every thread conducts I/O to a different file, the issues are not as significant.
- It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

API

- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables

Compiler Directives

- **Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag.**
- **OpenMP compiler directives are used for various purposes:**
  - **Spawning a parallel region**
  - **Dividing blocks of code among threads**
  - **Distributing loop iterations between threads**
  - **Serializing sections of code**
  - **Synchronization of work among threads**

Compiler Directives

- Compiler directives have the following syntax:

sentinel directive-name [clause, ...]

- Example

#pragma omp **parallel** shared(variable)

```
#include <iostream>
#include <omp.h>
main ()  {
  int var1, var2, var3;
  /* Serial code
  .
  Beginning of parallel region. Fork a team of threads.
  Specify variable scoping */
  #pragma omp parallel private(var1, var2) shared(var3)
  {
    /* Parallel region executed by all threads
              Other OpenMP directives
              Run-time Library calls
              All threads join master thread and disband
    */
  }
  /*  Resume serial code
  .
  */
}
```

Compiler Directives

- **Case sensitive**
- **Directives follow conventions of the C/C++ standards for compiler directives**
- **Only one directive-name may be specified per directive**
- **Each directive applies to at most one succeeding statement, which must be a *structured block*.**
- **A *structured block* is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.**
- **Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.**

General Code Structure

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

main ()  {

  int var1, var2, var3;

  /* Serial code

  .

  Beginning of parallel region. Fork a team of threads.

  Specify variable scoping */

  #pragma omp parallel private(var1, var2) \

                       shared(var3)

  {

    /* Parallel region executed by all threads

         Other OpenMP directives

         Run-time Library calls

         All threads join master thread and disband

    */

  }

  /*  Resume serial code

  .

  */
```

Compiling Open Programs

- **GNU GCC**

```
gcc / g++ -fopenmp -O2 -o mycode mycode.cpp
```

Different compilers support different OpenMP versions:

https://www.openmp.org/resources/openmp-compilers-tools/

Runtime Library Routines

- The OpenMP API includes an ever-growing number of run-time library routines.
- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution
- Example

```
#include <omp.h>

int omp_get_num_threads(void)
```

```
#include <iostream>
#include <omp.h>
main ()  {
  int var1, var2, var3;
  /* Serial code
  .
  Beginning of parallel region. Fork a team of threads.
  Specify variable scoping */
  #pragma omp parallel private(var1, var2) shared(var3)
  {
    /* Parallel region executed by all threads
               Other OpenMP directives
               Run-time Library calls
               All threads join master thread and disband
    */
    std::cout << "Number of available threads: " <<
                 omp_get_num_threads() << std::endl;
  }
  /*  Resume serial code
  .
  */
}
```

Environment Variables

- **OpenMP provides several environment variables for controlling the execution of parallel code at run-time.**
- **These environment variables can be used to control such things as:**
  - **Setting the number of threads**
  - **Specifying how loop interations are divided**
  - **Binding threads to processors**
  - **Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism**
  - **Enabling/disabling dynamic threads**
  - **Setting thread stack size**
  - **Setting thread wait policy**
- **Setting OpenMP environment variables is done the same way you set any other environment variables, and depends upon which shell you use.**
- **Example:**

```
export OMP_NUM_THREADS=8
```

```
export OMP_NUM_THREADS=8
./mycode  # run this executable with 8 threads
```

PARALLEL Region Construct

- **A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.**

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

```
#include <iostream>
#include <omp.h>
main ()  {
  int var1, var2, var3;
  /* Serial code
  .
  Beginning of parallel region. Fork a team of threads.
  Specify variable scoping */
  #pragma omp parallel private(var1, var2) shared(var3)
  {
    /* Parallel region executed by all threads
                 Other OpenMP directives
                 Run-time Library calls
                 All threads join master thread and disband
    */
  }
  /*  Resume serial code
  .
  */
}
```

PARALLEL Region Construct

- **If thread reaches PARALLEL directive: Team of threads created and thread becomes master of the team and member of the team (thread number 0)**
- **PARALLEL region code is *duplicated*: all threads execute the same code.**
- **Implicit barrier at end of PARALLEL region; only master thread continues past this point.**
- **If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.**

How Many 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_THREADS environment variable**
  5. **Implementation default - usually the number of CPUs on a node, though it could be dynamic (see next slide).**
- **Threads are numbered from 0 (master thread) to N-1**

Dynamic Threads & Clauses

Dynamic Threads

- **Use the `omp_get_dynamic()` library function to determine if dynamic threads are enabled.**
- **If supported, the two methods available for enabling dynamic threads are:**
  - **The `omp_set_dynamic()` library routine**
  - **Setting of the `OMP_DYNAMIC` environment variable to `TRUE`**

Clauses

- **IF clause: If present, it must evaluate to non-zero (C/C++) in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread.**

Nested Parallel Region

- **Use the `omp_get_nested()` library function to determine if nested parallel regions are enabled.**
- **The two methods available for enabling nested parallel regions (if supported) 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, consisting of one thread, by default.**

Restrictions

- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch (goto) into or out of a parallel region
- Only a single IF clause is permitted
- Only a single NUM_THREADS clause is permitted
- A program must not depend upon the ordering of the clauses

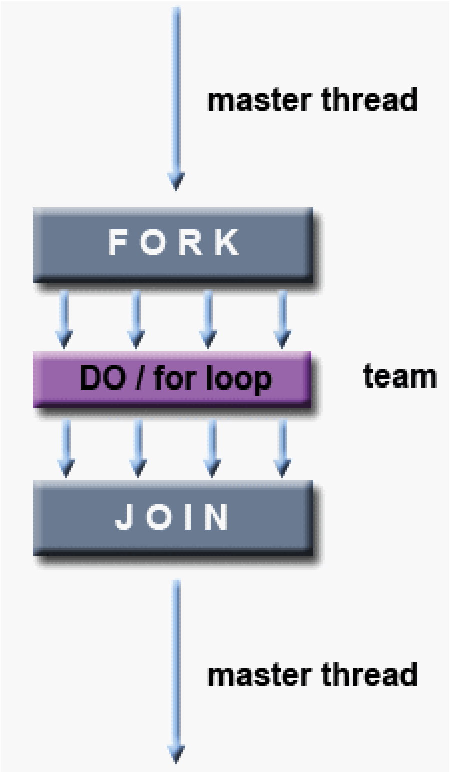
Example: Hello World

```
#include <iostream>
#include <omp.h>
main(int argc, char *argv[]) {
  int nthreads, tid;
  /* Fork a team of threads with each thread having a
  private tid variable */
  #pragma omp parallel private(tid)
  {
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    std::cout << "Hello World from thread " << tid << std::endl;
    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      std::cout << "Number of threads " << nthreads << std::endl;
    }
  }  /* All threads join master thread and terminate */
}
```

Warning: `omp_get_num_threads()` **will yield 1 in a serial scope. Use** `omp_get_max_threads()` **in serial scopes to get the number of available threads.**

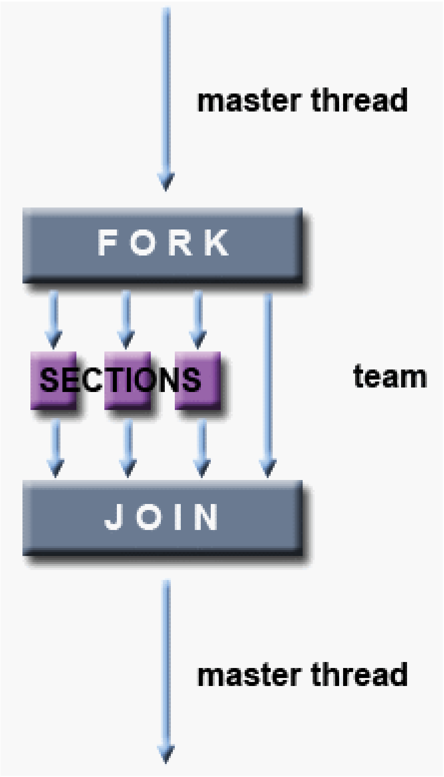
Work-Sharing Constructs

FOR LOOP



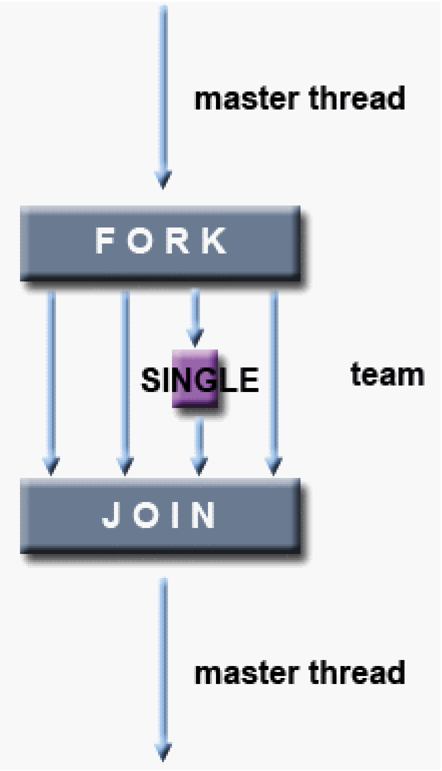
Loop iteration sharing
"data parallelism"

SECTIONS



Work separation
One section - one thread
“functional parallelism"

SINGLE



Selective serial execution

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 no implied barrier upon entry to a work-sharing construct, however, there is an implied barrier at the end of a work sharing construct.
- 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.
- Successive work-sharing constructs must be encountered in the same order by all members of a team.

Work-Sharing Constructs: FOR LOOP

- **The `for` directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.**

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

   for_loop
```

Work-Sharing Constructs: FOR LOOP

```
#include <iostream>
#include <omp.h>
main(int argc, char *argv[])
{
  int i;
  int N     = 1000;
  int chunk = 100;
  std::vector<float> a(N), b(N), c(N);
  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  #pragma omp parallel shared(a,b,c,chunk) \
                       private(i)
  {
    #pragma omp for
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  }   /* end of parallel region */
}
```

Simple vector-add program

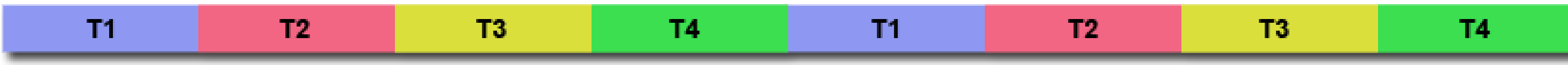
- **Vectors `a, b, c` and variable `N` will be shared by all threads.**
- **Variable `i` will be private to each thread (own unique copy).**
- **Loop iterations will be distributed statically**

Work-Sharing Constructs: FOR LOOP

[CLAUSE] Schedule:

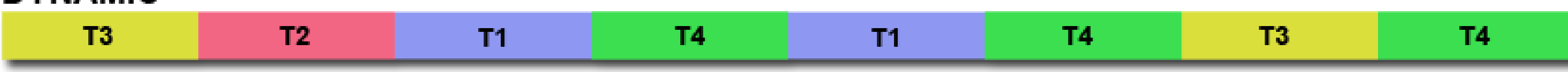
```
static / dynamic / guided / runtime / auto
```

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.

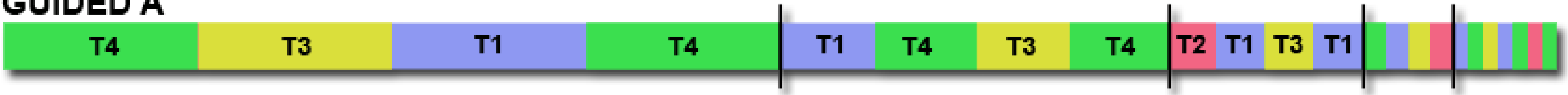
DYNAMIC



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.

Work-Sharing Constructs: FOR LOOP

GUIDED A



GUIDED B

Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to `dynamic` except that the block size decreases each time a parcel of work is given to a thread.

The size of the initial block is proportional to:

`number_of_iterations / number_of_threads`

Subsequent blocks are proportional to

`number_of_iterations_remaining / number_of_threads`

The chunk parameter defines the minimum block size. Default chunk size is 1.

<u>Note:</u> compilers differ in how GUIDED is implemented as shown in the "Guided A" and "Guided B" examples below.

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 delegated to the compiler and/or runtime system.

Work-Sharing Constructs: FOR LOOP

[CLAUSE] Nowait: If specified, then threads do not synchronize at the end of the parallel loop.

[CLAUSE] Ordered: Specifies that the iterations of the loop must be executed as they would be in a serial program.

[CLAUSE] Collapse: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause (discussed later). The order of the iterations in the collapsed iteration space is determined as though they were executed sequentially. May improve performance.

[CLAUSE] Data Scope Attributes: discussed later …

Work-Sharing Constructs: FOR LOOP

Main Restrictions:

- The loop iteration variable must be an integer and the loop control parameters must be the same for all threads.
- Program correctness must not depend upon which thread executes a particular iteration.
- It is illegal to branch (`goto`) out of a loop associated with a `for` directive.
- The chunk size must be specified as a loop invarient integer expression, as there is no synchronization during its evaluation by different threads.
- ORDERED, COLLAPSE and SCHEDULE clauses may appear once each.

Work-Sharing Constructs: FOR LOOP

```
#include <iostream>
#include <omp.h>
main(int argc, char *argv[])
{
  int i;
  int N     = 1000;
  int chunk = 100;
  std::vector<float> a(N), b(N), c(N);
  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  #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];
  }   /* end of parallel region */
}
```

Simple vector-add program

- **Vectors `a, b, c` and variable `N` will be shared by all threads.**
- **Variable `i` will be private to each thread (own unique copy).**
- **Loop iterations will be distributed dynamically in CHUNK sized pieces.**
- **No synchronization upon completing their individual work (NOWAIT).**

Work-Sharing Constructs: FOR LOOP COLLAPSE

```
int kl, ku, ks, jl, ju, js, il, iu,is;
int i, j, k;
#pragma omp for collapse(2) private(i,k,j)
for (k=kl; k<=ku; k+=ks)
  for (j=jl; j<=ju; j+=js)
    for (i=il; i<=iu; i+=is)
      some_function(a,i,j,k);
```

- **By default, only the k-loop is parallelized, now →**
- **k and j loops are associated with the loop construct: both loops *collapsed* into one loop with larger iteration space → divided among the threads.**
- **i loop is not associated with loop construct, not collapsed → executed in its entirety in every iteration of collapsed k-j loop.**
- **k and j: implicitly private, can be omitted from private**

Work-Sharing Constructs: SECTIONS

- **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 for a thread to execute more than one section if it is quick enough and the implementation permits such.**

```
#pragma omp sections [clause ...]  \
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
  {
  #pragma omp section
  structured_block
  #pragma omp section
  structured_block
  }
```

Work-Sharing Constructs: SECTIONS

Clauses

- **There is an implied barrier at the end of a SECTIONS directive, unless the `nowait` clause is used.**
- **Data Scope Attribute Clauses: discussed later …**

Restrictions

- **It is illegal to branch (`goto`) into or out of section blocks.**
- **SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive (no orphan SECTIONs).**

Work-Sharing Constructs: SECTIONS

```
#include <iostream>
#include <omp.h>
main(int argc, char *argv[]) {
  int i, int N = 1000;
  std::vector<float> a(N), b(N), c(N), d(N);
  /* Some initializations */
  for (i=0; i < N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
  }
  #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];
    }  /* end of sections */
  }  /* end of parallel region */
}
```

Work-Sharing Constructs: 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)

```
#pragma omp single [clause ...]  \
                   private (list)
                   firstprivate (list)
                   nowait
     structured_block
```

Work-Sharing Constructs: SINGLE

Clauses

- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a `nowait` clause is specified.
- Data Scope Attribute Clauses: discussed later …

Restrictions

- It is illegal to branch into or out of a SINGLE block.

Work-Sharing Constructs: SINGLE

```
#include <iostream>
#include <omp.h>
void work1() {}
void work2() {}
main(int argc, char *argv[]) {
  #pragma omp parallel
  {
    #pragma omp single
    std::cout << "Beginning work1." << std::endl;
    // barrier
    work1();
    #pragma omp single nowait
    std::cout << "Finished work1 and beginning work2." << std::endl;
    // no barrier
    work2();
  }
}
```

Combined Parallel Work-Sharing Constructs

- **OpenMP provides three directives that are merely conveniences:**
  - **PARALLEL FOR**
  - **PARALLEL SECTIONS**
- **Directives behave as an individual PARALLEL directive being immediately followed by a separate work-sharing directive.**

Combined PARALLEL FOR LOOP Construct

```
#include <iostream>
#include <omp.h>
main(int argc, char *argv[]) {
  int i, chunk=100, N=1000;
  std::vector<float> a(N), b(N), c(N);
  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  /* combined parallel vector addition */
  #pragma omp parallel for shared(a,b,c,chunk) private(i) \
                         schedule(static,chunk)
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
}
```

Versus:

```
#pragma omp parallel shared(a,b,c,chunk) private(i)
{
  #pragma omp for schedule(static,chunk)
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
}   /* end of parallel region */
```

Synchronization Constructs

```
THREAD 1:
update(x)
{
    x = x + 1
}
x = 0
update(x)
print(x)
```

```
THREAD 2:
update(x)
{
    x = x + 1
}
x = 0
update(x)
print(x)
```

Race Condition:
Update of x must be synchronized

One possible execution sequence:

1. **Thread 1 initializes x to 0 and calls update(x)**
2. **Thread 1 adds 1 to x.**
   x now equals 1
3. **Thread 2 initializes x to 0 and calls update(x)**
   x now equals 0
4. **Thread 1 prints x, which is equal to 0 instead of 1**
5. **Thread 2 adds 1 to x.**
   x now equals 1.
6. **Thread 2 prints x as 1.**

Synchronization Constructs: MASTER

- The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code
- There is no implied barrier associated with this directive

Restrictions

- It is illegal to branch into or out of MASTER block.

```
#pragma omp master

   structured_block
```

Synchronization Constructs: CRITICAL

- The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.
- If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.
- The optional name enables multiple different CRITICAL regions to exist.

TIP: Limit the use of CRITICAL sections as much as possible!

Restrictions

- It is illegal to branch into or out of a CRITICAL block.

```
#pragma omp critical [name]

   structured_block
```

Synchronization Constructs: CRITICAL

```
#include<iostream>
#include <omp.h>
main(int argc, char *argv[]) {
  int x = 0;
  #pragma omp parallel shared(x)
  {
    #pragma omp critical
    x = x + 1;
  }  /* end of parallel region */
}
```

Synchronization Constructs: BARRIER

- The BARRIER directive synchronizes all threads in the team.
- 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.

Restrictions

- All threads in a team (or none) must execute the BARRIER region.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

```
#pragma omp barrier
```

TIP: Limit the use of BARRIER sections as much as possible!

Synchronization Constructs: ATOMIC

- **The atomic construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values. In essence, this directive provides a mini-CRITICAL section.**

Restrictions

TIP: Limit the use of ATOMIC sections as much as possible!
If you must, prefer them over CRITICAL sections!

- **The directive applies only to a single, immediately following statement**
- **An atomic statement must follow a specific syntax. See the most recent OpenMP specs for this.**

Q5: What does the directive `#pragma omp atomic` do?

```
#pragma omp atomic  [ read | write | update | capture ]

   statement_expression
```

Synchronization Constructs: CRITICAL

```
#include<iostream>
#include <omp.h>
main(int argc, char *argv[]) {
  int x = 0;
  #pragma omp parallel shared(x)
  {
    #pragma omp atomic
    x = x + 1;
  }  /* end of parallel region */
}
```

Data Scope: THREADPRIVATE Directive

- The THREADPRIVATE directive specifies that variables are replicated, with each thread having its own copy.
- Global variables → local and persistent to a thread through the execution of multiple parallel regions.
- The directive must appear after the declaration of listed variables/common blocks. Each thread then gets its own copy of the variable/common block, so data written by one thread is not visible to other threads.
- On first entry to a parallel region, data in THREADPRIVATE variables and common blocks should be assumed undefined, unless a COPYIN (discussed later) clause is specified in the PARALLEL directive
- THREADPRIVATE variables differ from PRIVATE variables (discussed later) because they are able to persist between different parallel regions of a code.

```
#pragma omp threadprivate (list)
```

Data Scope: THREADPRIVATE Directive

```
#include <iostream>
#include <omp.h>
int  a, b, tid;
float x;
#pragma omp threadprivate(a, x)
main(int argc, char *argv[]) {
/* Explicitly turn off dynamic threads */
  omp_set_dynamic(0);
  std::cout << "1st Parallel Region:"
            << std::endl;
  #pragma omp parallel private(b,tid) {
    tid = omp_get_thread_num();
    a = tid;
    b = tid;
    x = 1.1 * tid +1.0;
    std::cout << "Thread " << tid << ": a,b,x = "
              << a << " " << b << " " << x << std::endl;
  }  /* end of parallel region */
  std::cout << "Master thread doing serial work here" << std::endl;
  std::cout << "2nd Parallel Region:" << std::endl;
  #pragma omp parallel private(tid) {
    tid = omp_get_thread_num();
    std::cout << "Thread " << tid << ": a,b,x = "
              << a << " " << b << " " << x << std::endl;
  }  /* end of parallel region */
}
```

```
1st Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 2:   a,b,x= 2 2 3.200000
Thread 3:   a,b,x= 3 3 4.300000
Thread 1:   a,b,x= 1 1 2.100000
Master thread doing serial work here
2nd Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 3:   a,b,x= 3 0 4.300000
Thread 1:   a,b,x= 1 0 2.100000
Thread 2:   a,b,x= 2 0 3.200000
```

Data Scope Attribute Clauses

- **Understanding data scopes is critically important!**
- **In OpenMP, most variables are shared by default.**
- **Global variables: file scope, static**
- **Private variables: loop indices, stack variables in subroutines called from parallel regions**
- ***OpenMPs Data Scope Attribute Clauses***
  - ***PRIVATE***
  - ***FIRSTPRIVATE***
  - ***LASTPRIVATE***
  - ***SHARED***
  - ***DEFAULT***
  - ***REDUCTION***
  - ***COPYIN***

Data Scope Attribute Clauses

- **Data Scope Attribute Clauses are used in conjunction with several directives (`PARALLEL`, `FOR`, and `SECTIONS`) to control the scoping of enclosed variables.**
- **These constructs provide the ability to control the data environment during execution of parallel constructs.**
  - **They define how and which data variables in the serial section of the program are transferred to the parallel regions of the program (and back)**
  - **They define which variables will be visible to all threads in the parallel regions and which variables will be privately allocated to all threads.**

Data Scope Attribute Clauses: PRIVATE

- **Declares variables in its list to be private to each thread.**
- **A new object of the same type is declared once for each thread in the team**
- **All references to the original object are replaced with references to the new object**
- **Should be assumed to be uninitialized for each thread**

```
private(list)
```

Data Scope Attribute Clauses: FIRSTPRIVATE

- The FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list.
- Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.

```
firstprivate(list)
```

Data Scope Attribute Clauses: LASTPRIVATE

- The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object.
- The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.
- For example, the team member which does the last SECTION of a SECTIONS context performs the copy with its own values

```
lastprivate(list)
```

Data Scope Attribute Clauses: SHARED

- **The SHARED clause declares variables in its list to be shared among all threads in the team.**
- **A shared variable exists in only one memory location and all threads can read or write to that address**
- **It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)**

```
shared(list)
```

Data Scope Attribute Clauses: DEFAULT

- The DEFAULT clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region.
- Specific variables can be exempted from the default using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses
- The C/C++ OpenMP specification does not include `private` or `firstprivate` as a possible default. However, actual implementations may provide this option.
- Using `none` as a default requires that the programmer explicitly scope all variables.
- Only one DEFAULT clause can be specified on a PARALLEL directive

```
default(shared | none)
```

Data Scope Attribute Clauses: REDUCTION

- **The REDUCTION clause performs a reduction operation on the variables that appear in its list.**
- **A private copy for each list variable is created and initialized for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.**
- **The type of a list item must be valid for the reduction operator.**
- **List items/variables can not be declared shared or private.**
- **Reduction operations may not be associative for real numbers.**

```
reduction(operator: list)
```

Data Scope Attribute Clauses: REDUCTION

```
#include <iostream>
#include <omp.h>
main(int argc, char *argv[])  {
  int   i, n = 100, chunk = 10;
  std::vector<float> a(100), b(100);
  float result = 0.0;
  for (i=0; i < n; i++) {
    a[i] = i * 1.0;
    b[i] = i * 2.0;
  }
  #pragma omp parallel for                  \
              default(shared) private(i)    \
              schedule(static,chunk)        \
              reduction(+:result)
  for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
  std::cout << "Final results = " << result << std::endl;
}
```

REDUCTION - Vector Dot Product:

- **Iterations of the parallel loop will be distributed in equal sized blocks to each thread in the team (`schedule static`)**
- **At the end of the parallel loop construct, all threads will add their values of `result` to update the master thread's global copy.**

Data Scope Attribute Clauses: REDUCTION

Operation	C/C++	Initialization
Addition	+	0
Multiplication	*	1
Subtraction	-	0
Logical AND	&&	.true. / 1
Logical OR	||	.false. / 0
AND bitwise	&	all bits on / ~0
OR bitwise	|	0
Exclusive OR bitwise	^	0
Equivalent		.true.
Not Equivalent		.false.
Maximum	max	Most negative #
Minimum	min	Largest positive #

New Quiz

1. What happens if the number of threads and the number of SECTIONs are different? More threads than SECTIONs? Less threads than SECTIONs?
2. Consider the following code:
   Which loops will be collapsed?
   Which loop iteration variables must be made `private`?

```
#pragma omp for collapse(3)
for (i=0; i<imax; i++)
  for (j=0; j<jmax; j++)
    for (k=0; k<kmax; k++)
      for (l=0; l<lmax; l++)
          for (m=0; m<mmax; m++)
```

New Quiz

3. **Consider the following code and substituting XXX with `private`, `firstprivate`, and `lastprivate`: What is the state of `x` before `x=i`? What will the `cout` statement output and why?**

```
#include <iostream>
#include <omp.h>
main(){
  int i, x=44;
  #pragma omp parallel for XXX(x)
  for(i=0;i<=10;i++)
    x=i;
  std::cout << "final x: " << x << std::endl;
}
```

New Quiz

4. **Which STL containers are usually implemented using hashes? What is the advantage of doing so?**
5. **How are elements accessed on a std::list? What is the consequence of this for an OpenMP parallel for?**

nssc@iue.tuwien.ac.at