OpenMP Programming

Piotr Luszczek

OpemMP Overview

Basic Premise of OpenMP

- Make development of threaded code
  - Easy
    - Threads are created, deployed, and destroyed with few code changes
  - Incremental
    - Only some parts of the code may need threading
    - By default, the rest of the code runs sequentially
  - Expose complex features when necessary
    - Direct locking of mutex locks
    - Accessing vector units
    - Using accelerators

Goals of OpenMP

- Standardization
  - Provide a standard among a variety of shared memory architectures/platforms
- 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.
- 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
  - Supports Fortran (77, 90, and 95, 2003), C, and C++
- Public forum for API and membership

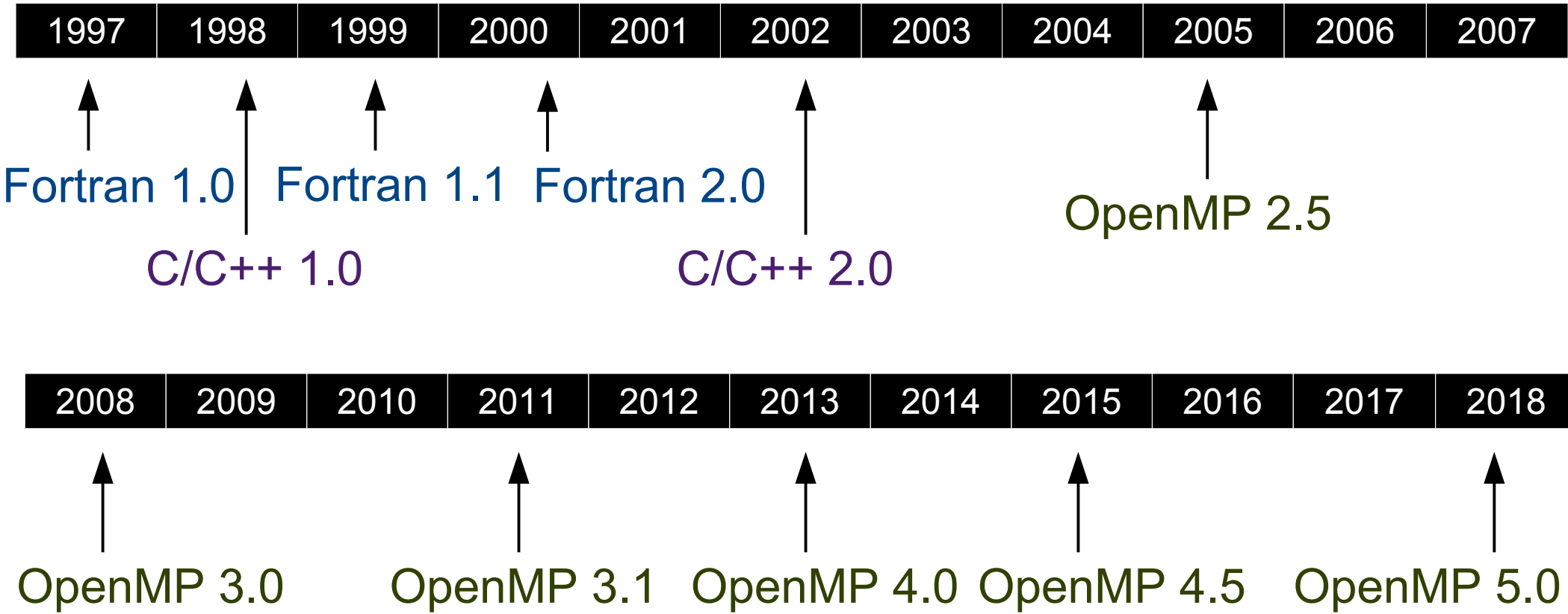
OpenMP is NOT...

- Not meant for distributed memory parallel systems (by itself)
  - It is often combined with MPI
- Not implemented identically by all vendors
  - Despite a lot of code reuse and sharing of ideas
- Not guaranteed to use shared memory efficiently
- Not required to check for:
  - data dependencies
  - data conflicts
  - race conditions, or
  - deadlocks
- Not required to check conformance of user code
- Not provide compiler-generated automatic parallelization and/or directives to the compiler to assist such parallelization
- Not providing synchronous I/O to the same file when executed in parallel
  - The programmer is responsible for synchronizing I/O

OpenMP Standard

- OpenMP is an industry standard
  - Freely available at www.openmp.org
  - Open Multi-Processing
- Supported languages
  - C
  - C++
  - Fortran
- Enabled during compilation
  - Intel, GNU, LLVM, …
    - -fopenmp
    - -fopenmp=gomp
    - -fopenmp=iomp
  - Visual Studio
    - /openomp
- OpenMP is supported by modern compilers
  - GNU gcc, gfortran
    - GOMP
  - LLVM clang
    - In progress
  - Intel icc, ifc
    - iomp library
  - IBM xlc, xlf
  - Cray compiler
  - PGI compiler (now part of NVIDIA): pgcc, pgfortran
  - Microsoft Visual Studio
    - Not in Express version in 2010

OpenMP Release History

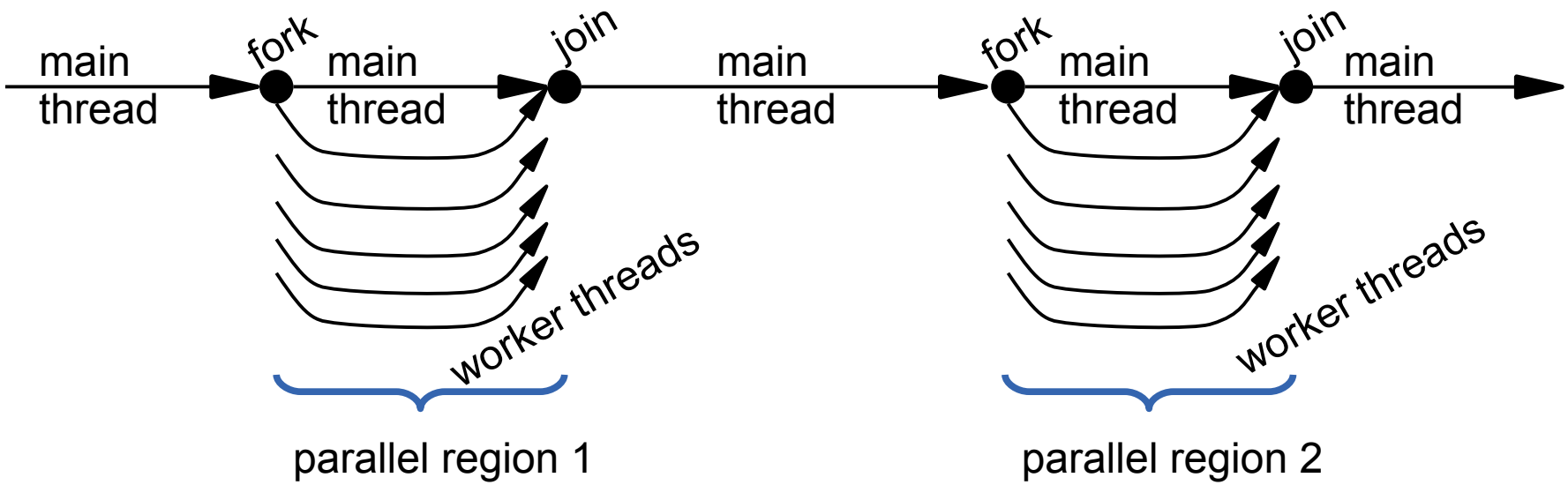


OpenMP Functionality Over Time

- OpenMP 1
  - Basic functionality of loop parallelization
- OpenMP 2
  - Enhancements to loop parallelization
- OpenMP 2.5
  - Unification of Fortran and C/C++ standards into a single document
- OpenMP 3.0
  - Sibling tasking model
- OpenMP 3.1
  - Fixes
- OpenMP 4.0
  - Data-depend task scheduler
  - Target off-load to hardware accelerators
- OpenMP 4.5
  - Task priorities
- OpenMP 5.0
  - Advanced memory model
  - Asynchronous tasking

OpemMP Parallelism Basics

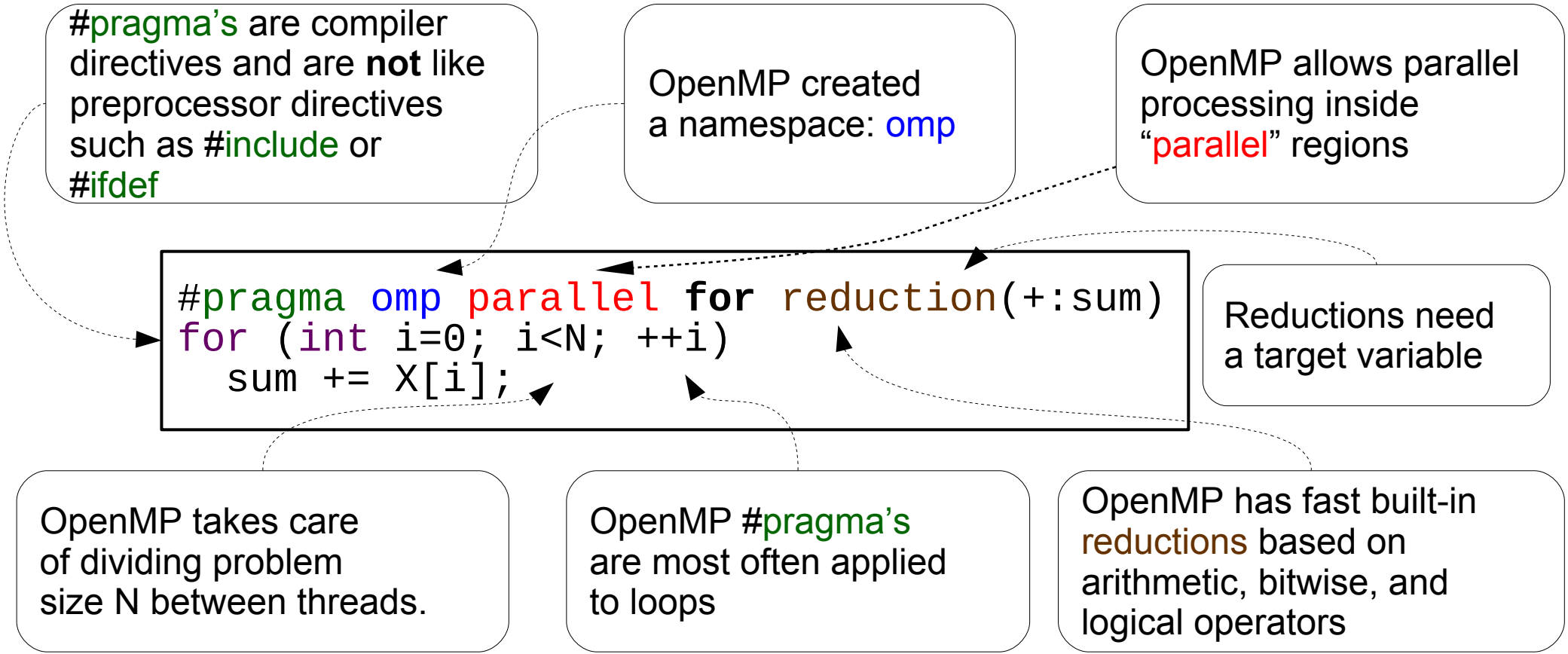
Bulk Synchronous Processing Model of Parallelism



OpenMP Syntax

```
#include <omp.h>
int main (void) {
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 */
/* ... */
}
```

Example: Largest Value (OpenMP)



Separate OpenMP Pragma's

#pragma parallel
marks a parallel region

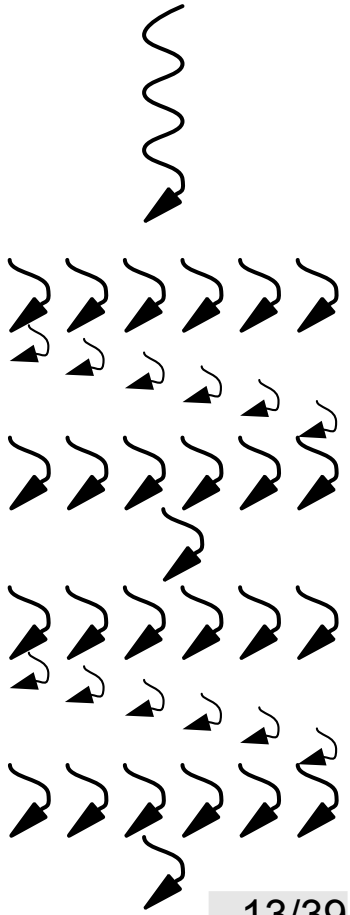
```
#pragma omp parallel
{ // begin of parallel region

#pragma omp for reduction(+:sum)
for (int i=0; i<N; ++i)
  sum += X[i];

#pragma omp for reduction(*:mul)
for (int i=0; i<N; ++i)
  mul *= X[i];

} // end of parallel region
```

There are a few OpenMP pragma's that may occur inside a parallel region.



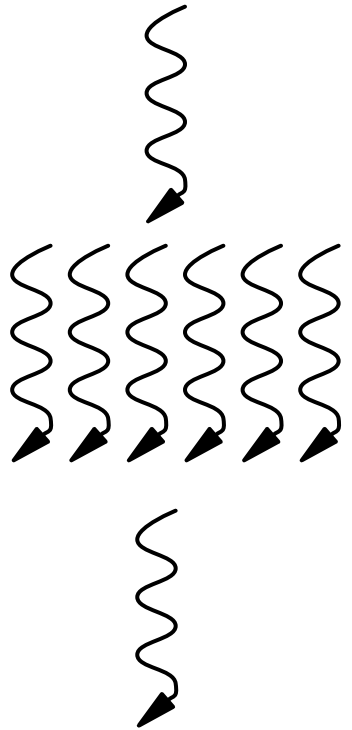
Running Multiple Threads

Parallel regions don't have to use other OpenMP pragmas.

```
#pragma omp parallel
{ // begin of parallel region

    printf( "Hello world!\n" );

} // end of parallel region
```



```
Hello world!
Hello world!
Hello world!
...
```

The output may be **scrambled** but each thread will print once.

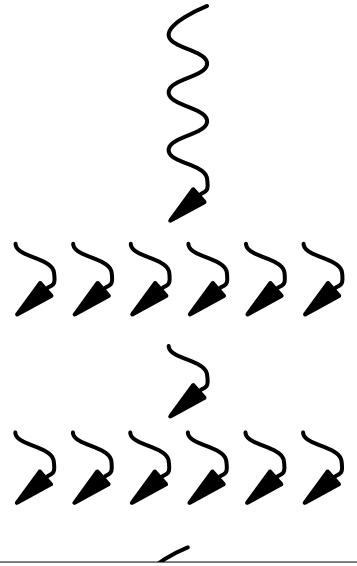
Dealing with I/O: One Thread for I/O

```
#pragma omp parallel
{ // begin of parallel region

    #pragma omp single
    printf( "Hello world!\n" );

} // end of parallel region
```

```
Hello world!
```



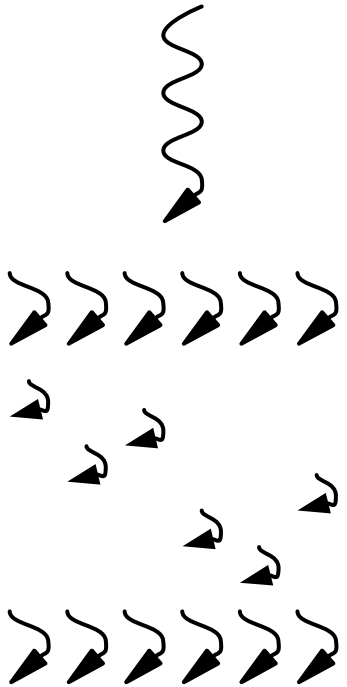
The output will **not** be **scrambled** and a single thread will print once at some point in time.

Dealing with I/O: Mutual Exclusion

```
#pragma omp parallel
{ // begin of parallel region

    #pragma omp critical
    {
    printf( “Hello world!\n” );
    fflush( stdout );
    }

} // end of parallel region
```



```
Hello world!
Hello world!
Hello world!
...
```

The output will **not** be **scrambled** and each thread will print once in some order.

Scoping of OpenMP Directives

```
int main(void) {
/* ... */
#pragma omp parallel
{
/* ... */
#pragma omp for
for (int i = 0; i<N ;++i) {
   /* ... */
   fnct1();
}
/* ... */
fnct2();
/* ... */
}
}
```

static extent: parallel and for directives in the same scope

dynamic extent: parallel region encloses orphaned directives at runtime

```
void fnct1() {
 /* ... */
#pragma omp critical
/* ... */
}
void fnct2() {
 /* ... */
#pragma omp sections
{
/* ... */
}
/* ... */
}
```

orphaned directive: parallel region is not visible

Parallel Region Syntax

- #pragma omp parallel
  - if (scalar expression)
  - private(list)
  - shared(list)
  - default(shared | none)
  - firstprivate(list)
  - reduction(operator:list)
  - copyin(list)
  - num_threads(integer expression)
- structured C/C++ block

- 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 cores on a node.
- Threads are numbered from 0 (master thread) to N − 1

PARALLEL Region Restrictions

- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch into or out of a parallel region
- Only a single IF clause is permitted
- Only a single NUM_THREADS clause is permitted

Nested Parallel Regions

- 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 by your OpenMP implementation:
  - A parallel region nested within another parallel region results in the creation of a new team, consisting of one thread

PARALLEL Region Example

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

Work Sharing Constructs

Work-sharing Constructs and Features

- In order to achieve speedup, the workload has to be divided between threads
- A work-sharing construct is the main method of dividing work
- Work-sharing constructs do not launch new threads
- There is no implied **barrier** upon entry to a work-sharing construct
- There is an implied barrier at the end of a work sharing construct
  - This improves maintaining corrects during development
  - May be disabled with NOWAIT
- A work-sharing construct must be enclosed dynamically within a parallel region
- Work-sharing constructs must be encountered by all threads in a team
  - Or all threads must skip it (with an IF statement, etc.)
- Successive work-sharing constructs must be encountered in the same order by all members of a team

Types of Work-sharing Constructs

- DO/for
  - Shares iterations of a loop across the team
  - Represents a type of “data parallelism”
- SECTIONS
  - Breaks work into separate, discrete sections
  - Each section is executed by a thread.
  - Can be used to implement a type of “functional parallelism”
- SINGLE and MASTER
  - Serializes a section of code
- TASK
  - Dynamically creates a task that may execute in parallel with other tasks
  - Enables Direct Acyclic Graph (DAG) scheduling

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

Restrictions on OpenMP Loops

```
#pragma omp for
for (index = <START> ; index { < <= >= > } <END>; { index++ ++index index-- --index index += inc index -= inc index = index + inc index = inc + index index = index - inc } )
```

Allowed (does not change iteration count)

`continue`

`break` `exit()` `goto` `return`

Not allowed

Changes iteration count

Working Around Restrictions

- The restrictions are in place so that the OpenMP runtime can compute the right schedule for all threads
  - Complicated loops require complicated math or cannot be computed at all
  - STL containers often cannot easily know their size to compute a balanced schedule for threads
- Example: go over powers of two
  - Bit shifting loop:

```
for (i = 1 << 31 ; i != 0 ; i >>= 1) {
}
```

replace with:

```
// compiler “sees” that there are 31 iterations
for (j = 31 ; j > 0 ; --j) {
  i = 1 << j;
}
```

Variable Scope: shared and private

```
#pragma omp parallel for shared(k) private(i)
for (int i = 0; i < 20; ++i)
  printf("%d\n", i+k);
```

nowait

```
for (int i0 = 0; i0 < 10; ++i0)
  printf("%d\n", i0+k);
```

```
for (int i1 = 10; i1 < 20; ++i1)
  printf("%d\n", i1+k);
```

```
#pragma omp barrier
```

unless nowait is used when opening a region

Variable Scope: shared and private

```
int j = 13, k=17;
#pragma omp parallel for private(i) firstprivate(j) lastprivate(k)
for (int i = 0; i < 20; ++i)
  printf("%d\n", k=i+j);
```

```
j0 = j; // copy from master thread
for (int i0 = 0; i0 < 10; ++i0)
  printf("%d\n", k0=i0+j0);
k = k0;
```

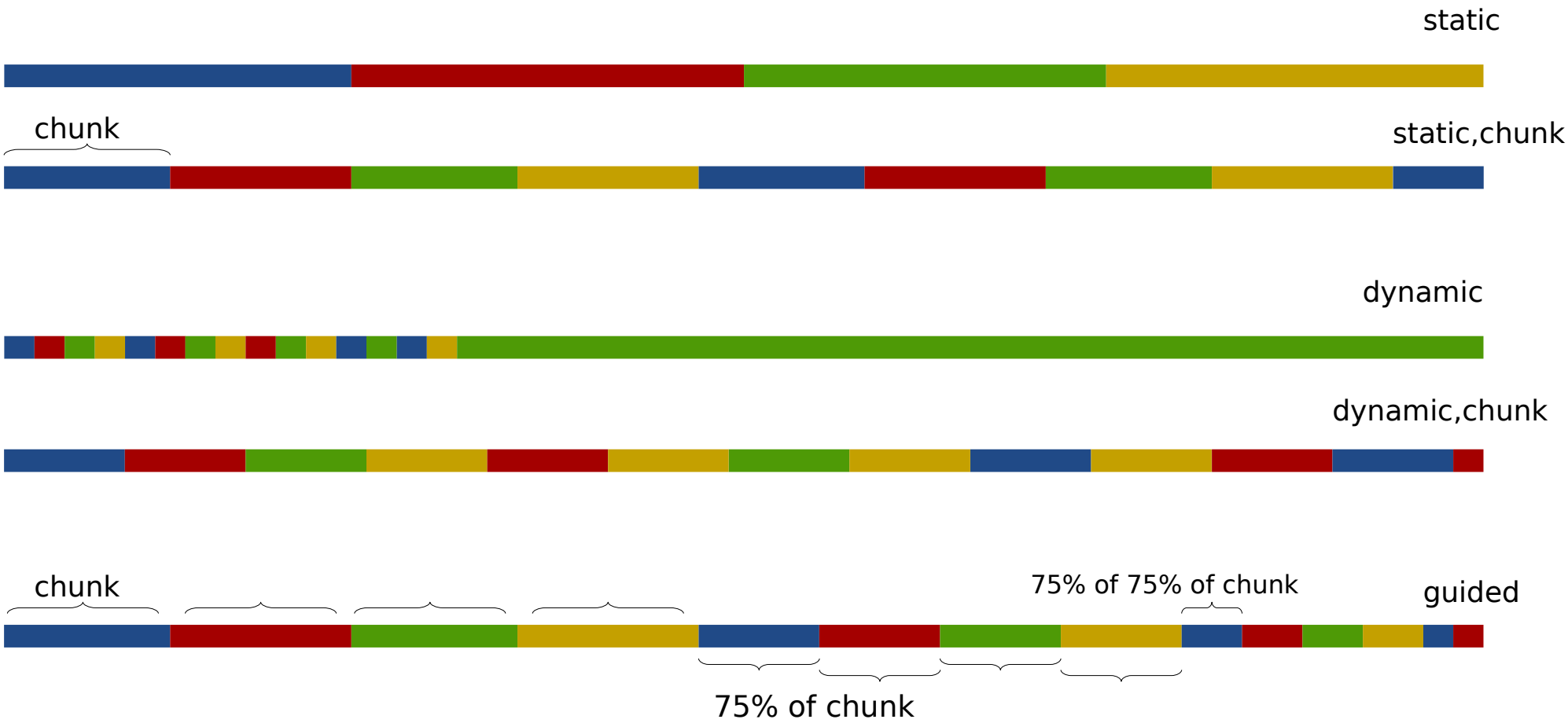
```
j1 = j; // copy from master thread
for (int i1 = 10; i1 < 20; ++i1)
  printf("%d\n", k1=i1+j1);
k = k1;
```

lastprivate can be used for copying private variable out of the "last" thread into the master thread.

Scheduling for Loops

- #pragma omp parallel **for** schedule(<T> [, <CHUNK>])
- Not all loops benefit from the same type of parallelism and/or are load balanced:
  for (N=2; N<100; ++N) matmatmul(N, a, b, c)
- schedule(static) – each thread gets #iters / THREADS
- schedule(static, C) – first thread gets C iterations, second thread gets the next C iterations, ...
- schedule(dynamic) – first thread gets an iteration and then gets another available iteration when its finished
- schedule(dynamic, C) – first thread gets C iterations, ...
- schedule(guided) – chunks exponentially decrease to 1
- schedule(guided,C) – chunks exponentially decrease to C
- schedule(runtime)
  - export/setenv OMP_SCHEDULE “static,1”

OpenMP Schedules' Details



Collapsing Nested Loops

- Multiple loop nests may be collapsed by OpenMP
  - Compiler automatically changes the code into single loop
  - The standard schedule types work on the reorganized loop
    - More opportunities for parallelism
- Consider matrix multiplication:

```
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j];
```

- With OpenMP:

```
#pragma omp parallel for collapse(3)
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j];
```

OpenMP Runtime Functions

Runtime Functions

```
#ifdef _OPENMP
#include <omp.h>
#endif

omp_set_num_threads(13);

printf("%d\n", omp_in_parallel()); // in parallel region? NO

#pragma omp parallel
{

printf("%d\n", omp_in_parallel());    // in parallel region? YES

printf("%f\n", omp_get_wtime());      // wall clock time

printf("%d\n", omp_get_num_threads());// number of active threads

printf("%d\n", omp_get_thread_num());// thread number (starting at 0)

printf("%d\n", omp_get_num_procs()); // number of processors

printf("%f\n", omp_get_wtime());     // wall clock time

}
```

Mutual Exclusion: Directives and Functions

```
#pragma omp parallel for
for (int i=0; i<N; ++i) {
  if (omp_get_thread_num() == 7)
    printf( “%d %d %d\n”, __LINE__, i, omp_get_thread_num() );

  #omp critical
  printf( “%d %d %d\n”, __LINE__, i, omp_get_thread_num() );

  #omp single
  printf( “%d %d %d\n”, __LINE__, i, omp_get_thread_num() );

  #omp master
  printf( “%d %d %d\n”, __LINE__, i, omp_get_thread_num() );
}
omp_lock_t L;
omp_init_lock( &L );
#pragma omp parallel for
{
  if (omp_test_lock( &L )) {printf(“Acquired without waiting\n”); omp_unset_lock(&L);}
  omp_set_lock( &L );
  printf(“Acquired\n”);
  omp_unset_lock( &L );
}
omp_destroy_lock( &L );
```

OpenMP Advanced Topics

OpenMP Memory Model

- OpenMP provides a **relaxed-consistency** and **temporary** view of thread memory
- Threads may cache their data and are not required to maintain exact consistency with the main memory all of the time
  - For efficiency, they rarely synchronize memory state
  - x86 hardware provides strong memory consistency
  - ARM and IBM processors have weaker consistency in hardware
- When all threads view a shared variable identically:
  - The programmer must ensure that the variable is FLUSHed by all threads as needed
  - FLUSH clause may be added to some directives
  - Using FLUSH creates a memory fence
    - It may be expensive because the compiler has to optimize less aggressively
    - It may be expensive because the hardware must synchronize cache memories

OpenMP Implementation Outline

- OpenMP runtime library is implemented using low-level primitives
  - POSIX threads
  - WinThreads
- OpenMP-aware compiler assists in inserting additional code that invokes the OpenMP runtime library
  - At start of the program: initialize the library
    - Just like MPI_Init() initializes MPI
- Every parallel region needs additional code from the compiler, either:
  - Call a separate function with the code inside the region
  - Bring all the threads into the function through long_jump() set_jump()

Two Possible Implementations

```
#pragma parallel
```

- Generated code with separate function:

```
local_args.N = N;
parallelRegion01(
   localArgs);
```

- Generated function:

```
static void
parallelRegion01(
struct Arg local_args){
  for(i=0;
      i<localArgs.N/T;
      ++i) ...
_omp_sum(local_sum,
&sum);
}
```

- Generated code with set_jump():

```
if (main_thread)
set_jmp(parallelReg01);

for (i=0;i<N/T;++i) {
  local_sum += X[i];
}
_omp_sum(local_sum,
&sum);

if (! main_thread)
long_jmp(threadsWait);
```