Introduction to Parallel & Distributed Computing

Lecture 3 Programming with OpenMP

Spring 2024

Instructor: 罗国杰
gluo@pku.edu.cn

Degree of Parallelism

- Definition: number of tasks that can execute in parallel
- May change during program execution
- Metrics
  - maximum degree of parallelism
    - largest # parallel tasks at any point in the execution
  - average degree of parallelism
    - average number of tasks that can be processed in parallel
- Degree of parallelism vs. task granularity
  - inverse relationship

Source: COMP 422 at Rice Univ., Spring 2016.

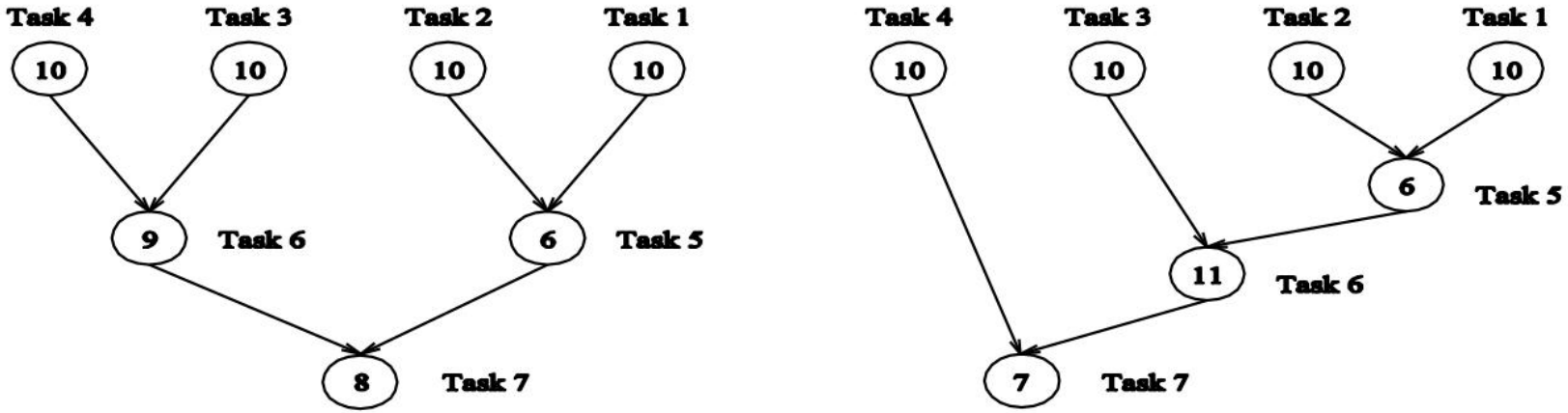
Critical Path

- Edge in task dependency graph represents task serialization
- Critical path = longest weighted path though graph
- Critical path length = lower bound on parallel execution time

Source: COMP 422 at Rice Univ., Spring 2016.

Critical Path Length

Examples: task dependency graphs



Note: number in vertex represents task cost

Questions:

- What are the tasks on the critical path for each dependency graph?
- What is the shortest parallel execution time for each decomposition?
- How many processors are needed to achieve the minimum time?
- What is the maximum degree of concurrency?
- What is the average parallelism?

Limits on Parallel Performance

- **What bounds parallel execution time?**
  - **minimum task granularity**
    - **e.g. dense matrix-vector multiplication $\leq n^2$ parallel tasks**
  - **dependencies between tasks**
  - **parallelization overheads**
    - **e.g., cost of communication between tasks**
  - **fraction of application work that can't be parallelized**
- **Measures of parallel performance**
  - **speedup = T_1/T_p**
  - **parallel efficiency = $T_1/(pT_p)$**

Source: COMP 422 at Rice Univ., Spring 2016.

Task Interaction Graphs

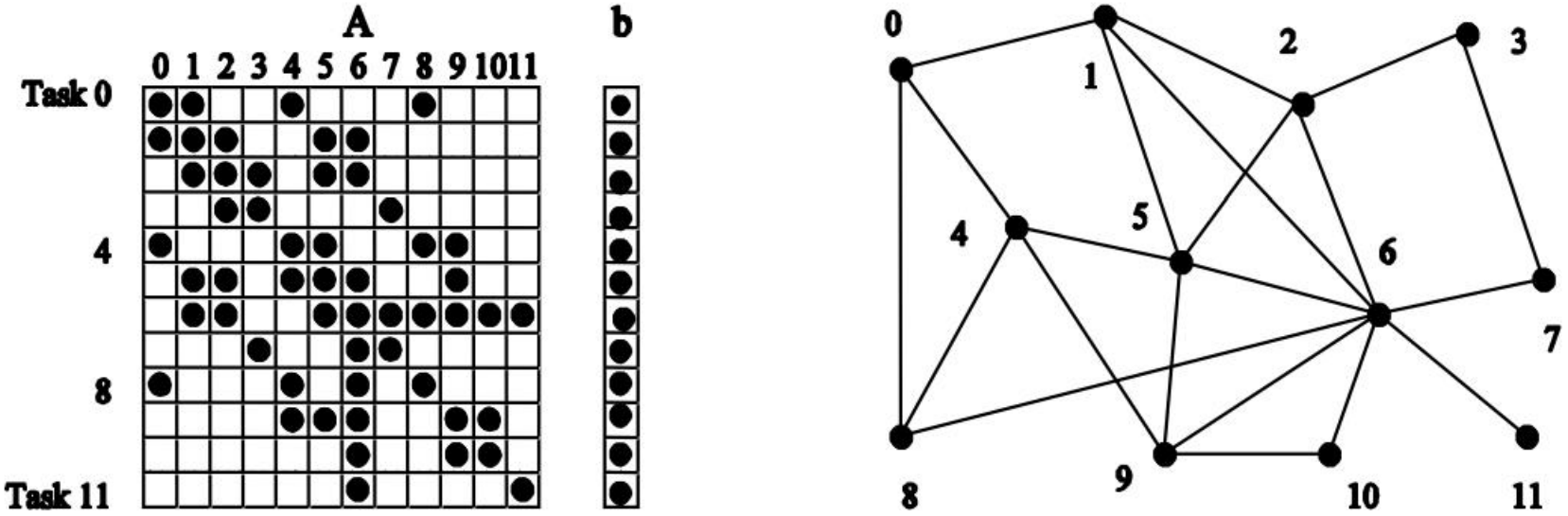
- Tasks generally exchange data with others
  - example: dense matrix-vector multiply
    - if vector b is not replicated in all tasks, tasks will have to communicate elements of b
- Task interaction graph
  - node = task
  - edge = interaction or data exchange
- Task interaction graphs vs. task dependency graphs
  - task interaction graphs represent data dependences
  - task dependency graphs represent control dependences

Source: COMP 422 at Rice Univ., Spring 2016.

Task Interaction Graph Example

- Computation of each result element = independent task
- Only non-zero elements of sparse matrix A participate
- If, b is partitioned among tasks …
  - structure of the task interaction graph = graph of the matrix A

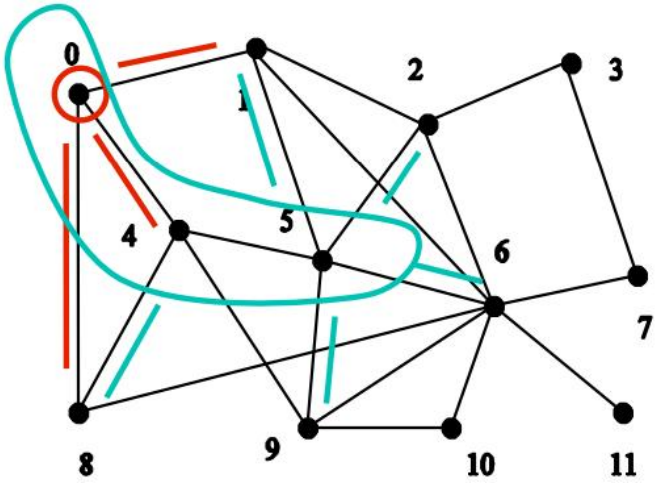
(i.e. the graph for which A represents the adjacency structure)



Source: COMP 422 at Rice Univ., Spring 2016.

Interaction Graphs, Granularity, & Communication

- Finer task granularity increases communication overhead
- Example: sparse matrix-vector product interaction graph



- Assumptions:
  - each node takes unit time to process
  - each interaction (edge) causes an overhead of a unit time
- If node 0 is a task: communication = 3; computation = 4
- If nodes 0, 4, and 5 are a task: communication = 5; computation = 15
  - coarser-grain decomposition → smaller communication/computation

Source: COMP 422 at Rice Univ., Spring 2016.

Tasks, Threads, and Mapping

- **Generally**
  - # of tasks > # threads available
- **Why threads rather than cores?**
  - aggregate tasks into threads
    - thread = processing or computing agent that performs work
    - assign collection of tasks and associated data to a thread
  - OS maps threads to physical cores
    - OS often enable one to bind a thread to a core
    - for multithreaded cores, the OS can bind multiple software threads to distinct hardware threads associated with a core

Source: COMP 422 at Rice Univ., Spring 2016.

Tasks, Threads, and Mapping

- **Mapping tasks to threads is critical for parallel performance**
- **On what basis should one choose mappings?**
  - **using task dependency graphs**
    - **schedule independent tasks on separate threads**

      minimum idling

      optimal load balance
  - **using task interaction graphs**
    - **want threads to have minimum interaction with one another**

      minimum communication

Source: COMP 422 at Rice Univ., Spring 2016.

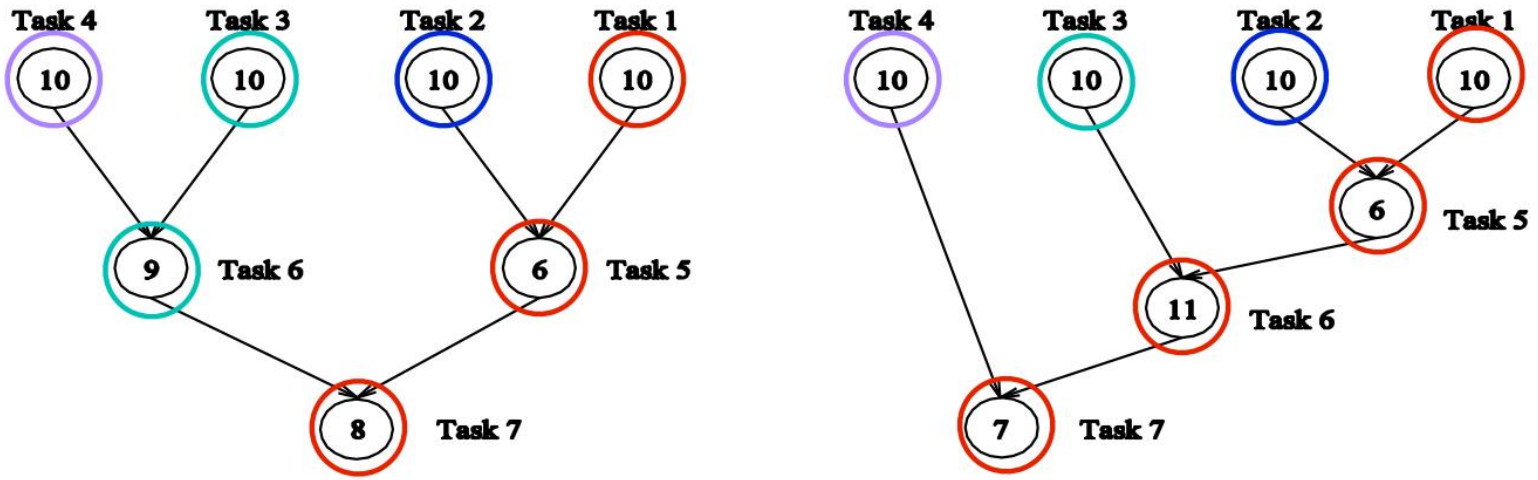
Tasks, Threads, and Mapping

A good mapping must minimize parallel execution time by

- **Mapping independent tasks to different threads**
- **Assigning tasks on critical path to threads ASAP**
- **Minimizing interactions between threads**
  - **map tasks with dense interactions to the same thread**
- **Difficulty: criteria often conflict with one another**
  - **e.g. no decomposition minimizes interactions but no speedup!**

Source: COMP 422 at Rice Univ., Spring 2016.

Tasks, Threads, and Mapping Example



Example: mapping database queries to threads

- **Consider the dependency graphs in levels**
  - **no nodes in a level depend upon one another**
  - **compute levels using topological sort**
- **Assign all tasks within a level to different threads**

Source: COMP 422 at Rice Univ., Spring 2016.

Topics for Today

- **Basic concepts**
  - dependency graph & interaction graph
  - degree of parallelism & critical path
  - speedup & efficiency
  - threads & mapping
- **OpenMP basics**

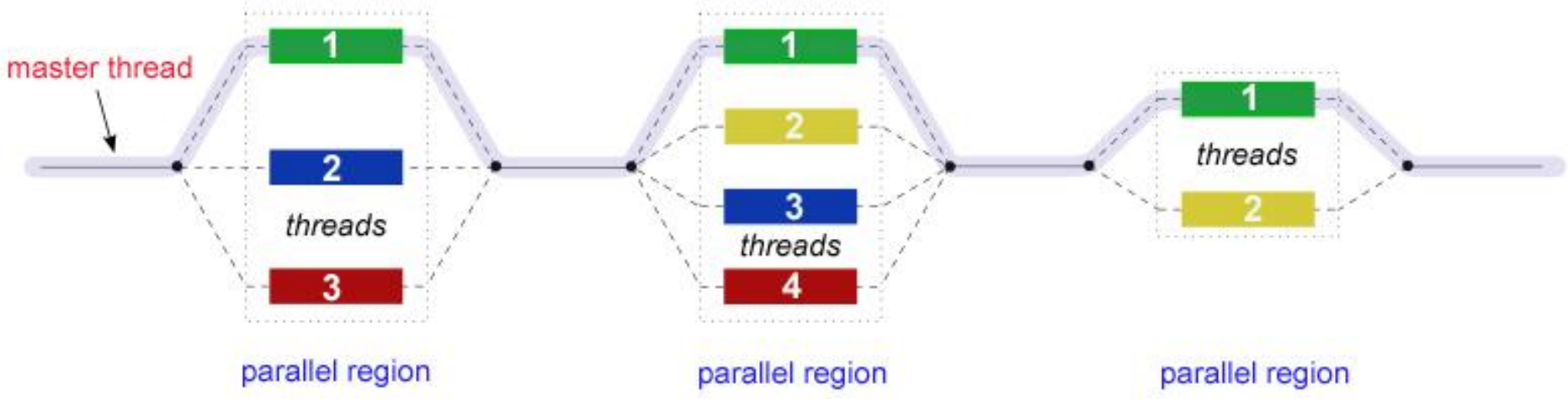
Source: COMP 422 at Rice Univ., Spring 2016.

What is OpenMP?

- An abbreviation for
  - Short version
    - Open Multi-Processing
  - Long version
    - Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia
- OpenMP is an API for parallel programming
  - First developed by the OpenMP Architecture Review Board (1997), now a standard
  - Designed for shared-memory multiprocessors
  - Set of compiler directives, library functions, and environment variables, but not a language
  - Can be used with C, C++, or Fortran
  - Based on fork/join model of threads

Fork-Join Model

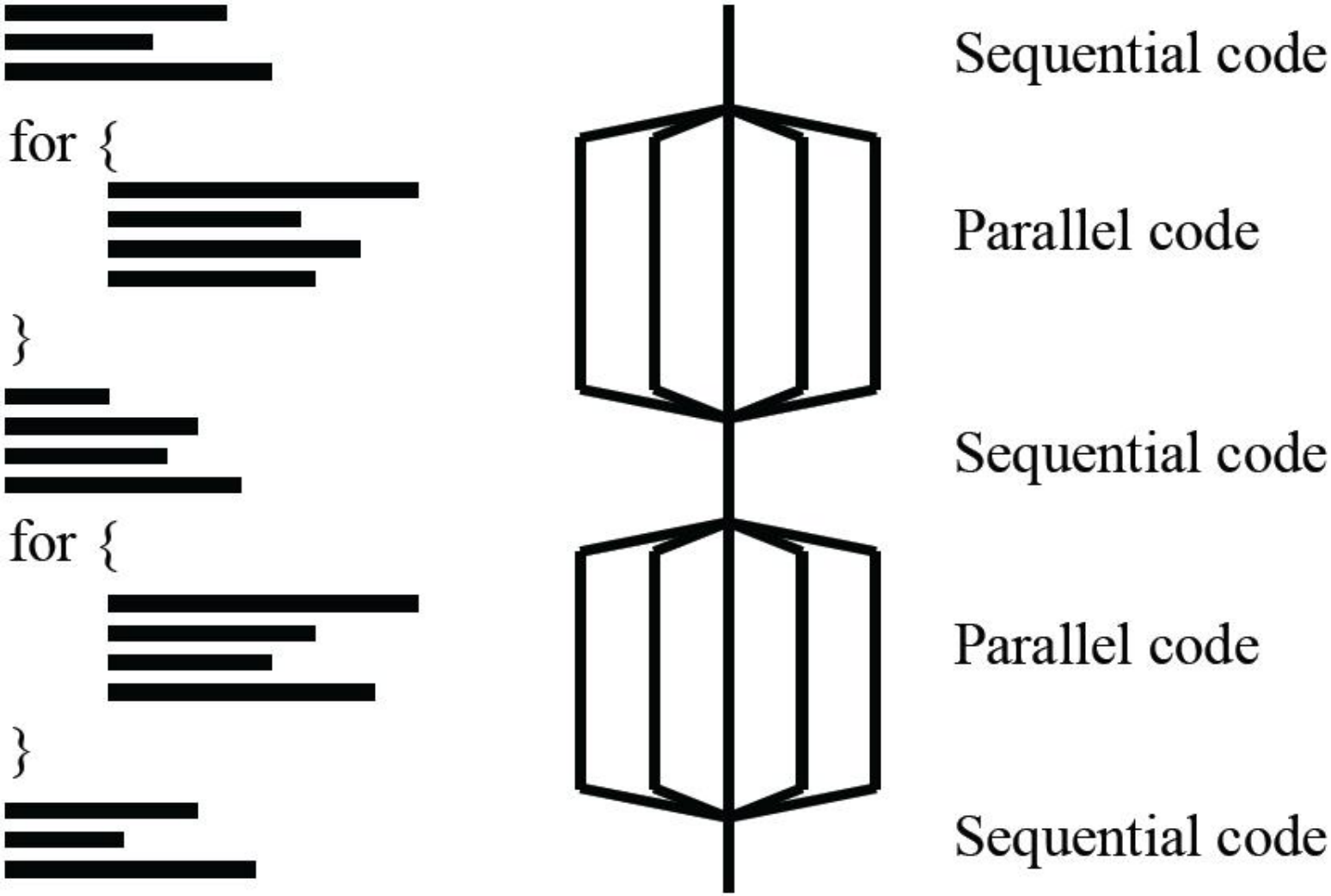
- **FORK**
  - The master thread then creates (or awakens) a team of parallel *threads*.
- **JOIN**
  - When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.



http://en.wikipedia.org/wiki/Flynn's_taxonomy

Blaise Barney (LLNL), Introduction to Parallel Computing

Relating Fork/Join to Code



OpenMP Components

- **Three API components**
  - **Compiler directives**
  - **Runtime library routines**
  - **Environment variables**

http://en.wikipedia.org/wiki/Flynn's_taxonomy

Syntax of Compiler Directives

- *pragma*: a C/C++ compiler directive
  - (other compiler directives: #include, #define, …)
  - Stands for "pragmatic information"
  - A way for the programmer to communicate with the compiler
  - Pragmas are handled by the preprocessor
  - Compilers are free to ignore pragmas
- All OpenMP pragmas have the syntax:

```
#pragma omp <directive-name> [clause, ...]
```

- Pragmas appear immediately *before* relevant construct

Hello World

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
  int nthreads, tid;
  /* Fork a team of threads giving private variables */
#pragma omp parallel private(nthreads, tid)
  {
    /* Obtain thread number */
    tid = omp_get_thread_num();
    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("Hello World from thread = %d\n", tid);
  }  /* All threads join master thread and disband */
}
```

Output – Non-deterministic!

```
[cong@lnxsrvg1 ~/cs133_examples]$ ./a.out
Hello World from thread = 5
Hello World from thread = 7
Number of threads = 16
Hello World from thread = 0
Hello World from thread = 8
Hello World from thread = 3
Hello World from thread = 1
Hello World from thread = 6
Hello World from thread = 15
Hello World from thread = 14
Hello World from thread = 10
Hello World from thread = 2
Hello World from thread = 9
Hello World from thread = 11
Hello World from thread = 13
Hello World from thread = 12
Hello World from thread = 4
[cong@lnxsrvg1 ~/cs133_examples]$ ./a.out
Hello World from thread = 6
Hello World from thread = 10
Hello World from thread = 8
Number of threads = 16
Hello World from thread = 0
Hello World from thread = 5
Hello World from thread = 14
Hello World from thread = 15
Hello World from thread = 13
Hello World from thread = 12
Hello World from thread = 4
Hello World from thread = 3
Hello World from thread = 9
Hello World from thread = 2
Hello World from thread = 7
Hello World from thread = 1
Hello World from thread = 11
```

OpenMP Compilers Perform the Translations to Threads (e.g. pthreads)

```
int a, b;
main() {
  // serial segment
  #pragma omp parallel num_threads (8) private (a) shared (b)
  {
    // parallel segment
  }
  // rest of serial segment
}
```

Sample OpenMP program

```
int a, b;
main() {
  // serial segment

  for (i = 0; i < 8; i++)
      pthread_create (......., internal_thread_fn_name, ...);

  for (i = 0; i < 8; i++)
      pthread_join (.......);

  // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;

  // parallel segment

}
```

Code inserted by the OpenMP compiler

Corresponding Pthreads translation

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Matching Threads with CPUs

- **Function *omp_get_num_procs* returens the number of physical processors available to the parallel program**

```
int omp_get_num_procs(void);
```

- **Function *omp_set_num_threads* allow you to set the number of threads that should be active in parallel sections of code**

```
void omp_set_num_threads(int t);
```

  - **The function can be called with different arguments at different points in the program**

More Runtime Libraries

Routine	Purpose
OMP_SET_NUM_THREADS	Sets the number of threads that will be used in the next parallel region
OMP_GET_NUM_THREADS	Returns the number of threads that are currently in the team executing the parallel region from which it is called
OMP_GET_MAX_THREADS	Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function
OMP_GET_THREAD_NUM	Returns the thread number of the thread, within the team, making this call.
OMP_GET_THREAD_LIMIT	Returns the maximum number of OpenMP threads available to a program
OMP_GET_NUM_PROCS	Returns the number of processors that are available to the program
OMP_IN_PARALLEL	Used to determine if the section of code which is executing is parallel or not
OMP_SET_DYNAMIC	Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions
OMP_GET_DYNAMIC	Used to determine if dynamic thread adjustment is enabled or not
OMP_SET_NESTED	Used to enable or disable nested parallelism
OMP_GET_NESTED	Used to determine if nested parallelism is enabled or not
OMP_SET_SCHEDULE	Sets the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive
OMP_GET_SCHEDULE	Returns the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive
OMP_SET_MAX_ACTIVE_LEVELS	Sets the maximum number of nested parallel regions
OMP_GET_MAX_ACTIVE_LEVELS	Returns the maximum number of nested parallel regions
OMP_GET_LEVEL	Returns the current level of nested parallel regions
OMP_GET_ANCESTOR_THREAD_NUM	Returns, for a given nested level of the current thread, the thread number of ancestor thread
OMP_GET_TEAM_SIZE	Returns, for a given nested level of the current thread, the size of the thread team
OMP_GET_ACTIVE_LEVEL	Returns the number of nested, active parallel regions enclosing the task that contains the call
OMP_IN_FINAL	Returns true if the routine is executed in the final task region; otherwise it returns false
OMP_INIT_LOCK	Initializes a lock associated with the lock variable
OMP_DESTROY_LOCK	Disassociates the given lock variable from any locks
OMP_SET_LOCK	Acquires ownership of a lock
OMP_UNSET_LOCK	Releases a lock
OMP_TEST_LOCK	Attempts to set a lock, but does not block if the lock is unavailable

2147483647

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

Pragma: *parallel for*

- **The compiler directive**

```
#pragma omp parallel for
```

tells the compiler that the *for* loop which immediately follows can be executed in parallel

  - **The number of loop iterations must be computable at run time before loop executes**
  - **Loop must not contain a *break*, *return*, or *exit***
  - **Loop must not contain a *goto* to a label outside loop**

Example: parallel for

```
int a[1000], b[1000], s[1000];
…
#pragma omp parallel for
for (i = 0; i < 1000; i ++)
  s[i] = a[i] + b[i];
```

- **Threads are assigned an independent set of iterations**
- **Threads must wait at the end of construct**

Which Loop to Make Parallel?

```
#define N g.vertices()
// input: Graph g
// output: all-pair shortest distance dist[N][N]
void FloydWarshall(Graph g, float dist[N][N])
  for (int i = 0; i< N ; i++)
    for (int j = 0; j < N; j++)
      dist[i][j] = +∞;
  // initialize dist[][] as the 1-hop distance matrix
  for (int i = 0; i < g.edges(); i++) {
    int u = g.edge(i).source();
    int v = g.edge(i).destination();
    dist[u][v] = g.edge(i).weight();
  }
  // kernel of Floyd-Warshall algorithm
  for (int k = 0; k < N; k++) {
    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        dist[i][j] = min(
            dist[i][j], dist[i][k] + dist[k][j]);
}
```

Which Loop to Make Parallel?

```
#define N g.vertices()
// input: Graph g
// output: all-pair shortest distance dist[N][N]
void FloydWarshall(Graph g, float dist[N][N])
  for (int i = 0; i< N ; i++)
    for (int j = 0; j < N; j++)
      dist[i][j] = +∞;
  // initialize dist[][] as the 1-hop distance matrix
  for (int i = 0; i < g.edges(); i++) {
    int u = g.edge(i).source();
    int v = g.edge(i).destination();
    dist[u][v] = g.edge(i).weight();
  }
  // kernel of Floyd-Warshall algorithm
  for (int k = 0; k < N; k++) {    // loop-carried dependences
    for (int i = 0; i < N; i++)    // can execute in parallel
      for (int j = 0; j < N; j++) // can execute in parallel
        dist[i][j] = min(
            dist[i][j], dist[i][k] + dist[k][j]);
}
```

Minimizing Threading Overhead

- **There is a fork/join for every instance of**

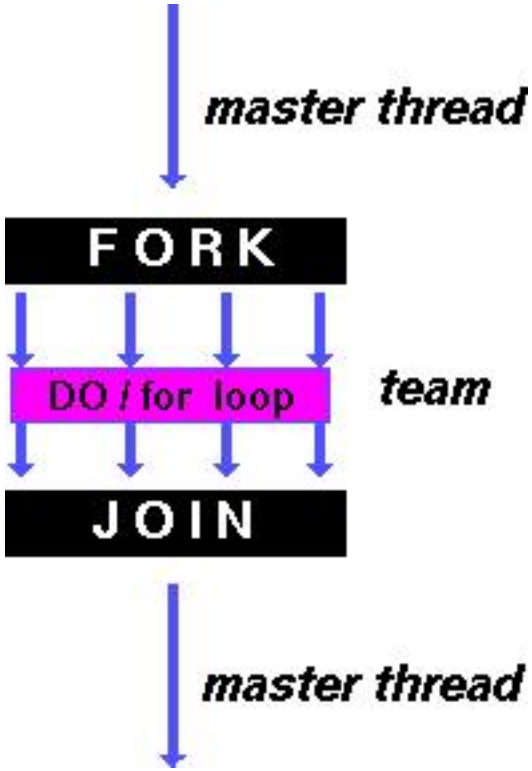
```
#pragma omp parallel for
for (...) {
   ...
}
```

- **Since fork/join is a source of overhead, we want to maximize the amount of work done for each fork/join; i.e., the *grain size***
- **Hence we choose to make the middle loop parallel**

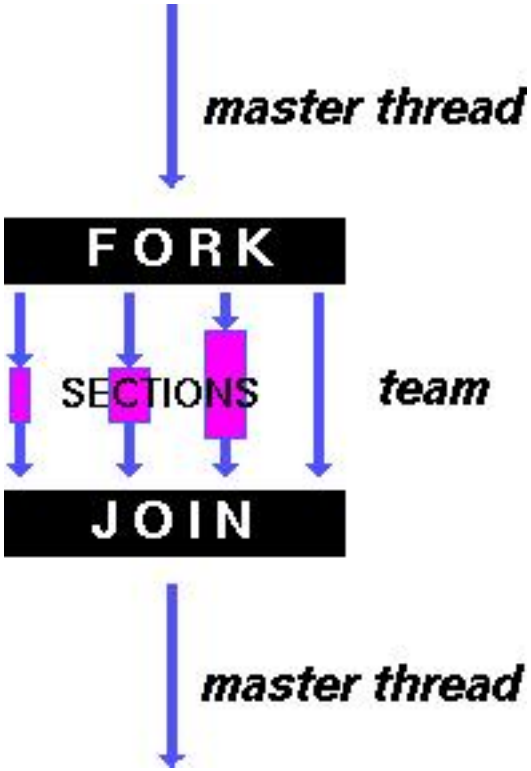
Example: Floy-Warshall Algorithm

```
#define N g.vertices()
// input: Graph g
// output: all-pair shortest distance dist[N][N]
void FloydWarshall(Graph g, float dist[N][N])
  for (int i = 0; i< N ; i++)
    for (int j = 0; j < N; j++)
      dist[i][j] = +∞;
  // initialize dist[][] as the 1-hop distance matrix
  for (int i = 0; i < g.edges(); i++) {
    int u = g.edge(i).source();
    int v = g.edge(i).destination();
    dist[u][v] = g.edge(i).weight();
  }
  // kernel of Floyd-Warshall algorithm
  for (int k = 0; k < N; k++) {   // loop-carried dependences
#pragma omp parallel for
    for (int i = 0; i < N; i++)   // can execute in parallel
      for (int j = 0; j < N; j++) // can execute in parallel
        dist[i][j] = min(
            dist[i][j], dist[i][k] + dist[k][j]);
}
```

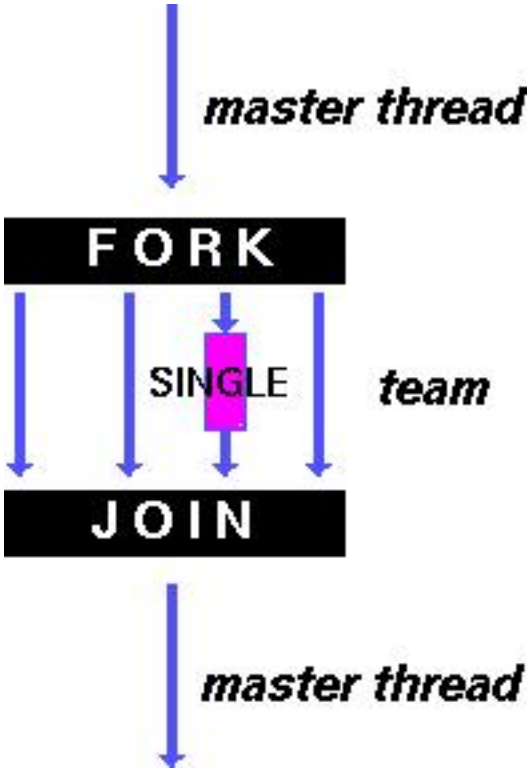
Work-Sharing Constructs



a type of "data parallelism".



a type of "functional parallelism".



serializes a section of code

Pragma: *parallel*

- **In the effort to increase grain size, sometimes the code that should be executed in parallel goes beyond a single *for* loop**
  - **The *parallel* pragma is used when a block of code should be executed in parallel**
  - **SPMD-style programming**
    - **Single program, multiple data**

Work-Sharing Constructs: Pragma *for*

- **The *for* pragma is used inside a block of code already marked with the *parallel* pragma**
  - **It indicates a *for* loop whose iterations should be divided among the active threads**
  - **There is a *barrier synchronization* of the threads at the end of the *for* loop**

Pragma parallel *and Pragma* for

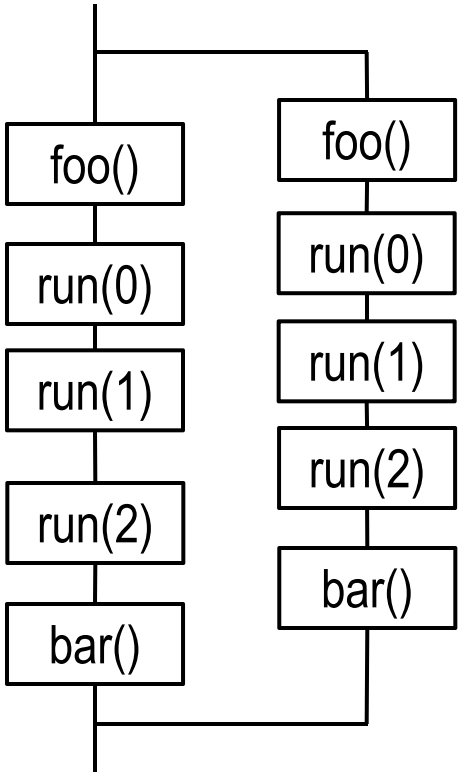
- #pragma omp for
  - **Used inside a block of code already marked by *parallel***
  - **Distribute the iterations to the active threads**

```
#pragma omp parallel \
 num_threads(2)
{
  foo();
  #pragma omp for
  for (int i=0; i<3; i++)
    run(i);
  bar();
}
```

```
#pragma omp parallel \
 num_threads(2)
{
  foo();
  for (int i=0; i<3; i++)
    run(i);
  bar();
}
```

Pragma parallel *and Pragma* for

- #pragma omp for
  - **Used inside a block of code already marked by *parallel***
  - **Distribute the iterations to the active threads**

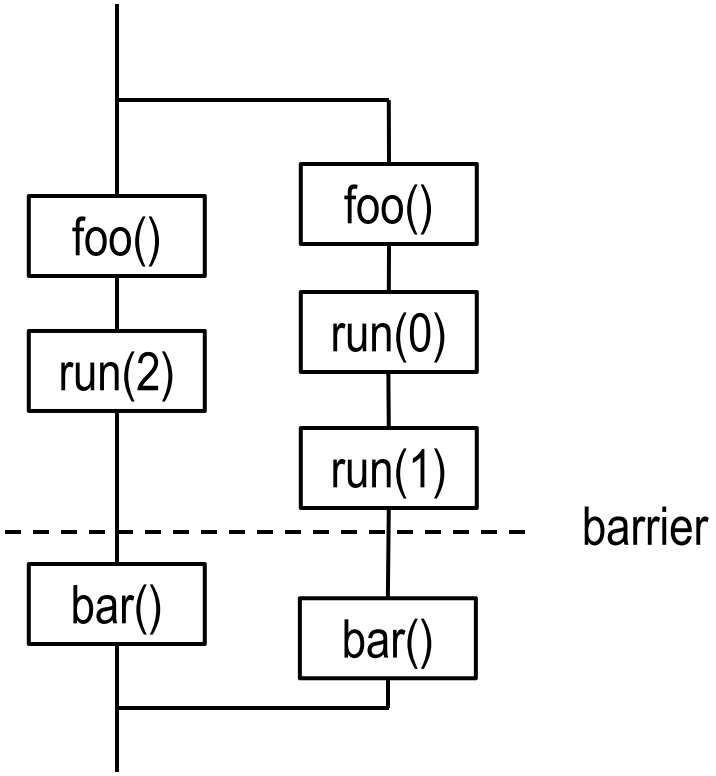



```
#pragma omp parallel \
 num_threads(2)
{
   foo();
   for (int i=0; i<3; i++)
     run(i);
   bar();
}
```

Pragma parallel *and Pragma* for

- #pragma omp for
  - **Used inside a block of code already marked by *parallel***
  - **Distribute the iterations to the active threads**

```
#pragma omp parallel \
 num_threads(2)
{
   foo();
   #pragma omp for
   for (int i=0; i<3; i++)
     run(i);
   bar();
}
```



Work-Sharing Constructs: Pragma *single*

- **The *single* pragma is used inside a parallel block of code**
  - **It tells the compiler that only a single thread should execute the statement or block of code immediately following**
  - **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 block**
    - **unless a nowait clause is specified (in next lectures)**

Example: *master* **and** *single nowait*

```
tid = omp_get_thread_num();
if (tid == 0) {
  nthreads = omp_get_num_threads();
  printf("Number of threads = %d\n", nthreads);
}
```

=

```
#pragma omp master
{
  nthreads = omp_get_num_threads();
  printf("Number of threads = %d\n", nthreads);
}
```

≈

```
#pragma omp single nowait
{
  nthreads = omp_get_num_threads();
  printf("Number of threads = %d\n", nthreads);
}
```

Work-Sharing Constructs: Pragma sections and section

- **Directive** sections **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.**

Example: sections and section

```
#pragma omp parallel shared(a,b,c,d) private(i)
{
#pragma omp sections
  {
#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 section */
```

Clause: *reduction*

- **Reductions are so common that OpenMP provides a reduction clause for the *parallel*, *for*, and *sections***

```
#pragma omp ... reduction (op : list)
```

  - **A private copy of each list variable is created and initialized depending on the *op***
    - **The identity value *op* (e.g., 0 for addition)**
  - **These copies are updated locally by threads**
  - **At end of construct, local copies are combined through *op* into a single value and combine the value in the original shared variable**

Intel, "Introduction to Parallel Programming," 2009

C/C++ Reduction Operation

- Reduction with an associative binary operator Þ

$$a_1 Þ a_2 Þ a_3 Þ \ldots Þ a_n$$

- A range of associative and commutative operators can be used with reduction
- Initial values are the ones that make sense

Operator	Initial Value
+	0
*	1
-	0
^	0

Operator	Initial Value
&	~0
|	0
&&	1
||	0

Intel, "Introduction to Parallel Programming," 2009

Reduction: a Normal Example

```
int sum = 3;
int prod = 5;
#pragma omp parallel for \
  reduction(+:sum)  \
  reduction(*:prod) \
  num_threads(2)
for (int i=1; i<=3; ++i) {
  int tid =
    omp_get_thread_num();
  sum += i;
  prod *= i;
  printf("thread(%d) "
         "sum=%d prod=%d\n",
         tid, sum, prod);
}
printf("results: "
       "sum=%d prod=%d\n",
       sum, prod);
```

- **Assume**
  - **thread 0 executes the 1st and 2nd iterations, and**
  - **thread 1 executes the 3rd iteration**
- **Possible outputs**

```
thread(1) sum=3 prod=3
thread(0) sum=1 prod=1
thread(0) sum=3 prod=2
results: sum=9 prod=30
```

Reduction: an Artificial Example

```
int sum = 3;
int prod = 5;
#pragma omp parallel for \
  reduction(+:sum)  \
  reduction(*:prod) \
  num_threads(2)
for (int i=1; i<=3; ++i) {
  int tid =
    omp_get_thread_num();
  sum |= i;
  prod += i;
  printf("thread(%d) "
         "sum=%d prod=%d\n",
         tid, sum, prod);
}
printf("results: "
       "sum=%d prod=%d\n",
       sum, prod);
```

- **Assume**
  - **thread 0 executes the 1st and 2nd iterations, and**
  - **thread 1 executes the 3rd iteration**
- **Possible outputs**

```
thread(1) sum=3 prod=4
thread(0) sum=1 prod=2
thread(0) sum=3 prod=4
results: sum=9 prod=80
```

What Happened to Reduction Variables?

```
int sum = 3;
int prod = 5;
#pragma omp parallel for \
  reduction(+:sum)  \
  reduction(*:prod) \
  num_threads(2)
for (int i=1; i<=3; ++i) {
  int tid =
    omp_get_thread_num();
  sum |= i;
  prod += i;
  printf("thread(%d) "
         "sum=%d prod=%d\n",
         tid, sum, prod);
}
printf("results: "
       "sum=%d prod=%d\n",
       sum, prod);
```

- **initial** $sum_{private} = 0$
  - **for the reduction op. +**
- **initial** $prod_{private} = 1$
  - **for the reduction op. ***
- **At the end of** parallel for **with the** reduction **clause**
  - $sum_{shared} += \Sigma_{thread} sum_{private}$
  - $prod_{shared} *= \Pi_{thread} prod_{private}$

Orphaned for in a Parallel Region

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

#define VECLEN 100
float a[VECLEN], b[VECLEN], sum;

float dotprod () {
  int i,tid;
  tid = omp_get_thread_num();
#pragma omp for reduction(+:sum)
  for (i=0; i < VECLEN; i++) {
    sum = sum + (a[i]*b[i]);
    printf(
      "  tid= %d i=%d\n",
      tid, i);
  }
}
```

```
int main (int argc, char *argv[])
{
  int i;
  for (i=0; i < VECLEN; i++)
    a[i] = b[i] = 1.0 * i;

  sum = 0.0;
#pragma omp parallel
  dotprod(); // run in parallel
  printf("Sum = %f\n",sum);

  sum = 0.0;
  dotprod(); // run in sequential
  printf("Sum = %f\n",sum);

  return 0;
}
```

Strengths and Weaknesses of OpenMP

- **Strengths**
  - Incremental parallelization & sequential equivalence
  - Well-suited for domain decompositions
  - Available on *nix and Windows
- **Weaknesses**
  - Not well-tailored for functional decompositions
  - Compilers do not have to check for such errors as deadlocks and race conditions

Further Readings

- OpenMP resources at the official website
  - https://www.openmp.org/resources/
- LLNL's OpenMP training materials
  - https://hpc-tutorials.llnl.gov/openmp/