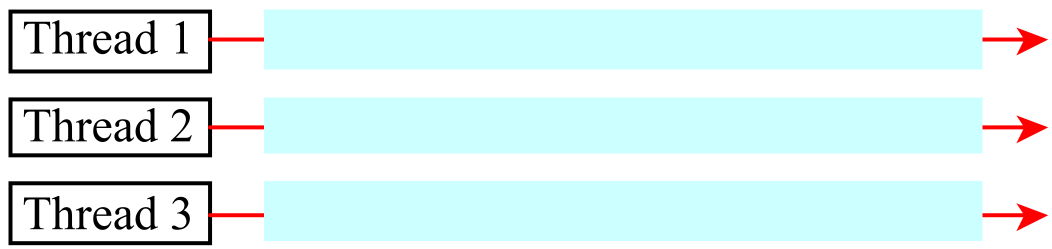
Using OpenMP : Portable Shared Memory Parallel Programming

Multi-Threading

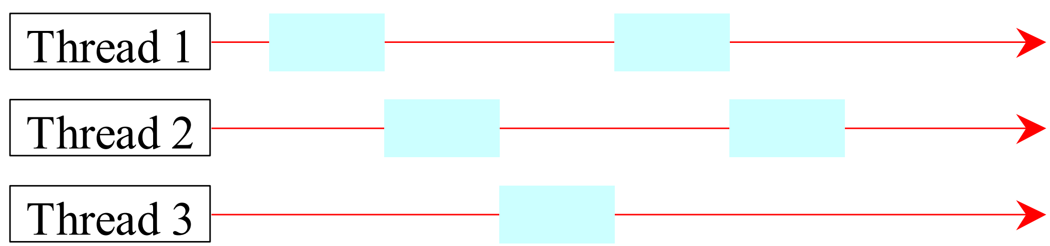
C/C++ supports the creation of programs with concurrent flows of control.

Threads -- independent flows of control.

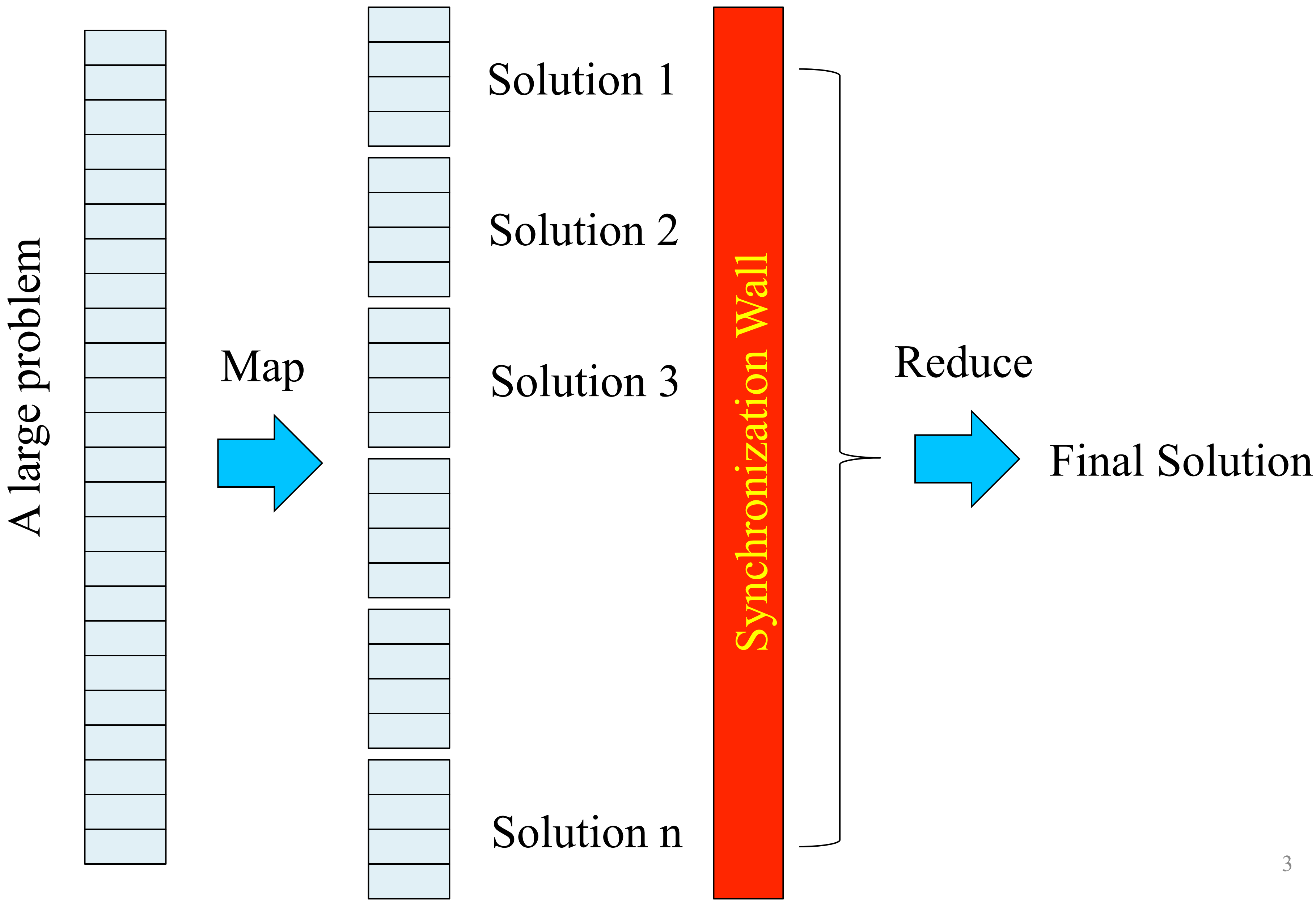
Multiple threads on multiple CPUs



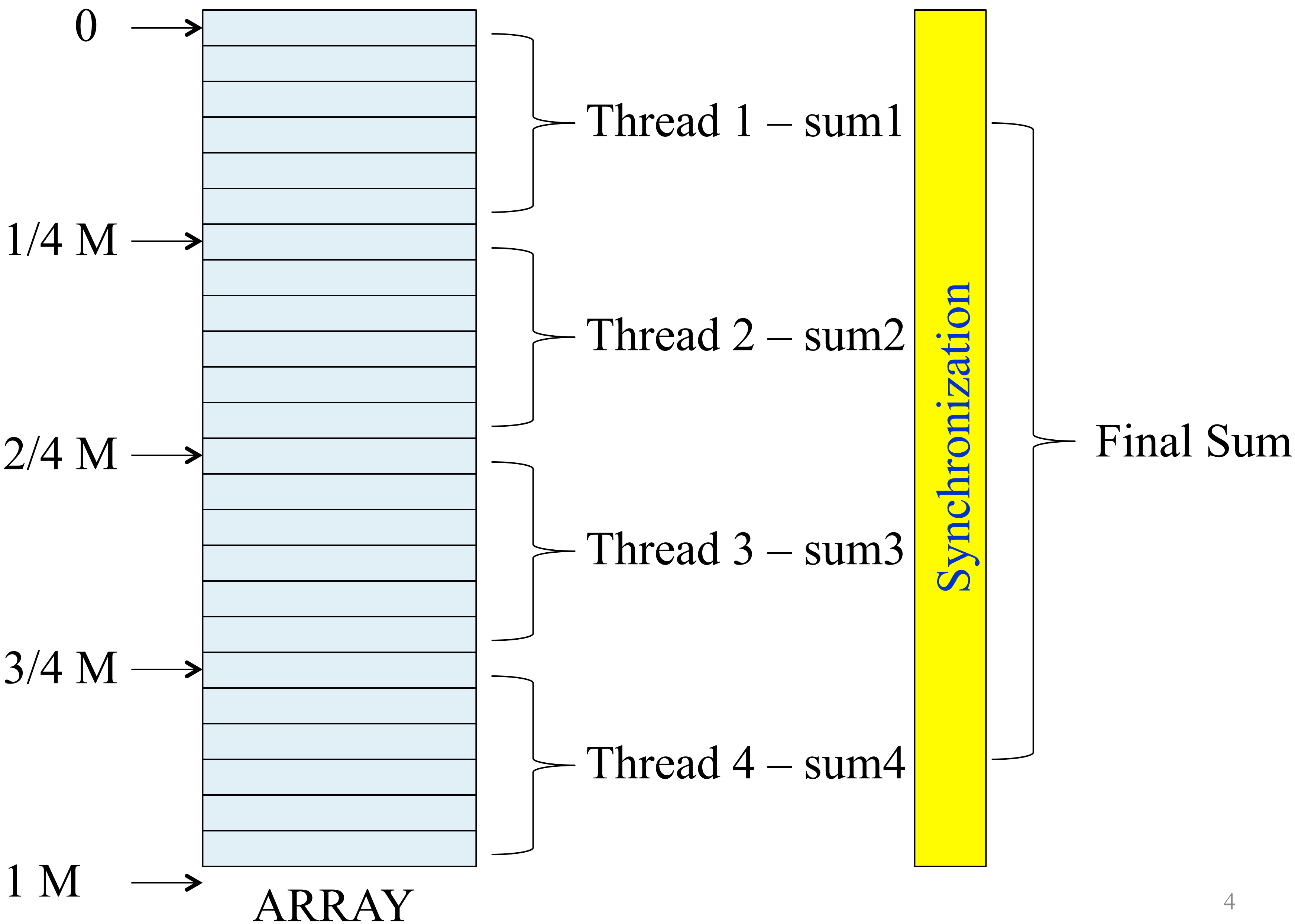
Multiple threads sharing a single CPU



Parallel Computing



Adding a Million Numbers



Multi-Threading using OpenMP (Open Multiprocessing)

In the execution time, we can request the number of threads to be used.

Based on their availability, it might execute with the number of threads, or less.

How many prime numbers are between 2 and N?

```
int prime_number ( int n )
{
  int i, j, prime, total = 0;
  for ( i = 2; i <= n; i++ )
  {
    prime = 1;
    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )    //if at least one number can divide i, then i is not a prime
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  return total;
}
```

```
Run with 1 thread.


  Number of processors available = 4
  Number of threads =              1

  Call PRIME_NUMBER to count the primes from 1 to N.

         N        Pi          Time

         5         3       0.000077
        50        15       0.000004
       500        95       0.000090
      5000       669       0.005964
     50000      5133       0.460443
    500000     41538      37.758831

Run with 2 threads.


  Number of processors available = 4
  Number of threads =              2

  Call PRIME_NUMBER to count the primes from 1 to N.

         N        Pi          Time

         5         3       0.002848
        50        15       0.002694
       500        95       0.002800
      5000       669       0.007945
     50000      5133       0.341860
    500000     41538      27.729894

Run with 4 threads.


  Number of processors available = 4
  Number of threads =              4

  Call PRIME_NUMBER to count the primes from 1 to N.

         N        Pi          Time

         5         3       0.005415
        50        15       0.002687
       500        95       0.002782
      5000       669       0.006338
     50000      5133       0.287292
    500000     41538      16.389043

-bash-3.2$
```

Multi-Threading with OpenMP

Include the header file

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

For the program file prime.c/prime.cpp compile:

```
gcc -fopenmp prime.c
g++ -fopenmp prime.cpp
```

Multi-Threading with OpenMP

Then execute with a number of threads:

```
export OMP_NUM_THREADS=1
export OMP_NUM_THREADS=2
./a.out
```

```
gcc -fopenmp –o prime prime.c
g++ -fopenmp –o prime prime.cpp
```

Then

```
./prime
```

```
int prime_number ( int n )
{
 int i, j, prime, total = 0;

# pragma omp parallel \
 shared ( n ) \
 private ( i, j, prime )

# pragma omp for reduction ( + : total )
 for ( i = 2; i <= n; i++ )
 {
  prime = 1;
  for ( j = 2; j < i; j++ )
  {
   if ( i % j == 0 )
   {
    prime = 0;
    break;
   }
  }
  total = total + prime;
 }
 return total;
}
```

The variable n is shared among threads.
The variables i, j, and prime will have an individual value for each thread.

The work done in a “for” loop inside a parallel region to be divided among threads.
At the end of parallel region, total will be reduced using + operation.

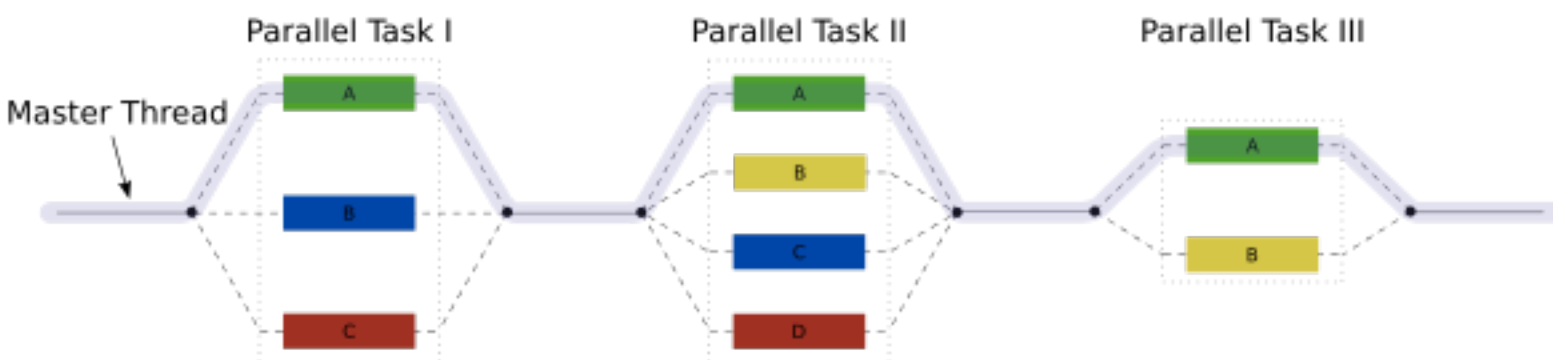
```c
//Checking the number of available processors

  int num_procs = omp_get_num_procs ( );
  int max_threads = omp_get_max_threads ( );

  printf ( "  Number of processors available = %d\n", num_procs );
  printf ( "  Number of threads =              %d\n", max_threads );
```

Overview of OpenMP

- OpenMP API developed to enable shared memory parallel programming
- Uses the fork-join programming model



Writing an OpenMP Program

- For C/C++, pragmas are provided by the OpenMP API to control parallelism
- In OpenMP these are called directives
- They always start with #pragma omp, then a specific keyword that identifies the directive, followed by zero or more clauses

Parallel Construct

- The parallel construct plays a crucial role in OpenMP: a program without a parallel construct will be executed sequentially

```
#pragma omp parallel [clause[[,] clause]…]
    Structured block
```

Sharing Work among Threads

- C/C++ has 3 work-sharing constructs (clauses omitted)

Functionality	C/C++ Syntax
Distribute iterations over the threads	#pragma omp for
Distribute independent work units	#pragma omp sections
Only one thread executes the code block	#pragma omp single

Loop Construct

```
#pragma omp parallel shared(n,a,b) private(i) num_threads(4)
{
        #pragma omp for
        for( i = 0; i < n; i++ )
        {
                a[i] = i;
        }
        #pragma omp for
        for( i = 0; i < n; i++ )
        {
                b[i] = 2 * a[i];
        }
} /* -- End of parallel region -- */
```

Combined Constructs

Full version	Combined construct

#pragma omp parallel
{
#pragma omp for
for-loop
}
	#pragma omp parallel for
	for-loop

#pragma omp parallel
{
#pragma omp sections
{
[#pragma omp section]
structured block
[#pragma omp section]
structured block
…
}
}
	#pragma omp parallel sections
	{
	[#pragma omp section]
	structured block
	[#pragma omp section]
	structured block
	…
	}

Shared Clause

- The shared clause is used to specify which data is shared among the threads executing the region it is associated with
- There is one unique instance of these variables, and each thread can freely read or modify the values

```
#pragma omp parallel shared(n,a) private(i) num_threads(4)
{
        #pragma omp for
        for( i = 0; i < n; i++ )
        {
                a[i] = i;
```

Private Clause

- There may be other data objects in a parallel region or work-sharing construct for which threads should be given their own copies
- The value of private data are undefined upon entry to and exit from the specific construct

```
#pragma omp parallel shared(n,a) private(i) num_threads(4)
{
        #pragma omp for
        for( i = 0; i < n; i++ )
        {
                a[i] = i;
```

Lastprivate Clause

- It ensures that the last value of a data object listed is accessible after the corresponding construct has completed execution
- What is “last”?
  - The object will have the value from the iteration of the loop that would be last in a sequential execution

```
#pragma omp parallel for private(i) lastprivate(a) num_threads(4)
        for( i = 0; i < n; i++ )
        {
                a = i+1;
        } /* -- End of parallel region -- */
//What is the value of a here??
```

Firstprivate Clause

- Variables that are declared to be firstprivate are private variables, but they are pre-initialized with the value of the variable with the same name before the construct

```
int  a = 55;
#pragma omp parallel for firstprivate(a) lastprivate(a) private(i) num_threads(4)
      for( i = 0; i < n; i++ )
      {
            a += i+1;
       } /* -- End of parallel region -- */
```

Default Clause

- The default clause is used to give variables a default data-sharing attribute
  - default(shared) assigns the shared attribute to all variables referenced in the construct
  - If default(none) is specified, the programmer is forced to specify a data-sharing attribute for each variable in the construct

Nowait Clause

- There is an implicit barrier at the end of a work-sharing construct
- The nowait clause overrides that feature of OpenMP
  - i.e., when threads reach the end of the construct, they immediately proceed to perform other work

Schedule Clause

- The schedule clause is supported on the loop construct only
- It is used to control the manner in which loop iterations are distributed over the threads
- schedule(kind[,chunk_size])

```
#pragma omp parallel for default(none) schedule(runtime) private(i) shared(n)
num_threads(4)
        for( i = 0; i < n; i++ )
        {
……….
```

Supported Schedule Kinds

Kind	Description
Static	Iterations are divided into chunks of size chunk_size. Chunks are assigned to the threads statically in a round-robin manner, in the order of thread number. When no chunk_size is specified, the iteration space is divided into chunks approximately equal in size. Each thread is assigned at most one chunk.
dynamic	The iterations are assigned to threads as the threads request them. The thread executes the chunk of iterations (controlled through the chunk_size parameter), then requests another chunk until there are no more chunks to work on. When no chunk_size is specified, it defaults to 1.
guided	Similar to dynamic, but 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 of k (k>1), the size of each chunk is determined the same way, with the restriction that the chunks do not contain fewer then k iterations.
runtime	If this schedule is selected, the decision regarding scheduling kind is made at run time. The schedule and (optional) chunk size are set

Barrier Construct

- A barrier is a point in the execution of a program where threads wait for each other
  - No thread in the team of threads may proceed beyond a barrier until all threads in the team have reached that point

```
#pragma omp parallel private(tid) num_threads(4)
{
        tid = omp_get_thread_num();
        if( tid < omp_get_num_threads()/2 )
                system( "sleep 3" );
        #pragma omp barrier
        (void) print_time( tid, "after" );
}
```

Ordered Construct

- Another synchronization construct, the ordered construct, allows one to execute a structured block within a parallel loop in sequential order
- The syntax is:

```
#pragma omp ordered
    structured block
```

- This ensures that the code within the associated structured block is executed in sequential order
- The code outside this block runs in parallel

Critical Construct

- The critical construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously
  - An optional name may be given to the critical construct

```
#pragma omp parallel shared(n,a,sum) private(tid,local_sum) num_threads(4)
{           local_sum = 0;
            #pragma omp for
            for( i = 0; i < n; i++ )
            {    local_sum += a[i];
            }
            #pragma omp critical (update_sum)
            {      sum += local_sum;
```

Reduction Clause

- OpenMP provides a reduction clause for specifying some forms of recurrence calculations so that they can be performed in parallel without code modification
- The programmer must identify the operations and the variables that hold the result values
  - reduction(operator:list)

```
#pragma omp parallel for default(none) shared(n,a) reduction(+:sum)
        for( i = 0; i < n; i++ )
                sum += a[i];
```

Atomic Construct

- The atomic construct can be an efficient alternative to the critical region
- In contrast to other constructs, it is applied only to the (single) assignment statement that immediately follows it

```
#pragma omp parallel for shared(n,ic) private(i) num_threads(4)
        for( i = 0; i < n; i++ )
        {
                #pragma omp atomic
                        ic++;
        } /* -- End of parallel region -- */
```

Performance Considerations

- T_1 = the execution time of an application on 1 processor
- In an ideal situation, the execution time on P processors should be T_1/P
- If T_P denotes the execution time on P processors,
- then the ratio $S = T_1/T_P$ is called the parallel speedup

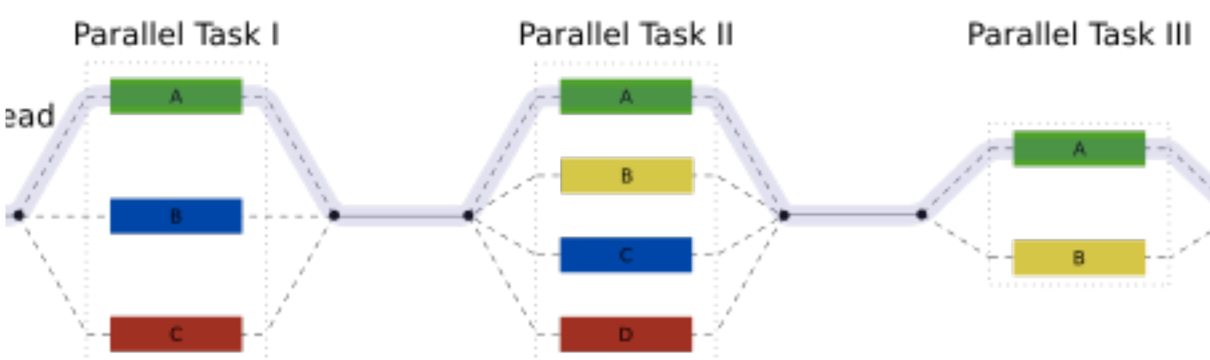
Amdahl's Law

- This effect, known as Amdahl's law, is formulated as

$$S = 1/(f_{par}/P + (1-f_{par}))$$

where f_{par} is the parallel fraction of the code and P is the number of processors

- In the ideal case, when all the code runs in parallel, f_{par} =1 and the expected speedup is P



Amdahl's Law

- Obstacles to speedup is the overhead introduced by
  - forking and joining threads,
  - thread synchronization, and
  - memory accesses