UCIrvine | The Henry Samueli School of Engineering

Shared memory machines; OpenMP

EECS 221: Intro to High-Performance Computing

Aparna Chandramowlishwaran

April 13, 2017

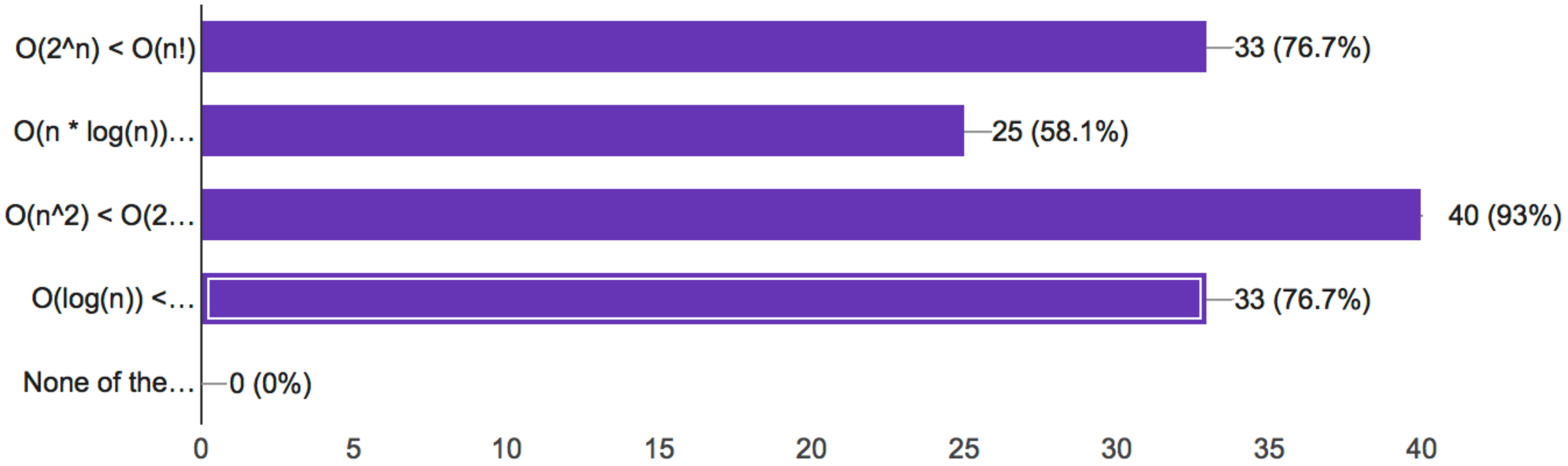
Observations on Calibration Quiz/Survey

Observations

- Background in parallel computing: ~1/5 have some experience
- Programming language preference: C

- Basic Coding: Good!
- Big-O: Mostly OK, with one common mistake
- Performance bottleneck: Lots of answers

Which of these big-Oh statements is true? (43 responses)



Which is larger?

$$\mathcal{O}\left(n \log n\right)$$

$$\mathcal{O}\left(n^{1.1}\right)$$

$\mathcal{O}\left(n \log n\right)$

$\mathcal{O}\left(n^{1.1}\right)$

Value

2^{25}
2^{23}
2^{21}
2^{19}
2^{17}
2^{15}
2^{13}
2^{11}
2^{9}
2^{7}
2^{5}
2^{3}
2^{1}

2^{1} 2^{2} 2^{3} 2^{4} 2^{5} 2^{6} 2^{7} 2^{8} 2^{9} 2^{10} 2^{11} 2^{12} 2^{13} 2^{14} 2^{15} 2^{16} 2^{17} 2^{18} 2^{19} 2^{20}

N

Function

n log n

n^1.1

$\mathcal{O}\left(n \log n\right)$

$\mathcal{O}\left(n^{1.1}\right)$

Value

2^{109}
2^{103}
2^{97}
2^{91}
2^{85}
2^{79}
2^{73}
2^{67}
2^{61}
2^{55}
2^{49}
2^{43}
2^{37}
2^{31}
2^{25}
2^{19}
2^{13}
2^{7}
2^{1}

2^{1} 2^{6} 2^{11} 2^{16} 2^{21} 2^{26} 2^{31} 2^{36} 2^{41} 2^{46} 2^{51} 2^{56} 2^{61} 2^{66} 2^{71} 2^{76} 2^{81} 2^{86} 2^{91} 2^{96}

N

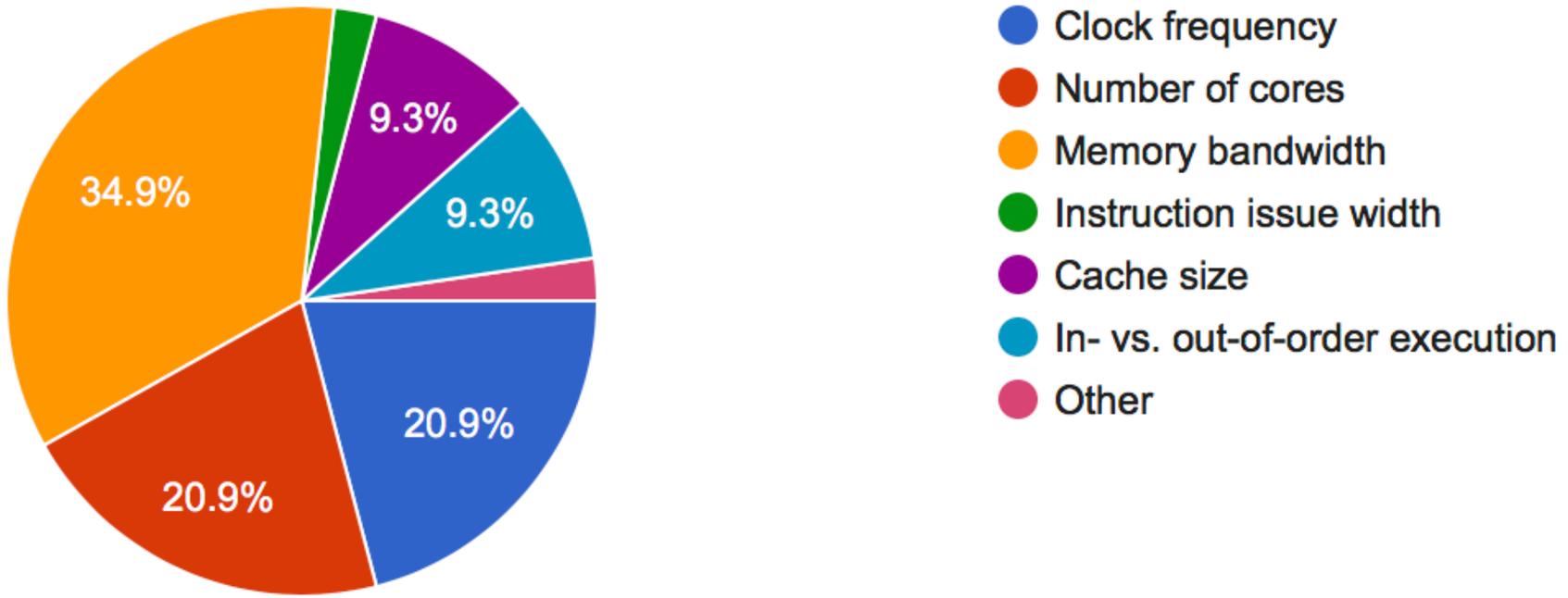
Function

n log n

n^1.1

In your vector-add code above, which of the following system characteristics is *most likely* to limit performance, when 'n' is very large?

(43 responses)



- Amdahl's Law:
  Estimating best-case speedup

- Speedup vs efficiency:
  Strong vs weak scaling (Gustafson's Law)

- *Shared memory programming model*
  OpenMP

Amdahl's Law

- **f** = Fraction of work done sequentially
- **1-f** = Fraction of work parallelizable
- **p** = Number of processors

$$S_p(f) = \frac{1}{f + \frac{1-f}{p}}$$

$$\lim_{p \to \infty} S_p(f) = \frac{1}{f}$$

Amdahl's Law

Exercise: Suppose sorting takes **70%** of the execution time of a program. Replace sequential sort with merge sort that scales perfectly on *p* processors. How many processors do you need to get a speedup of **4x**?

$$S_p(f) = \frac{1}{f + \frac{1-f}{p}}$$

$$\lim_{p \to \infty} S_p(f) = \frac{1}{f}$$

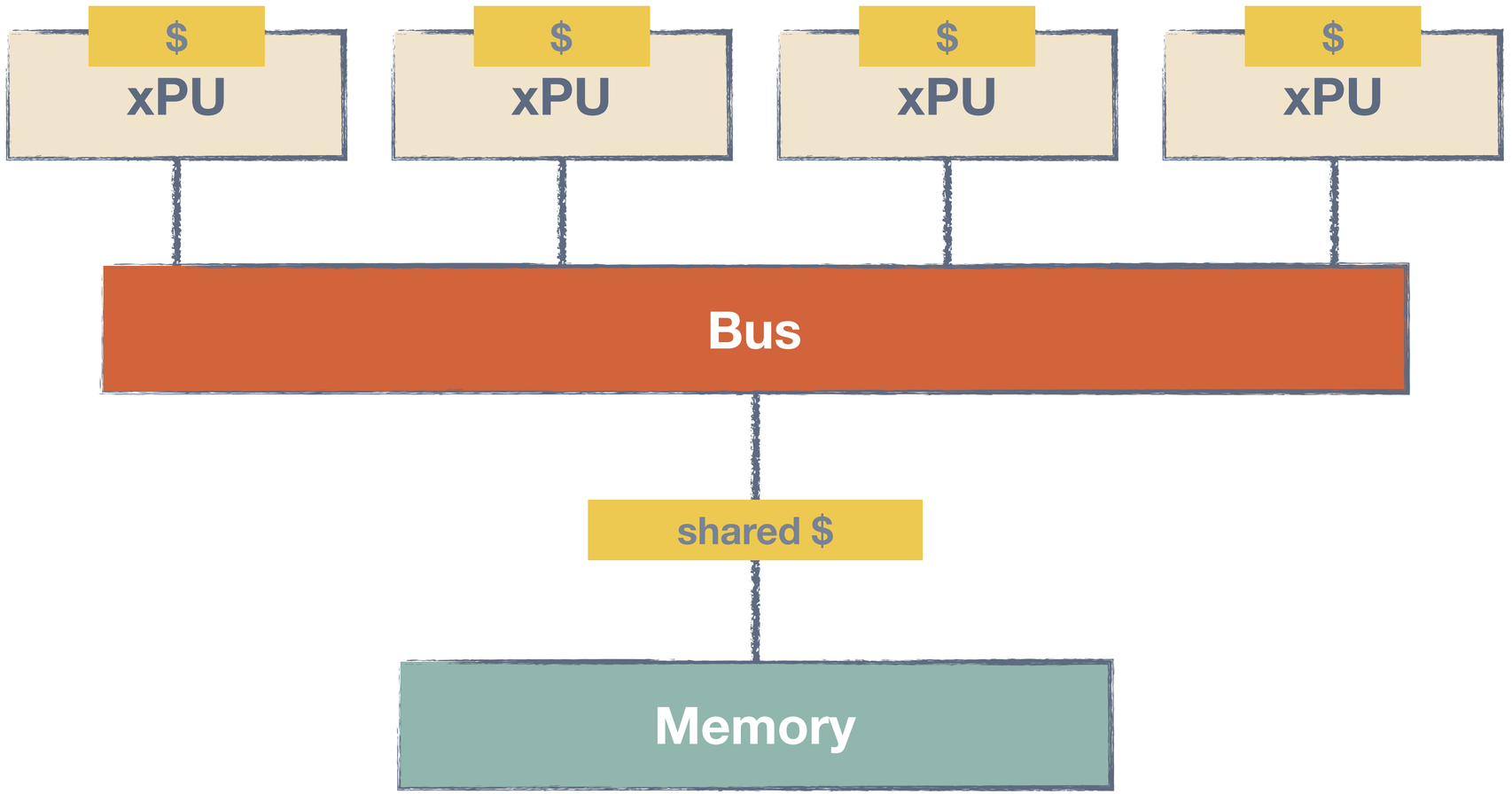
Speedup vs Efficiency

A fixed problem size limits the maximum speedup. But given more parallel hardware, one might choose instead to solve a larger problem. Scaling the problem is called weak scaling (vs. strong). A suitable measure in this case is not speedup but parallel efficiency, which we hope approaches a constant as p → ∞.

$$E_p(n) = \frac{S_p(n)}{p}$$

$$E_p(n) = \frac{W(n)}{p.T_p(n)}$$

$$E_p(n) \geq \frac{1}{\frac{D(n)}{W(n)}p + 1}$$

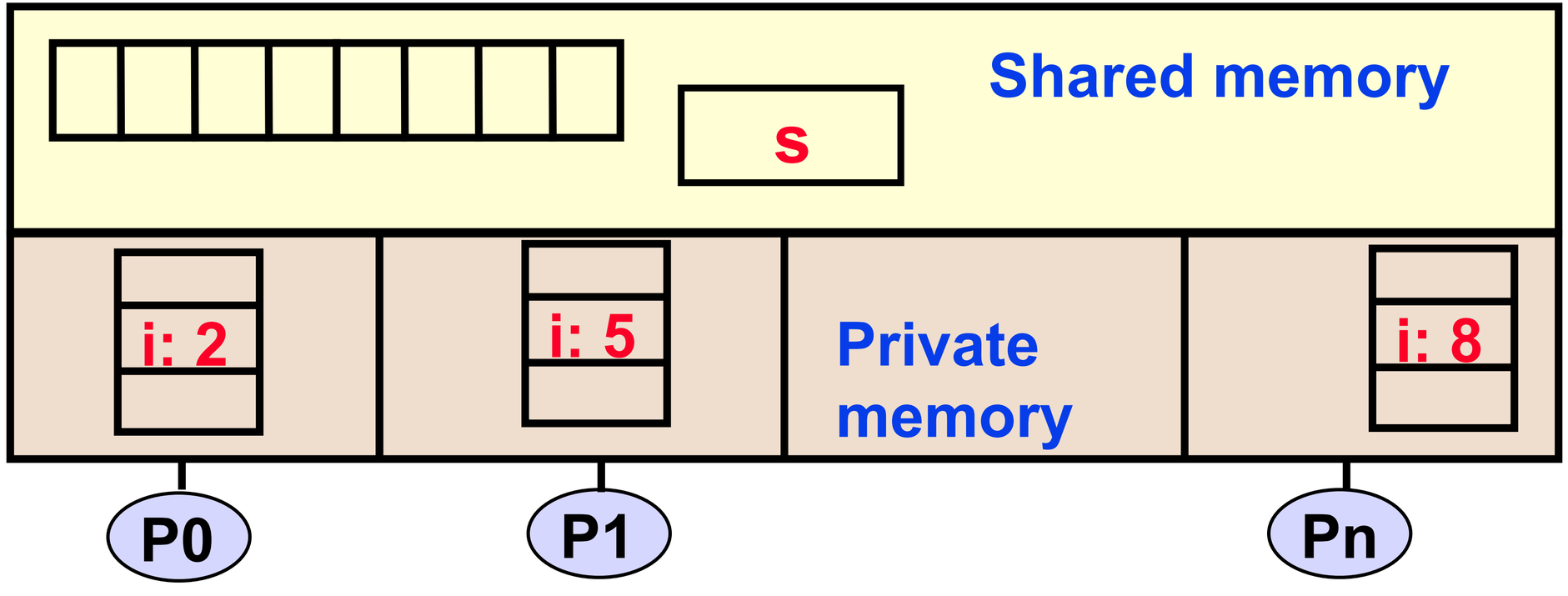


Shared Memory

All processors are connected to a shared memory.
Advantage: Uniform memory access (UMA)
Disadvantage: Difficult to scale to large number of processors

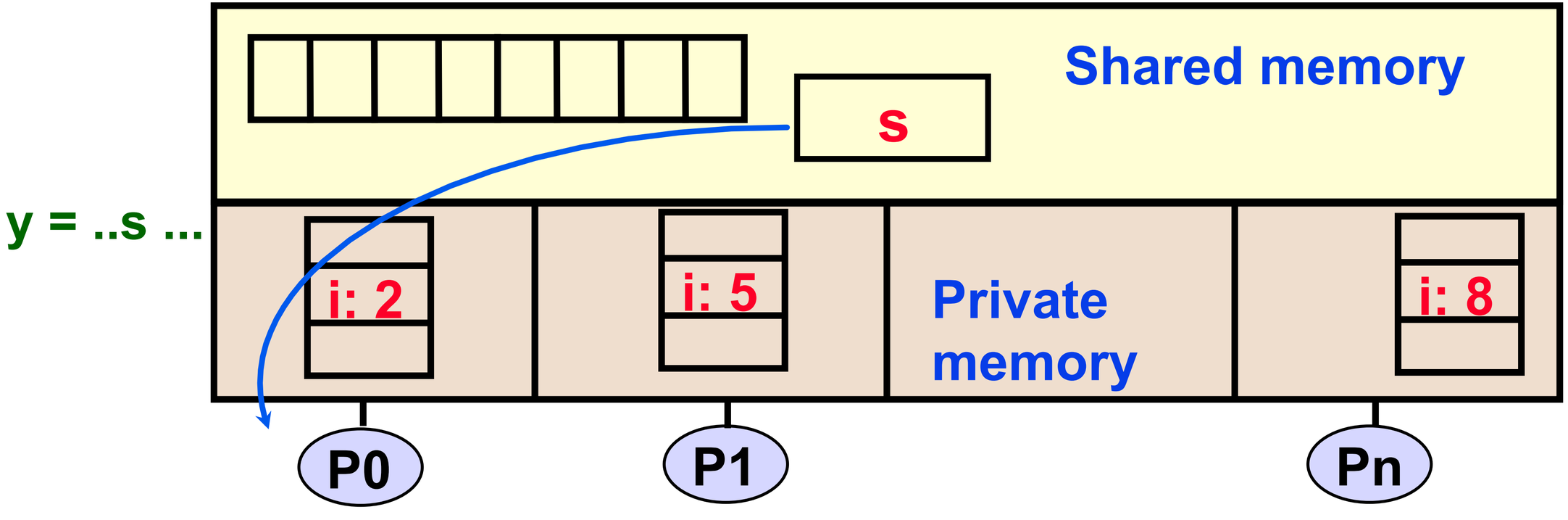
Shared Memory Model

- Program is a collection of threads of control
- Threads communicate implicitly by reading and writing shared variables
- Threads coordinate by synchronizing on shared variables



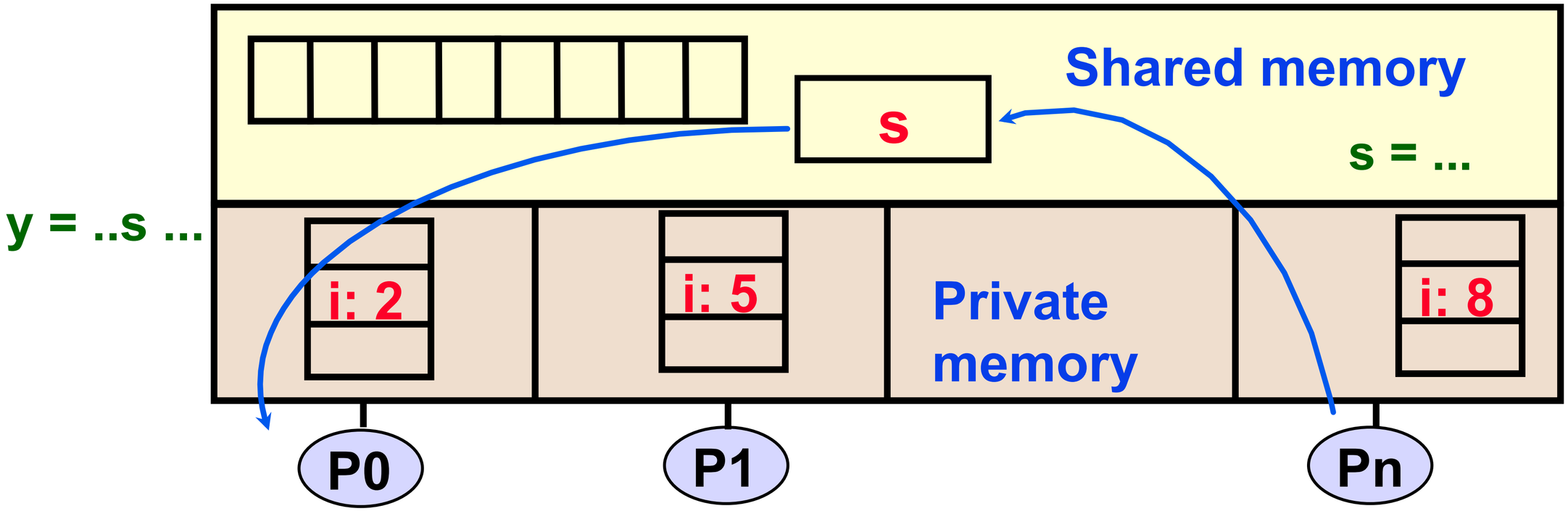
Shared Memory Model

- Program is a collection of threads of control
- Threads communicate implicitly by reading and writing shared variables
- Threads coordinate by synchronizing on shared variables



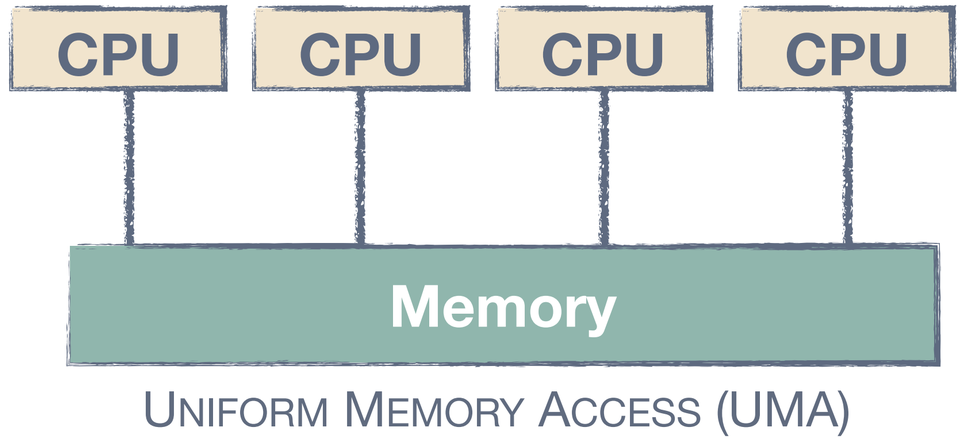
Shared Memory Model

- Program is a collection of threads of control
- Threads communicate implicitly by reading and writing shared variables
- Threads coordinate by synchronizing on shared variables

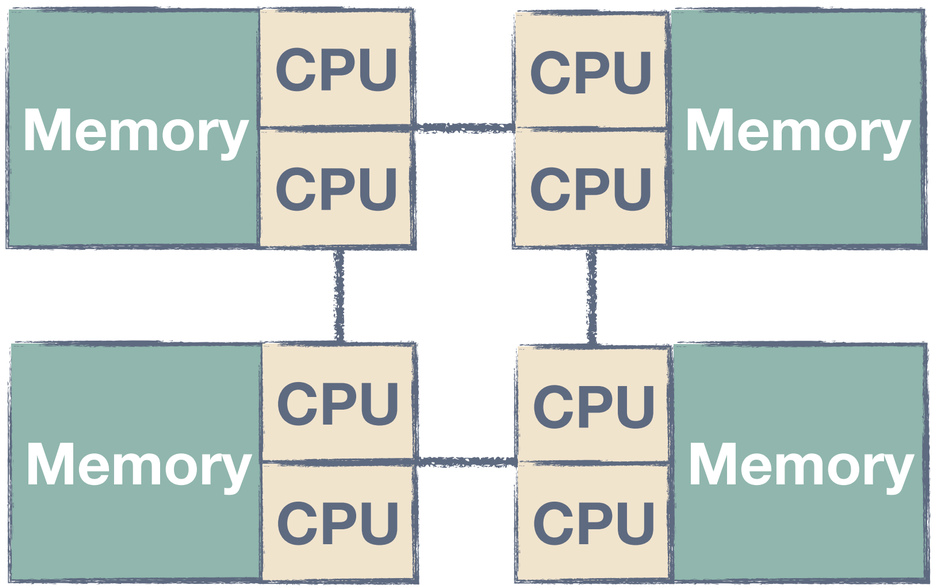


OpenMP programming model

- OpenMP is designed for shared memory machines. The underlying architecture can be shared memory UMA or NUMA.
- Comprised of three components:
  - Compiler directives
  - Library routines
  - Environment variables
- Parallelism through the use of threads
- Explicit programming model, offering the programmer full control over parallelization

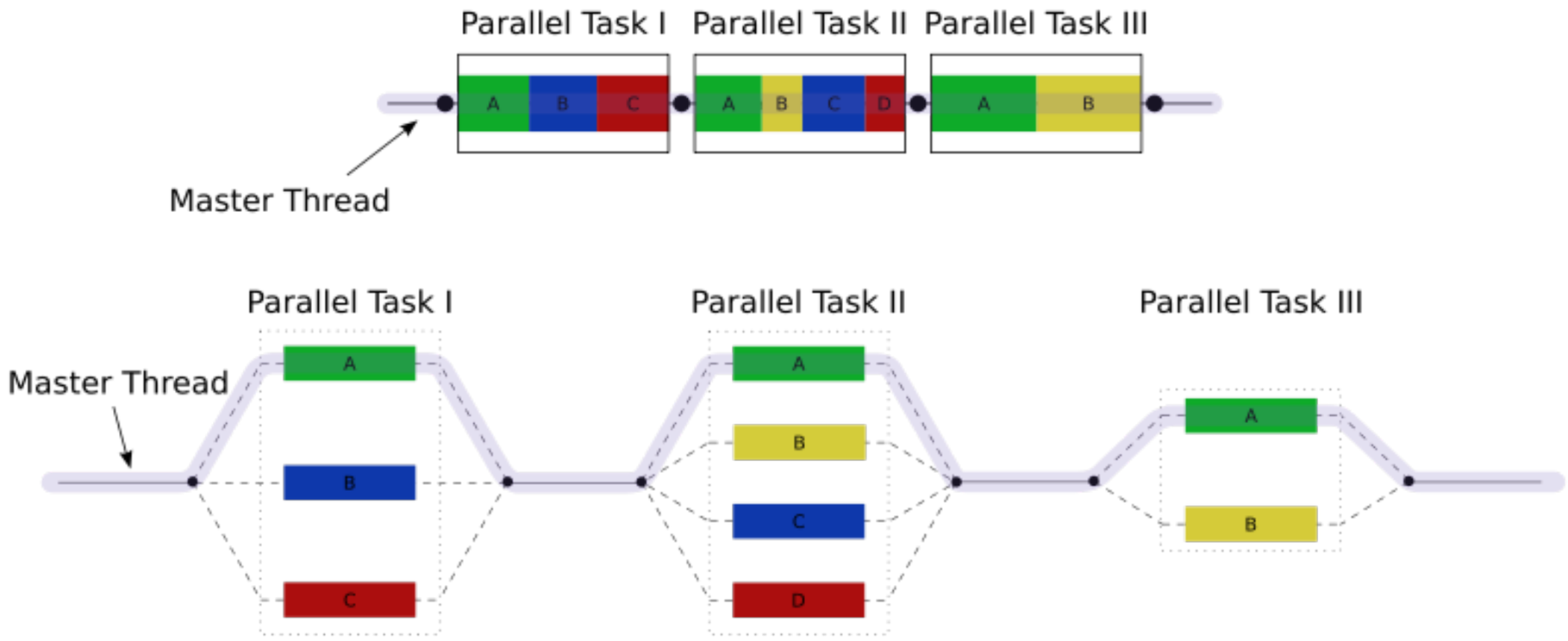


Uniform Memory Access (UMA)

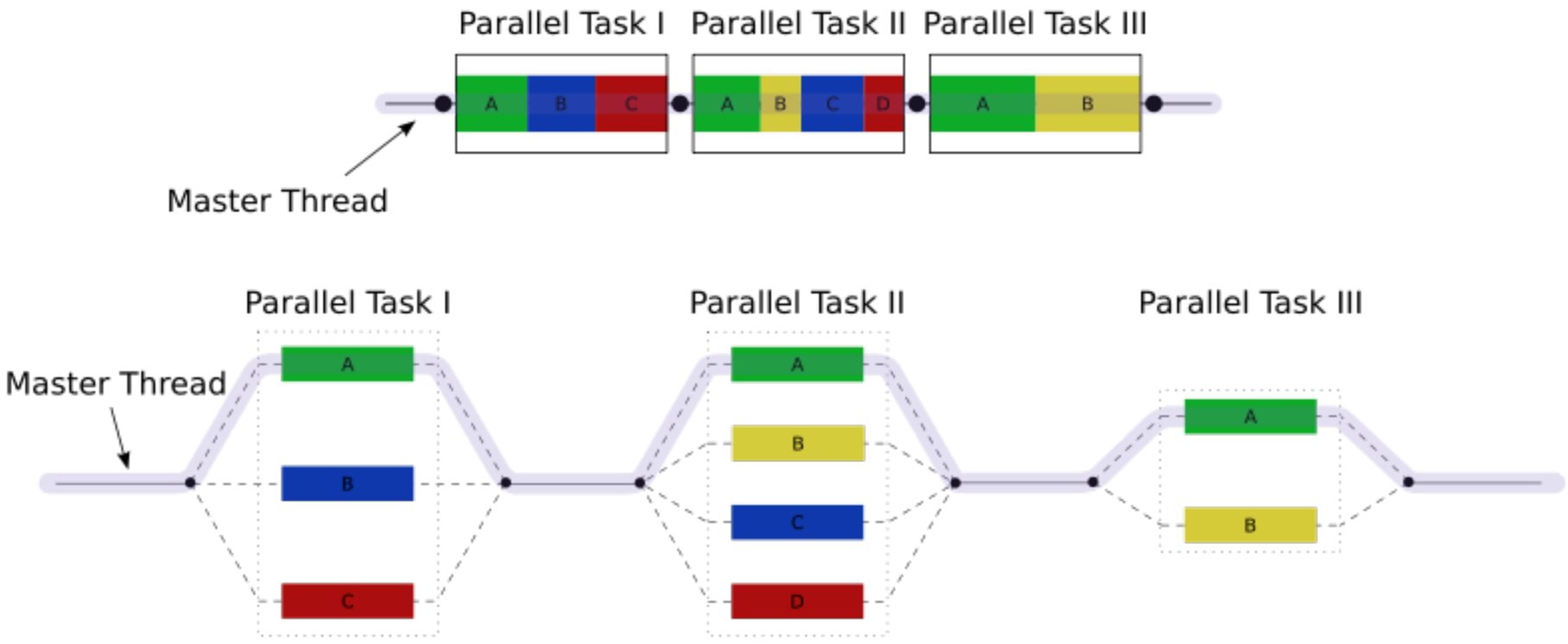


Non-Uniform Memory Access (NUMA)

Fork-Join model

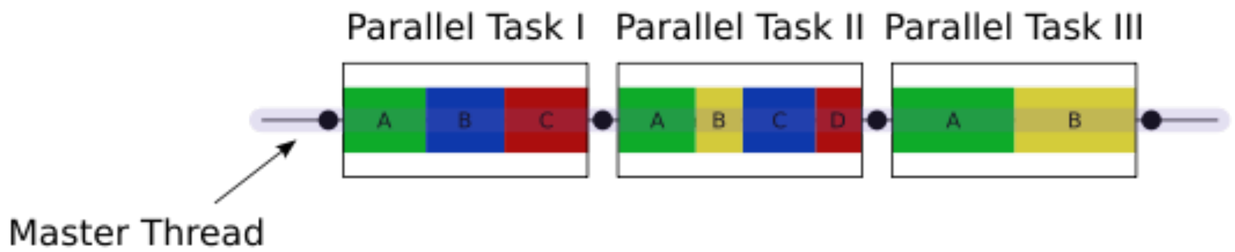


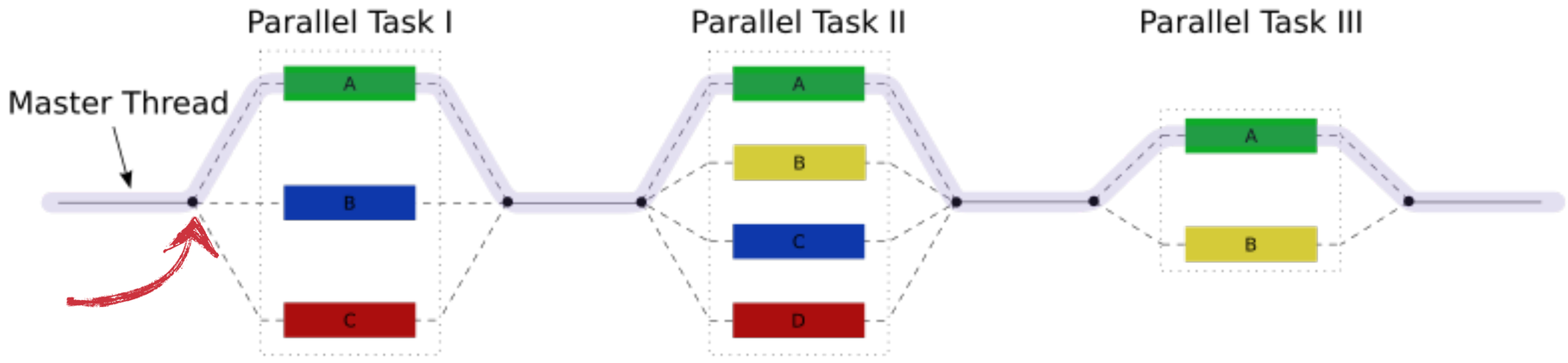
Fork-Join model



FORK: master thread then creates a team of parallel threads.
JOIN: team of threads execute the statements in the parallel region, then synchronize and terminate.

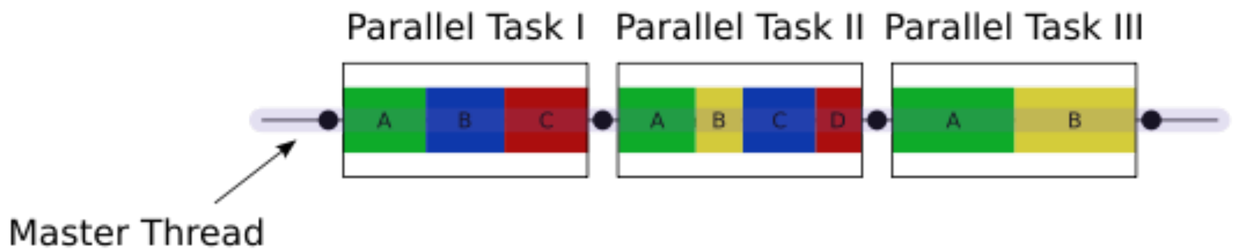
Fork-Join model

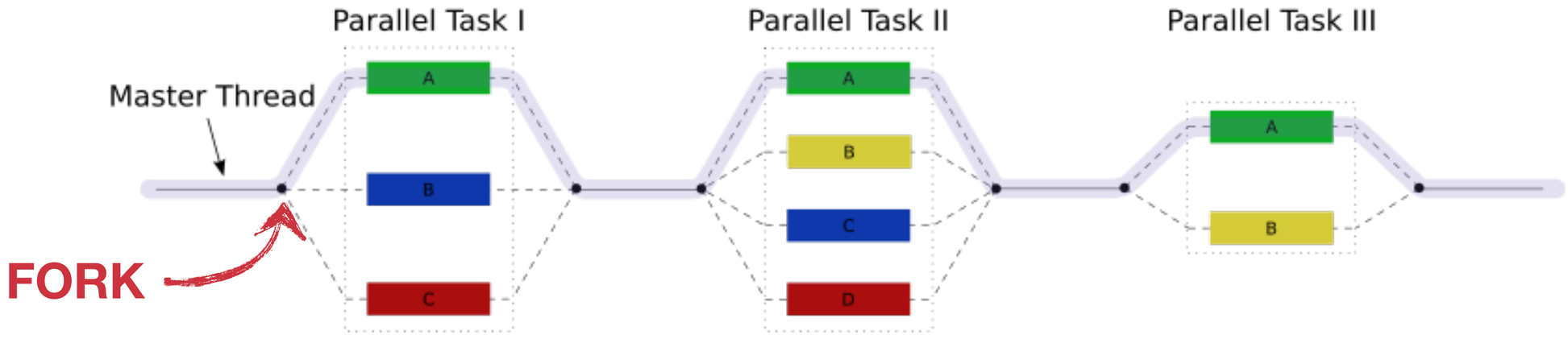




FORK: master thread then creates a team of parallel threads.
JOIN: team of threads execute the statements in the parallel region, then synchronize and terminate.

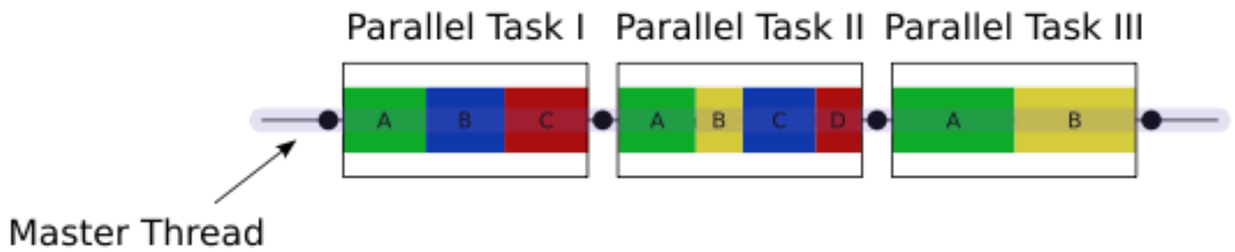
Fork-Join model

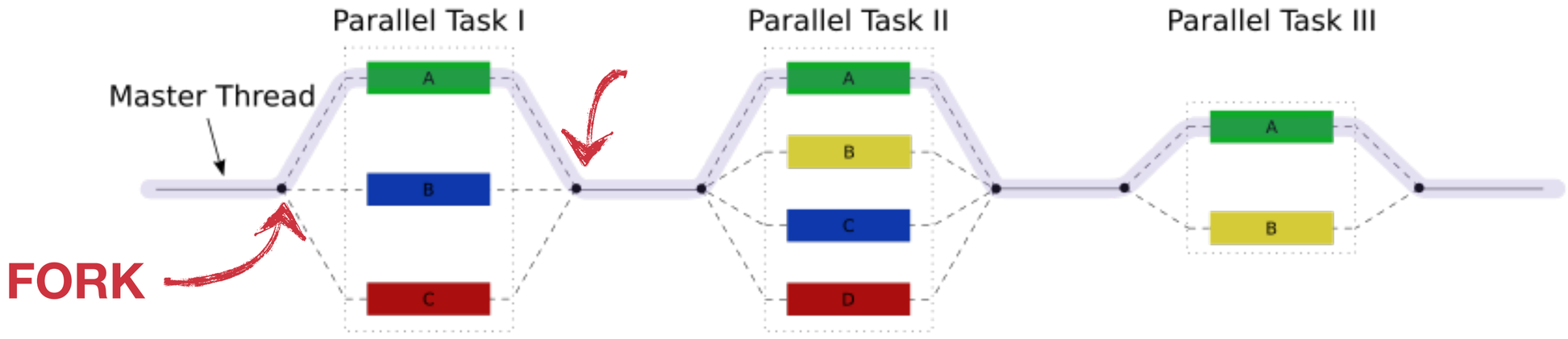




FORK: master thread then creates a team of parallel threads.
JOIN: team of threads execute the statements in the parallel region, then synchronize and terminate.

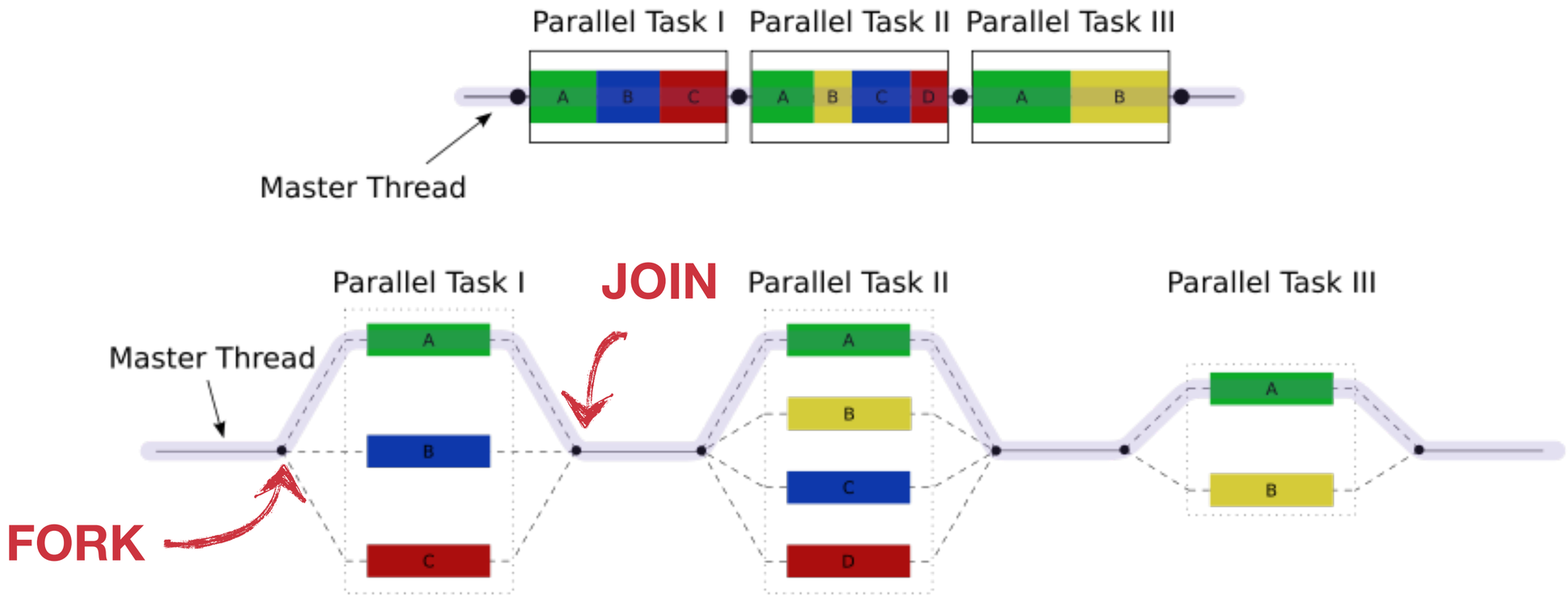
Fork-Join model





FORK: master thread then creates a team of parallel threads.
JOIN: team of threads execute the statements in the parallel region, then synchronize and terminate.

Fork-Join model



FORK: master thread then creates a team of parallel threads.
JOIN: team of threads execute the statements in the parallel region, then synchronize and terminate.

“Hello World” example

```
int main()
{



printf (“hello, world!\n”); // Execute in parallel

return 0;
}
```

“Hello World” example

```
#include <omp.h>

int main()
{



printf (“hello, world!\n”); // Execute in parallel

return 0;
}
```

“Hello World” example

```
#include <omp.h>

int main()
{


#pragma omp parallel
{
printf (“hello, world!\n”); // Execute in parallel
} // Implicit barrier/join
return 0;
}
```

“Hello World” example

```
#include <omp.h>

int main()
{
omp_set_num_threads (16);    // Can also use OMP_NUM_THREADS
                             // environment variable
#pragma omp parallel
{
printf (“hello, world!\n”); // Execute in parallel
} // Implicit barrier/join
return 0;
}
```

“Hello World” example

```
#include <omp.h>

int main()
{
omp_set_num_threads (16);    // Can also use OMP_NUM_THREADS
                             // environment variable
#pragma omp parallel
{
printf (“hello, world!\n”); // Execute in parallel
} // Implicit barrier/join
return 0;
}
```

Compiling:

```
gcc -fopenmp
icc -openmp
pgcc -mp
```

Parallel loops

```
for (i = 0; i < n; ++i) {
  a[i] += i;
}
```

Parallel loops

```
for (i = 0; i < n; ++i) {
  a[i] += i;
}
```

```
#pragma omp parallel
{
  #pragma omp for shared (a,n) private (i)
  for (i = 0; i < n; ++i) {
    a[i] += i;
  } // Implicit barrier/join
} // Implicit barrier/join
```

Parallel loops

```
for (i = 0; i < n; ++i) {
  a[i] += i;
}
```

Parallel loops

```
for (i = 0; i < n; ++i) {
  a[i] += i;
}
```

```
#pragma omp parallel for shared (a,n) private (i)
for (i = 0; i < n; ++i) {
  a[i] += i;
} // Implicit barrier/join
```

Parallel loops with dependencies

```
sum = 0;
for (i = 0; i < n; ++i) {
  sum += i;
}
```

Parallel loops with dependencies

```
sum = 0;
for (i = 0; i < n; ++i) {
  sum += i;
}
```

```
#pragma omp parallel for shared (sum,n) private (i)
for (i = 0; i < n; ++i) {
  // Data race condition!
  sum += i;
}
```

Parallel loops with dependencies

```
sum = 0;
for (i = 0; i < n; ++i) {
  sum += i;
}
```

```
#pragma omp parallel for shared (sum,n) private (i)
for (i = 0; i < n; ++i) {
  #pragma omp critical
  sum += i;
}
```

Parallel loops with dependencies

```
sum = 0;
for (i = 0; i < n; ++i) {
  sum += i;
}
```

```
#pragma omp parallel for reduction (+:sum)
for (i = 0; i < n; ++i) {
  sum += i;
}
```

Parallel loop scheduling

- Static: *chunk* iterations per thread and statically assigned

  #pragma omp parallel for schedule (static, chunk)

- Dynamic: *chunk* iterations per thread and dynamically assigned

  #pragma omp parallel for schedule (dynamic, chunk)

- Guided: *chunk* iterations per thread initially, decreases with each allocation and dynamically assigned

  #pragma omp parallel for schedule (guided, chunk)

- Runtime: set environment variable **OMP_SCHEDULE**

Synchronization primitive 1: Single

```
#pragma omp parallel shared (a, n) private (i)
{
  #pragma omp single
  for (i = 0; i < n; ++i) {
    a[i] += i;
  } // Implicit barrier/join
} // Implicit barrier/join
```

Synchronization primitive 1: Single

```
#pragma omp parallel shared (a, n) private (i)
{
  #pragma omp single nowait
  for (i = 0; i < n; ++i) {
    a[i] += i;
  }
} // Implicit barrier/join
```

Synchronization primitive 2: Master

```
#pragma omp parallel shared (a, n) private (i)
{
  #pragma omp master
  for (i = 0; i < n; ++i) {
    a[i] += i;
  } // No Implicit barrier/join
} // Implicit barrier/join
```

Synchronization primitive 3: Barrier

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

  #pragma omp for
  for (i = 0; i < n; ++i) {
    b[i] = a[i] * 2;
  }
} // Implicit barrier/join
```

Synchronization primitive 3: Barrier

```
#pragma omp parallel shared (a, b, n) private (i)
{
  #pragma omp for nowait
  for (i = 0; i < n; ++i) {
    a[i] += i;
  }
  #pragma omp barrier
  #pragma omp for
  for (i = 0; i < n; ++i) {
    b[i] = a[i] * 2;
  }
} // Implicit barrier/join
```

Synchronization primitives

Master	#pragma omp **master** { …. }
Single	#pragma omp **single** { …. }
Critical	#pragma omp **critical** { …. }
Barrier	#pragma omp **barrier**
Locks	omp_set_lock(l) …. omp_unset_lock(l)

Tasking (OpenMP 3.0+)

```
// At the call site:

  answer = fib (n);

// …

int fib (int n) {
  if (n <= G) fib__seq (n);
  int f1, f2;

  f1 = fib (n-1);
  f2 = fib (n-2);

  return f1 + f2;
}
```

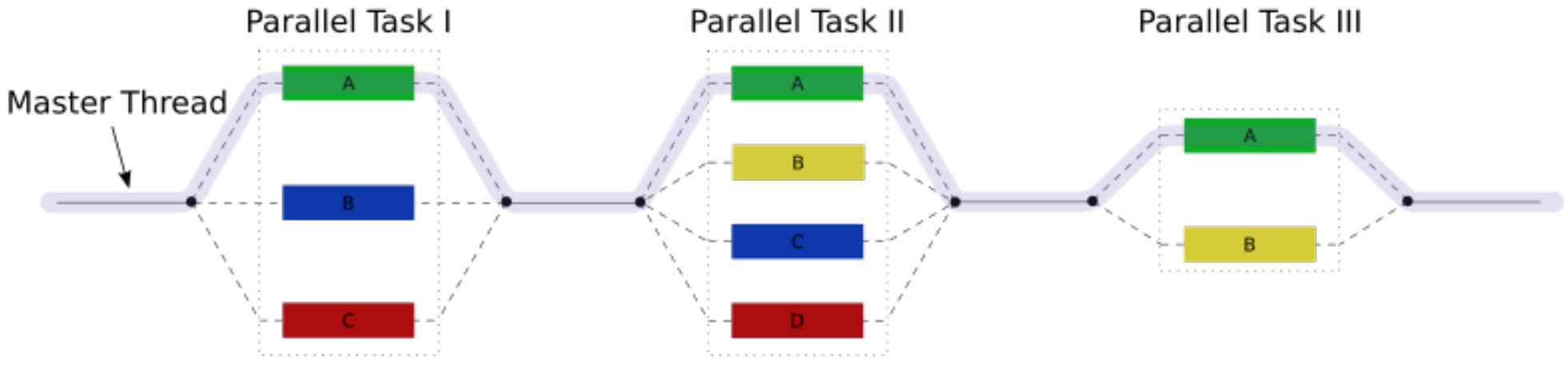
Tasking (OpenMP 3.0+)

```
// At the call site:
#pragma omp parallel
#pragma omp single nowait
  answer = fib (n);

// …

int fib (int n) {
  if (n <= G) fib__seq (n);
  int f1, f2;

  f1 = fib (n-1);
  f2 = fib (n-2);

  return f1 + f2;
}
```

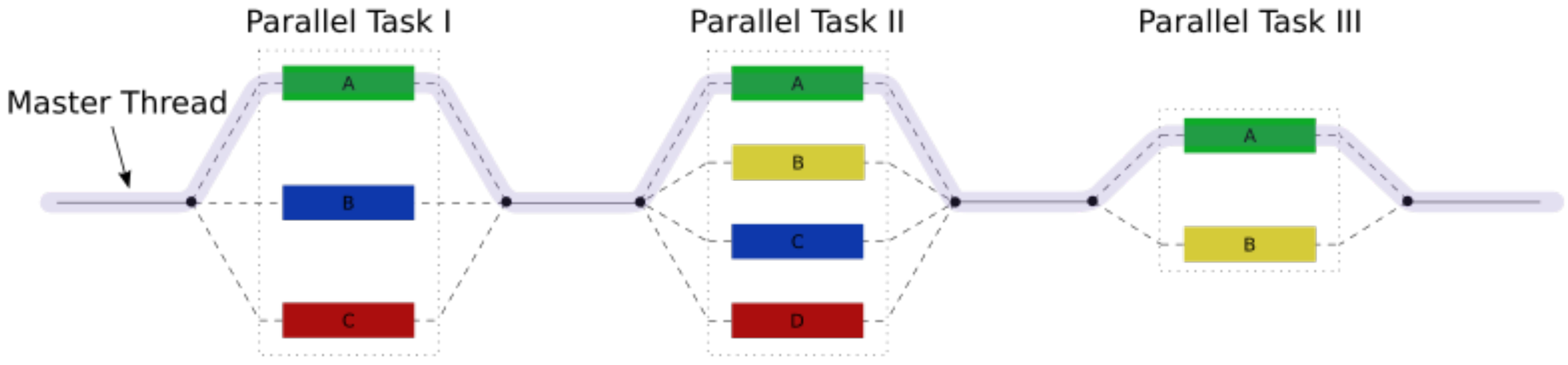
Tasking (OpenMP 3.0+)

```
// At the call site:
#pragma omp parallel
#pragma omp single nowait
  answer = fib (n);

// …

int fib (int n) {
  if (n <= G) fib__seq (n);
  int f1, f2;
#pragma omp task
  f1 = fib (n-1);
  f2 = fib (n-2);
#pragma omp taskwait
  return f1 + f2;
}
```

Controlling number of threads

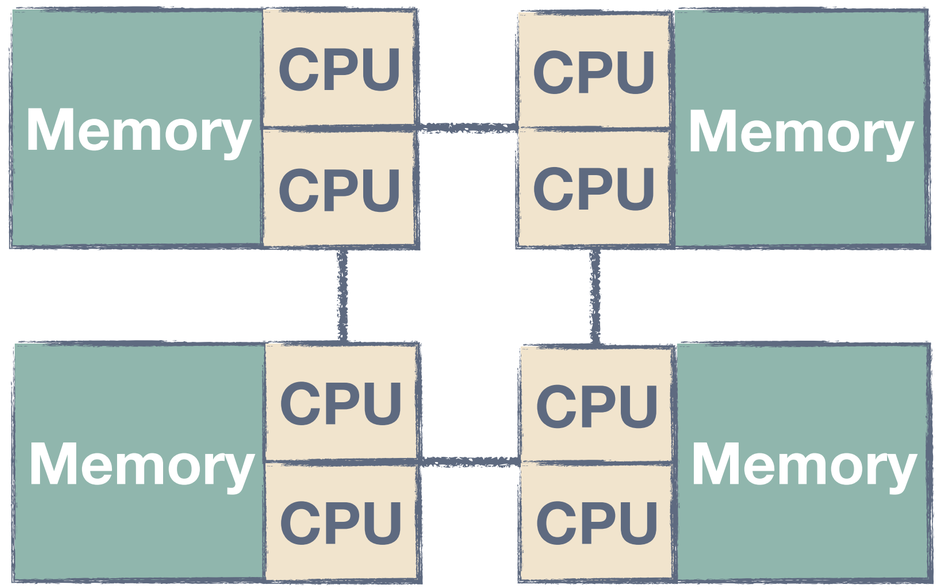


Controlling number of threads

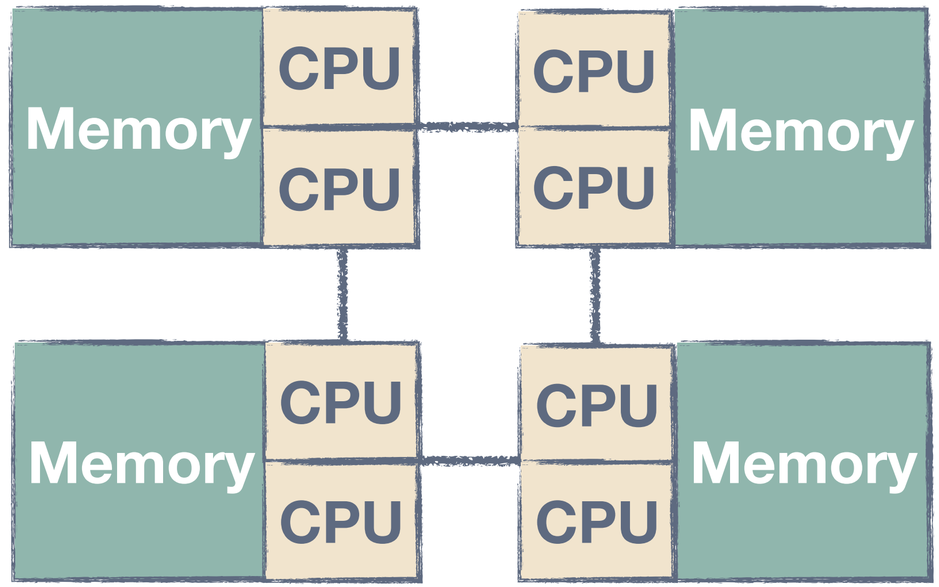



```
#pragma omp parallel num_threads(4)
{
  #pragma omp for shared (a,n) private (i)
  for (i = 0; i < n; ++i) {
    a[i] += i;
  } // Implicit barrier/join
} // Implicit barrier/join
```

NUMA: Linux “first-touch” policy



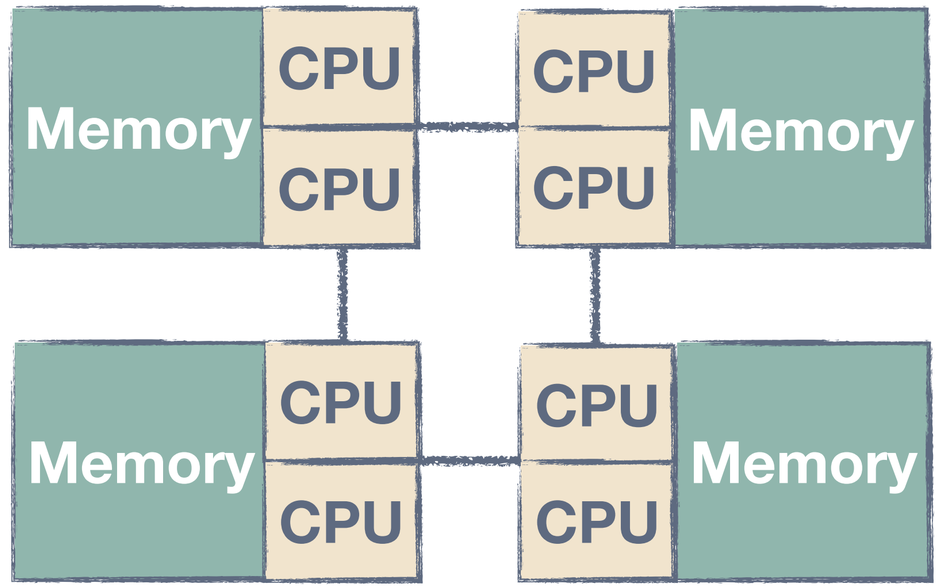
NUMA: Linux “first-touch” policy




```
#pragma omp parallel for shared (a,n) private (i)
for (i = 0; i < n; ++i) {
  a[i] += i;
} // Implicit barrier/join
```

NUMA: Linux “first-touch” policy

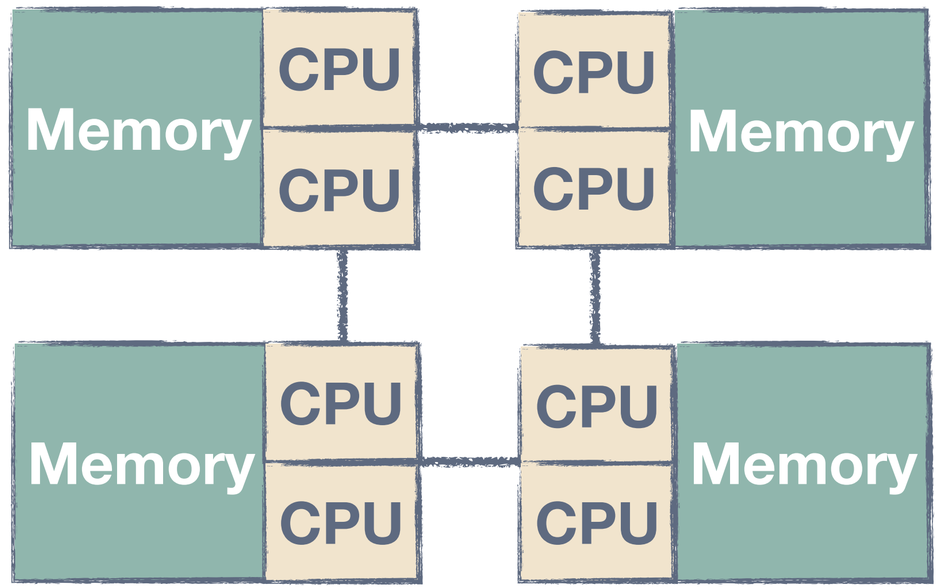
```
a = /* allocate memory */;

for (i = 0; i < n; ++i) {
 a[i] = /* assign initial value */;
}
```




```
#pragma omp parallel for shared (a,n) private (i)
for (i = 0; i < n; ++i) {
  a[i] += i;
} // Implicit barrier/join
```

NUMA: Linux “first-touch” policy

```
a = /* allocate memory */;
#pragma omp parallel for schedule(static)
for (i = 0; i < n; ++i) {
 a[i] = /* assign initial value */;
}
```




```
#pragma omp parallel for shared (a,n) private (i) schedule(static)
for (i = 0; i < n; ++i) {
  a[i] += i;
} // Implicit barrier/join
```