Parallel software

And how they run on parallel hardware

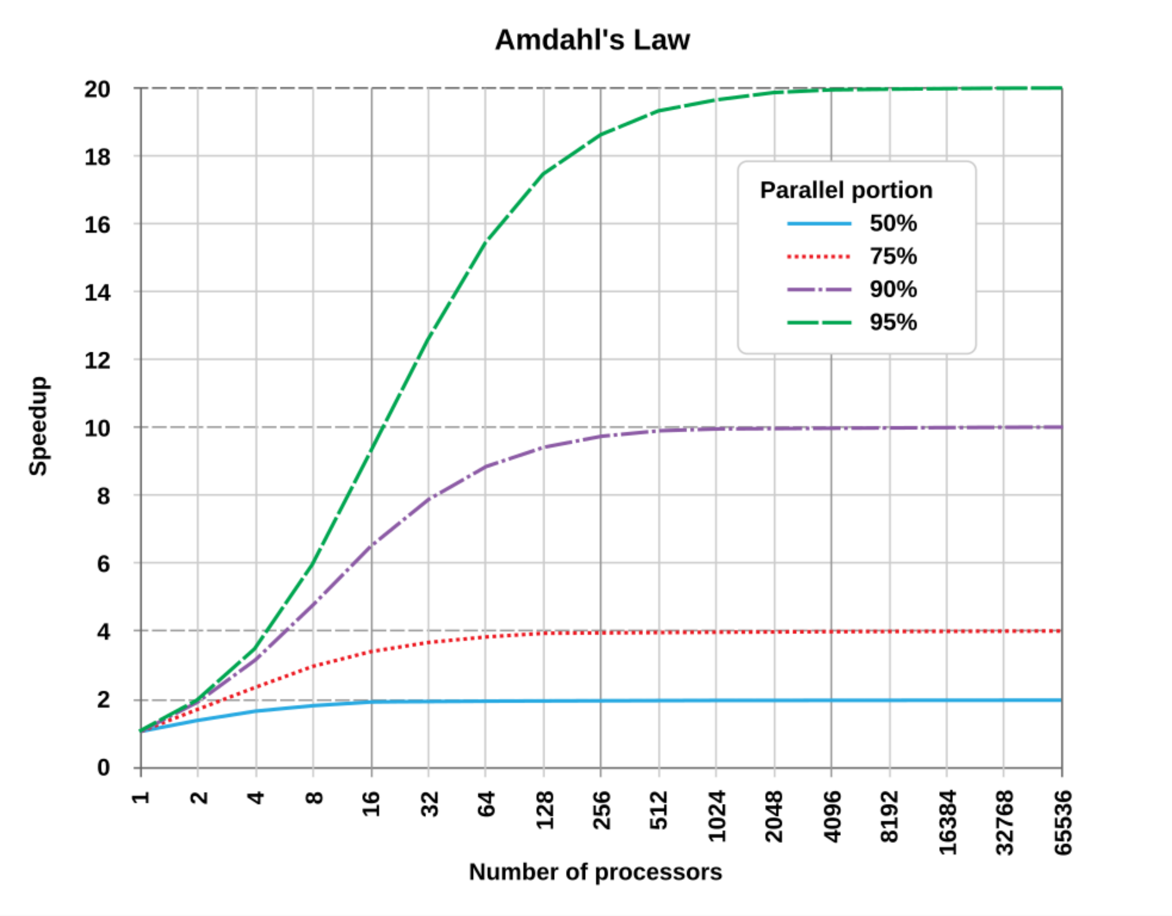
Amdahl's Law

The speedup from enhancing one part of a system is limited by the fraction of time the improved part is used.

- Let $Speedup = \frac{Time_o}{Time_e}$
  - o is original
  - e is enhanced
- Amdahl's law: Enhance fraction of computation (f) by some speedup (S):

$$Speedup_e(f,S) = \frac{1}{(1-f) + {}^{f}/_{S}}$$

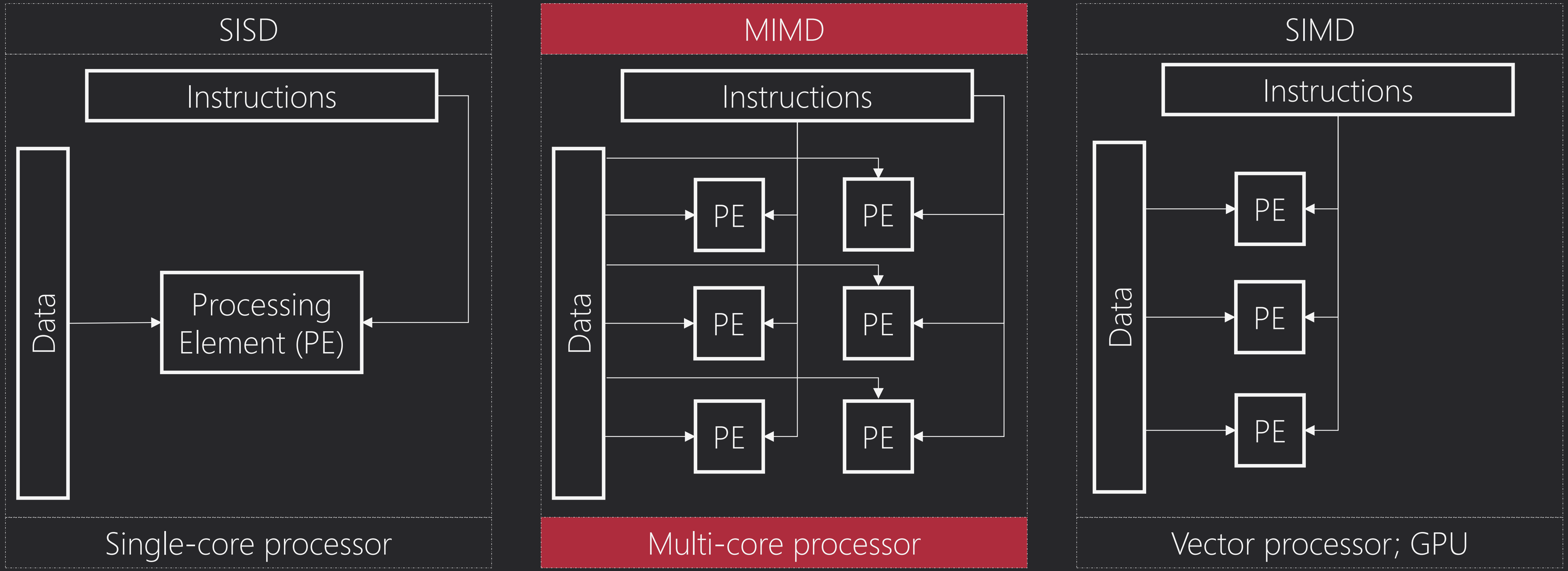
- Implications of Amdahl's law
  - Small f means enhancement has little effect
  - Even with very large S, speedup is bounded by $\frac{1}{1-f}$



By Daniels220, CC BY-SA 3.0

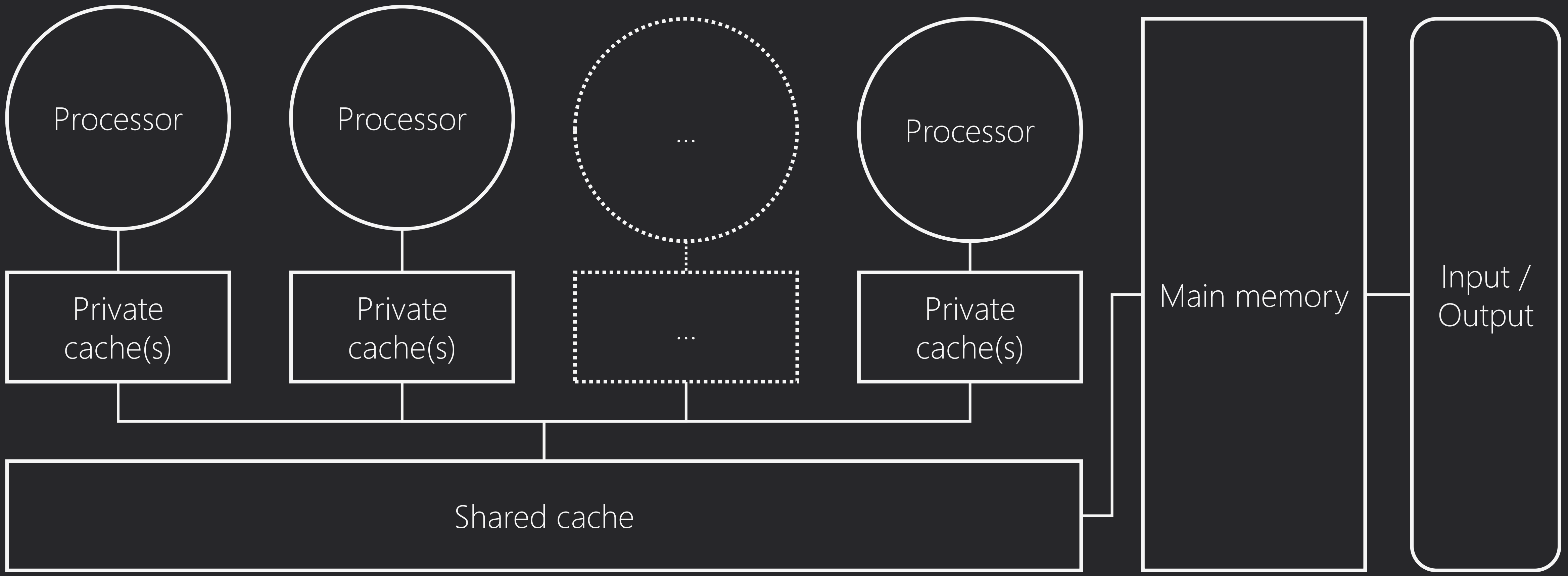
Flynn's Taxonomy

- Single instruction, single data (SISD)
- Multiple instruction, multiple data (MIMD)
- Single instruction, multiple data (SIMD)



Thread-level parallelism (TLP)

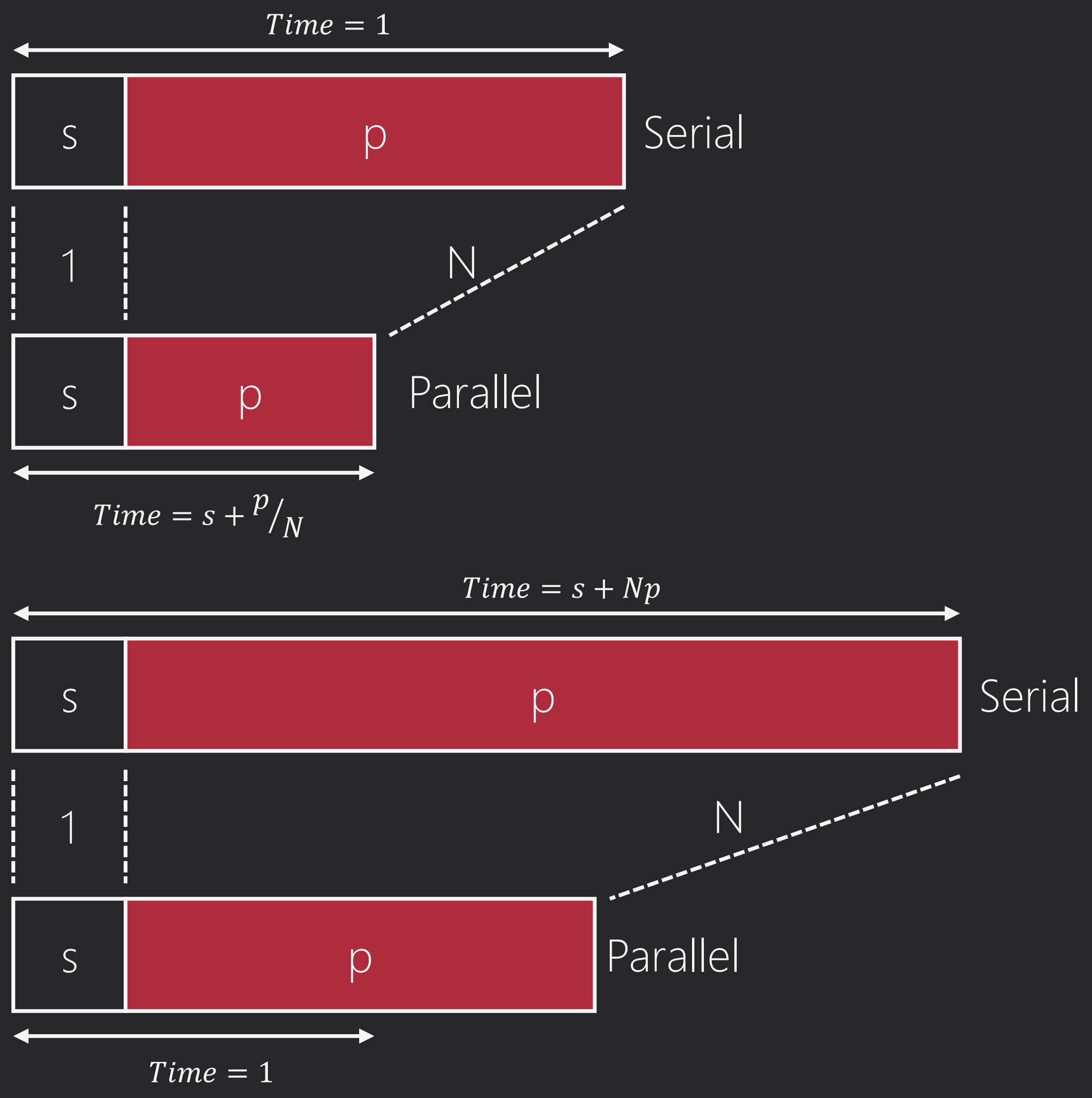
- Threads can be executed in parallel. But programmers must be explicit about what each thread does
- Weeks 6 to 9



What is Gustafson's Law?

- s: serial fraction of time
- p: parallel fraction of time
- N: number of cores

- Amdahl's law assumes a fixed problem size
  - $Speedup = \frac{1}{s+\frac{p}{N}}$
- Gustafson's law assumes *the problem size scales with the number of processors*
  - $Speedup = s + Np = N + s(1 - N)$



What is scaling?

Strong scaling

- Measuring speed up while the problem size (W) is fixed, regardless of the number of processors (P)
- i.e, the amount of work to do by each processor is: $W/_P$
- Useful model for applications whose working set do not grow (much) over time

Weak scaling

- Measuring speed up when the problem size (W) grows with the number of processors (P)
- i.e., the amount of work to do by each processor is: W
- Useful model for applications whose working sets grow commensurate with processing power (i.e., number of cores)

Parallel programming

An overview of abstractions for parallel programming

Why is writing parallel software challenging?

- Load balancing (or work partitioning)
  - How do you ensure that each processor gets an equal division of work?

- Communication (or coordination)
  - Many algorithms require data to be communicated across processors

- Incentive
  - The program already works for single-core and runs well enough…

How do we go from sequential to parallel?

Method

- Identify which program segments can be run in parallel
- Two sequential segments, S1 and S2, can be run in parallel iff S1 and S2 are independent
  - i.e., S2 does not need data from S1

Common patterns

- Data-level parallelism in loops
  - No loop-carried dependencies (each iteration’s computation is independent)
- Task-level parallelism
  - Functions are tasks performed in parallel
  - The ordering of tasks is based on dependencies between them
- A function pipeline
  - Useful in streaming applications

What is a thread?

Definition

- A thread is a control flow through a program

- A sequential program has one control flow
- A multi-threaded program has multiple control flows

Effect

- Each thread has its own PC
- Threads may run in parallel

- Threads share resources with other threads
  - Hardware
  - Memory and data
- Sharing data needs to be done correctly

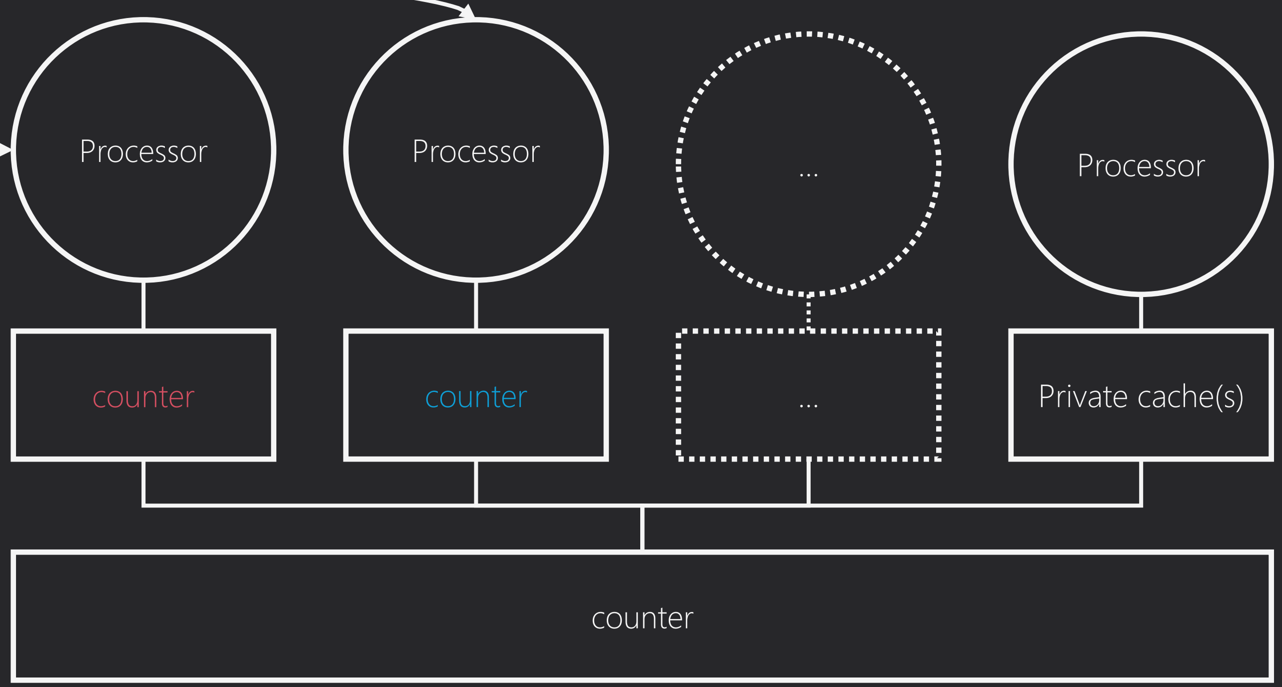
How do threads execute in parallel?

Thread 2

```
counter++
```

Thread 1

```
counter++
```



Assembly

```
LW r1, counter
addi r1, r1, 1
SW r1, counter
```

What is synchronization?

- In a shared memory system, threads communicate implicitly
  - Load and store instructions

- Synchronization is a mechanism
  - That makes communication explicit
  - That avoids incorrect interleaving of loads and stores

- Later: how does hardware support synchronization?

Creating parallel programs

From sequential to parallel

Examples of sequential algorithms

Sum all elements of an array

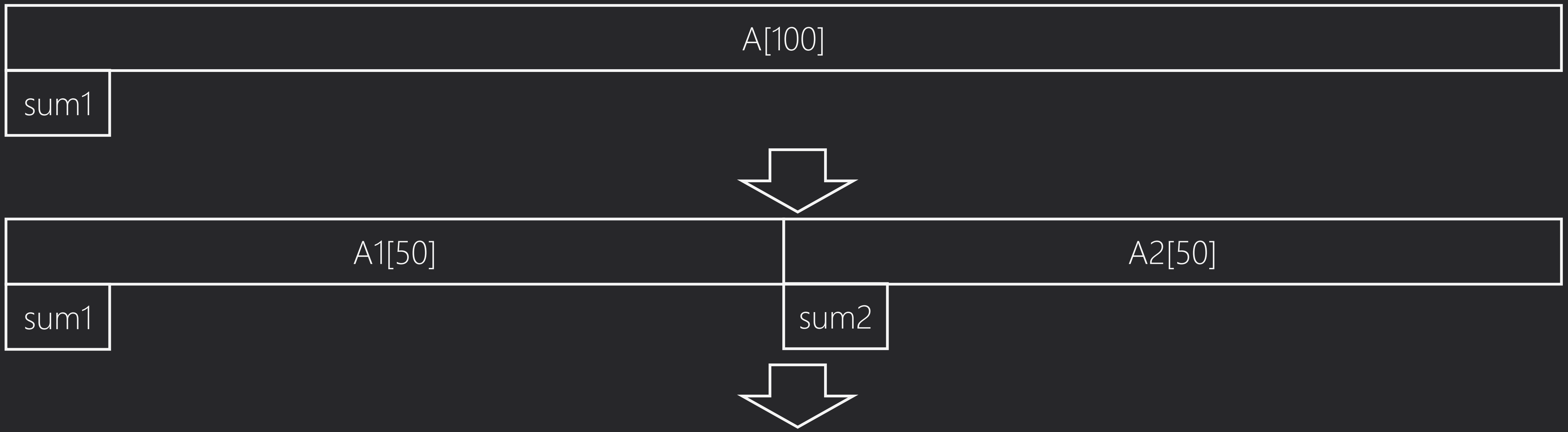
```
sum = 0
for i in range(N):
      sum += A[i]
```

Matrix multiplication and summation

```
sum = 0
for i in range(N):
  for j in range(N):
    C[i][j] = 0;

    for k in range(N):
      C[i][j] += A[i][k]*B[k][j]

    sum += C[i][j]
```

Dividing the data in half

```
sum = 0
for i in range(N):
    sum += A[i]
```




```
sum1 = 0
for i in range(N/2):
    sum1 += A1[i]
```

```
sum2 = 0
for i in range(N/2):
    sum2 += A2[i]
```

```
sum = sum1 + sum2
```

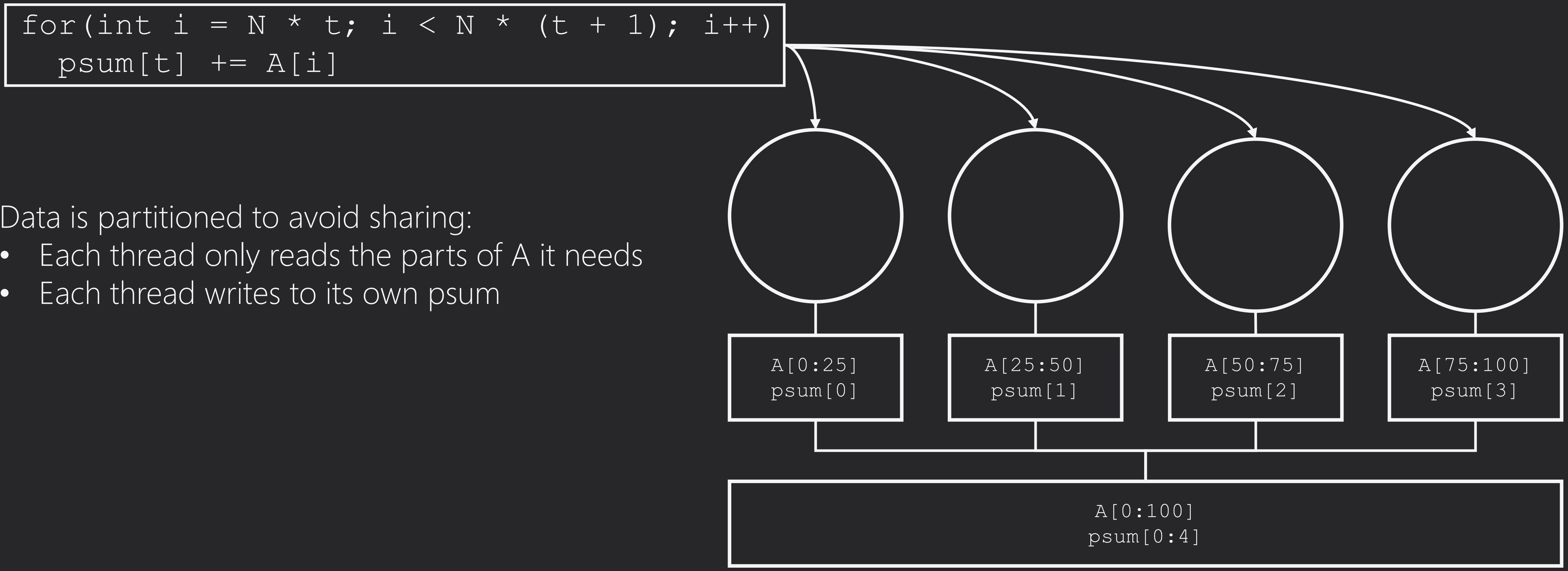
Dividing the data across T threads

- OpenMP is an API that simplifies parallel program
  - Use “decorators” to indicate parallel pattern
- OpenMP will…
  - Create a pool of threads (to re-use)
  - Assign work to threads
  - Follow the decorated patterns
- Easy to use, difficult to debug

```
#define N 100    // elements in array
#define T 4      // number of threads
#pragma omp parallel num_threads(T)

// create arrays for psum[T] and A[N]

#pragma omp parallel for
for (int t = 0; t < T; t++)
  for(int i = N * t; i < N * (t + 1); i++)
    psum[t] += A[i]

#pragma omp parallel for reduction(+: sum)
for (int t = 0; t < T; t += 1)
  sum += psum[t]
```

Running the sum on a parallel processor

```
#pragma omp parallel for
for (int t = 0; t < T; t++)
```



Data is partitioned to avoid sharing:

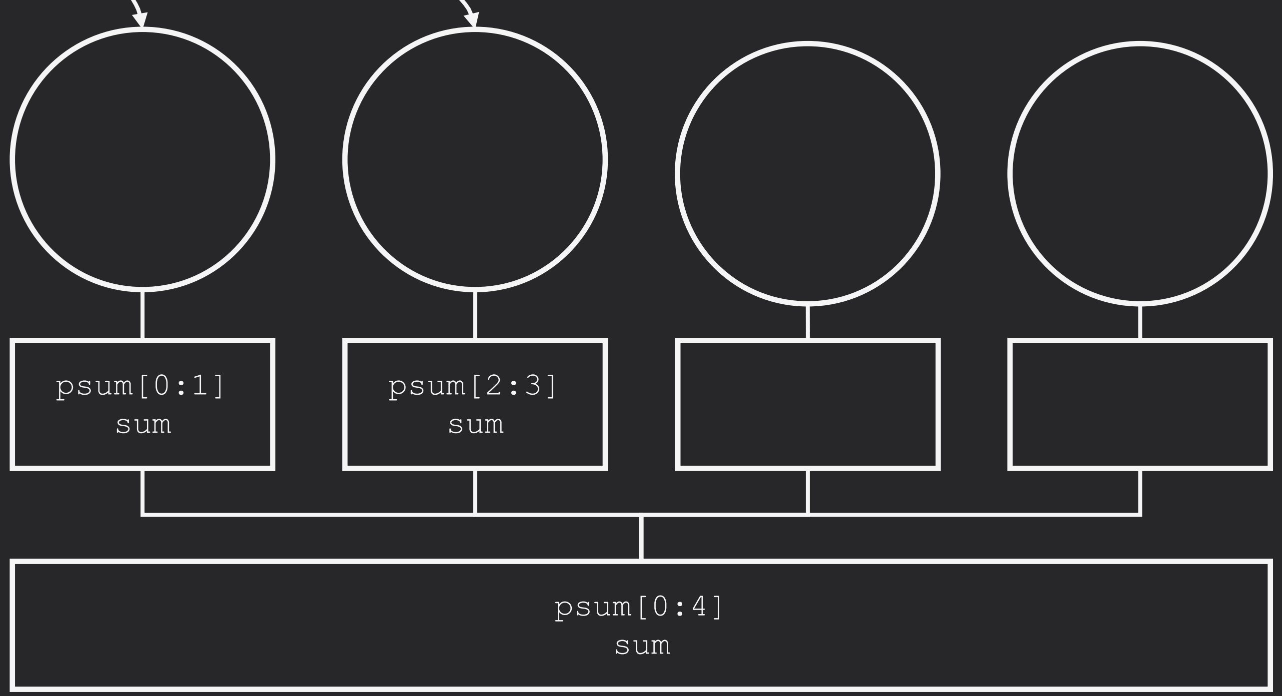
- Each thread only reads the parts of A it needs
- Each thread writes to its own psum

Running the reduction on a parallel processor

```
#pragma omp parallel for reduction(+: sum)
for (int t = 0; t < T; t += 1)
  sum += psum[t]
```

Some data is partitioned to avoid sharing.

Needs synchronization on sum – handled by OpenMP



Dividing the matrix across T threads

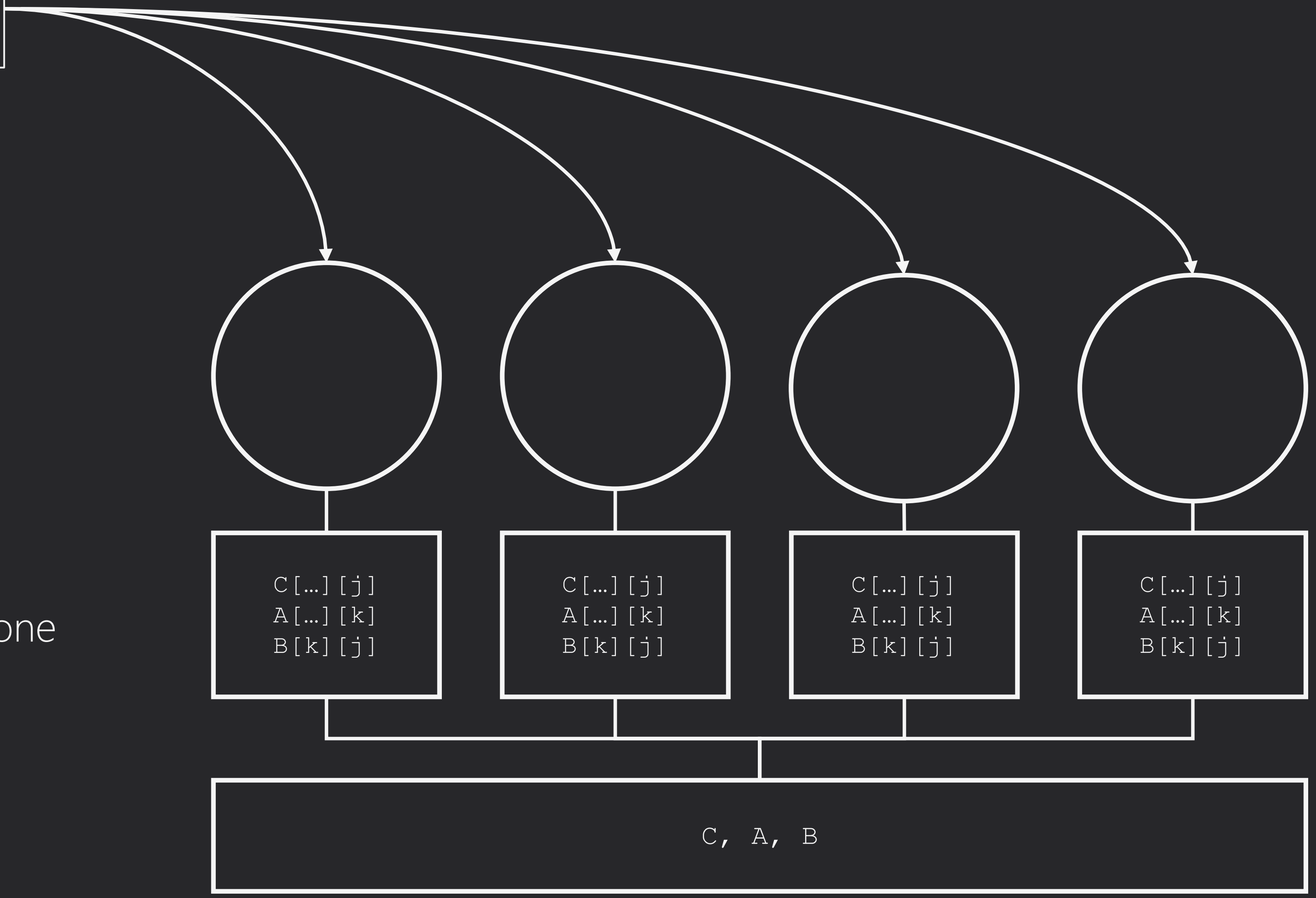
- Each thread
  - Executes the code on the right
  - Loops over a part of the matrix
  - Calculates a partial sum

- What about the final sum?

```
#define N 100     // elements in matrix
#define T 4       // number of threads

tid = get_thread_id()
start = tid * N / T
end = low + N / T

psum = 0
for (i = start; i < end; i++)
  for j in range(N):
    C[i][j] = 0;

    for k in range(N):
      C[i][j] += A[i][k]*B[k][j]

    psum += C[i][j]
```

Running the sum on a parallel processor

```
for k in range(N):
  C[i][j] += A[i][k]*B[k][j]
```



Data is partitioned to avoid sharing:
- Each thread only writes a part of C

Other data is shared "nicely":
- No writes to A or B
- Data brought into the shared cache by one thread may help another (e.g., B)

Summing up the partial sums

- Use a more flexible API: POSIX threads
- What is a barrier?
  - A meeting point
  - “Wait here until all other threads reach this point”
- What is a lock?
  - “Only I will update sum at this time”
  - Serializes updates to sum across all threads

```
// … see earlier slide …

psum = 0
for (i = start; i < end; i++)
  for j in range(N):
    C[i][j] = 0;

    for k in range(N):
      C[i][j] += A[i][k]*B[k][j]

    psum += C[i][j]

barrier()
lock()
sum += psum;
unlock();
```

Conclusion

A summary and parting thoughts

Parallel programming

- Programmers must explicitly,
  - Divide program into multiple threads
  - Define communication between threads
  - Be aware of implicit communication, so that they
  - include synchronization to shared memory where interleavings are problematic

- Different APIs are available
  - OpenMP (easy to use, difficult to debug)
  - POSIX threads (more flexible, difficult to debug)

What do architects need to do?

- Correctness
  - Ensure sequential programs run as expected on parallel processors
  - Ensure parallel programs run correctly (assuming correct synchronization)

- Support
  - Provide support in the hardware for synchronization

- Performance
  - Ensure communication across processors is fast