Introduction to Parallel Programming – Part 1

Recognizing Potential Parallelism

Intel Software College

Objectives

At the end of this module you should be able to:

- Define parallel computing
- Explain why parallel computing is becoming mainstream
- Explain why explicit parallel programming is necessary
- Identify opportunities for parallelism in code segments and applications

What Can We Do with Faster Computers?

- Solve problems faster
  - Reduce turn-around time of big jobs
  - Increase responsiveness of interactive apps
- Get better solutions in same amount of time
  - Increase resolution of models
  - Make model more sophisticated

What Is Parallel Computing?

Attempt to speed solution of a particular task by

1. Dividing task into sub-tasks
2. Executing sub-tasks simultaneously on multiple processors

Successful attempts require both

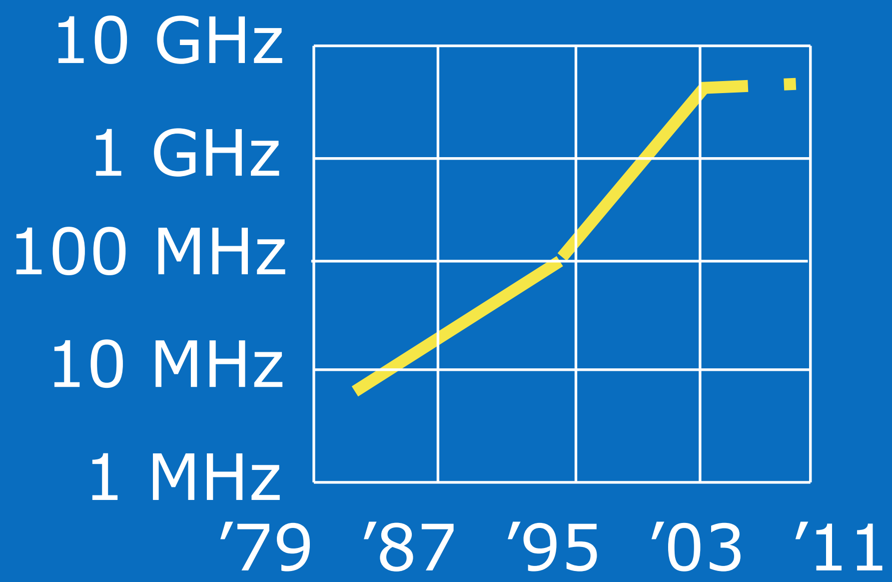
1. Understanding of where parallelism can be effective
2. Knowledge of how to design and implement good solutions

Why Parallel Computing?

"The free lunch is over." —Herb Sutter

We want applications to execute faster

Clock speeds no longer increasing exponentially



Clock Speeds Have Flattened Out

Problems caused by higher speeds
- Excessive power consumption
- Heat dissipation
- Current leakage

Power consumption critical for mobile devices

Mobile computing platforms increasingly important
- Retail laptop sales now exceed desktop sales
- Laptops may be 35% of PC market in 2007

Execution Optimization

Popular optimizations to increase CPU speed
- Instruction prefetching
- Instruction reordering
- Pipelined functional units
- Branch prediction
- Functional unit allocation
- Register allocation
- Hyperthreading

Added sophistication $\Rightarrow$ more silicon devoted to control hardware

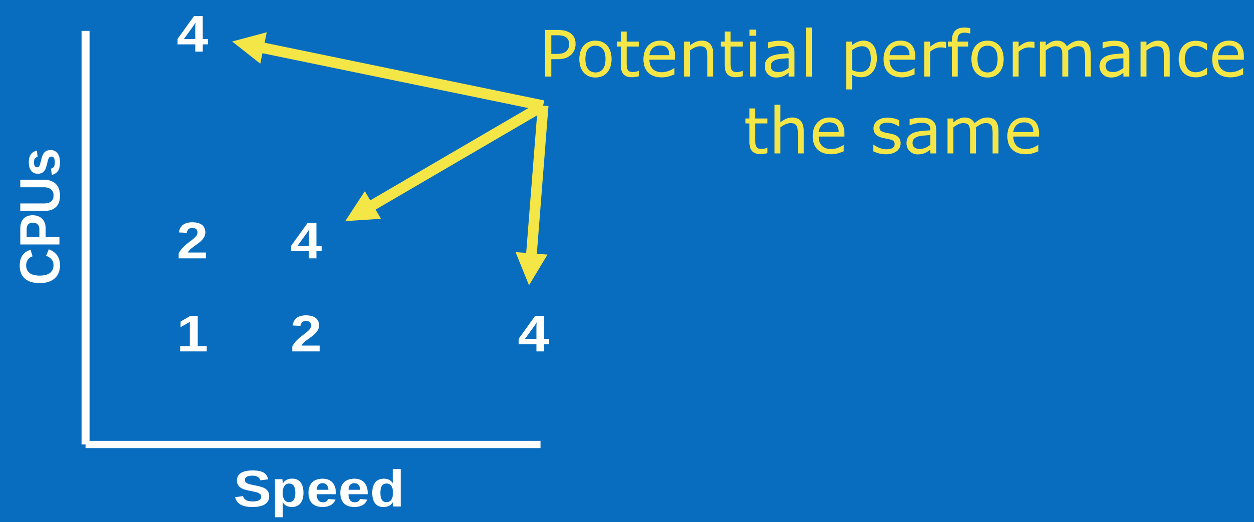
Multi-core Architectures

Potential performance = CPU speed × # of CPUs

Strategy:

Limit CPU speed and sophistication

Put multiple CPUs ("cores") on a single chip



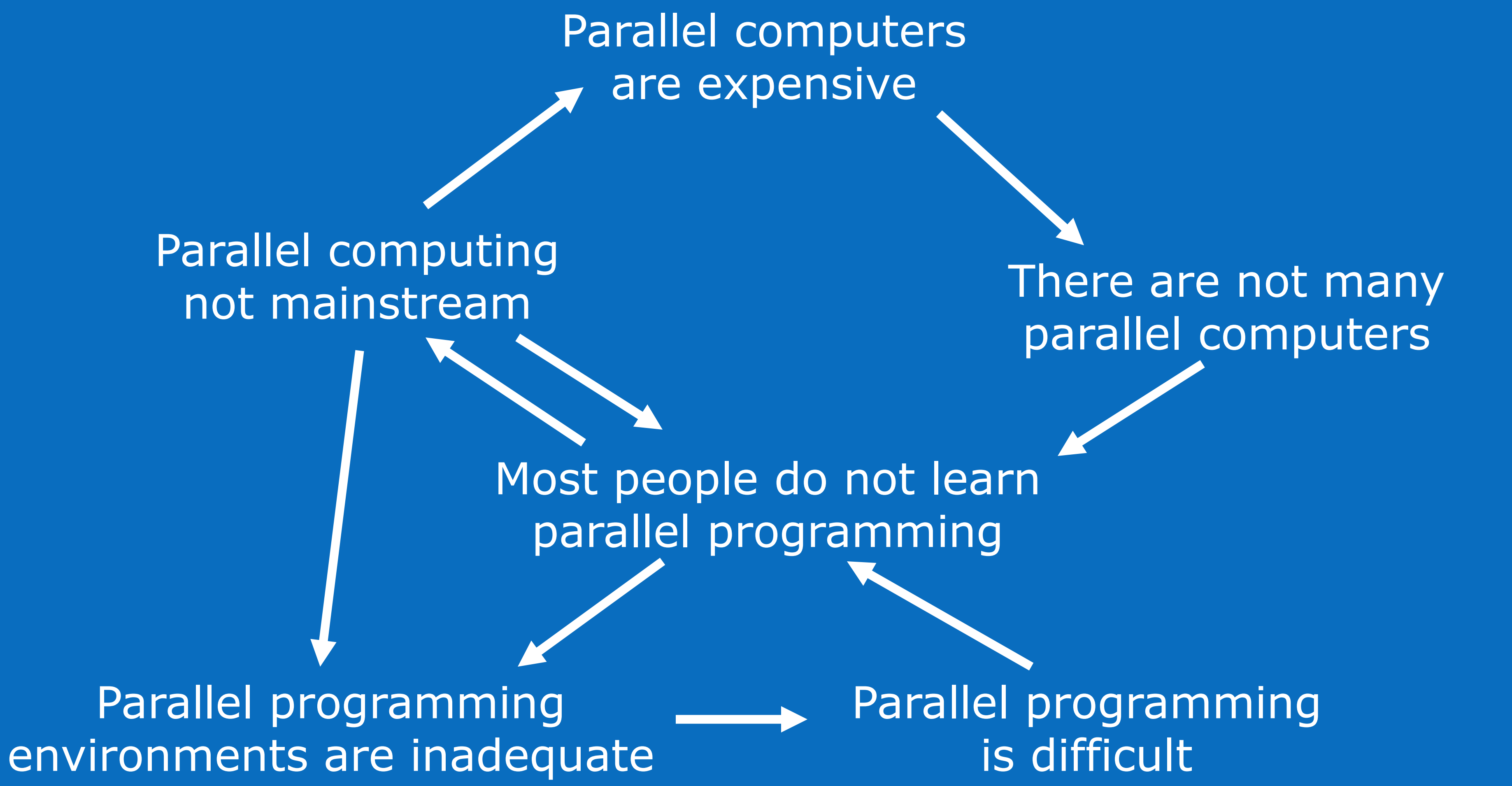
History of Parallel Computing, Part I

Multiple-processor systems supporting parallel computing

1960s: Experimental systems

1980s: Microprocessor-based commercial systems

Old Dynamic of Parallel Computing



Sequential Language Approach

Problem has inherent parallelism

Programming language cannot express parallelism

Programmer hides parallelism in sequential constructs

Compiler and/or hardware must find hidden parallelism

Sadly, doesn't work

Alternative Approach: Programmer and Compiler Work Together

Problem has inherent parallelism

Programmer has way to express parallelism

Compiler translates program for multiple processors

Nothing Radical about a Programmer/Compiler Team

Programmers of modern CPUs must take architecture and compiler into account in order to get peak performance

"…you can actively reorganize data and algorithms to take advantage of architectural capabilities…" *Introduction to Microarchitectural Optimization for Itanium® 2 Processors*, p. 3

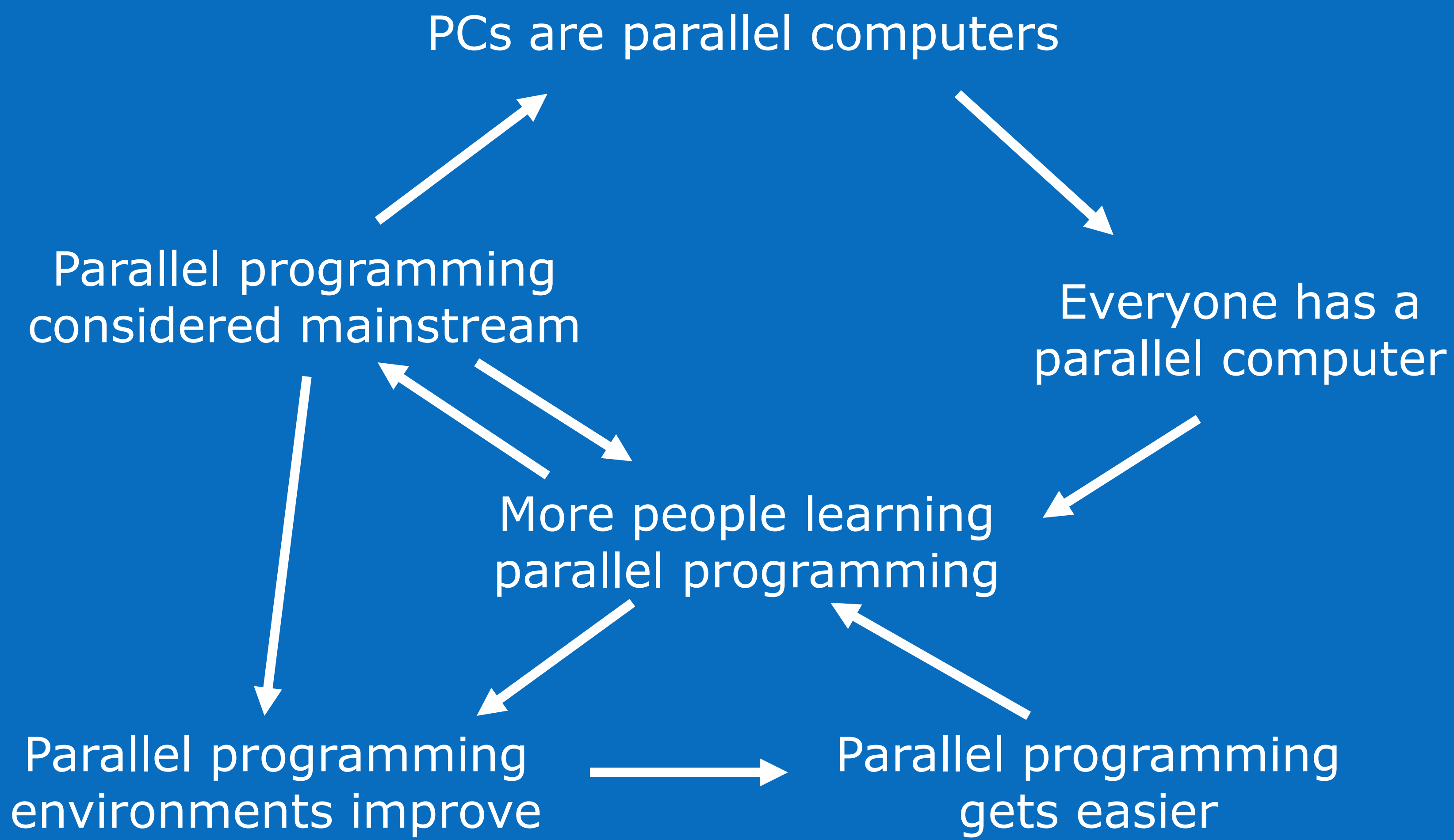
History of Parallel Computing, Part II

2004: Intel demos Montecito dual-core CPU

2006: Intel demos Clovertown quad-core CPU

Clovertown scalable to 32+ cores in a single package

New Dynamic of Parallel Computing



Methodology

Study problem, sequential program, or code segment

Look for opportunities for parallelism

Try to keep all processors busy doing useful work

Ways of Exploiting Parallelism

Domain decomposition

Task decomposition

Pipelining

Domain Decomposition

First, decide how data elements should be divided among processors

Second, decide which tasks each processor should be doing

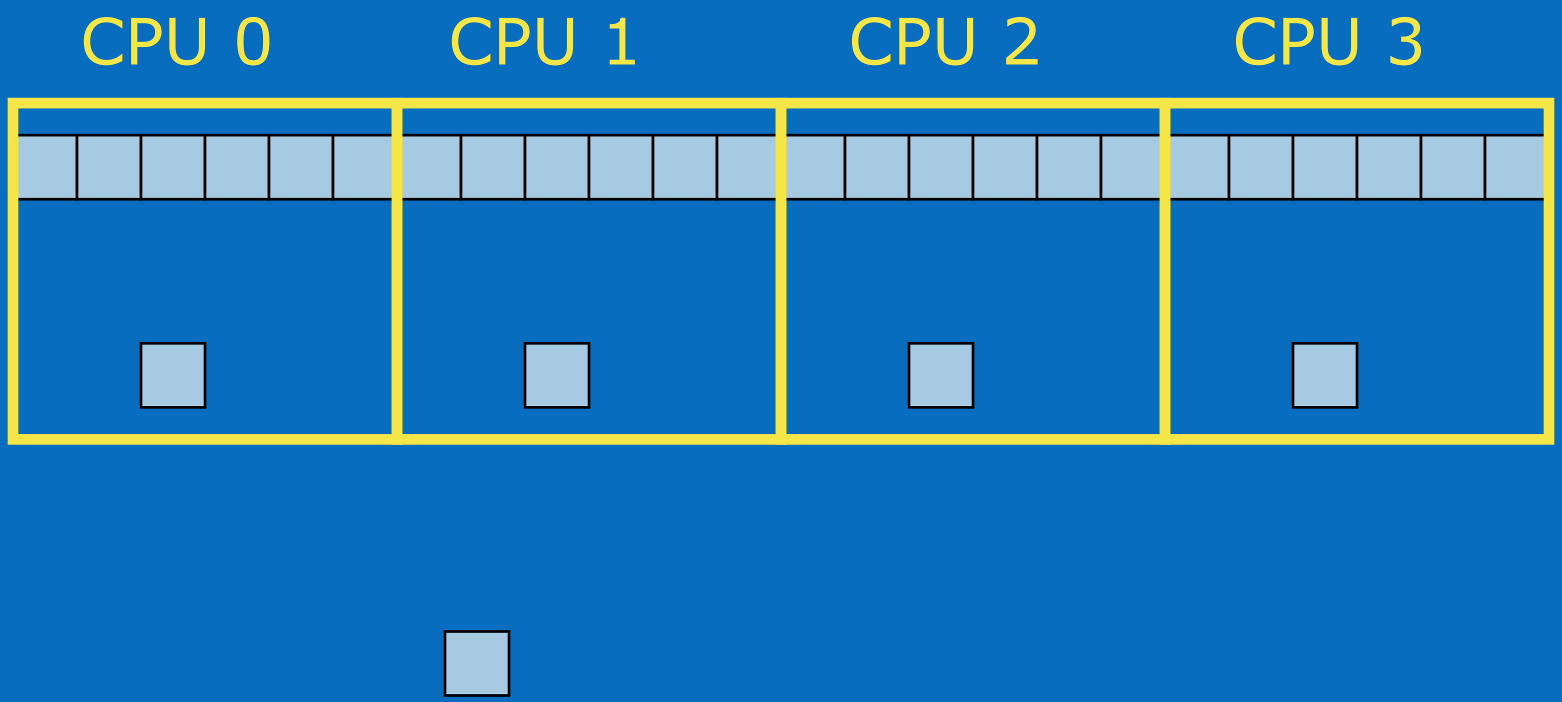
Example: Vector addition

Domain Decomposition

Find the largest element of an array

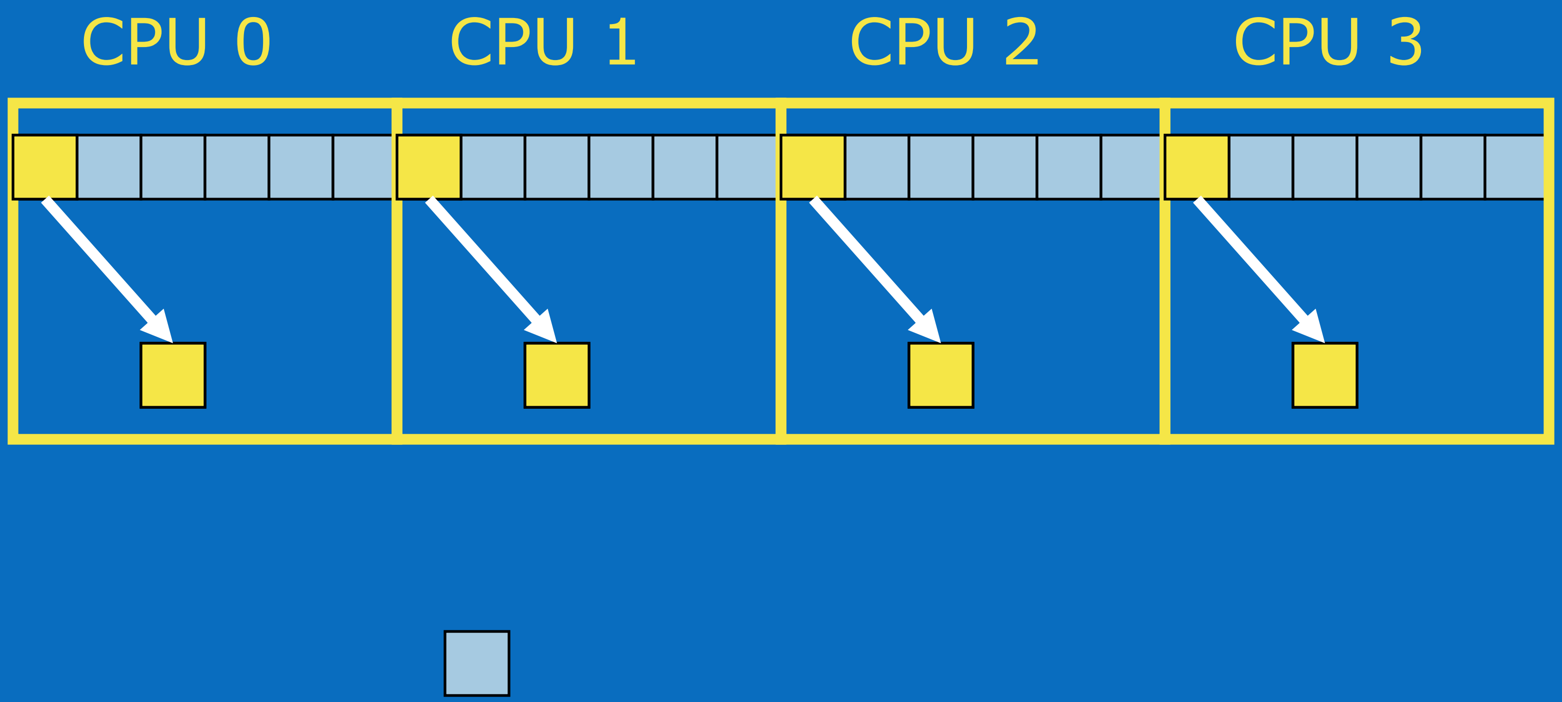
Domain Decomposition

Find the largest element of an array



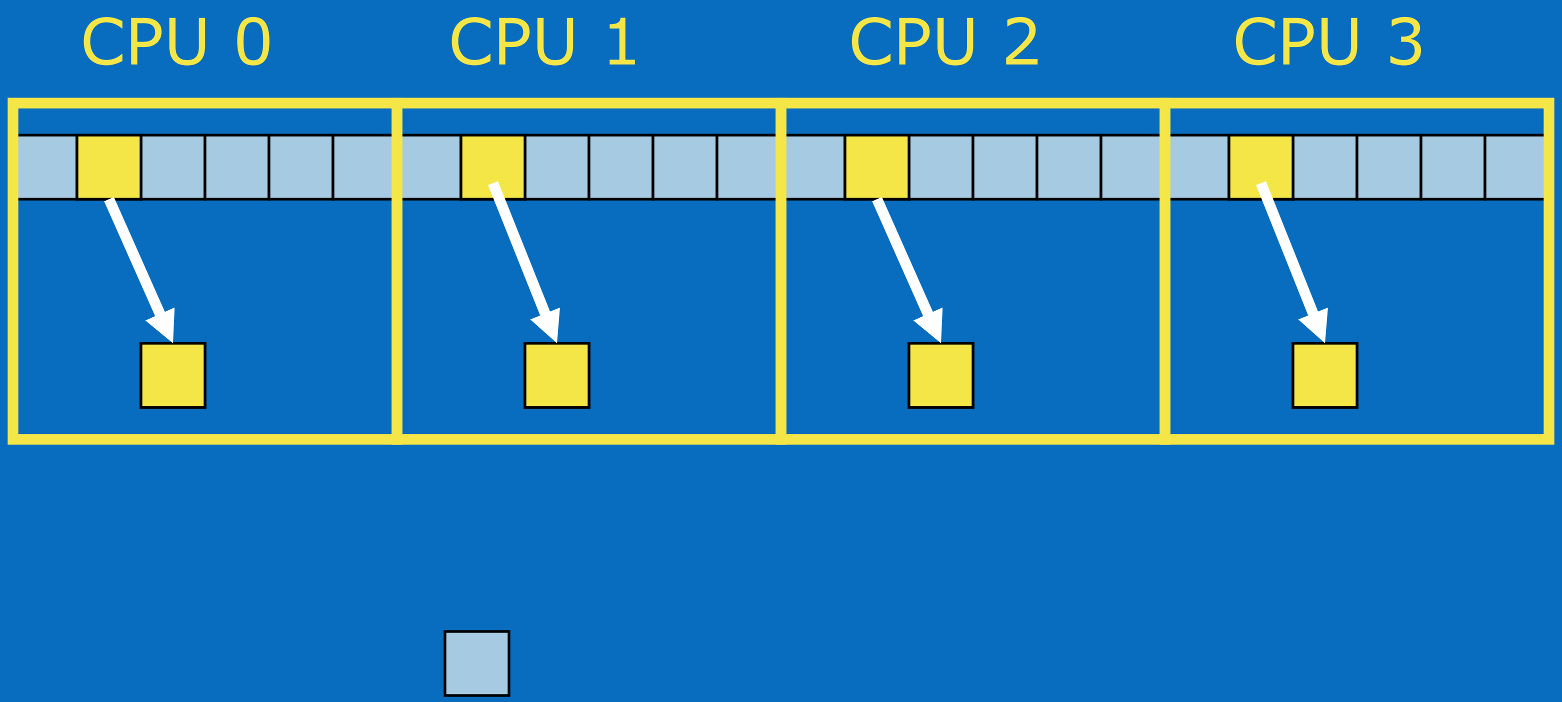
Domain Decomposition

Find the largest element of an array



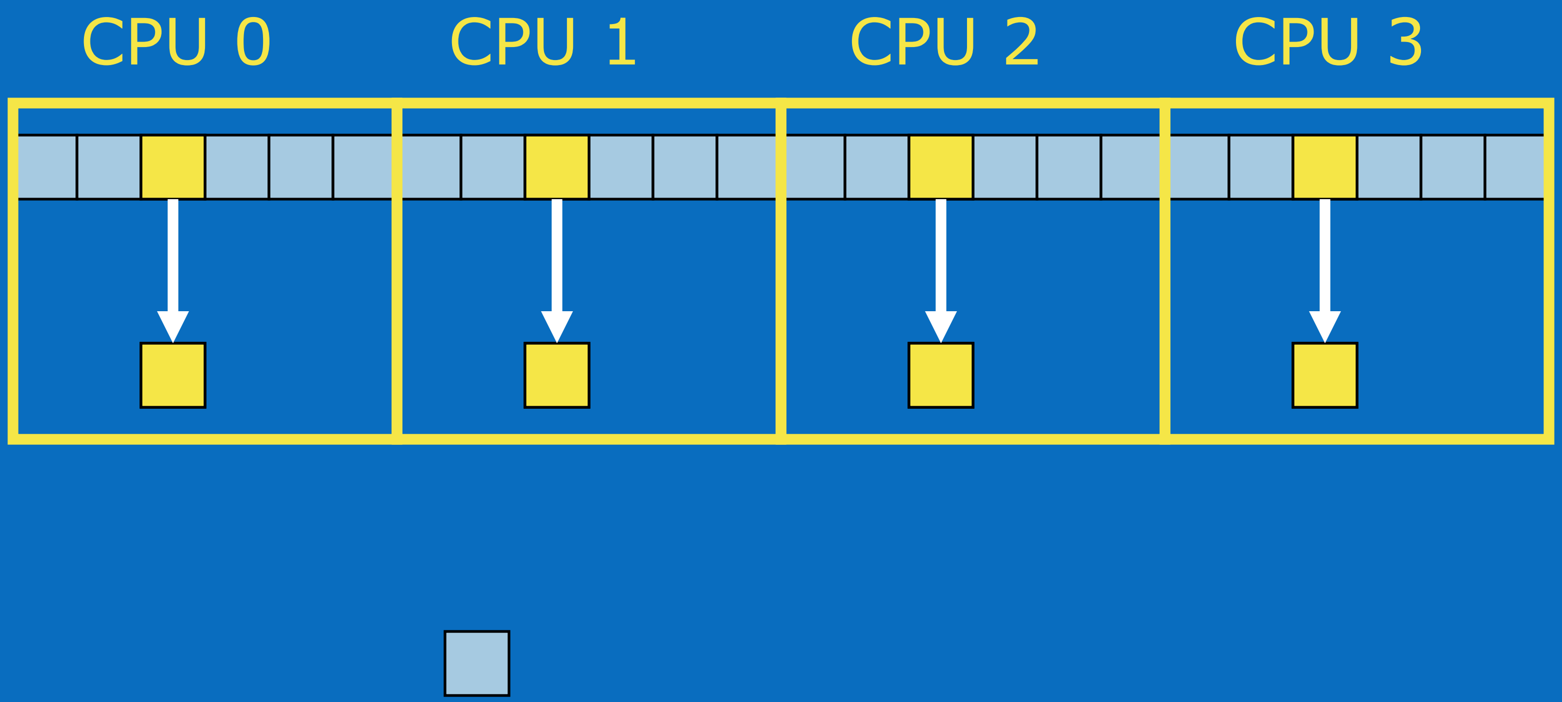
Domain Decomposition

Find the largest element of an array



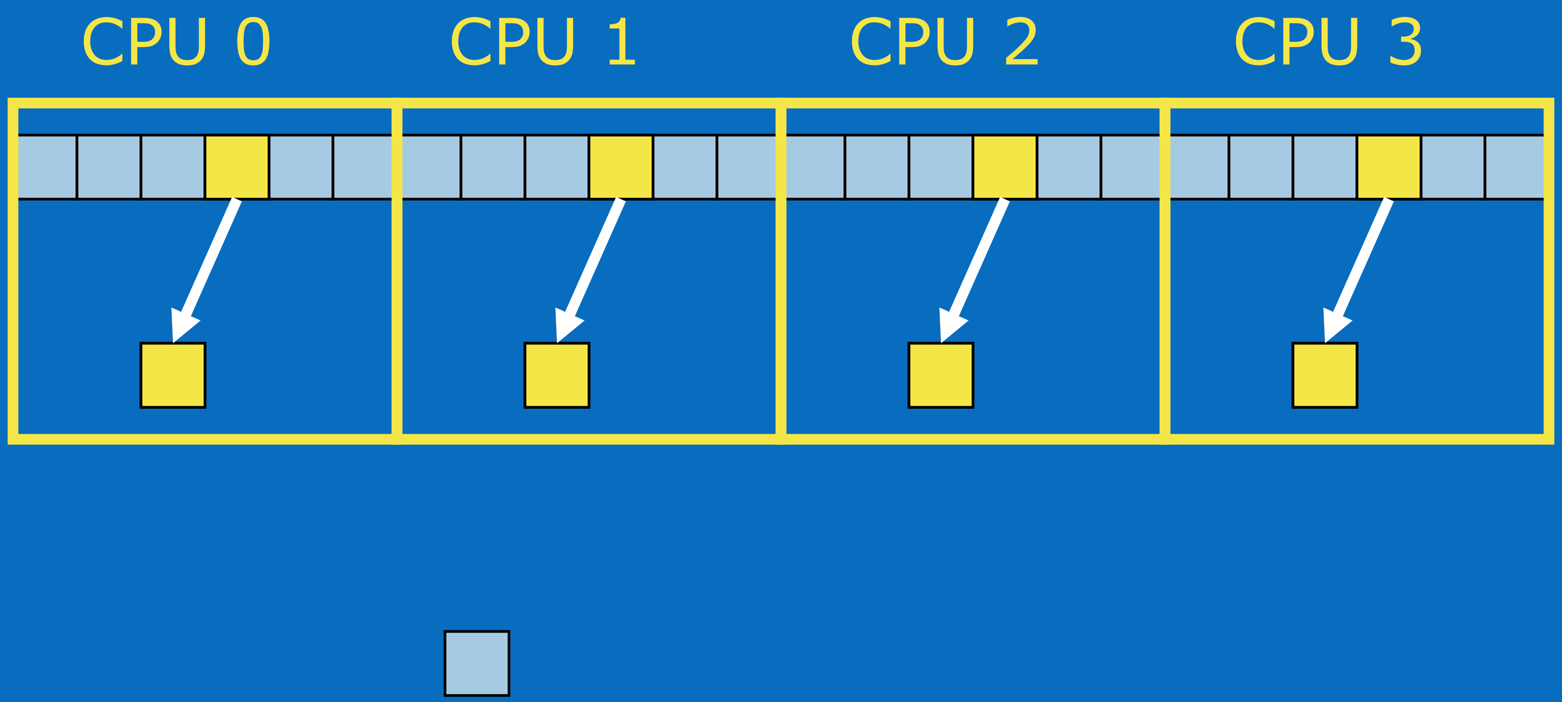
Domain Decomposition

Find the largest element of an array



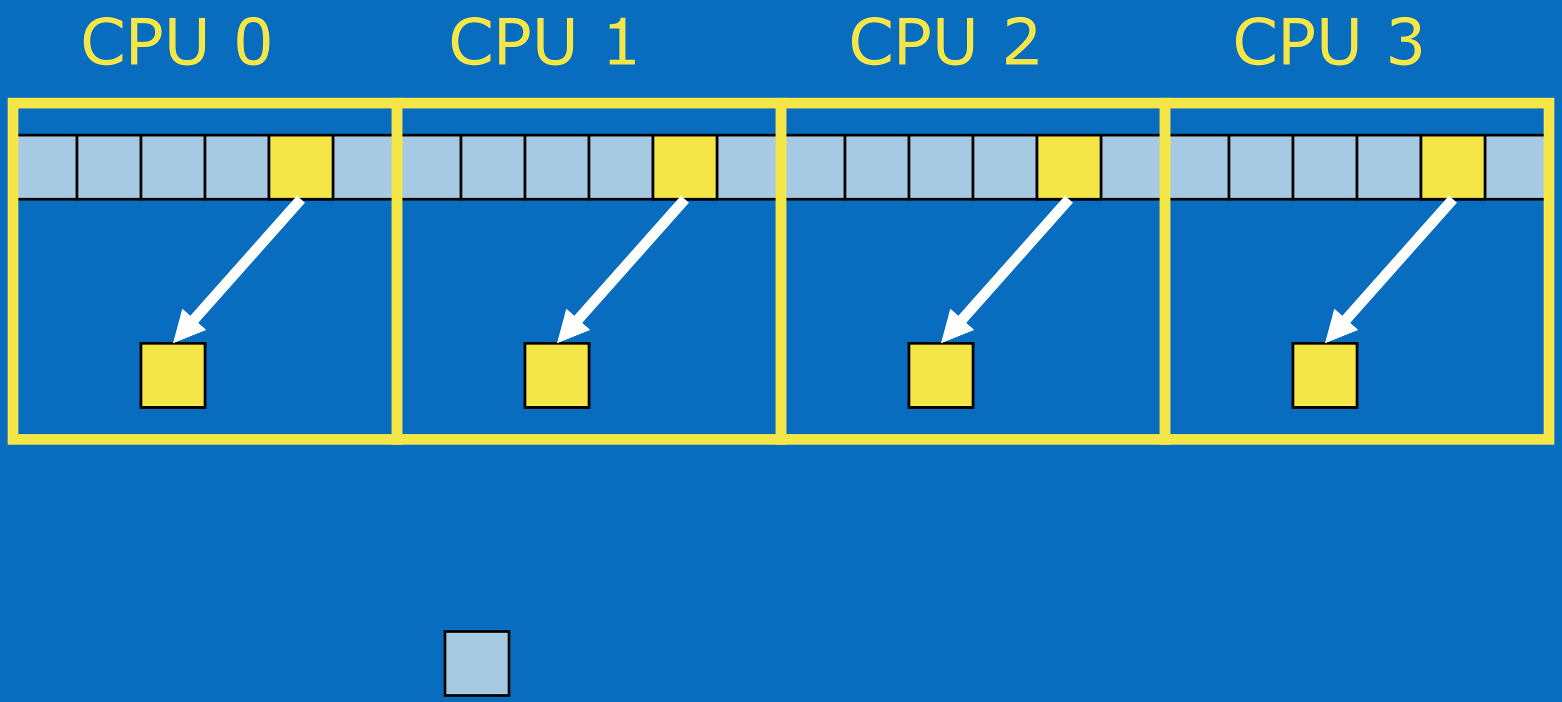
Domain Decomposition

Find the largest element of an array



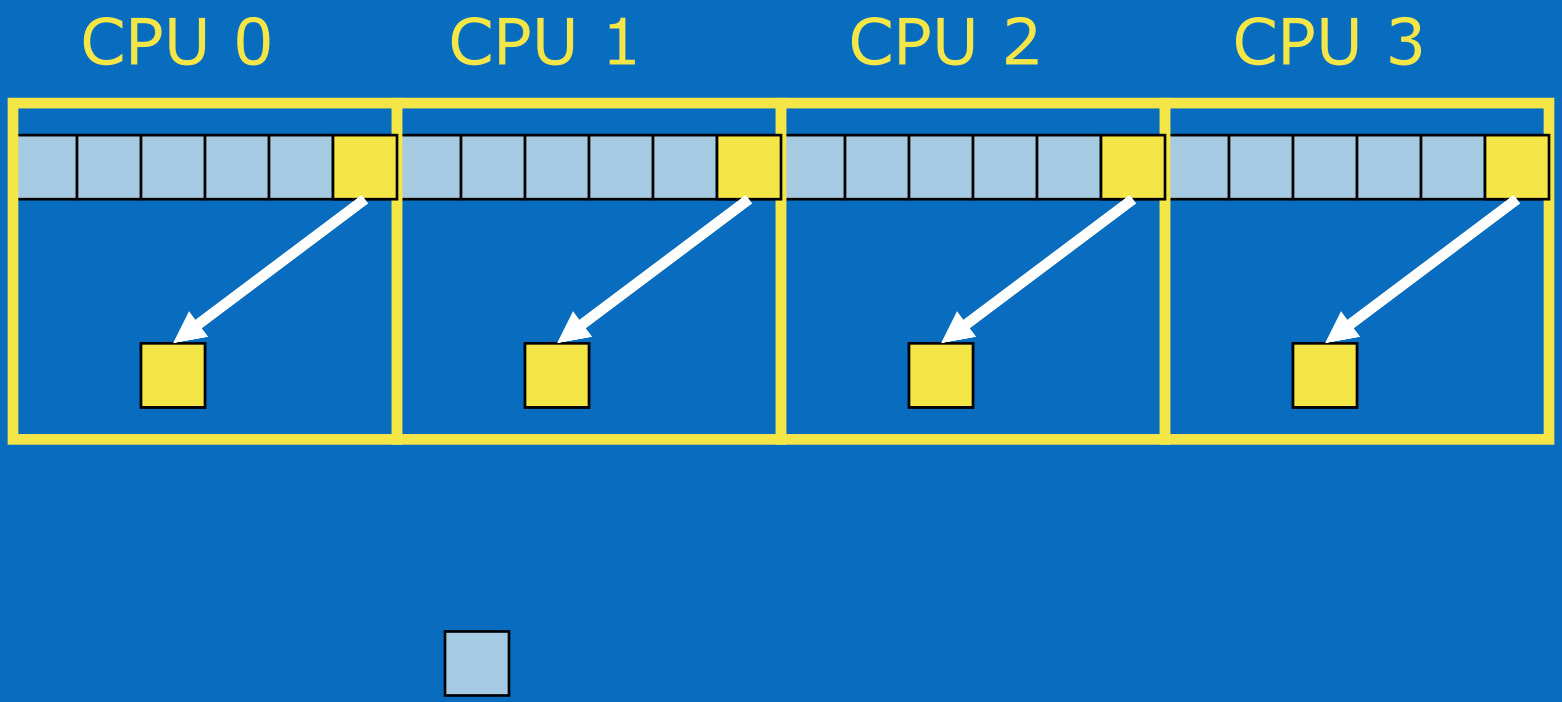
Domain Decomposition

Find the largest element of an array



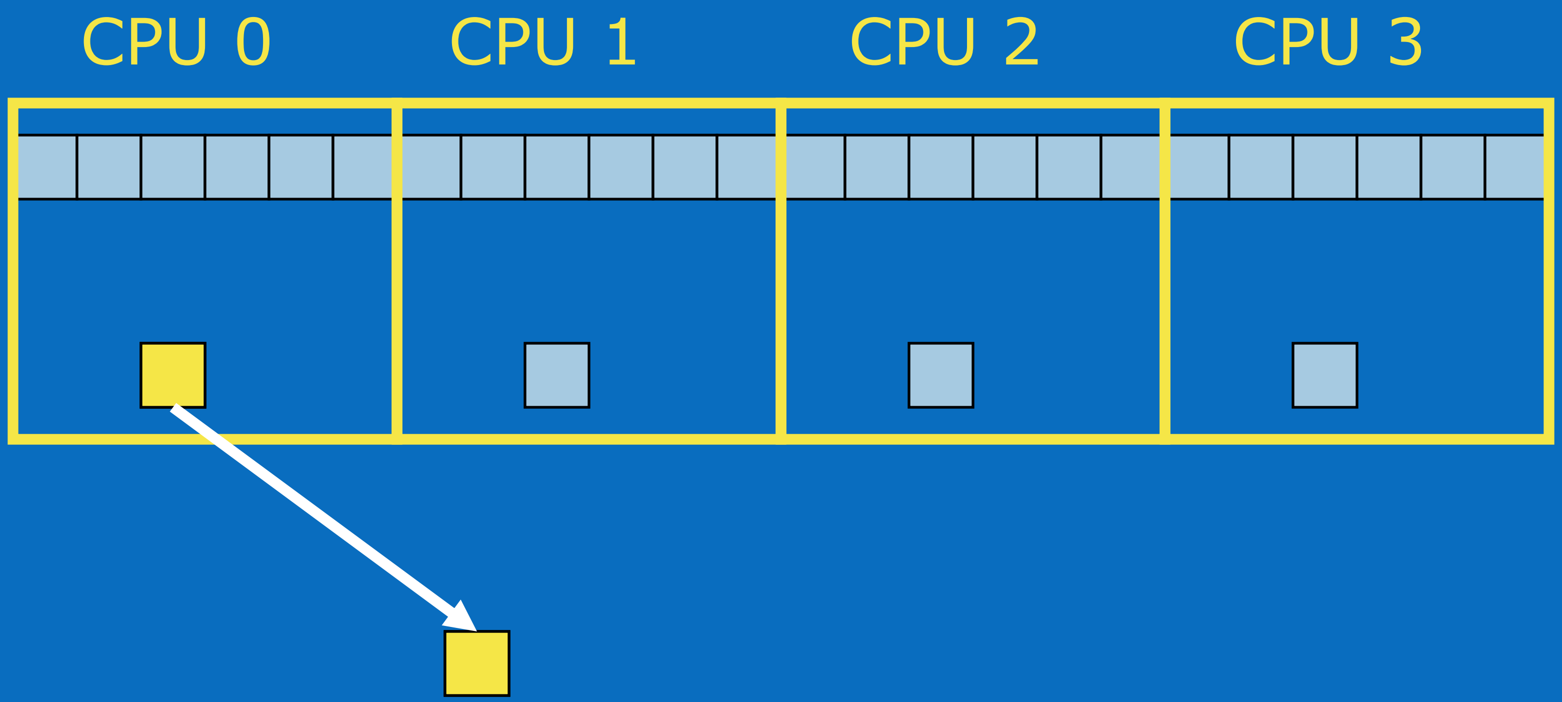
Domain Decomposition

Find the largest element of an array



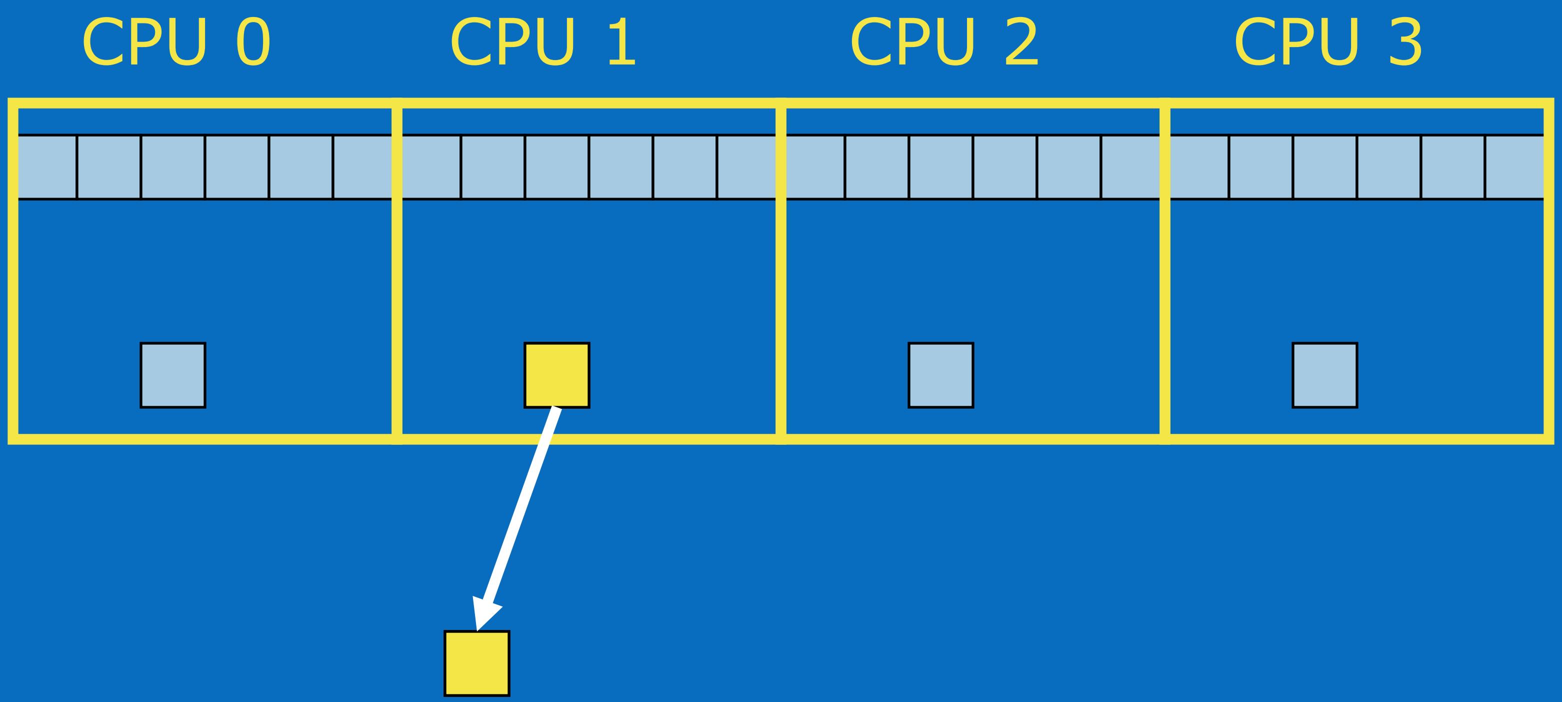
Domain Decomposition

Find the largest element of an array



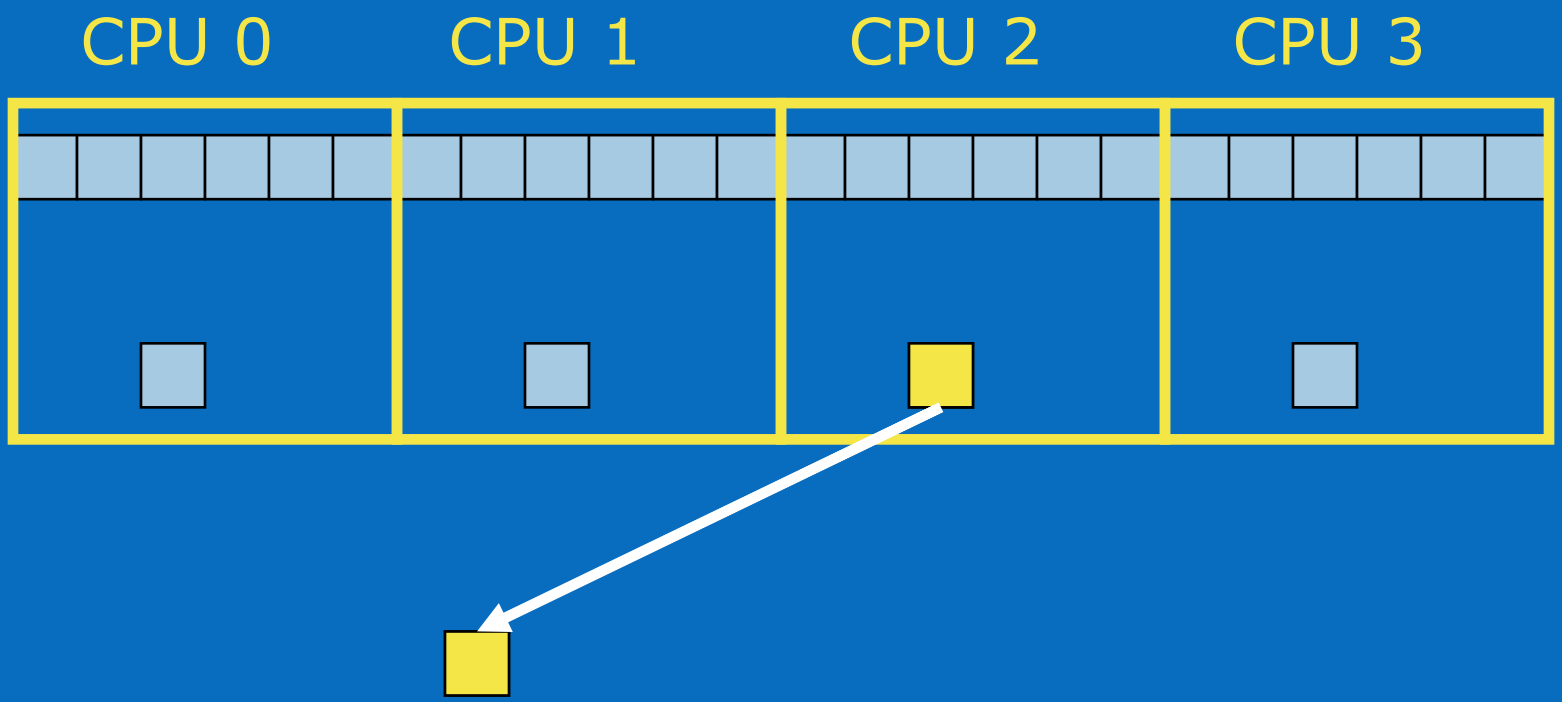
Domain Decomposition

Find the largest element of an array



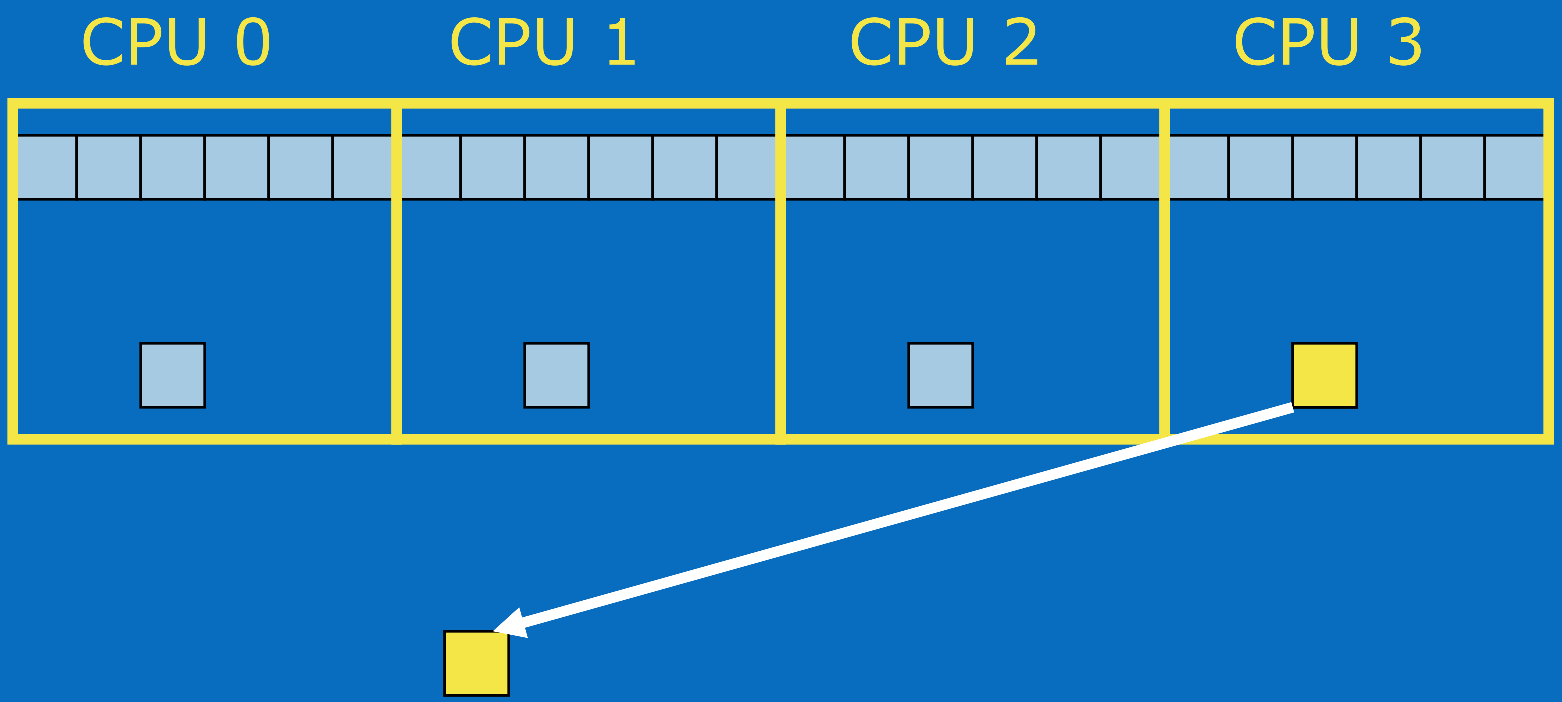
Domain Decomposition

Find the largest element of an array



Domain Decomposition

Find the largest element of an array



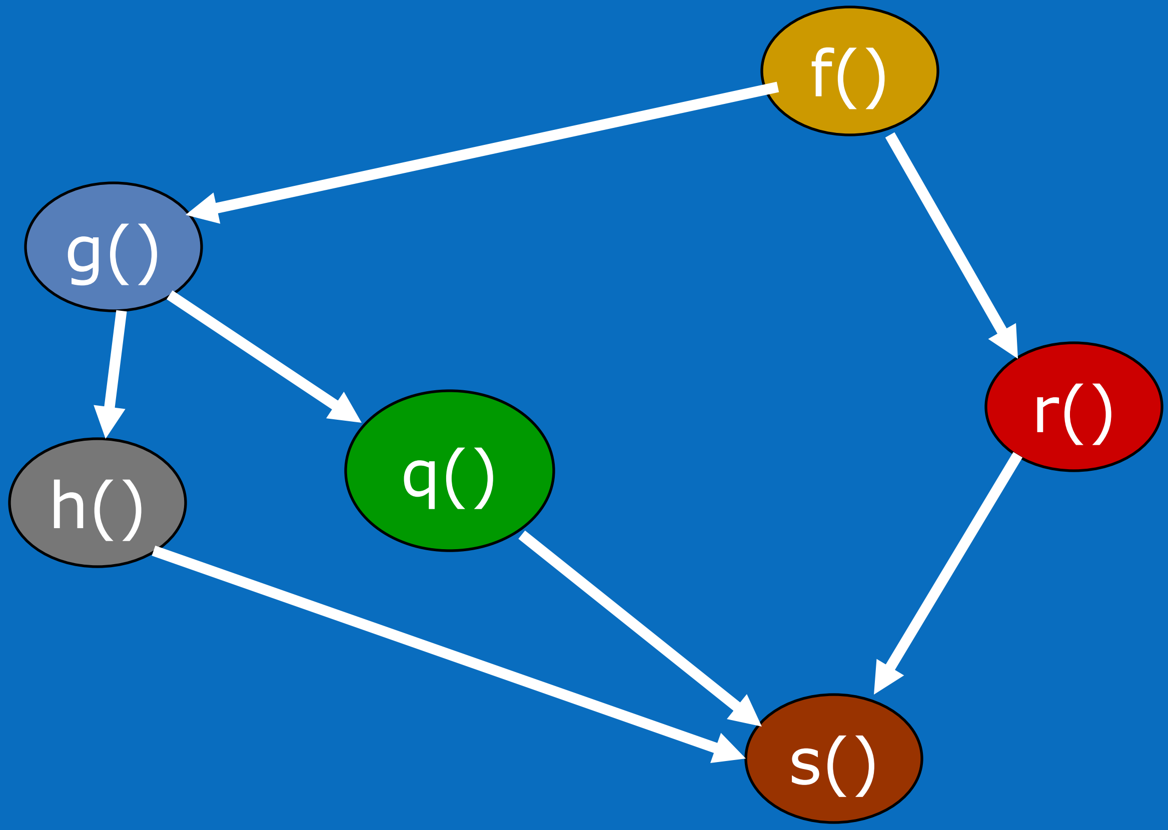
Task (Functional) Decomposition

First, divide tasks among processors

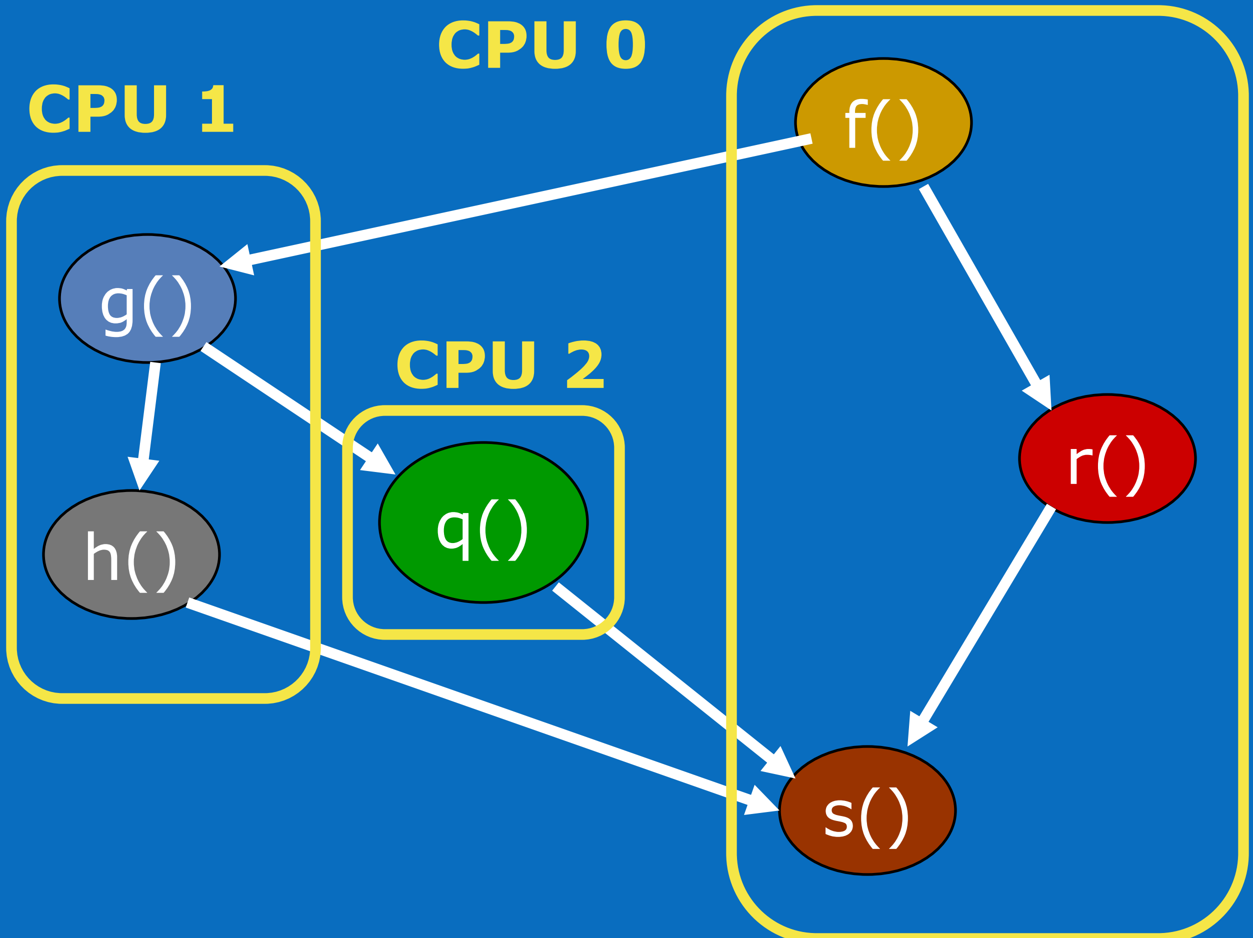
Second, decide which data elements are going to be accessed (read and/or written) by which processors

Example: Event-handler for GUI

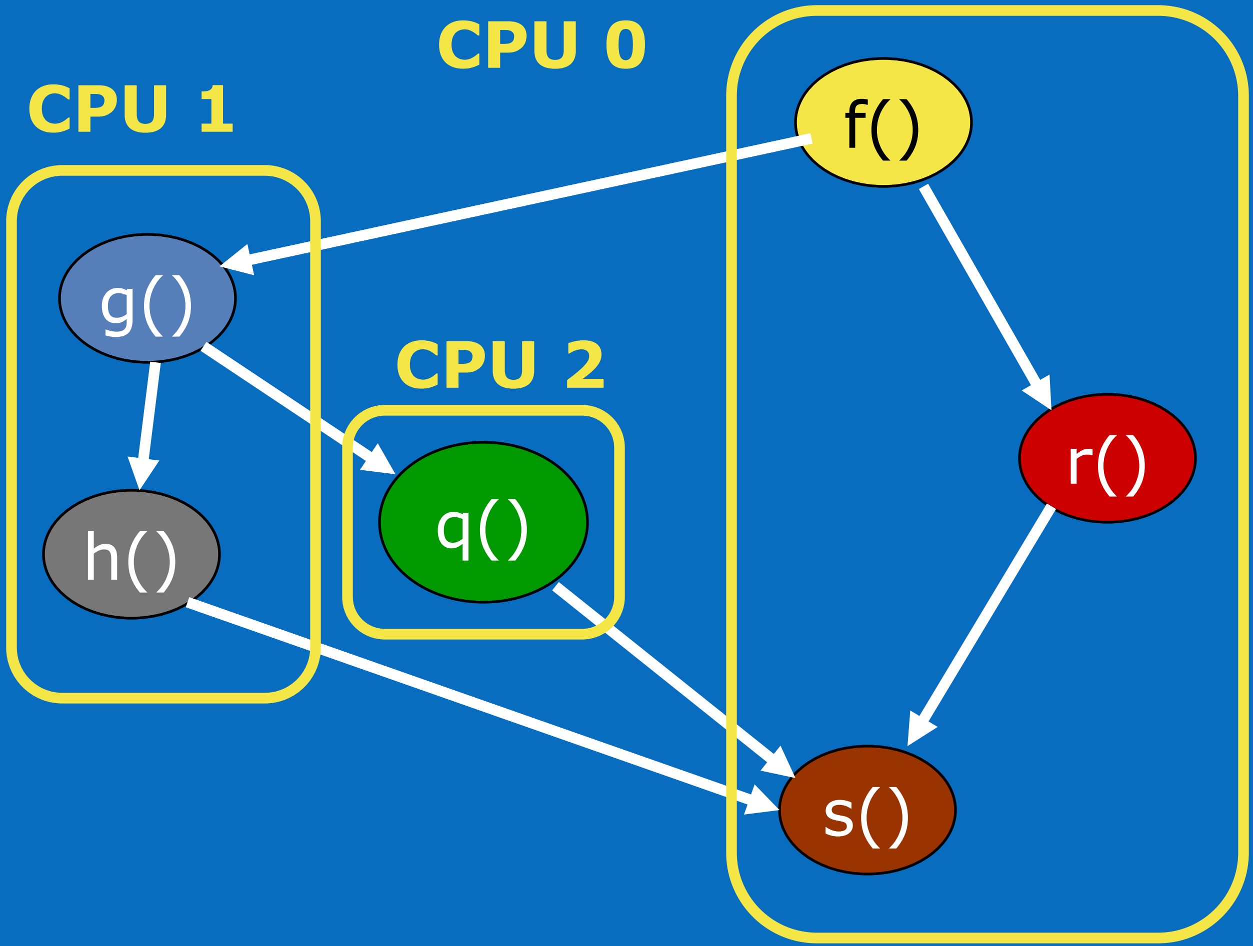
Task Decomposition



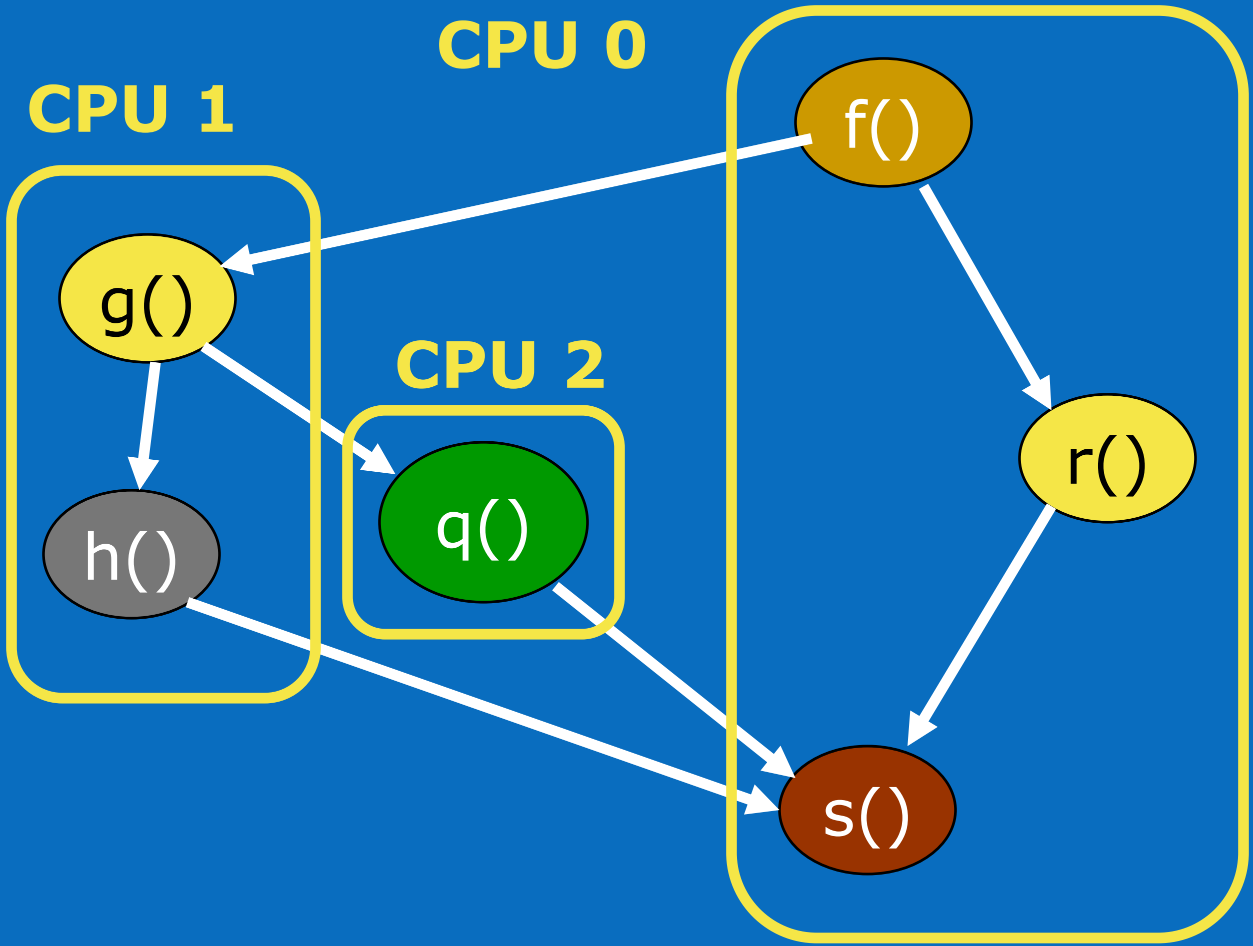
Task Decomposition



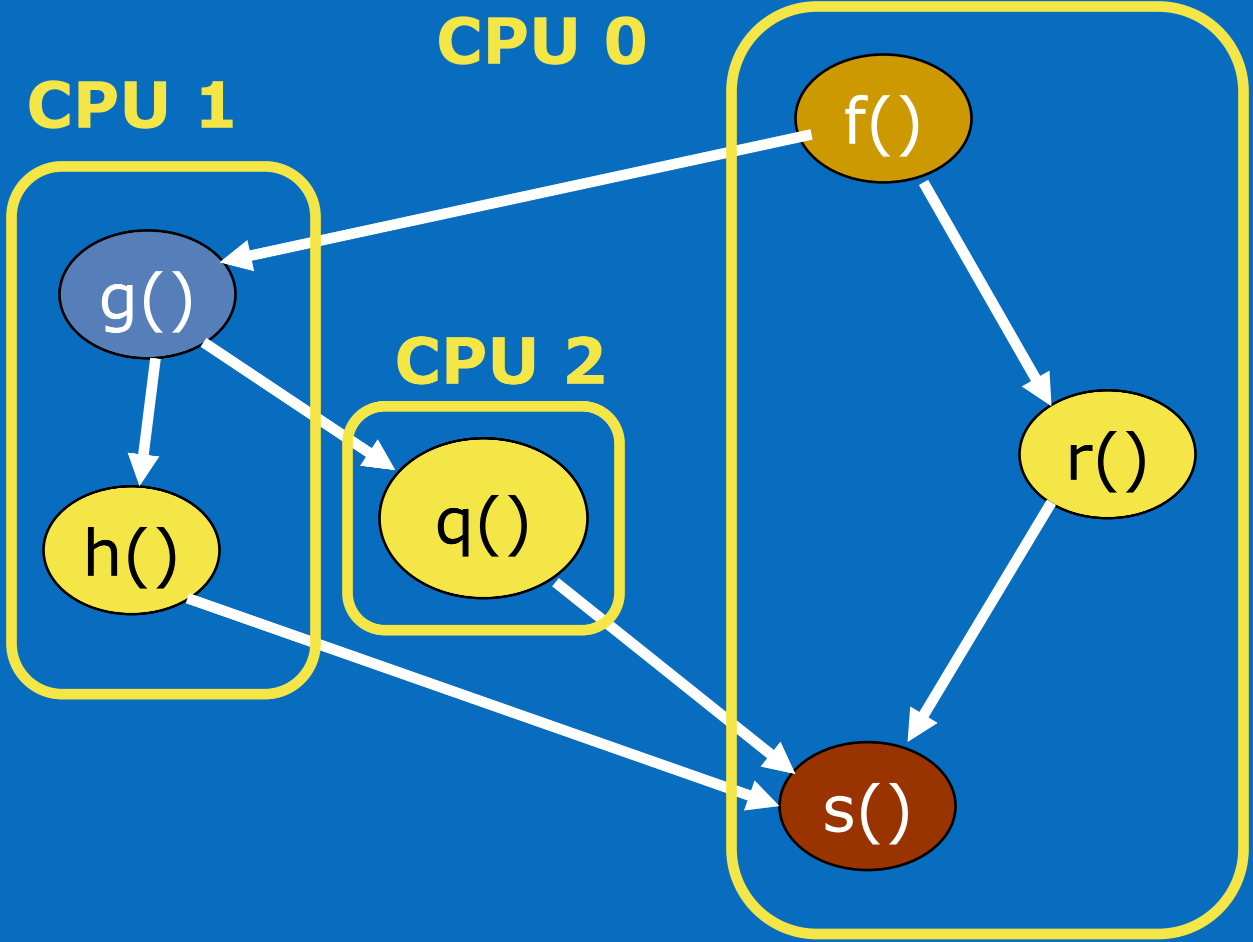
Task Decomposition



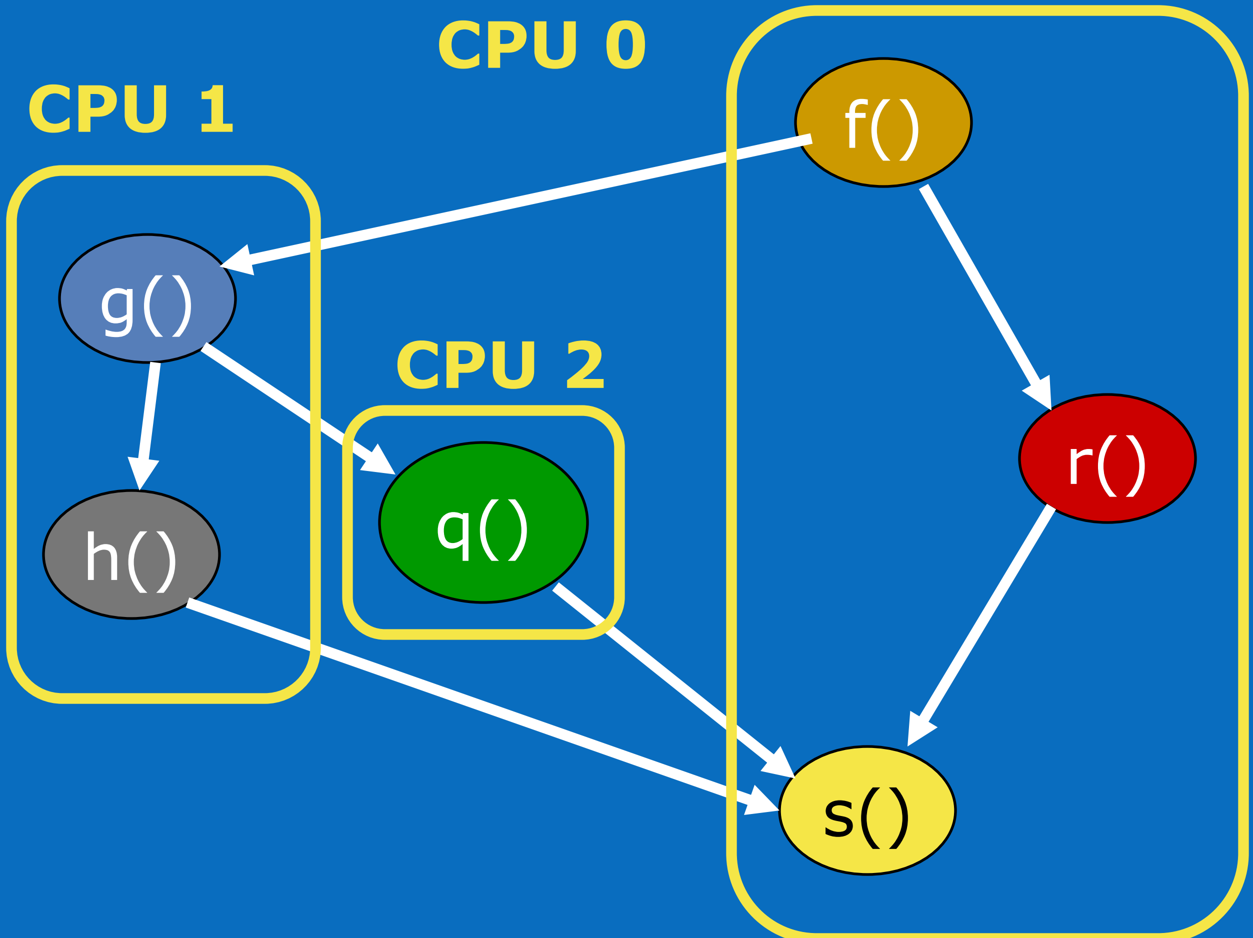
Task Decomposition



Task Decomposition



Task Decomposition

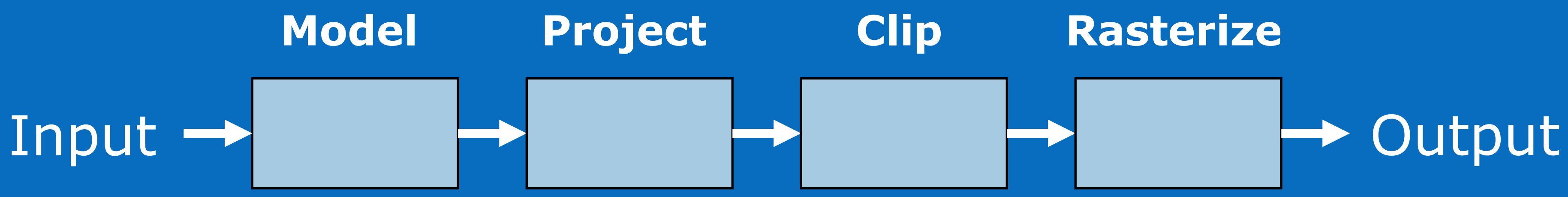


Pipelining

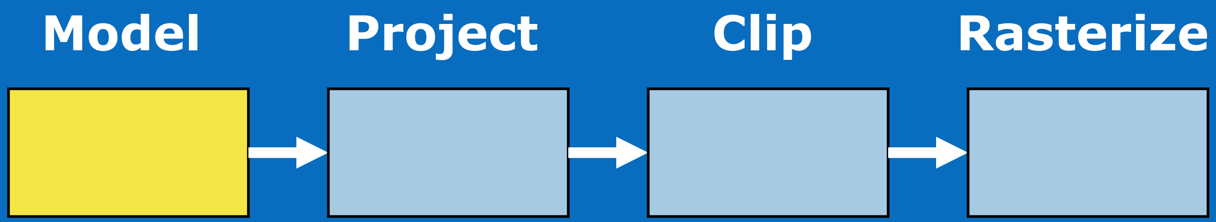
Special kind of task decomposition

"Assembly line" parallelism

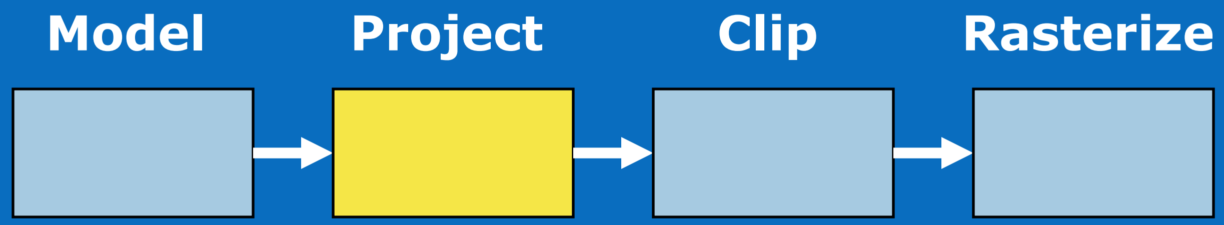
Example: 3D rendering in computer graphics



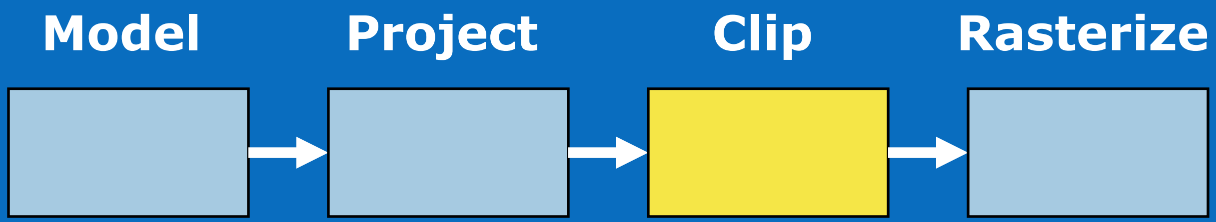
Processing One Data Set (Step 1)



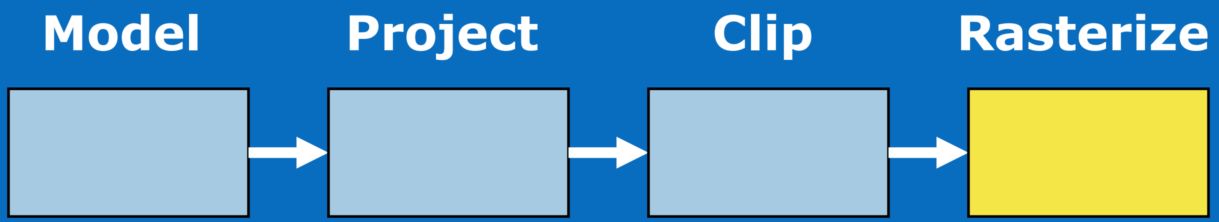
Processing One Data Set (Step 2)



Processing One Data Set (Step 3)

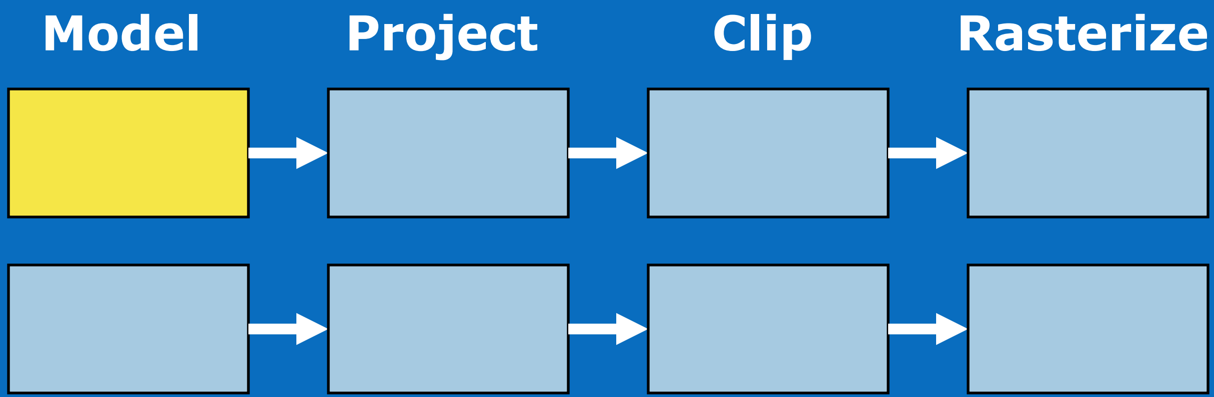


Processing One Data Set (Step 4)

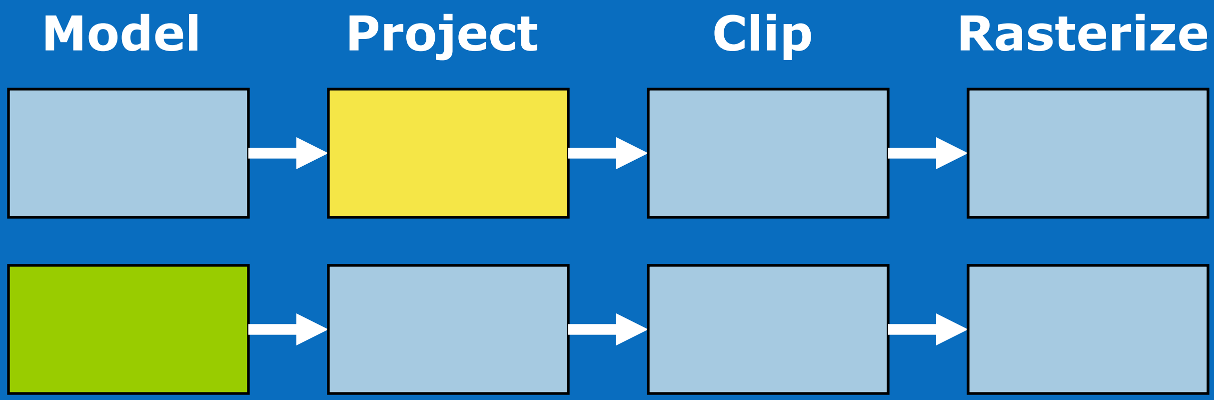


The pipeline processes 1 data set in 4 steps

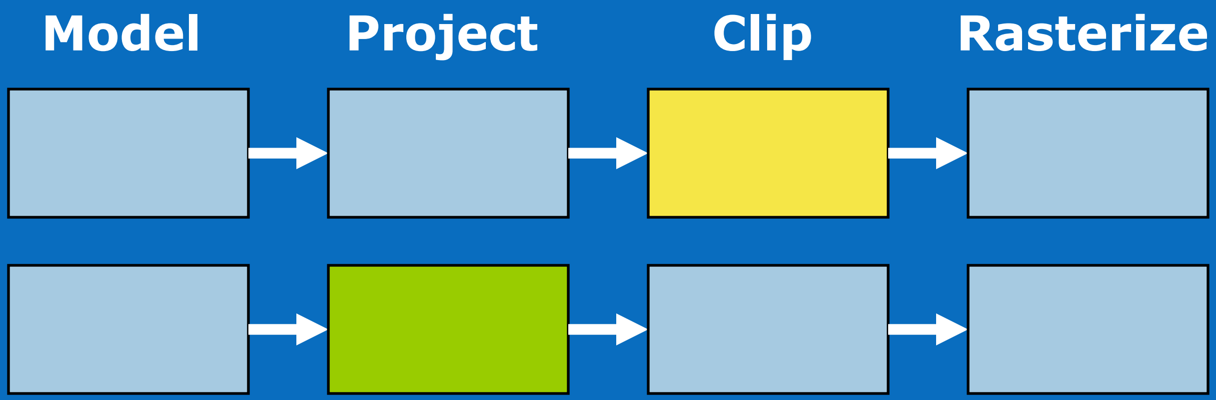
Processing Two Data Sets (Step 1)



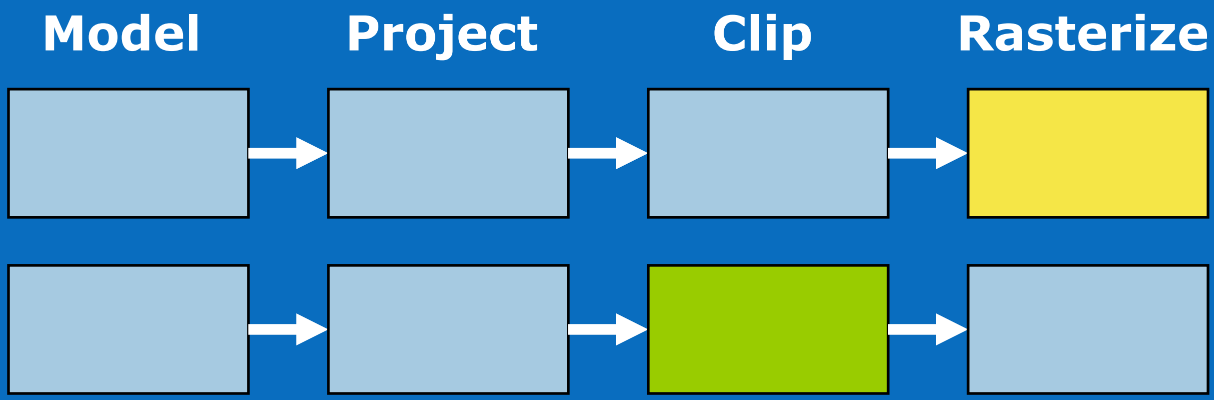
Processing Two Data Sets (Time 2)



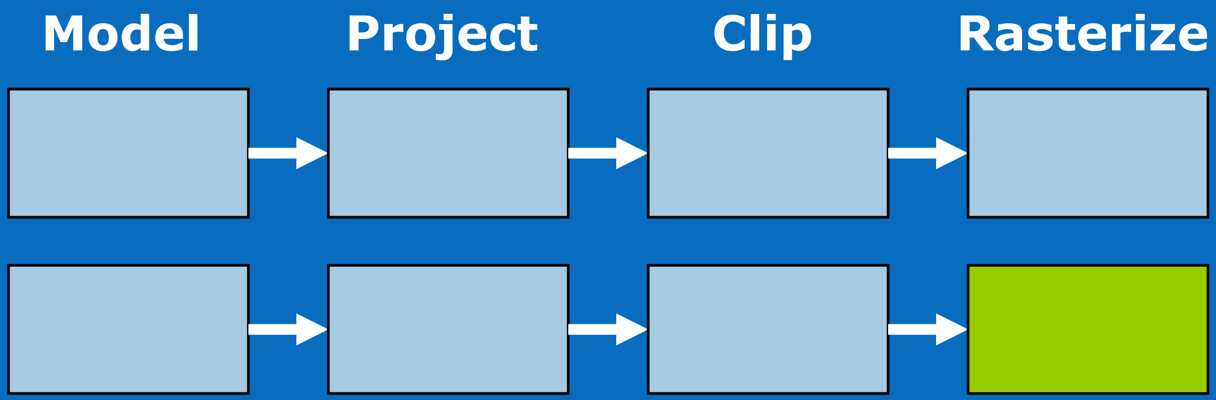
Processing Two Data Sets (Step 3)



Processing Two Data Sets (Step 4)

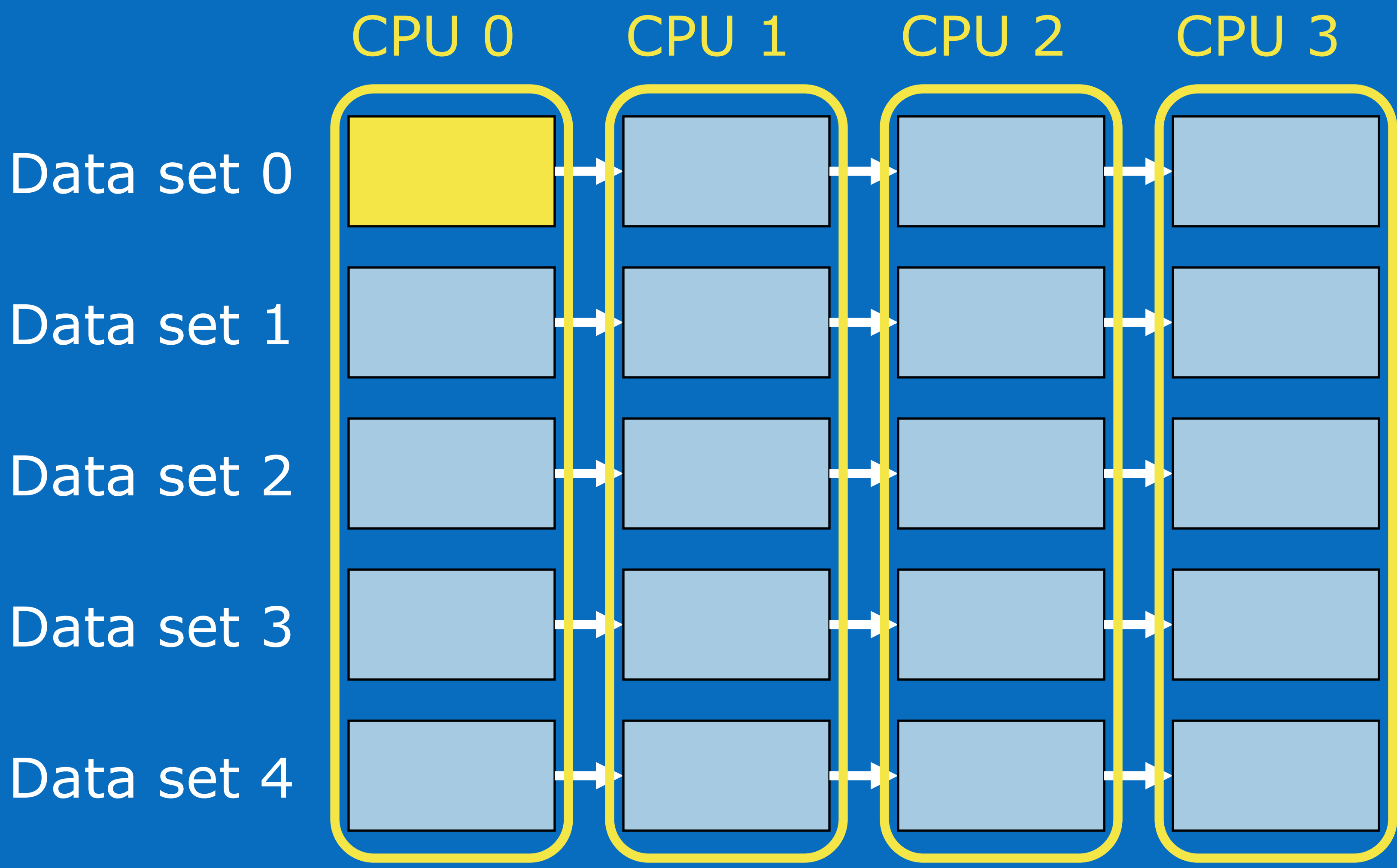


Processing Two Data Sets (Step 5)

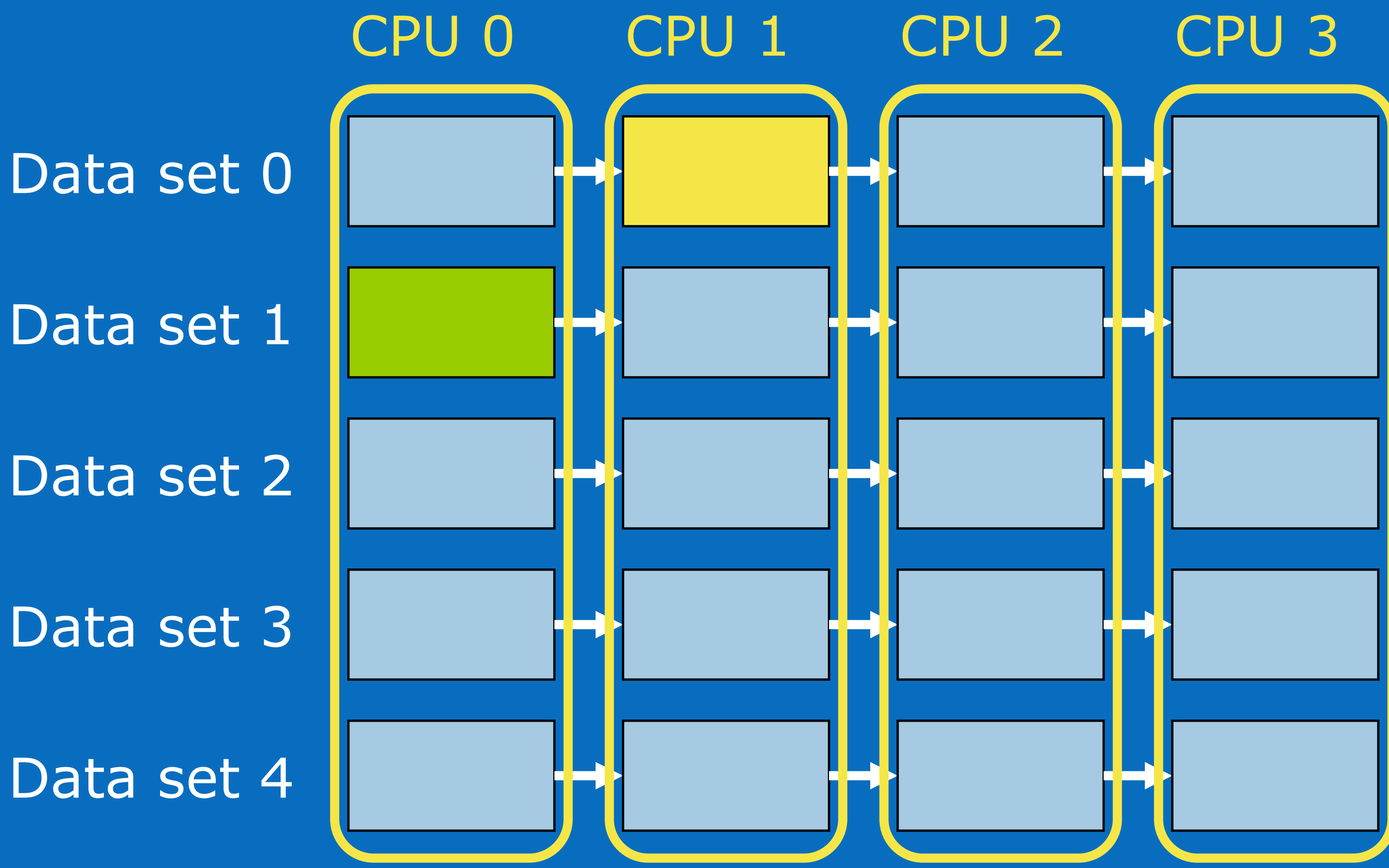


The pipeline processes 2 data sets in 5 steps

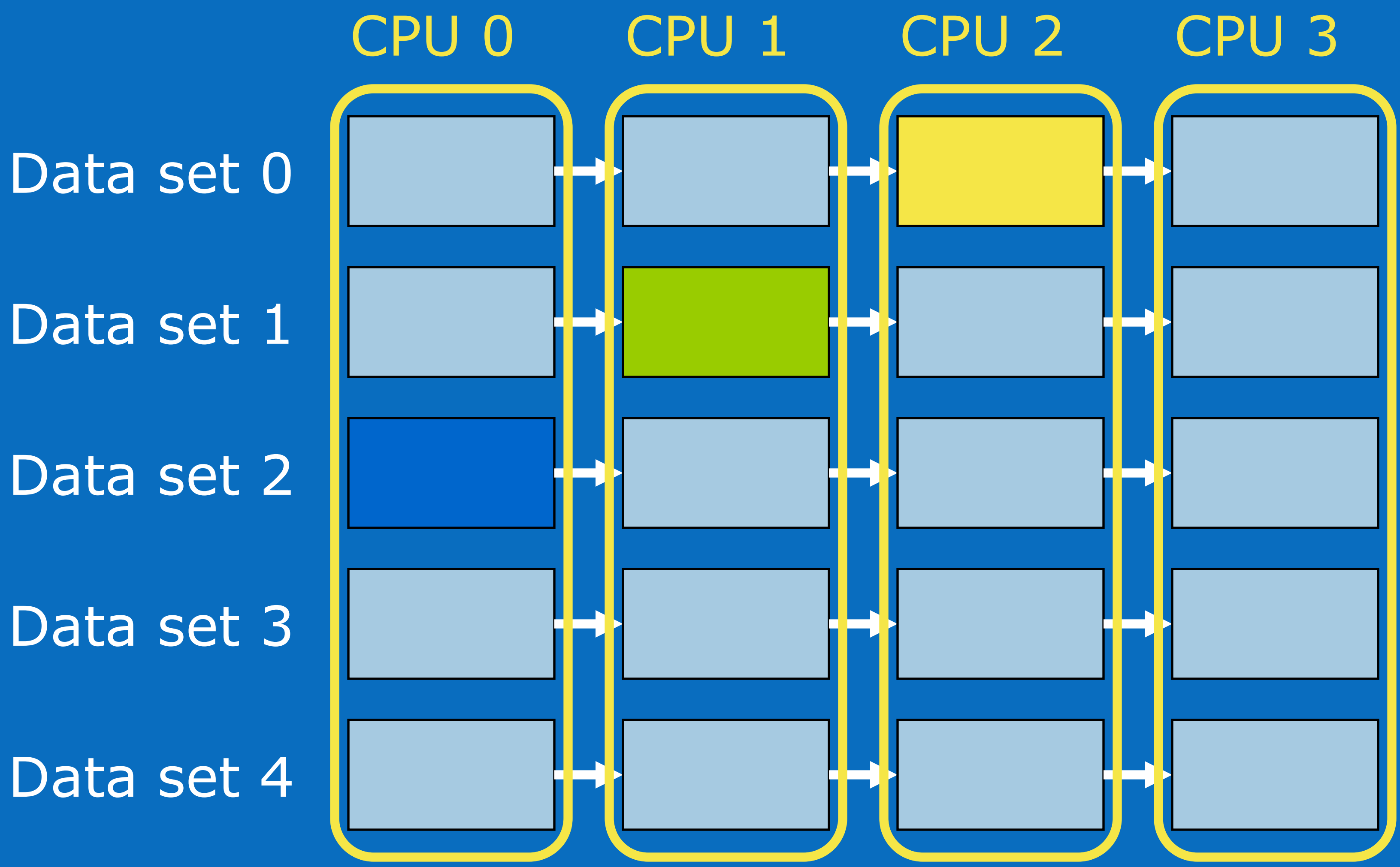
Pipelining Five Data Sets (Step 1)



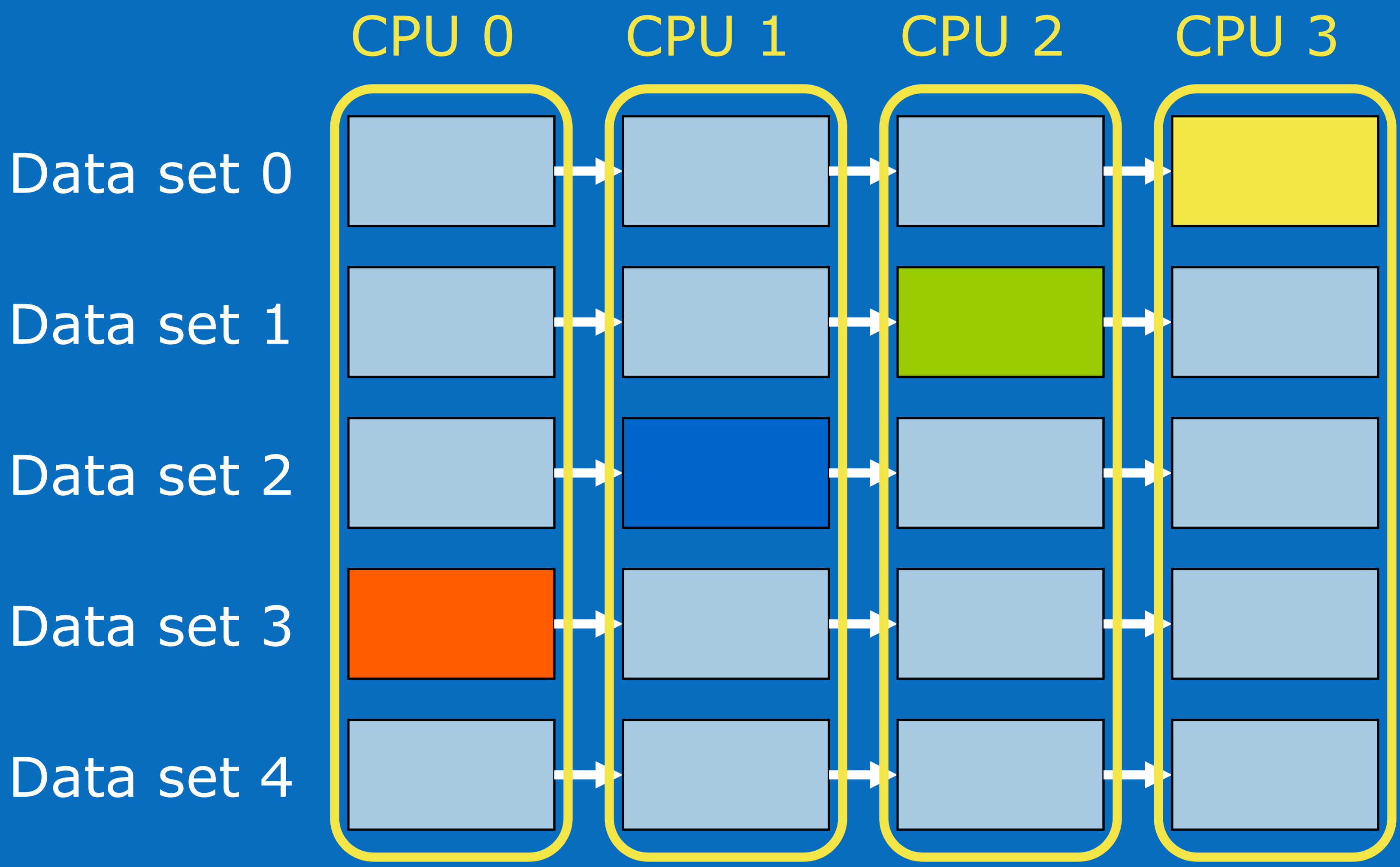
Pipelining Five Data Sets (Step 2)



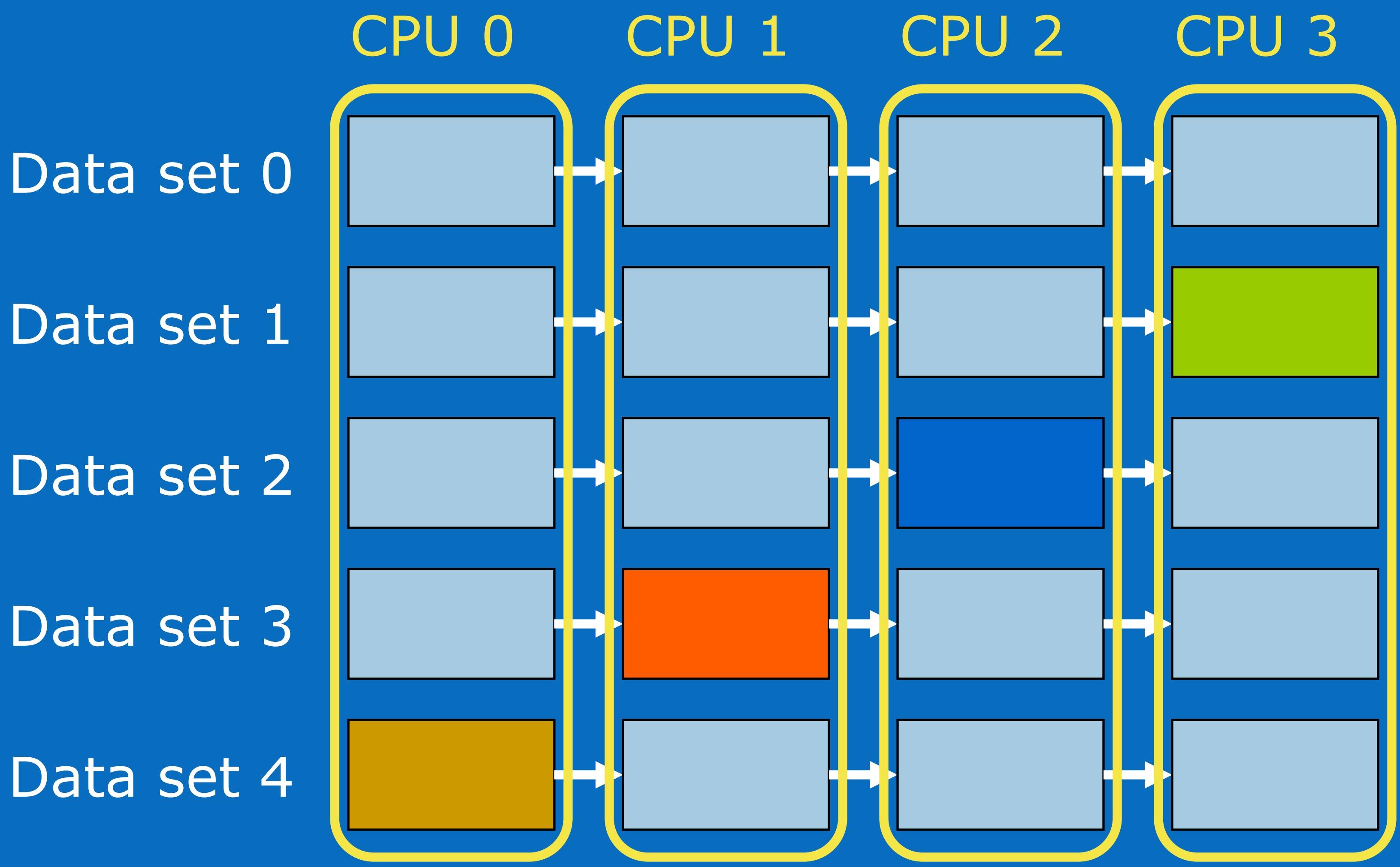
Pipelining Five Data Sets (Step 3)



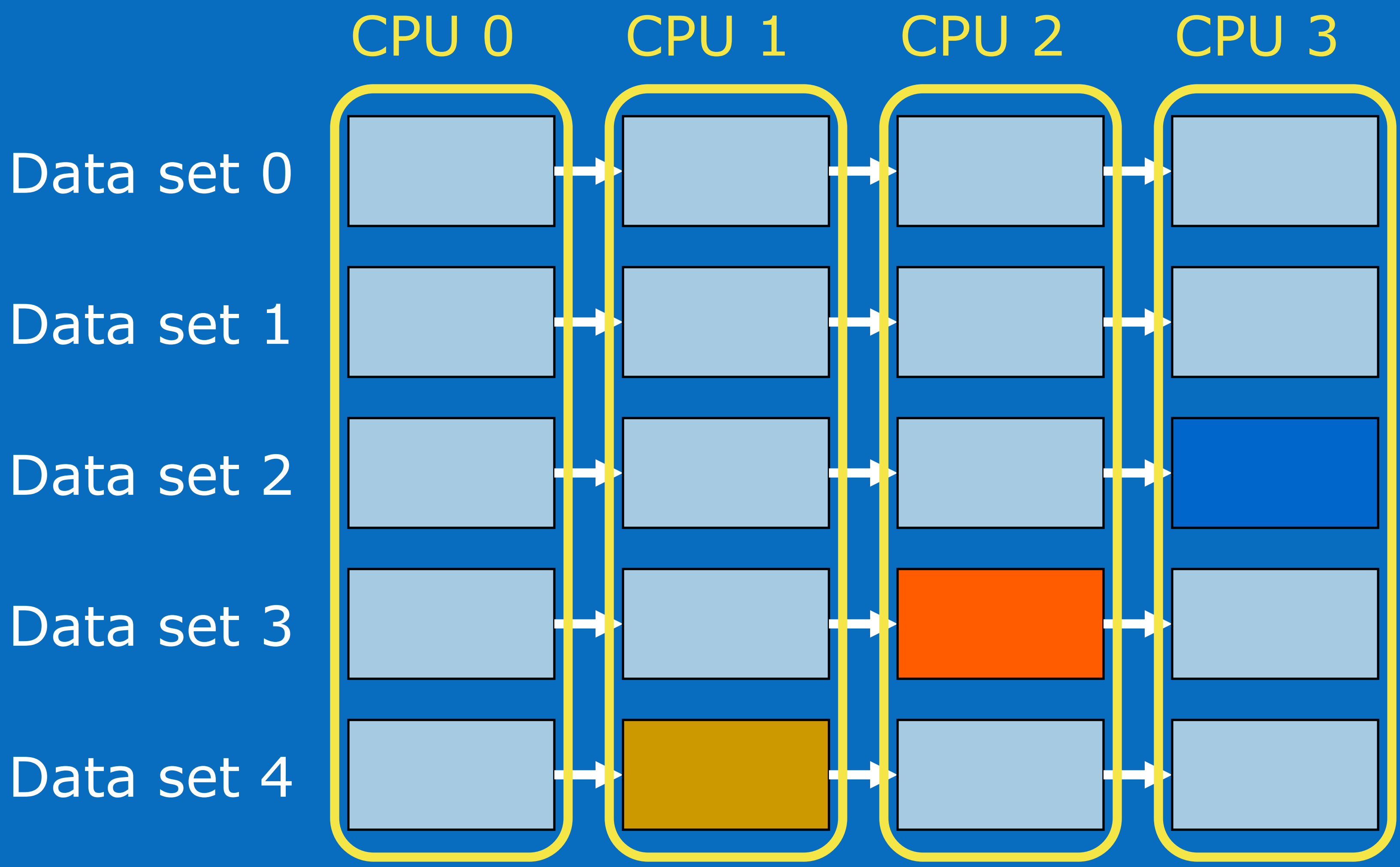
Pipelining Five Data Sets (Step 4)



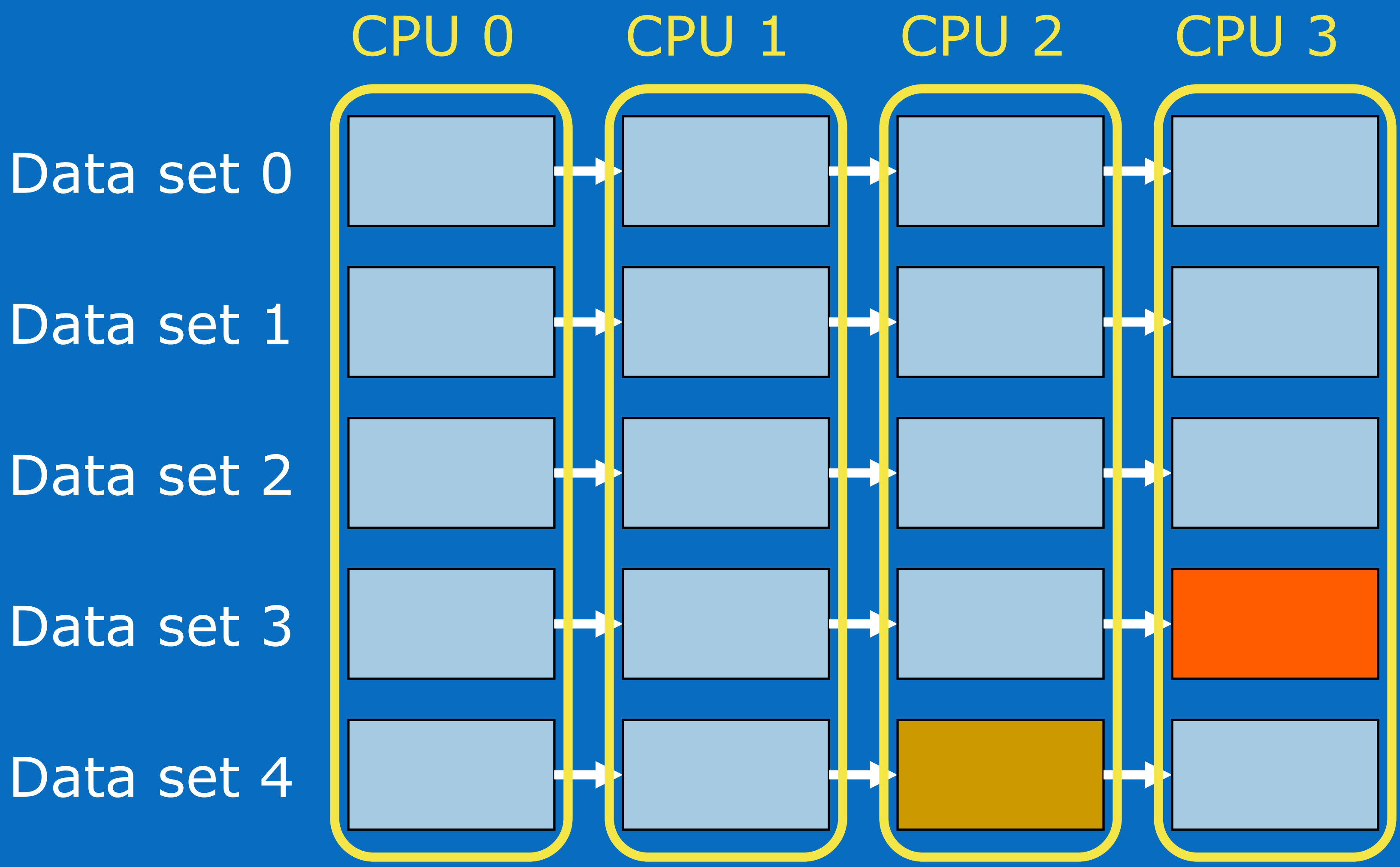
Pipelining Five Data Sets (Step 5)



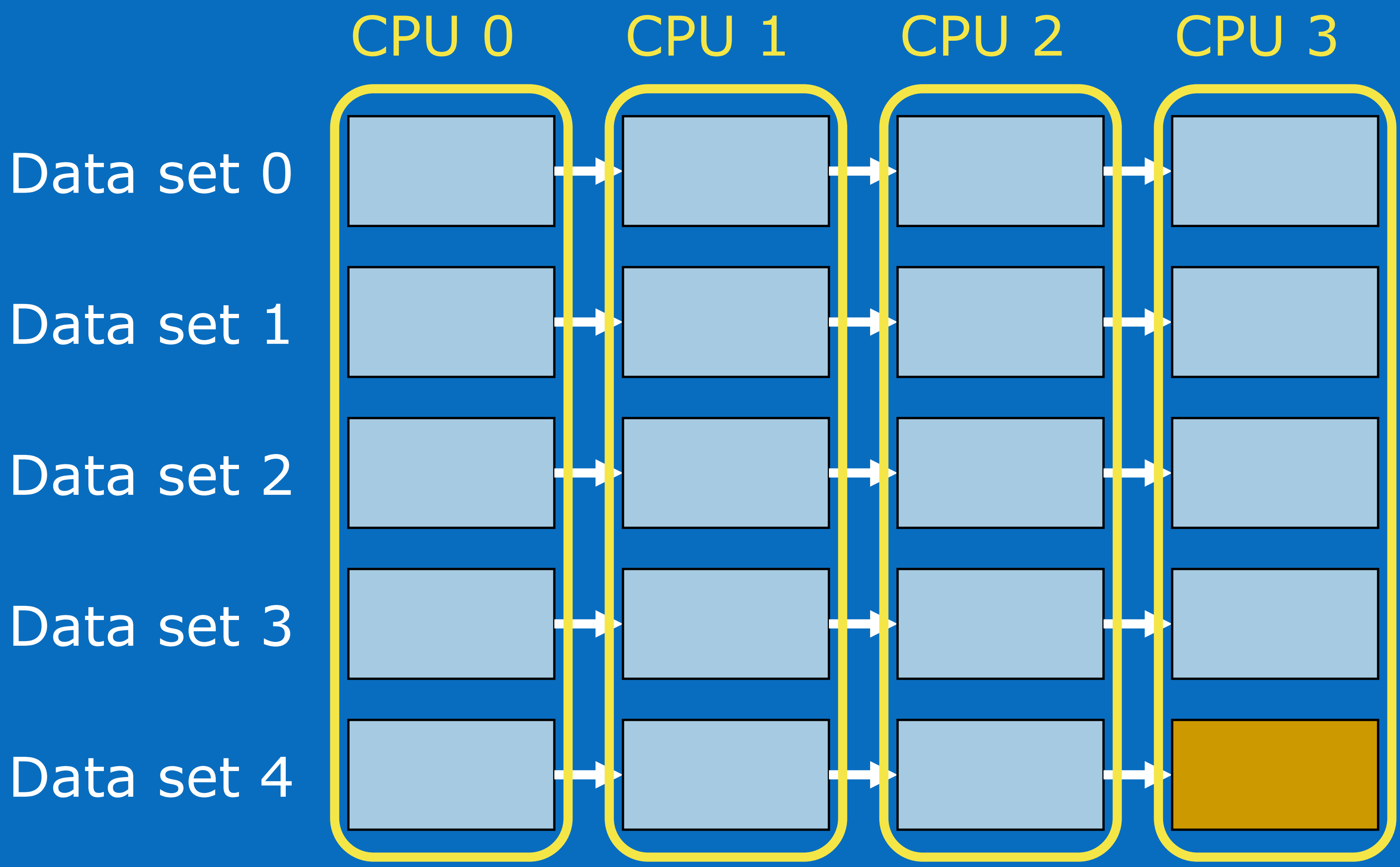
Pipelining Five Data Sets (Step 6)



Pipelining Five Data Sets (Step 7)



Pipelining Five Data Sets (Step 8)



Dependence Graph

Graph = (nodes, arrows)

Node for each

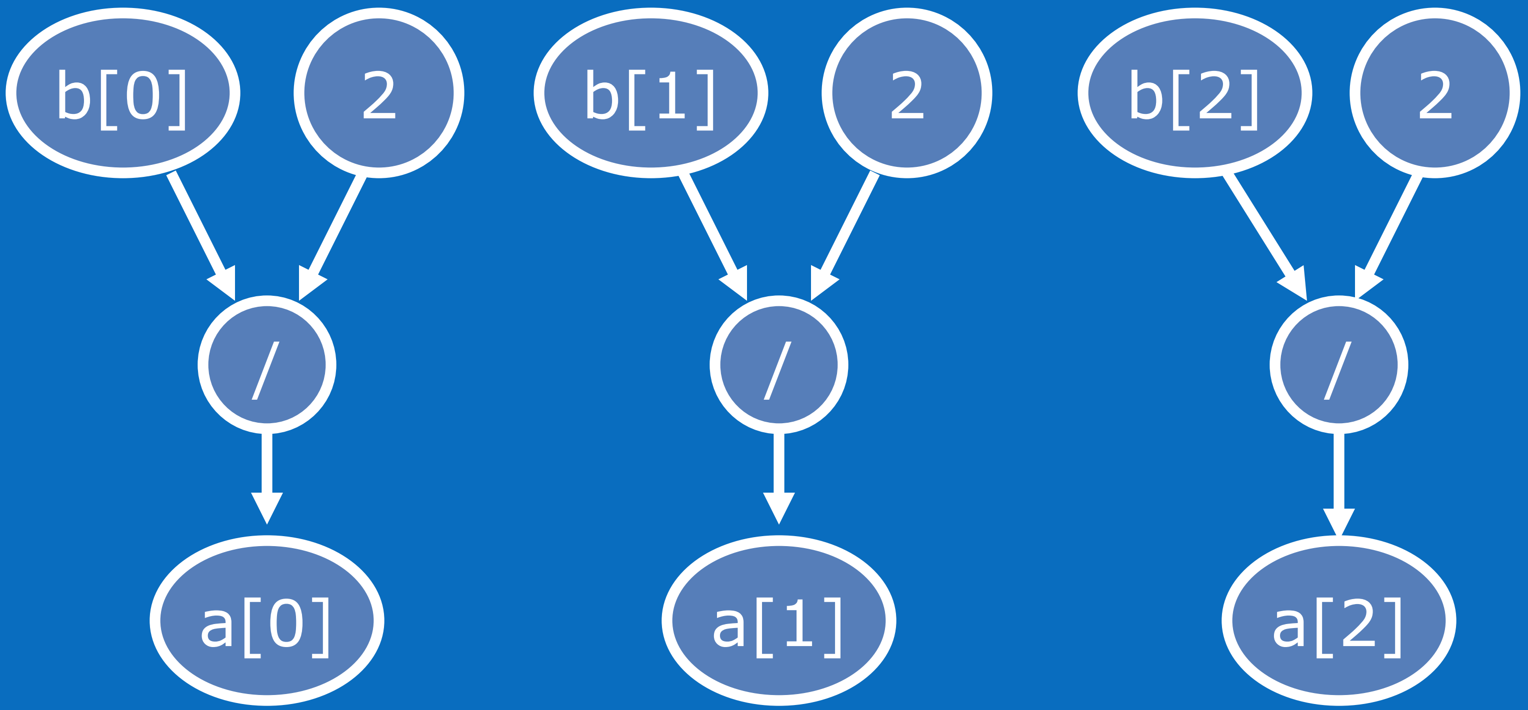
- Variable assignment (except index variables)
- Constant
- Operator or function call

Arrows indicate use of variables and constants

- Data flow
- Control flow

Dependence Graph Example #1

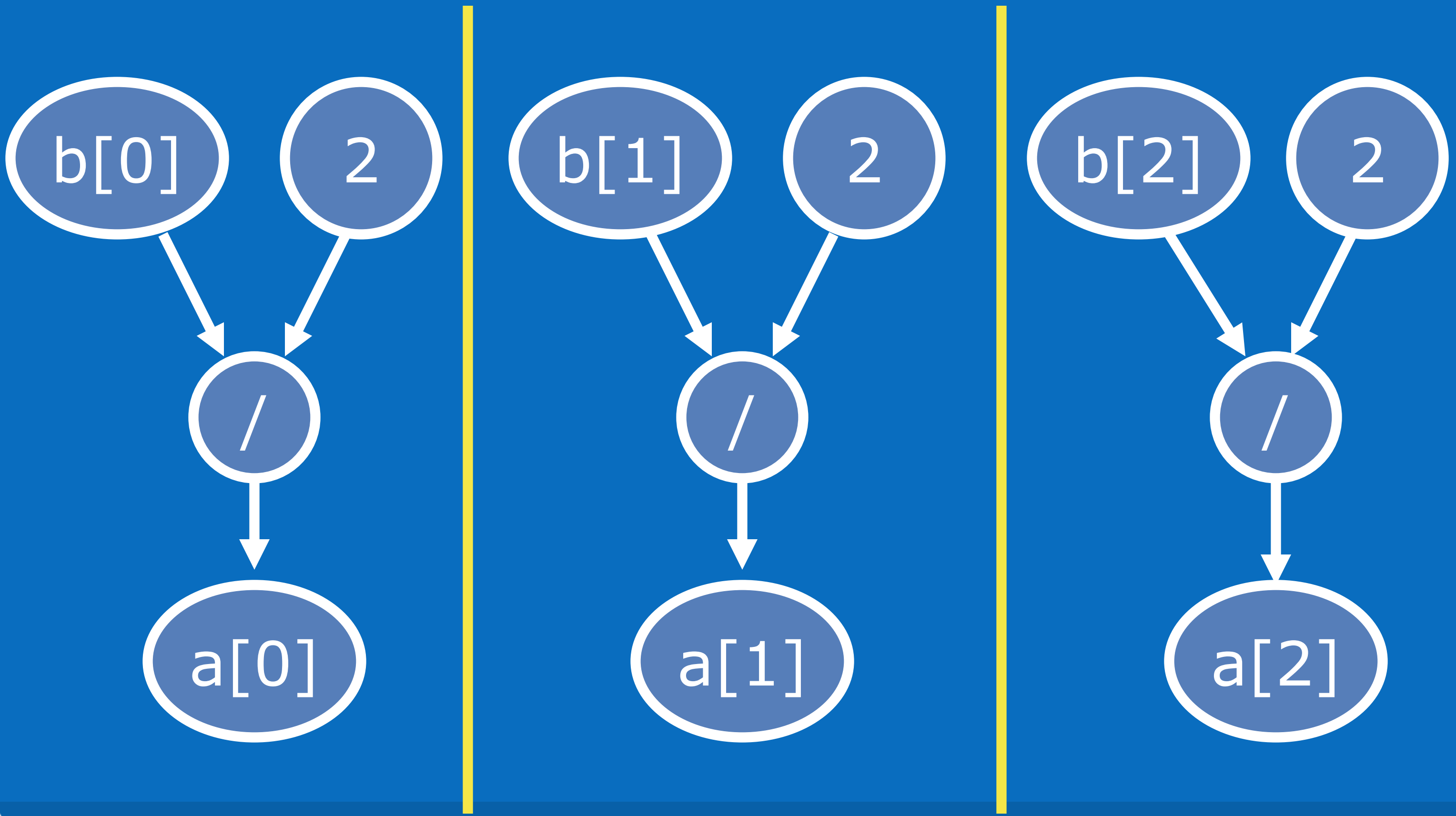
```
for (i = 0; i < 3; i++)
   a[i] = b[i] / 2.0;
```



Dependence Graph Example #1

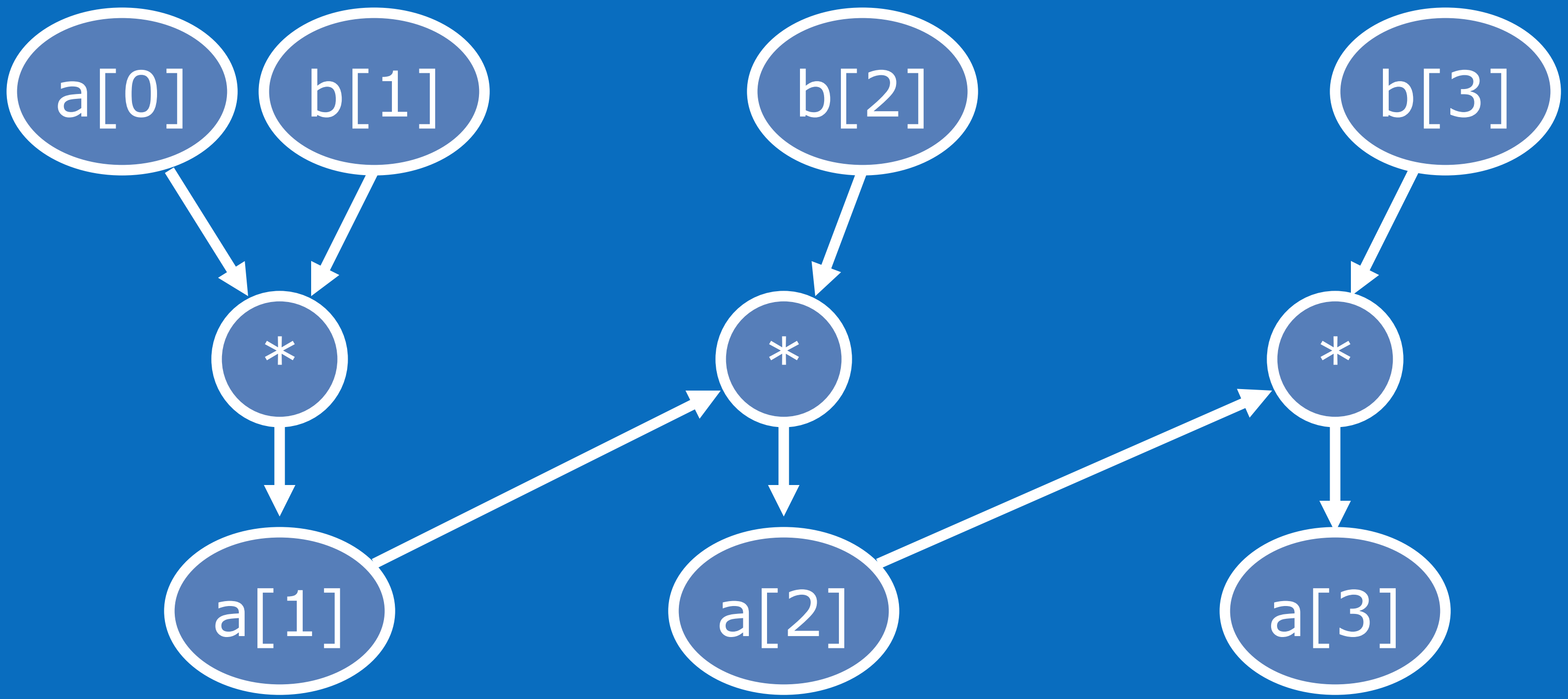
```
for (i = 0; i < 3; i++)
    a[i] = b[i] / 2.0;
```

Domain decomposition possible



Dependence Graph Example #2

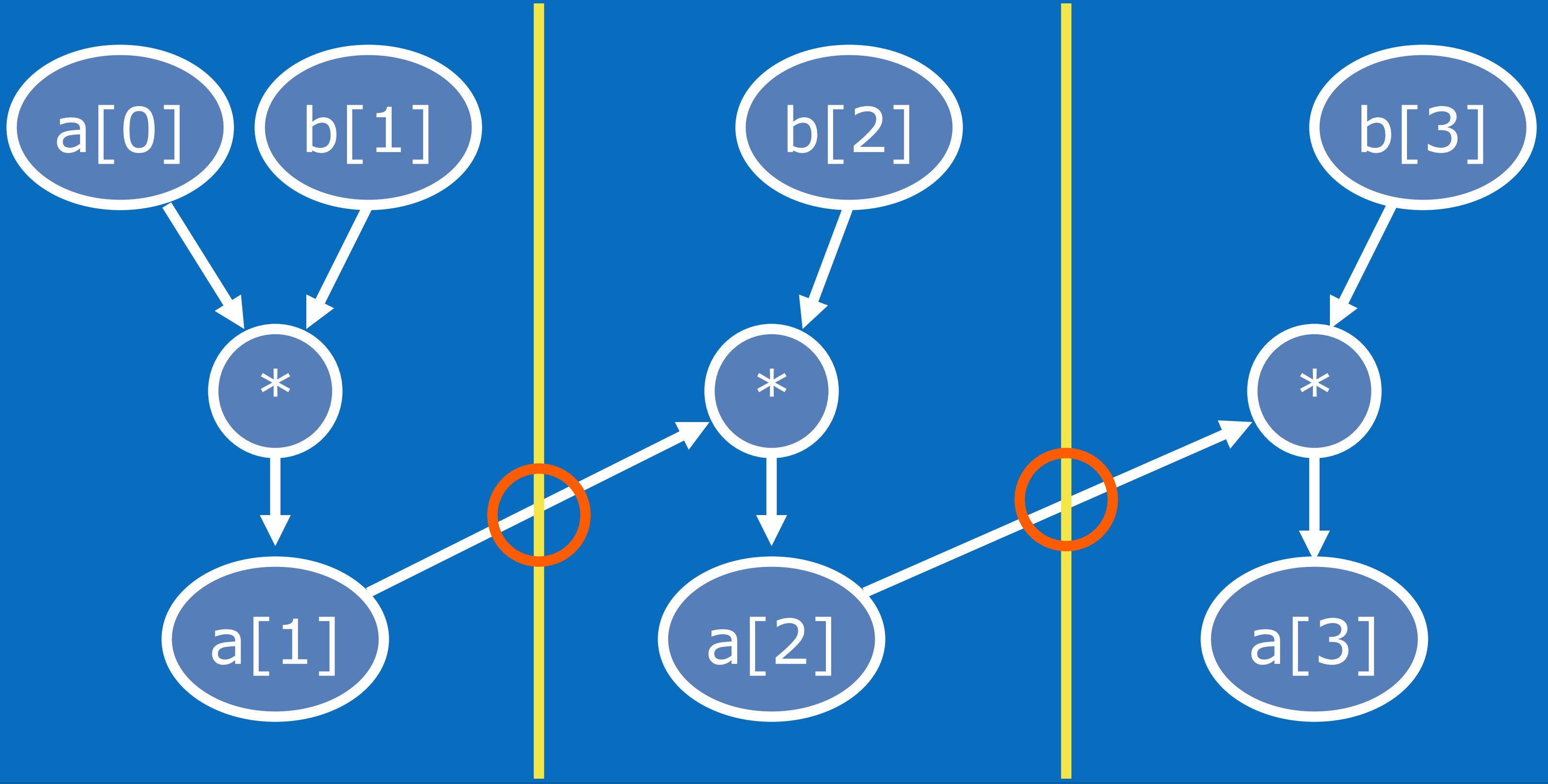
```
for (i = 1; i < 4; i++)
   a[i] = a[i-1] * b[i];
```



Dependence Graph Example #2

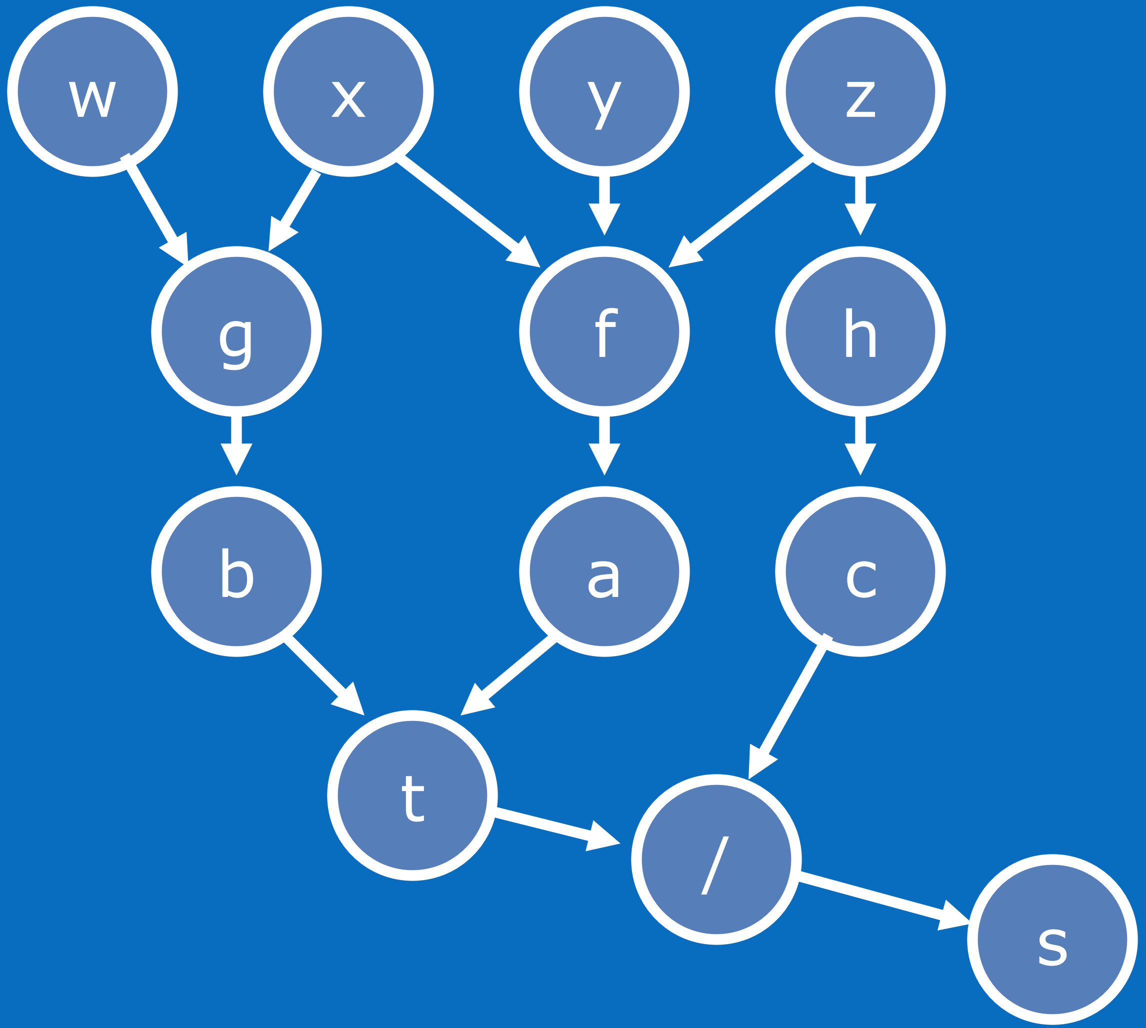
```
for (i = 1; i < 4; i++)
   a[i] = a[i-1] * b[i];
```

No domain decomposition



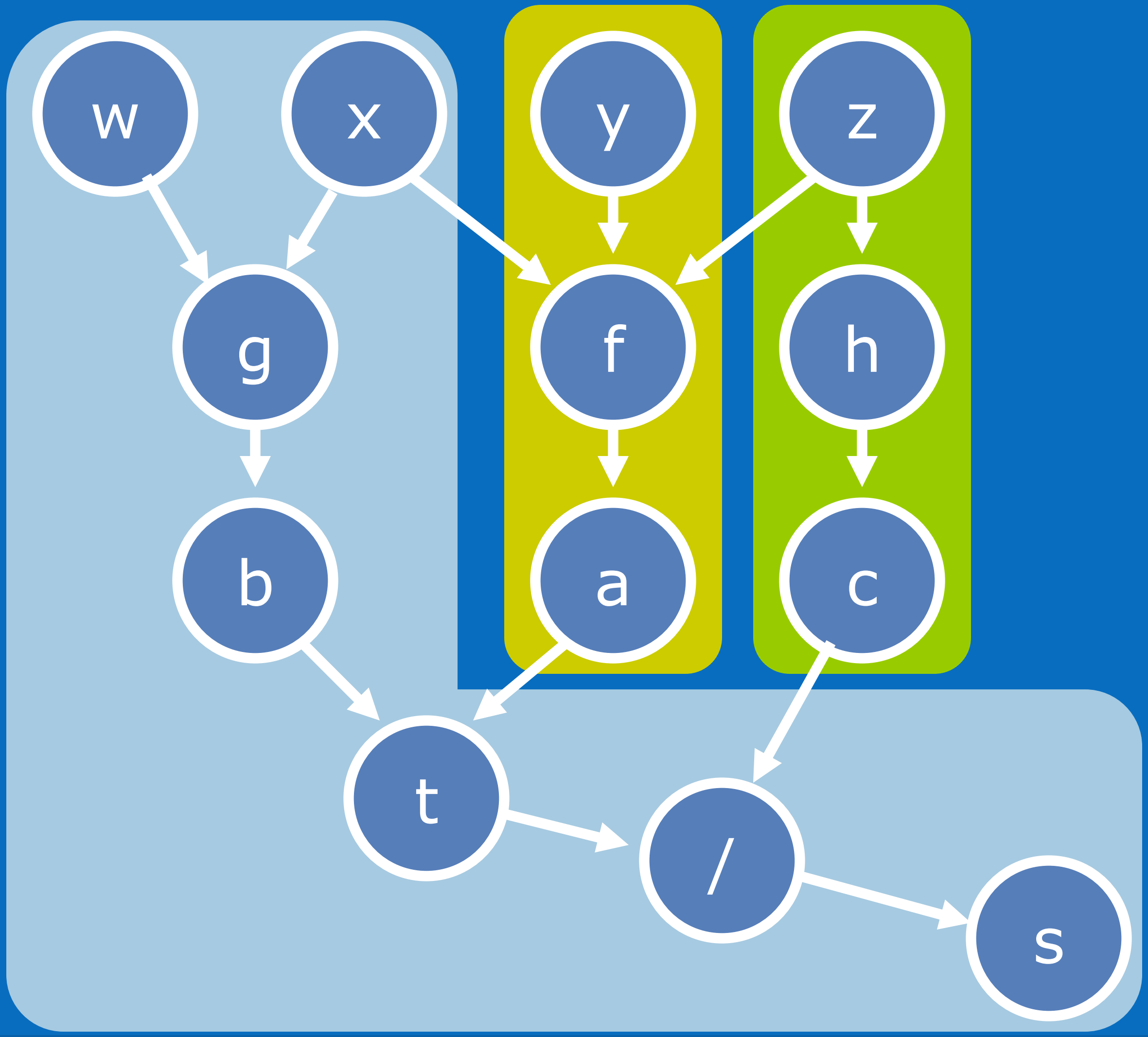
Dependence Graph Example #3

a = f(x, y, z);
b = g(w, x);
t = a + b;
c = h(z);
s = t / c;



Dependence Graph Example #3

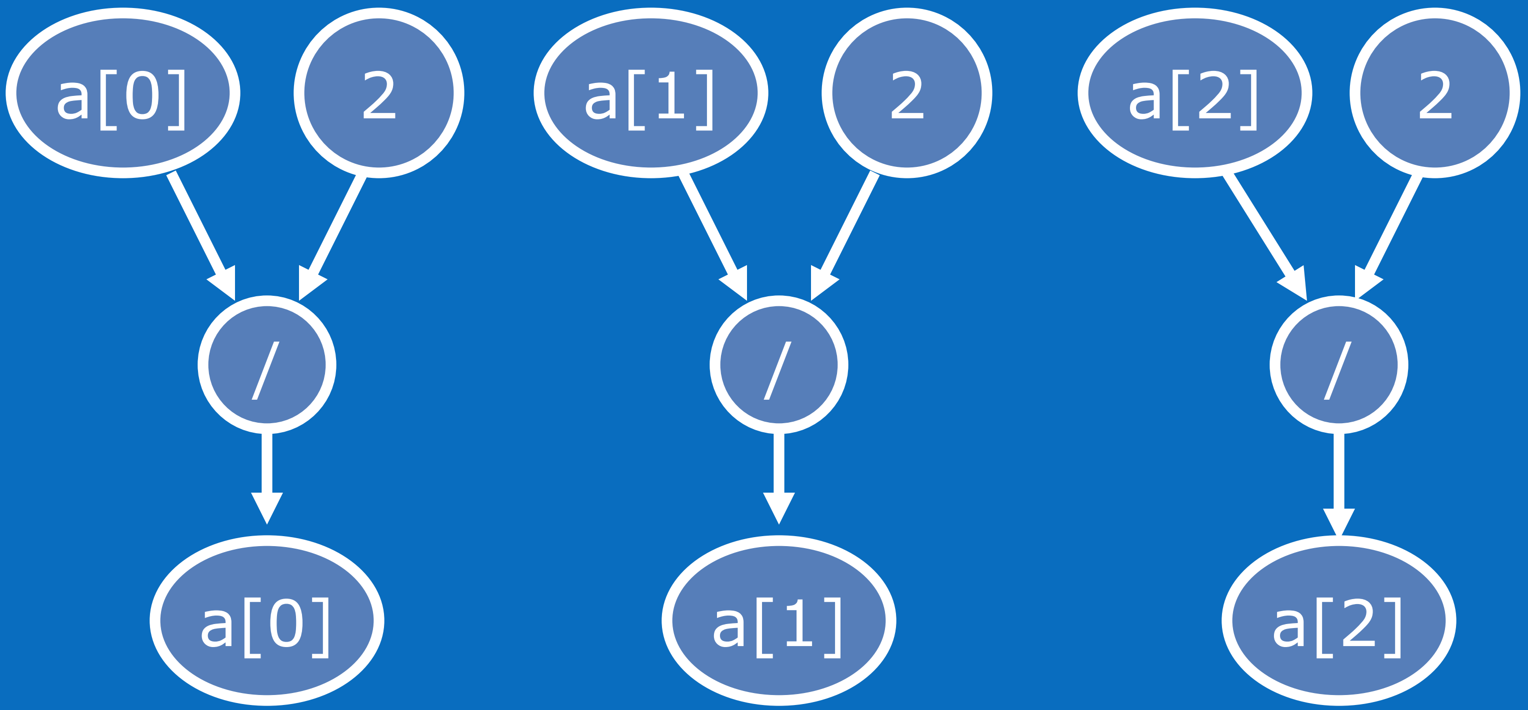
a = f(x, y, z);
b = g(w, x);
t = a + b;
c = h(z);
s = t / c;



Task decomposition with 3 CPUs.

Dependence Graph Example #4

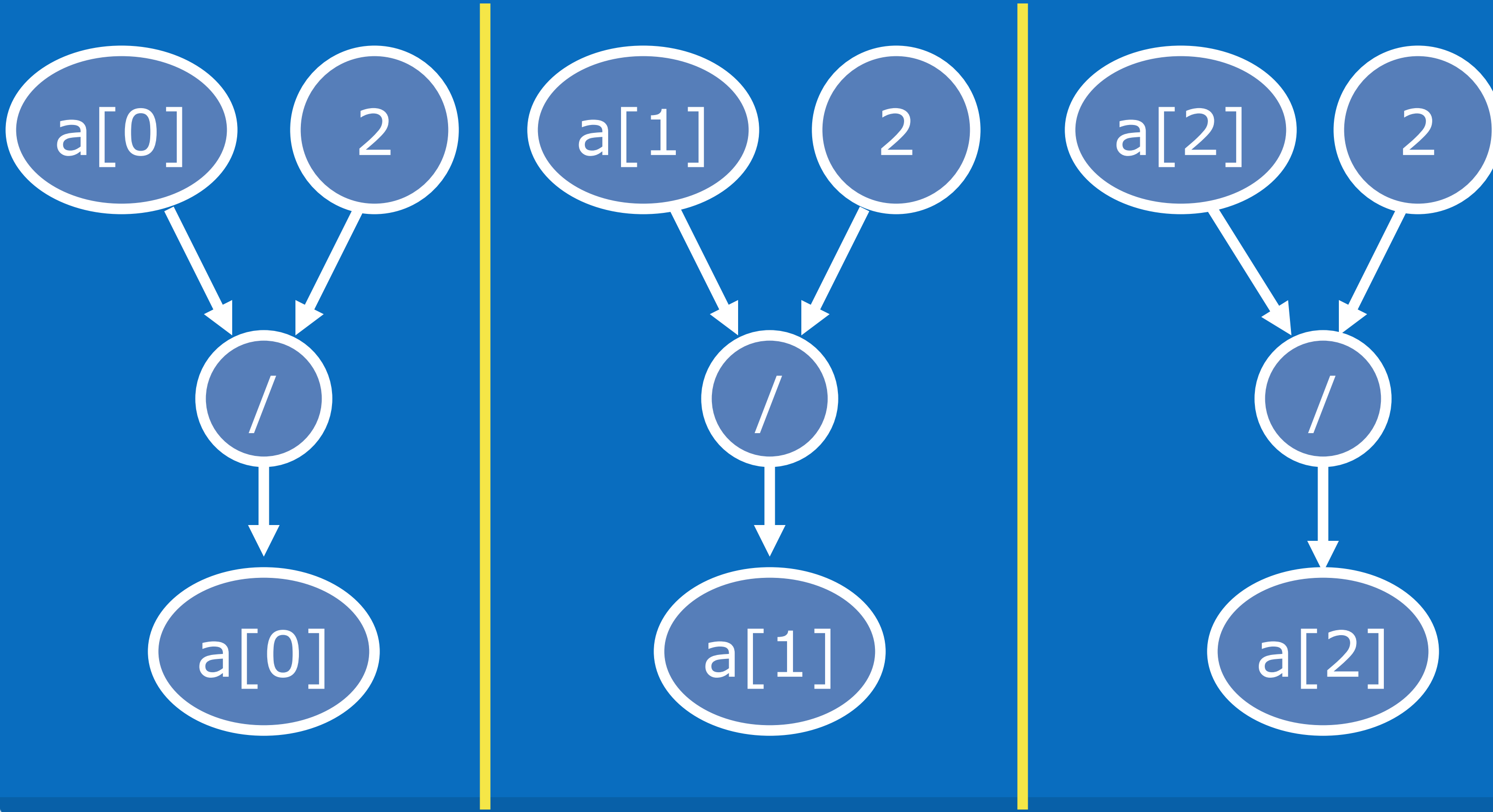
```
for (i = 0; i < 3; i++)
   a[i] = a[i] / 2.0;
```



Dependence Graph Example #4

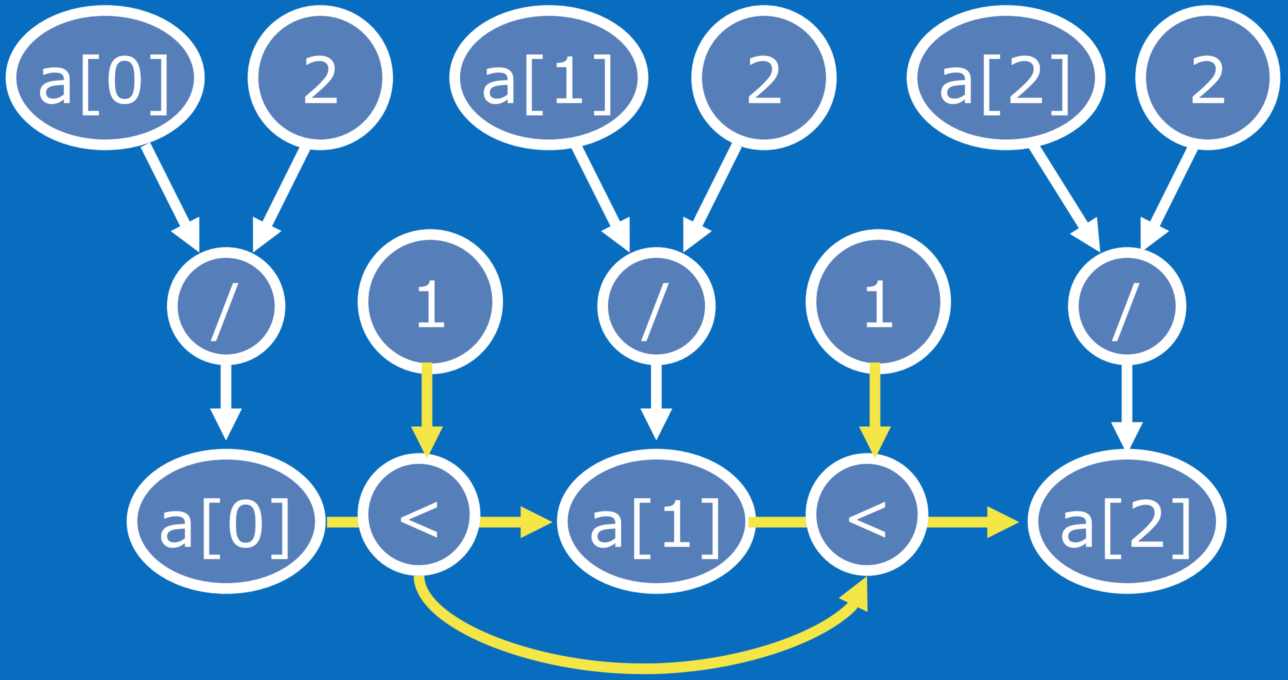
```
for (i = 0; i < 3; i++)
   a[i] = a[i] / 2.0;
```

Domain decomposition



Dependence Graph Example #5

```
for (i = 0; i < 3; i++) {
   a[i] = a[i] / 2.0;
   if (a[i] < 1.0) break;
}
```



Can You Find the Parallelism?

Resizing a photo

Searching a document for all instances of a word

Updating a spreadsheet

Compiling a program

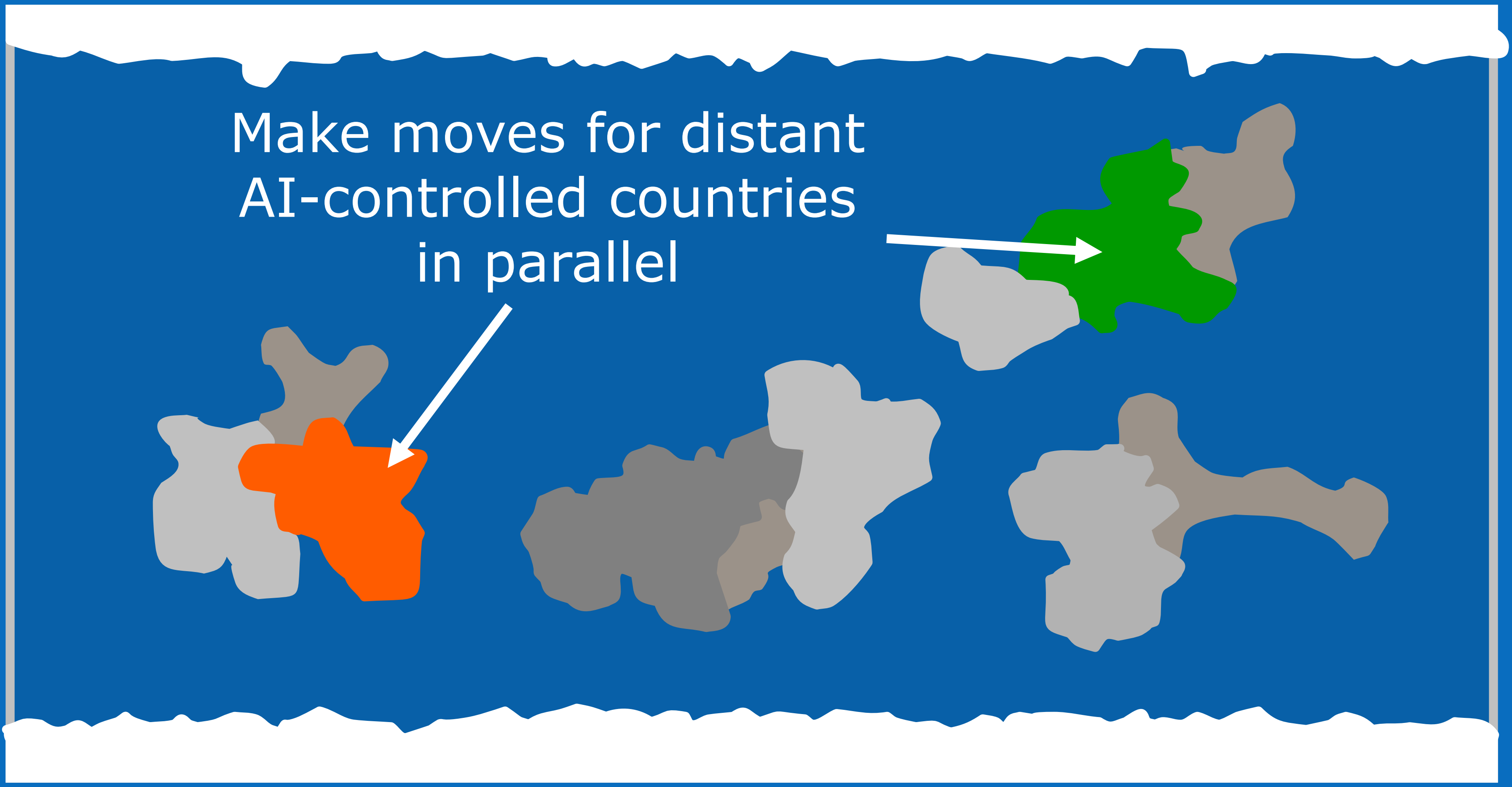
Prefetching pages in a Web browser

Using a word processor to type a report

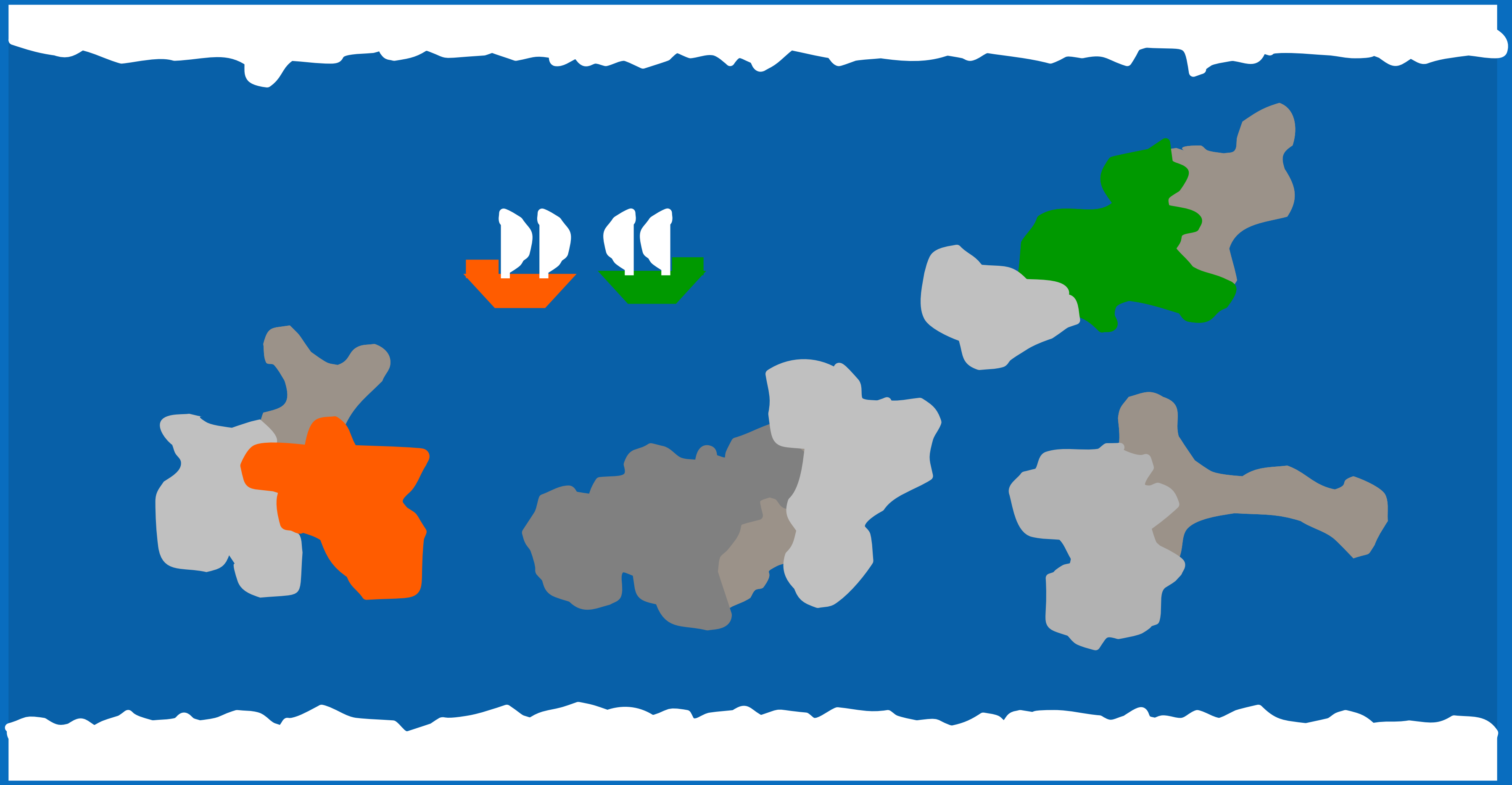
Good/Bad Opportunities for a Parallel Solution

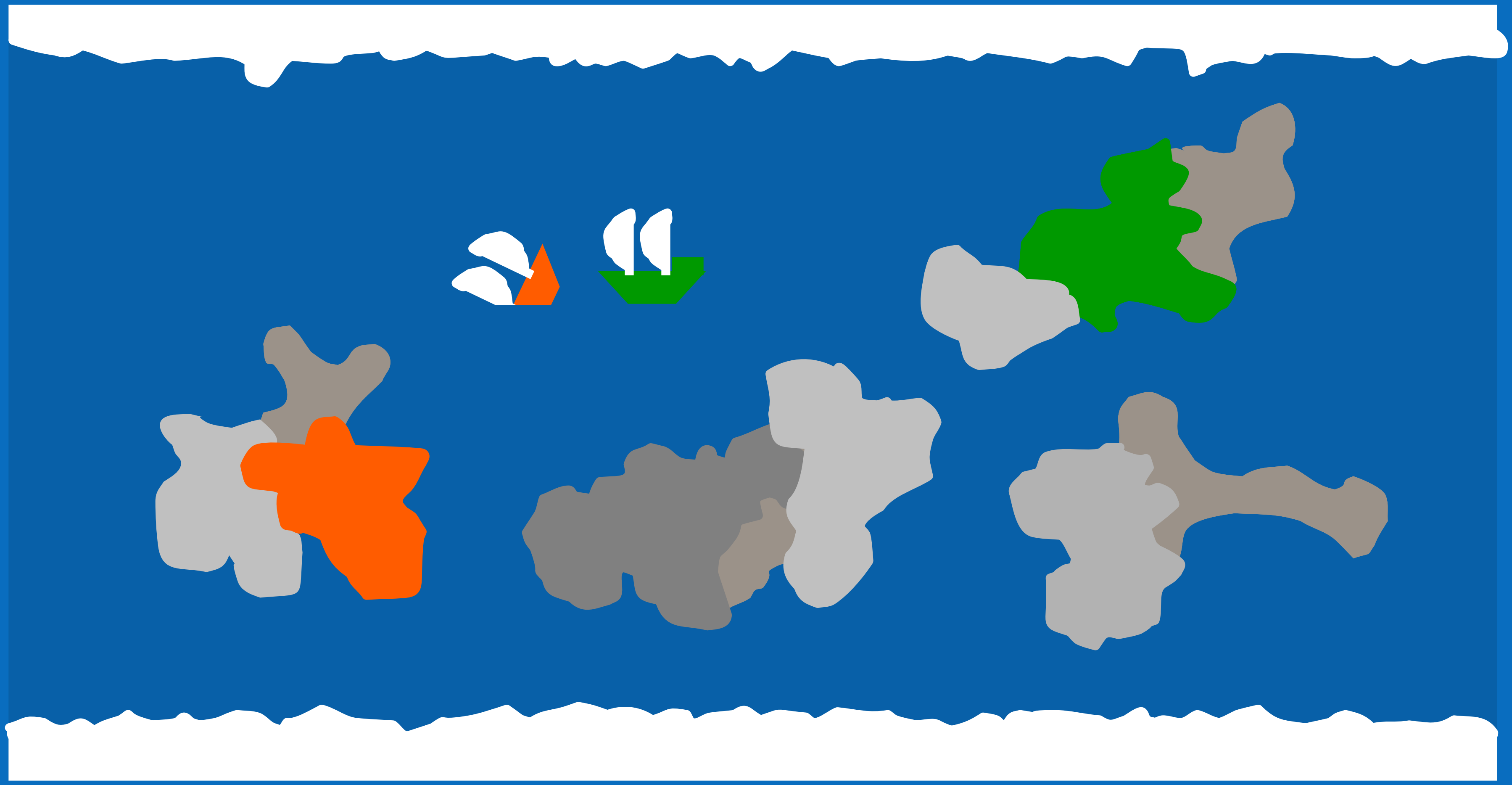
Parallel Solution Easier	Parallel Solution More Difficult or Even Impossible
Larger data sets	Smaller data sets
Dense matrices	Sparse matrices
Dividing space among processors	Dividing time among processors

Speculative Computation in a Turn-Based Strategy Game

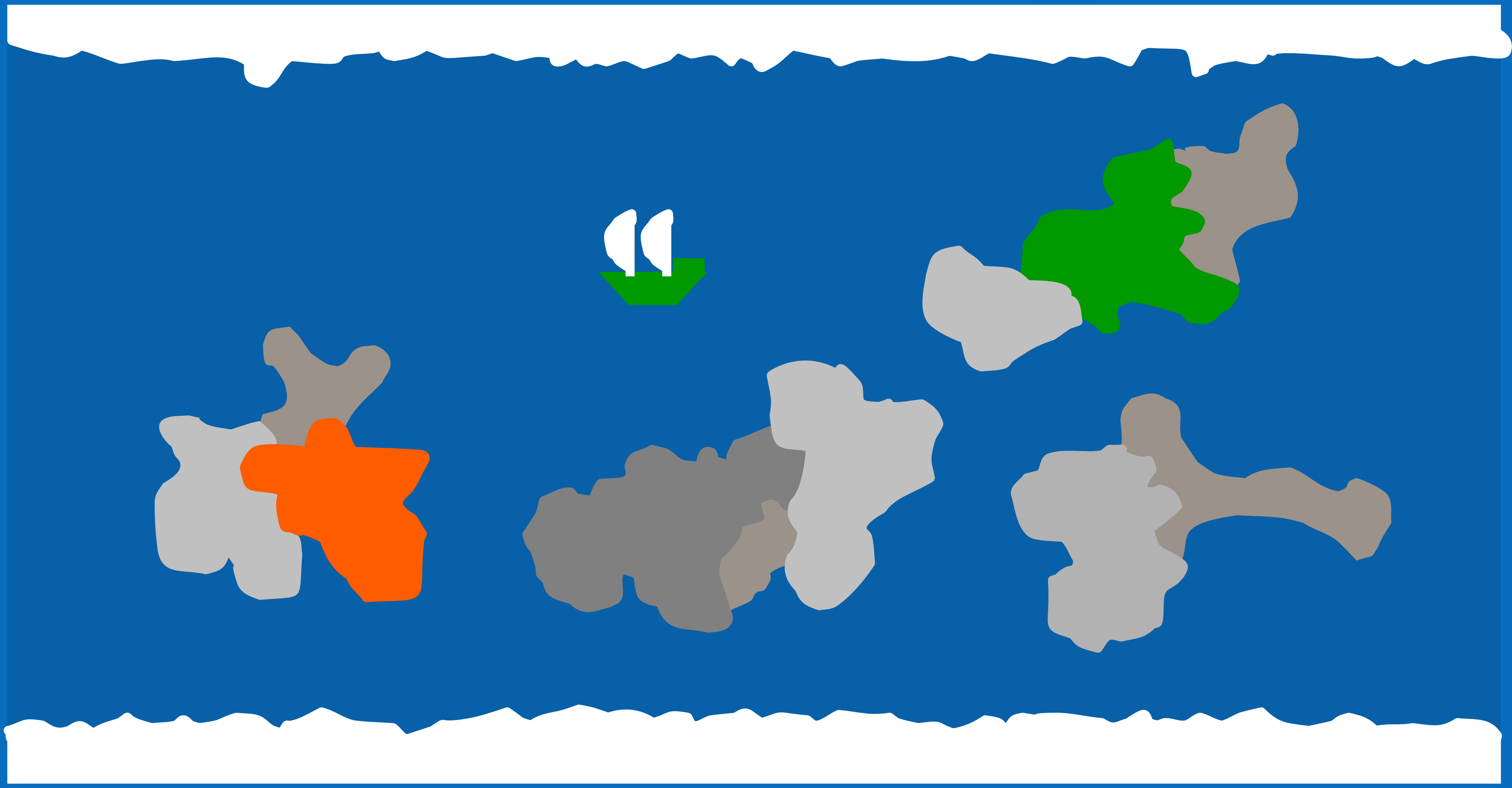


Risk: Unexpected Interaction





Orange Cannot Move a Ship that Has Already Been Sunk by Green



Solution: Reverse Time

Must be able to “undo” an erroneous, speculative computation

Analogous to what is done in hardware after incorrect branch prediction

Speculative computations typically do not have a big payoff in parallel computing

References

Richard H. Carver and Kuo-Chung Tai, *Modern Multithreading: Implementing, Testing, and Debugging Java and C++/Pthreads/ Win32 Programs*, Wiley-Interscience (2006).

Robert L. Mitchell, “Decline of the Desktop,” *Computerworld* (September 26, 2005).

Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill (2004).

Herb Sutter, “The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software,” *Dr. Dobb’s Journal* 30, 3 (March 2005).

intel®

Leap ahead™