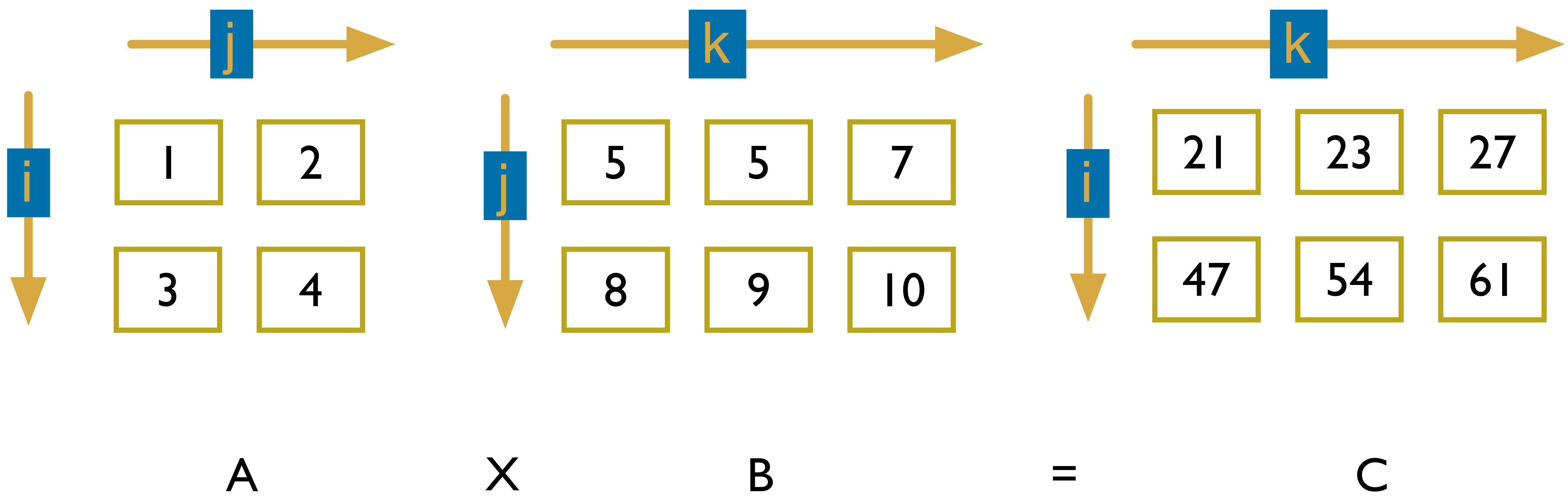
Removing Unneeded Dependencies in Loops

- Example. We know, intuitively, we can parallelise this a great deal:
    - each element in C is independent



Sample Serial Solution

```
for i = 1  to 2 {
  for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[i,j] * b[j,k];
    c[i,k] = sum;
  }
}
```

How can we transform it?

- We can work out how to transform it by locating the dependencies in it.
- Some dependencies are intrinsic to the solution, but
- Some are *artefacts* of the *way* we are *solving* the problem;
    - If we can identify them, perhaps we can modify or remove them.

Try three execution agents:

```
with k = 1;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 2;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 3;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

Issues:

- The variable ***sum***, as written, is common to all three programs.
- Solution:
  - Make ***sum*** private to each program to avoid this dependency.

Try Six Execution Agents

```
with k = 1, i=1;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   sum = 0.0;
   for j = 1 to 2
     sum = sum + a[1,j] * b[j,k];
   c[i,k] = sum;
 ~~}~~
}
```

```
with k = 2, i=1;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   sum = 0.0;
   for j = 1 to 2
     sum = sum + a[1,j] * b[j,k];
   c[i,k] = sum;
 ~~}~~
}
```

```
with k = 3, i=1;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   sum = 0.0;
   for j = 1 to 2
     sum = sum + a[1,j] * b[j,k];
   c[i,k] = sum;
 ~~}~~
}
```

```
with k = 1, i=2;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   sum = 0.0;
   for j = 1 to 2
     sum = sum + a[1,j] * b[j,k];
   c[i,k] = sum;
 ~~}~~
}
```

```
with k = 2, i=2;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   sum = 0.0;
   for j = 1 to 2
     sum = sum + a[1,j] * b[j,k];
   c[i,k] = sum;
 ~~}~~
}
```

```
with k = 3, i=2;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   sum = 0.0;
   for j = 1 to 2
     sum = sum + a[1,j] * b[j,k];
   c[i,k] = sum;
 ~~}~~
}
```

Summarising:

- We could parallelise the original algorithm with some care:
    - Private Variables to avoid unnecessary dependencies
    - But we may need to combine private results at the end to get a global answer.
- Actually, we could break this 'Embarrassingly Parallel' problem into tiny separate pieces; maybe too small. *(How will we know?)*

How fast can we go?

- We have a sequential program that runs too slow
- We have extra hardware resources that could be used to speed it up.
- How fast can we go?

Do the math ….

$T(n)$	Time to run program with n parallel processors
T	Shorthand for $T(1)$
$S(n)$	Speedup with n processors
$E(n)$	Efficiency of Speedup
p	Proportion of T spent executing parallelisable part.
s	Speedup possible for parallelisable part

Speedup and Efficiency

Maximum speedup:

$$T(n) \geq T(1)/n$$

Speedup:

$$S(n) = T(1)/T(n)$$

Efficiency:

$$E(n) = S(n)/n$$

Program Time and Effective Speedup

- Time to run program without parallelism:

$$T = (1-p)T + pT$$

- Parallel (effective) speedup vs. processor count:

$$s \leq n$$

Speedup related to *n* and *s*.

Time to run program with speedup s of parallelisable part:

$$T(n) = (1-p)T + pT/s = (1-p+p/s)T$$

Speedup when running program with parallelisable speedup s:

$$\begin{aligned} S(n) &= T/(1-p+p/s)T \\ &= 1/(1-p+p/s) \end{aligned}$$

Amdahl's Law

$$S(n) = \frac{1}{1 - p + (p/s)} \qquad s \leq n$$

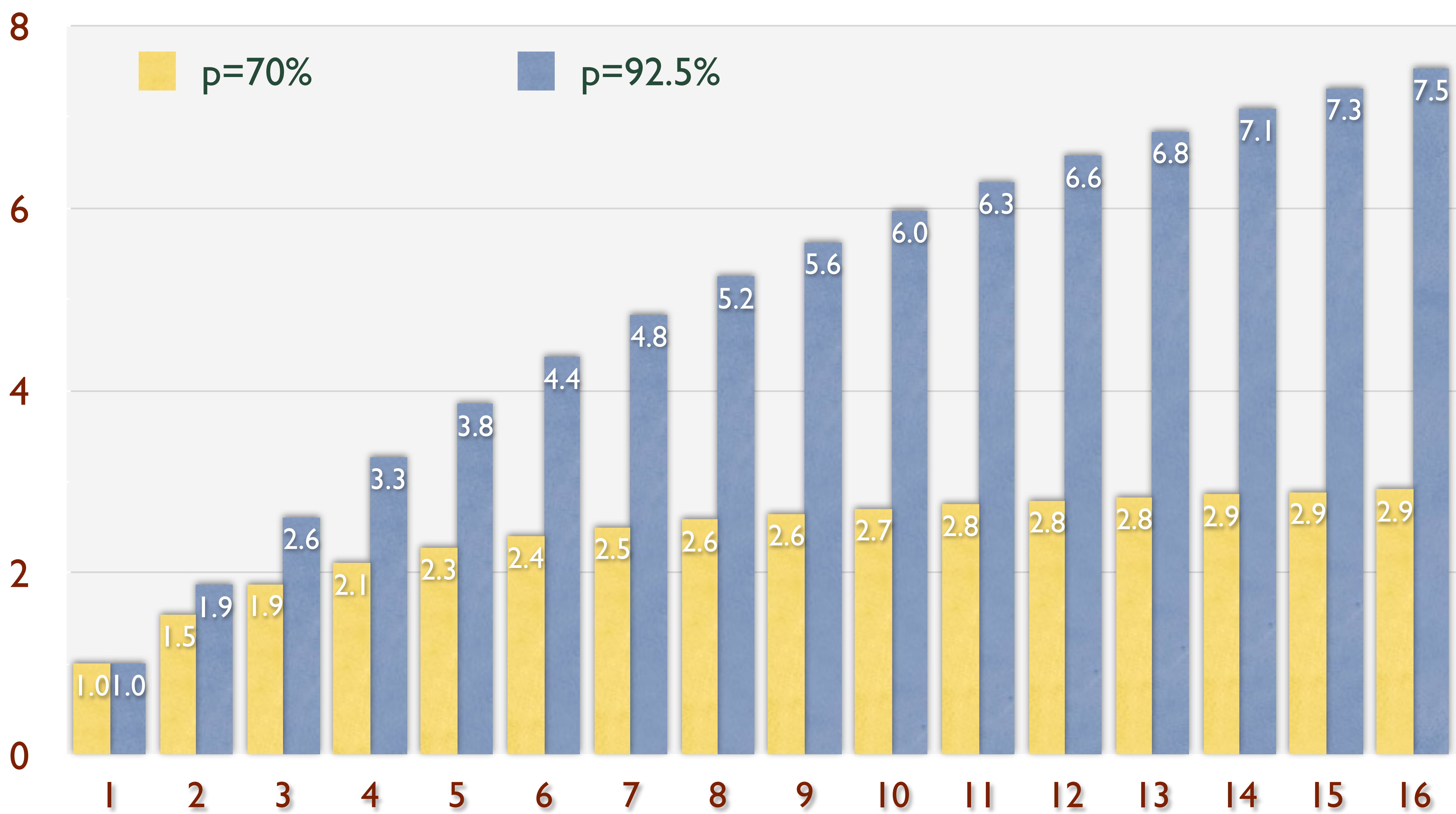
p - proportion of single processor time we can parallelise

n - number of parallel processors

s - parallel speedup ratio

$S(n)$ - overall program speedup

Graphically, Bad News



Amdahl's Law as n (and s) get large

$$S(n) = \frac{1}{1 - p + (p/s)} \quad s \leq n$$

Limit as n, s get very large, so $p/s \to 0$:

$$\frac{1}{1 - p}$$

p=0.925, $S(n) \longrightarrow$ 13.333... ; *p=0.75, $S(n) \longrightarrow$ 4*

Implications

- Even a small fraction of sequential code in a program can seriously interfere with speedup.
  - Note that the code protected by a mutex can only run sequentially!
  - If code has wait a while for a mutex, then that waiting time, has to be added in.
- To maximise performance, inherently sequential code has to be minimised.