

18-447 Lecture 23:

Illusiveness of Parallel Performance

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today
 - peel back simplifying assumptions to understand parallel performance (or the lack of)
- Notices
 - HW5, due Friday 4/28 midnight
 - get going on Lab 4, now less than 2 weeks left
 - **Final Exam**, May 4 Thu, 8:30am-11:30am
- Readings
 - P&H Ch 6
 - LogP: a practical model of parallel computation, Culler, et al. (advanced optional)

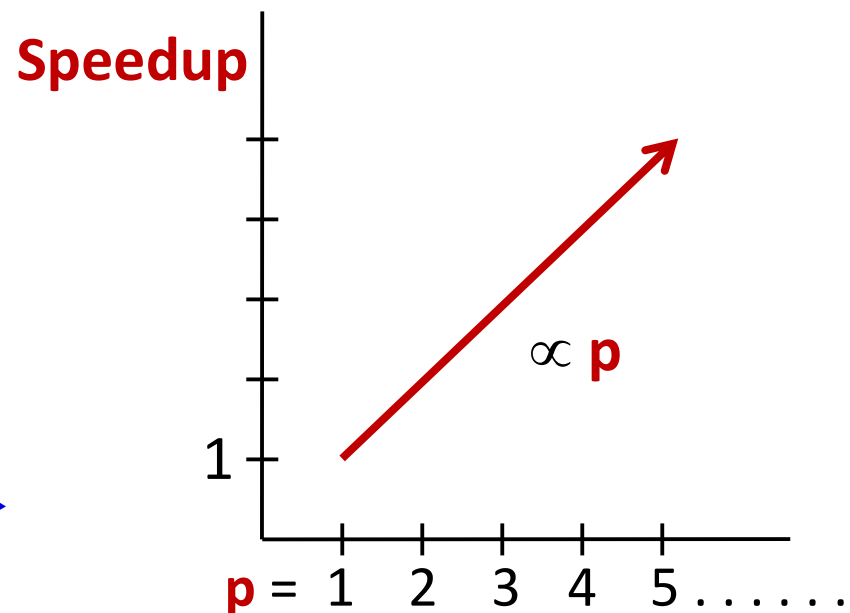
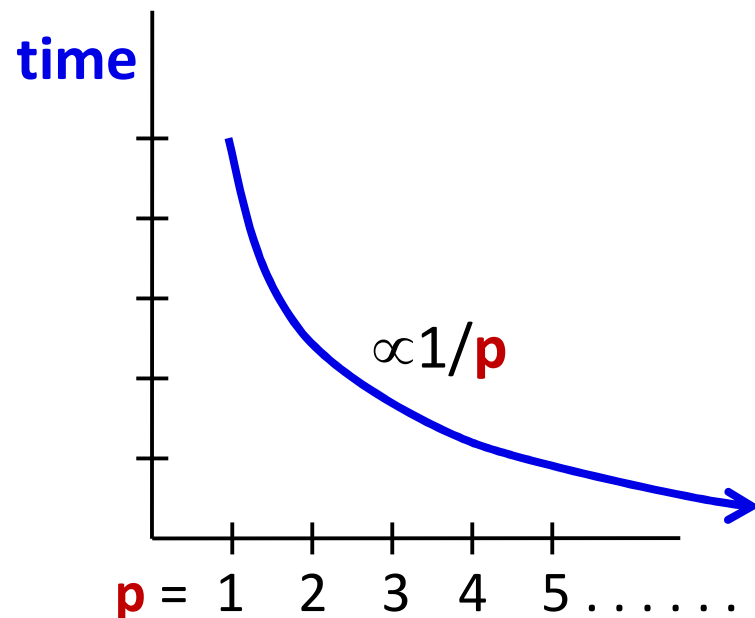
Format of Final Exam

- Comprehensive in coverage, HW, labs, assigned readings (from textbooks and papers)
- Types of questions
 - freebies: remember the materials
 - >> **probing: understand the materials** <<
 - applied: apply the materials in original interpretation
- ****150 minutes, 150 points****
 - point values calibrated to time needed
 - closed-book, three 8½x11-in² hand-written crib sheets
 - no electronics
 - use pencil or black/blue ink only

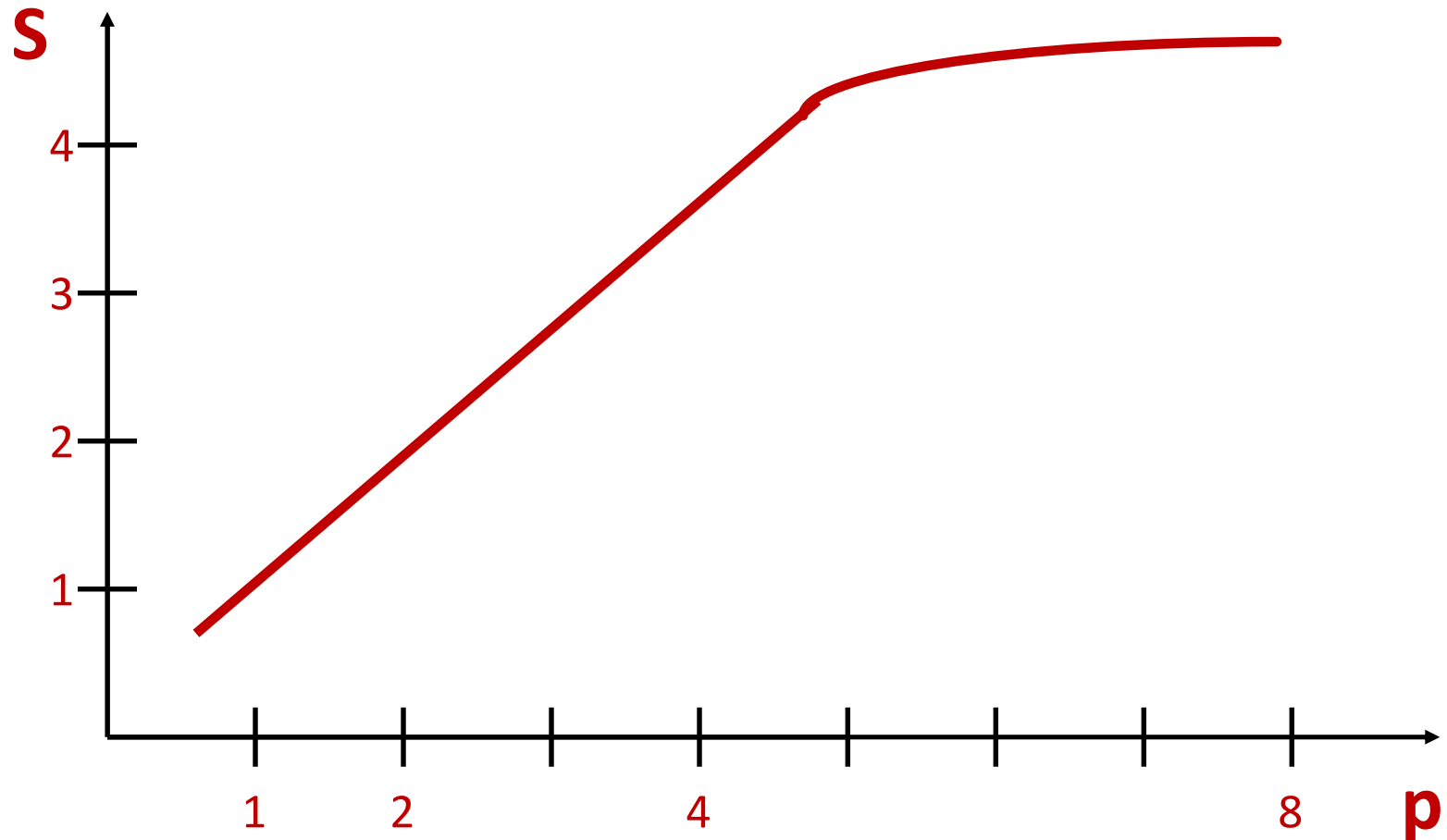
“Ideal” Linear Parallel Speedup

- Ideally, parallel speedup is linear with **p**

$$\text{Speedup} = \frac{\text{time}_{\text{sequential}}}{\text{time}_{\text{parallel}}}$$



Non-Ideal Speed Up



*Never get to high speedup
regardless of p !!*

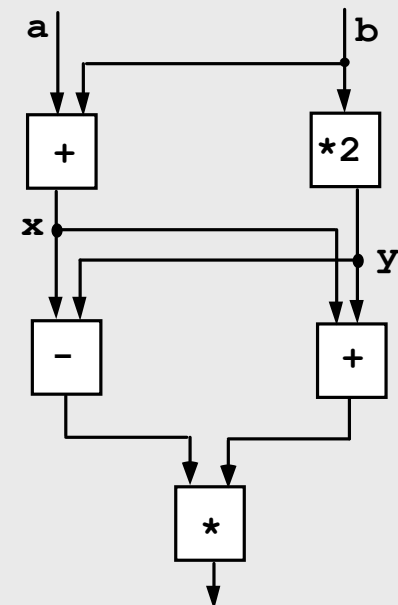
Parallelism Defined

- T_1 (work measured in time):
 - time to do work with 1 PE
- T_∞ (critical path):
 - time to do work with infinite PEs
 - T_∞ bounded by dataflow dependence
- Average parallelism:
 - $P_{avg} = T_1 / T_\infty$
- For a system with p PEs

$$T_p \geq \max\{T_1/p, T_\infty\}$$

- When $P_{avg} \gg p$
 - $T_p \approx T_1/p$, aka “linear speedup”

```
x = a + b;
y = b * 2
z = (x - y) * (x + y)
```

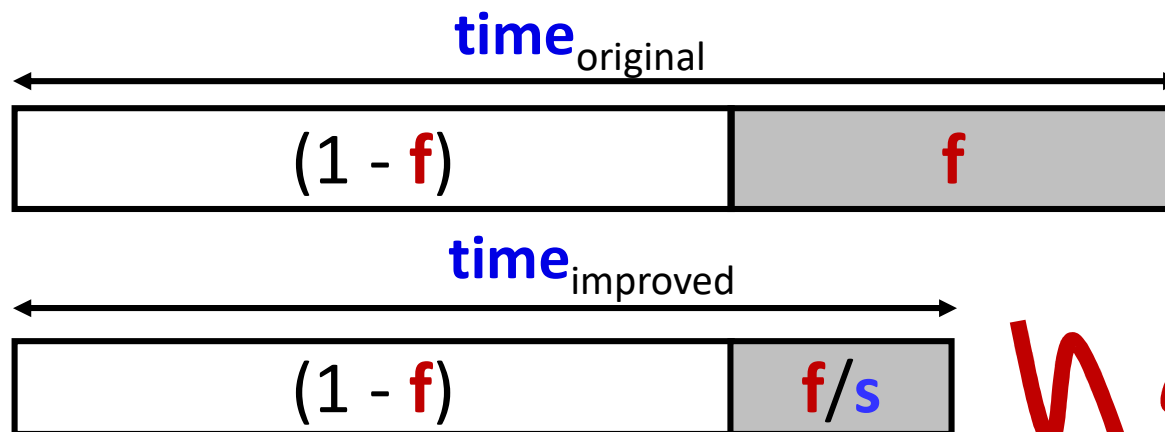


[Shiloach&Vishkin]

Recall

Amdahl's Law: a lesson on speedup

- If only a fraction **f** (of time) is speedup by **s**



$$\text{time}_{\text{improved}} = \text{time}_{\text{original}} \cdot ((1-f) + f/s)$$

$$S_{\text{effective}} = 1 / ((1-f) + f/s)$$

- if **f** is small, **s** doesn't matter
- even when **f** is large, diminishing return on **s**;
eventually “1-**f**” dominates

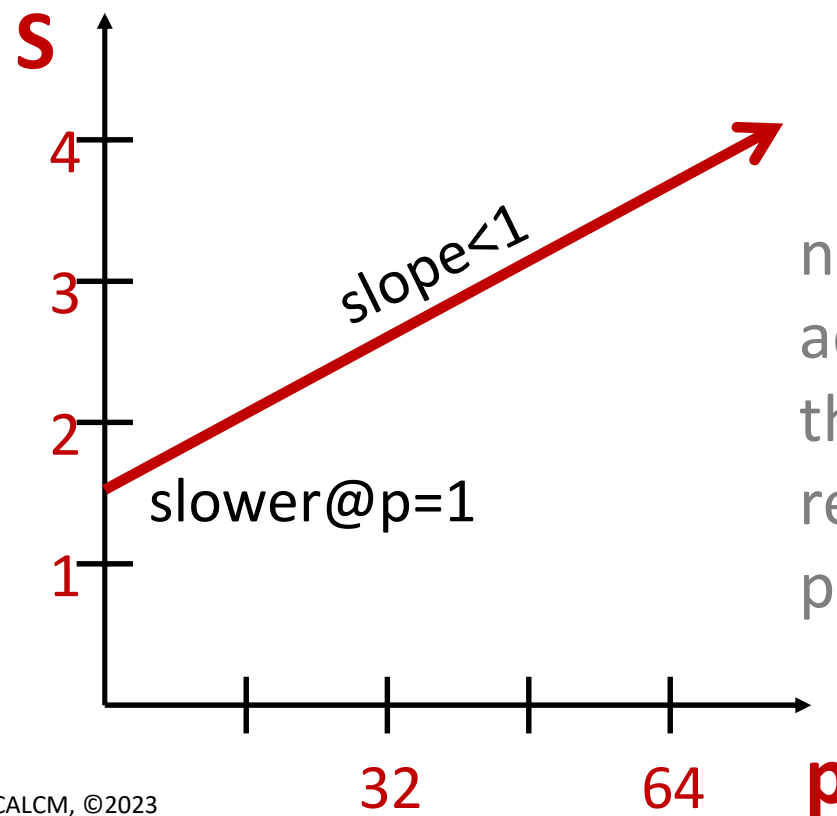
Recall

Non-Ideal Speed Up

Cheapest algo may not be the most scalable, s.t.

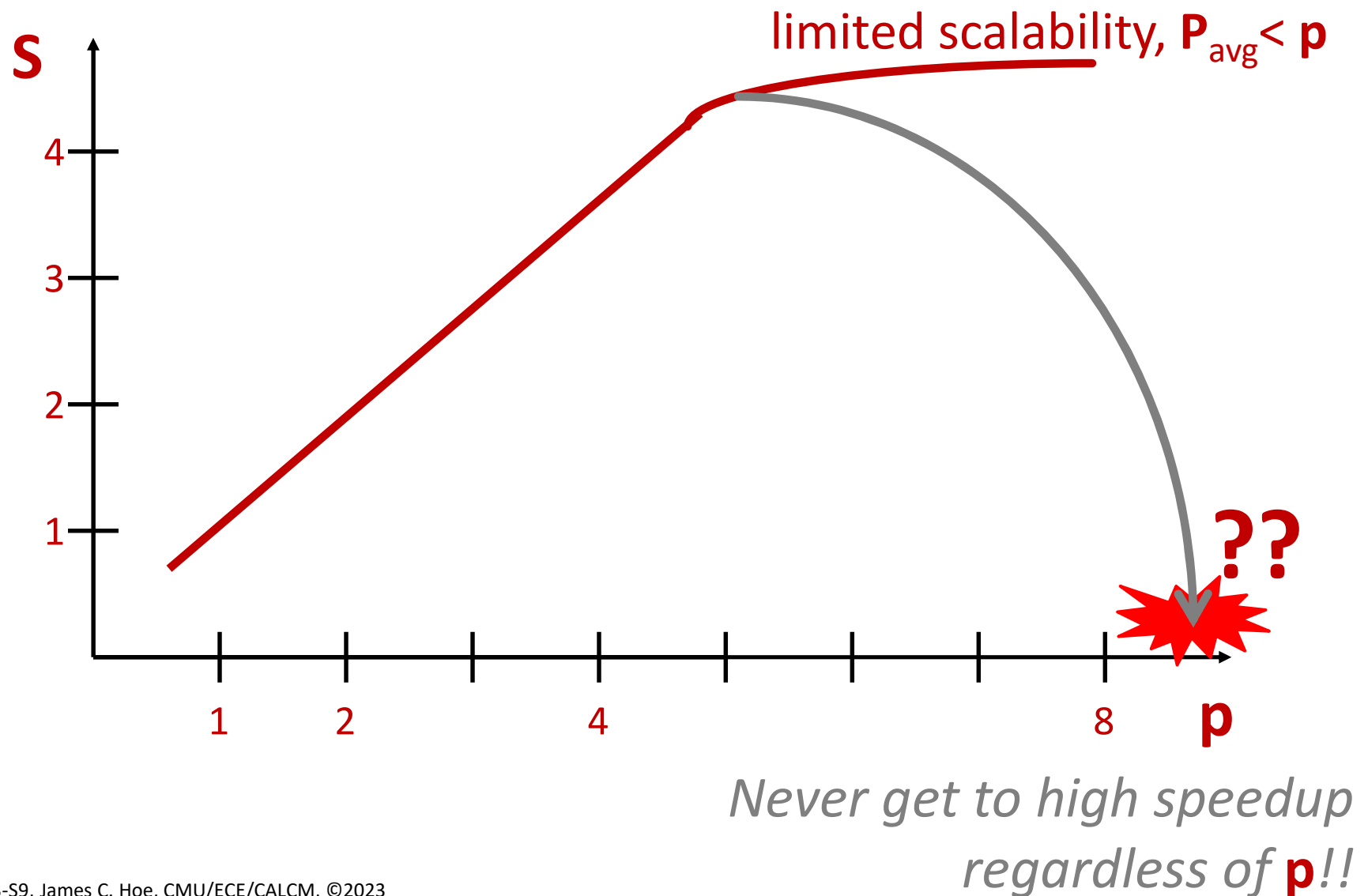
$\text{time}_{\text{parallel-algo}@p=1} = K \cdot \text{time}_{\text{sequential-algo}}$ where $K > 1$
and

$$\text{Speedup} = p/K$$



not efficient but
acceptable if it is
the only way to
reach required
performance

Non-Ideal Speed Up



Communication not Free

- PE may spend extra time
 - in the act of sending or receiving data
 - waiting for data to be transferred from another PE
 - latency: data coming from far away
 - bandwidth: data coming thru finite channel
 - waiting for another PE to get to a particular point of the computation (a.k.a. synchronization)

How does communication cost grow with T_1 ?

How does communication cost grow with p ?

Aside: Strong vs. Weak Scaling

- **Strong Scaling**

- what is S_p as p increases for constant work, T_1
run same workload faster on new larger system
- harder to speedup as (1) p grows toward P_{avg} and
 (2) communication cost increases with p

- **Weak Scaling**

- what is S_p as p increases for larger work, $T_1' = p \cdot T_1$
run a larger workload faster on new larger system

- $S_p = \text{time}_{\text{sequential}}(p \cdot T_1) / \text{time}_{\text{parallel}}(p \cdot T_1)$


- Which is easier depends on

- how P_{avg} scales with work size T_1'
- relative scaling of bottlenecks (*storage, BW, etc*)

Continuing from Last Lecture

- Parallel Thread Code (Last Lecture)

```
void *sumParallel(void *_id) {
    long id=(long) _id;
    psum[id]=0;
    for(long i=0;i<(ARRAY_SIZE/p);i++)
        psum[id]+=A[id*(ARRAY_SIZE/p) + i];
}
```



- Assumed “+” takes 1 unit-time; **everything else** free

$$T_1 = 10,000$$

$$T_\infty = \lceil \log_2 10,000 \rceil = 14$$

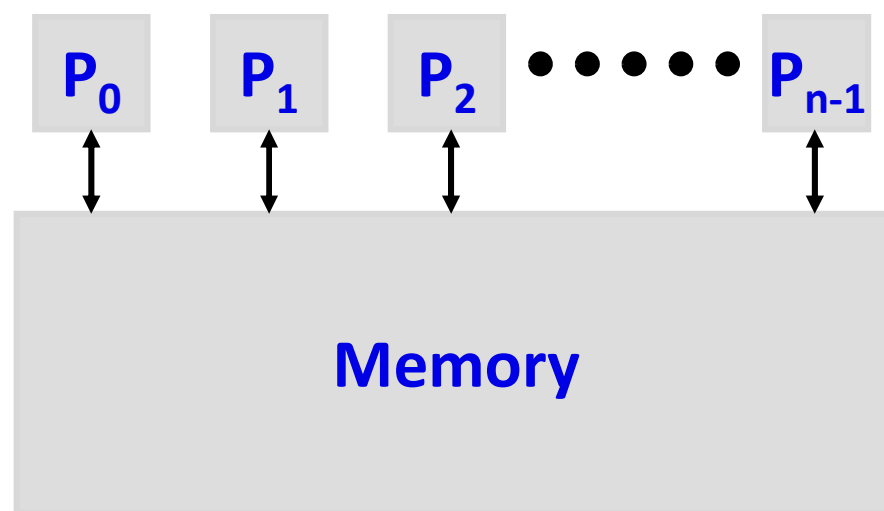
$$P_{\text{average}} = 714$$

What would you predict is the real speedup on a 28-core ECE server?

Need for more detailed analysis

- What cost were left out in “**everything else**”?
 - explicit cost: need to charge for all operations (branches, LW/SW, pointer calculations)
 - implicit cost: ****communication and synchronization****
- PRAM-like models (Parallel Random Access Machine) capture cost/rate of parallel processing but assume
 - **zero latency** and **infinite bandwidth** to share data between processors
 - **zero overhead** cycles to send and receive

Useful when analyzing complexity but not for performance finetuning



Arithmetic Intensity: Modeling Communication as “Lump” Cost

Arithmetic Intensity

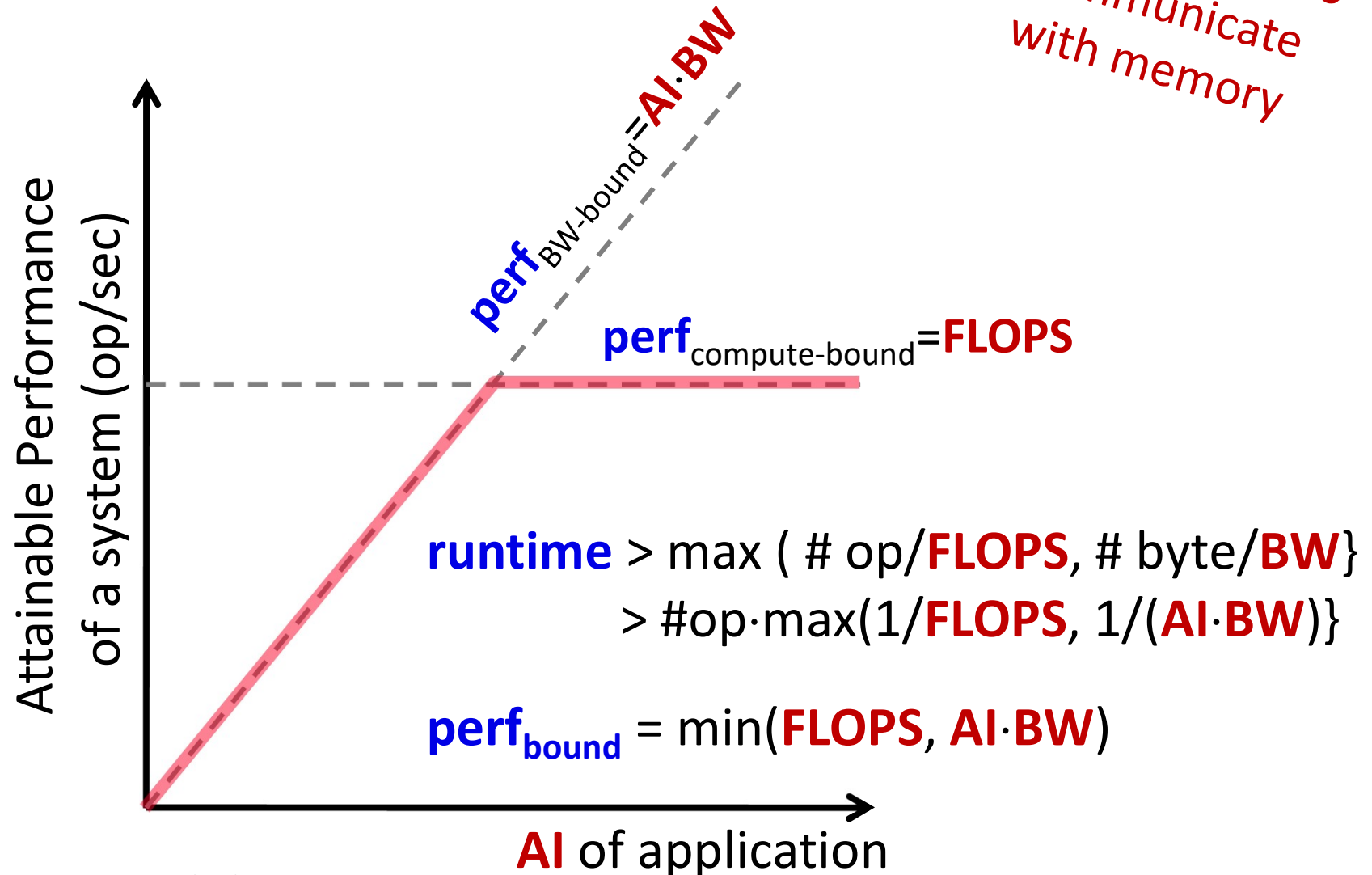
- An algorithm has a cost in terms of operation count
 - **runtime**_{compute-bound} = # operations / **FLOPS**
- An algorithm also has a cost in terms of number of bytes communicated (ld/st or send/receive)
 - **runtime**_{BW-bound} = # bytes / **BW**
- Which one dominates depends on
 - ratio of **FLOPS** and **BW** of platform
 - ratio of ops and bytes of algorithm
- Average **Arithmetic Intensity (AI)**
 - how many ops performed per byte accessed
 - # operations / # bytes

FLOPS=floating-point operations per second

Roofline Performance Model

[Williams&Patterson, 2006]

*cost for CPU to
communicate
with memory*



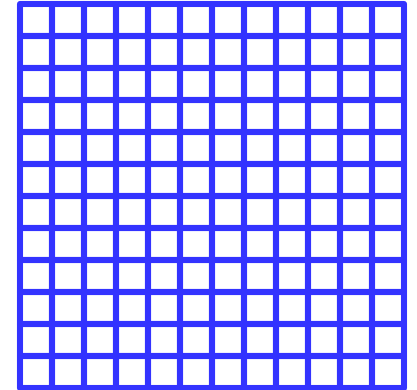
Parallel Sum Revisited with **AI**

- Last lecture we said
 - 100 threads perform 100 +’s each in parallel, and
 - between 1~7 (plus a few) +’s each in the parallel reduction
 - $T_{100} = 100 + 7$
 - $S_{100} = 93.5$
- Now we see (*assume 1 op per cycle per thread*)
 - **AI** is a constant, 1 op / 8 bytes (for doubles)
 - Let BW_{cyc} be total bandwidth (byte/cycle) shared by threads on a multicore
$$Perf_p < \min\{ p \text{ ops/cycle}, AI * BW_{cyc} \}$$
 - useless to parallelize beyond $p > BW_{cyc}/8$

What about a multi-socket system?

Interesting **AI** Example: MMM

```
for (i=0; i<N; i++)  
  for (j=0; j<N; j++)  
    for (k=0; k<N; k++)  
      C[i][j] += A[i][k] * B[k][j];
```



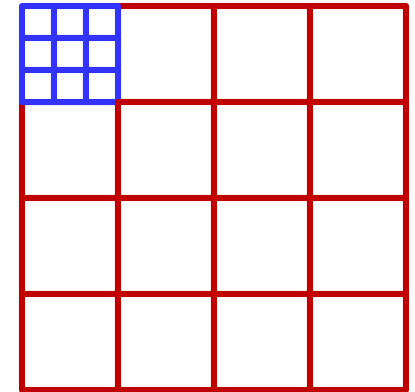
- N^2 data-parallel dot-product's
- Assume N is large s.t. 1 row/col too large for on-chip
- Operation count: N^3 float-mult and N^3 float-add
- External memory access (assume 4-byte floats)
 - $2N^3$ 4-byte reads (of A and B) from DRAM
 - ... N^2 4-byte writes (of C) to DRAM ...
- Arithmetic Intensity $\approx 2N^3 / (4 \cdot 2N^3) = 1/4$

More Interesting **AI** Example: MMM

```

for (i0=0; i0<N; i0+=Nb)
  for (j0=0; j0<N; j0+=Nb)
    for (k0=0; k0<N; k0+=Nb) {
      for (i=i0; i<i0+Nb; i++)
        for (j=j0; j<j0+Nb; j++)
          for (k=k0; k<k0+Nb; k++)
            C[i][j] += A[i][k] * B[k][j];
    }

```



- Imagine a ' N/N_b ' x ' N/N_b ' **MATRIX** of $N_b \times N_b$ matrices
 - inner-triple is straightforward **matrix-matrix** mult
 - outer-triple is **MATRIX-MATRIX** mult
- To improve **AI**, hold $N_b \times N_b$ sub-matrices on-chip for data-reuse

need to copy block (not shown)

AI of blocked MMM Kernel ($N_b \times N_b$)

```

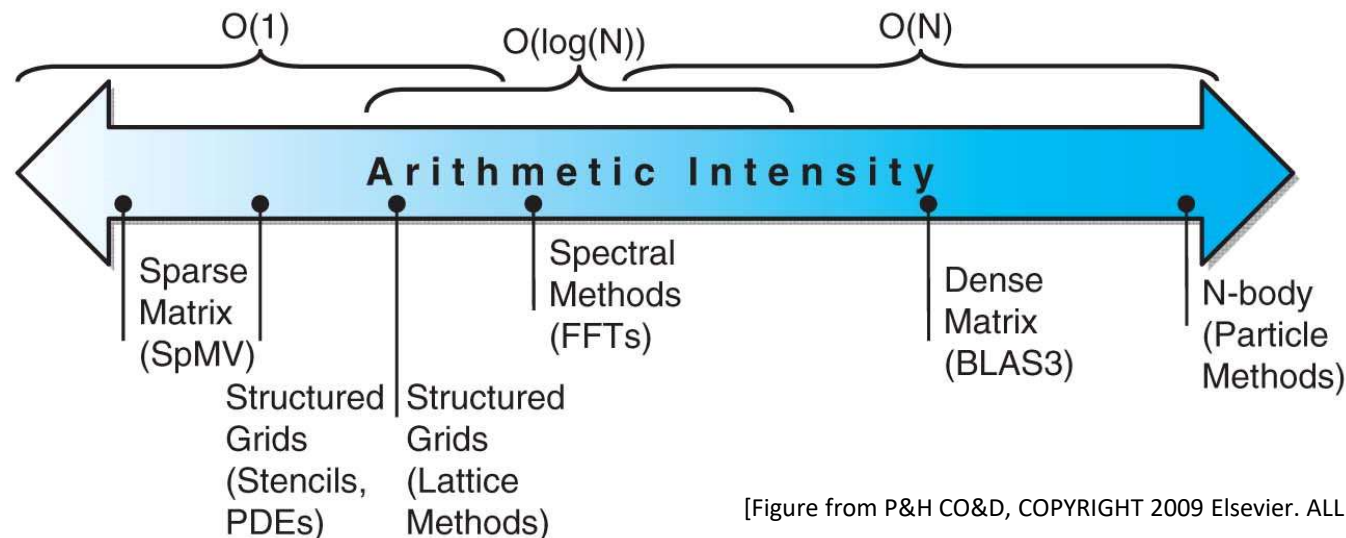
for (i=i0; i<i0+Nb; i++)
  for (j=j0; j<j0+Nb; j++) {
    t=C[i][j];
    for (k=k0; k<k0+Nb; k++)
      t+=A[i][k]*B[k][j];
    C[i][j]=t;
  }

```

need to copy
block (not shown)

- Operation count: N_b^3 float-mult and N_b^3 float-add
- When **A**, **B** fit in scratchpad ($2 \times N_b^2 \times 4$ bytes)
 - $2N_b^3$ 4-byte on-chip reads (**A**, **B**) (fast)
 - $3N_b^2$ 4-byte off-chip DRAM read **A**, **B**, **C** (slow)
 - N_b^2 4-byte off-chip DRAM writeback **C** (slow)
- Arithmetic Intensity = $2N_b^3 / (4 \cdot 4N_b^2) = N_b / 8$

AI and Scaling



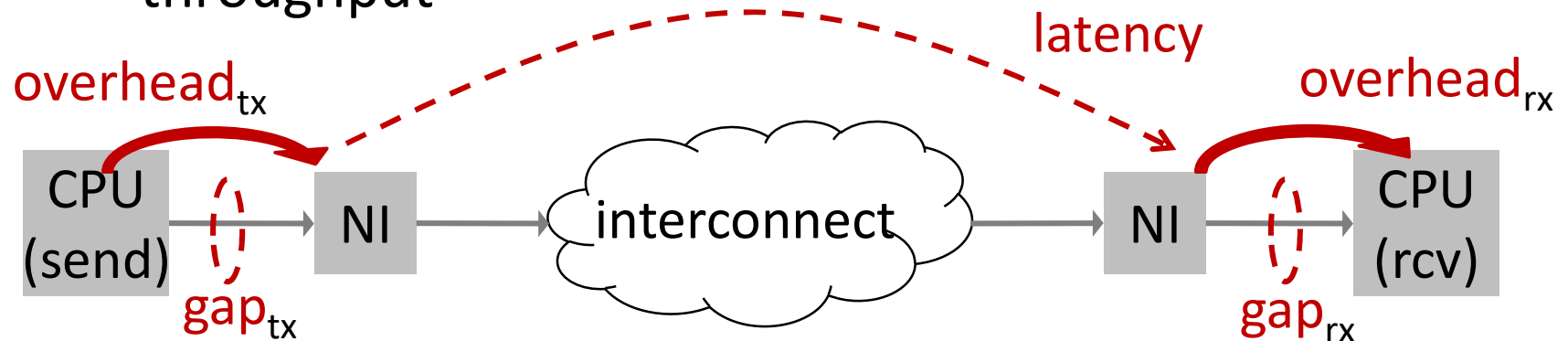
[Figure from P&H CO&D, COPYRIGHT 2009 Elsevier. ALL RIGHTS RESERVED.]

- **AI** is a function of algorithm and problem size
- Higher **AI** means more work per communication and therefore easier to scale
- Recall strong vs. weak scaling
 - strong=increase perf on fixed problem sizes
 - weak=increase perf on proportional problem sizes
 - weak scaling easier if **AI** grows with problem size

LogP Model: Components of Communication Cost

LogP

- A parallel machine model with explicit communication cost
 - **Latency**: transit time between sender and receiver
 - **overhead**: time used up to setup a send or a receive (cycles not doing computation)
 - **gap**: wait time in between successive data units sent or received due to limited transfer bandwidth
 - **Processors**: number of processors, i.e., computation throughput



Message Passing Example

```

if (id==0)           //assume node-0 has A initially
    for (i=1;i<p;i=i+1)
        SEND(i, &A[SHARE*i], SHARE*sizeof(double));
else
    RECEIVE(0,A[])   //receive into local array

sum=0;
for(i=0;i<SHARE;i=i+1) sum=sum+A[i];

remain=p;
do {
    BARRIER();
    half=(remain+1)/2;
    if (id>=half&&id<remain) SEND(id-half,sum,8);
    if (id<(remain/2)) {
        RECEIVE(id+half,&temp);
        sum=sum+temp;
    }
    remain=half;
} while (remain>1);

```

SHARE=HOWMANY/p

Review

[based on P&H Ch 6 example]

Parallel Sum Revisited with LogP

How long?

```

1: if (id==0)
2:     for (i=1;i<100;i=i+1)
3:         SEND(i, &A[100*i], 100*sizeof(double));
4: else RECEIVE(0, A[])
  
```

- o • assuming no back-pressure, **node-0** finishes sending to **node-99** after $99 \times$ overhead of **SEND()**
- L • first byte arrives at **node-99** some network latency later
- g • the complete message arrives at **node-99** after $100 \times \text{sizeof(double)} / \text{network_bandwidth}$
- o • **node-99** finally ready to compute after the overhead to **RECEIVE()**

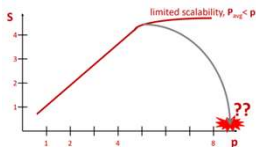
What if $100 \times \text{sizeof(double)} / \text{network_bandwidth}$ greater than the overhead to **SEND()**?

Parallel Sum Revisited with LogP

How long?

```
sum=0 ;
for (i=0 ; i<100 ; i=i+1)  sum=sum+A[i] ;
```

- ideally, this step is computed **p**=100 times faster than summing 10,000 numbers by one processor
- big picture thinking, e.g.,

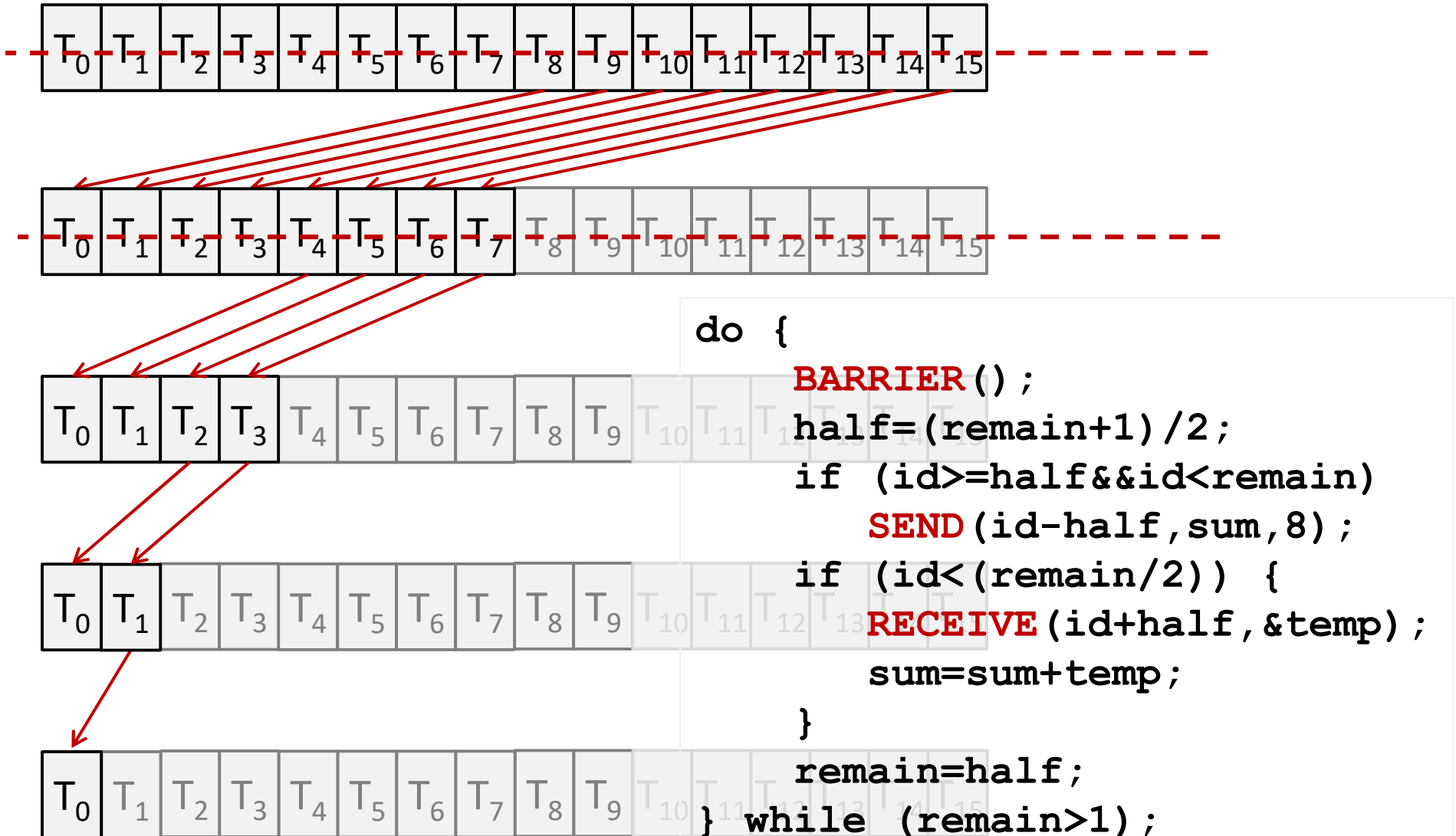


- is the time saved worth the data distribution cost?
- if not, actually faster if parallelized less

- fine-tooth comb thinking, e.g.,
 - **node-1** begins work first; **node-99** begins work last
 \Rightarrow minimize overall finish time by assigning more work to **node-1** and less work to **node-99**
 - maybe latency and bandwidth are different to different nodes

Performance tuning is a craft

Parallel Sum Revisited with LogP



Parallel Sum Revisited with LogP

```
do {
  BARRIER();
  half=(remain+1)/2;
  if (id>=half&&id<remain) SEND(id-half,sum,8);
  if (id<(remain/2)) {
    RECEIVE(id+half,&temp);
    sum=sum+temp;
  }
  remain=half;
} while (remain>1);
```

- do we need to synchronize each round?

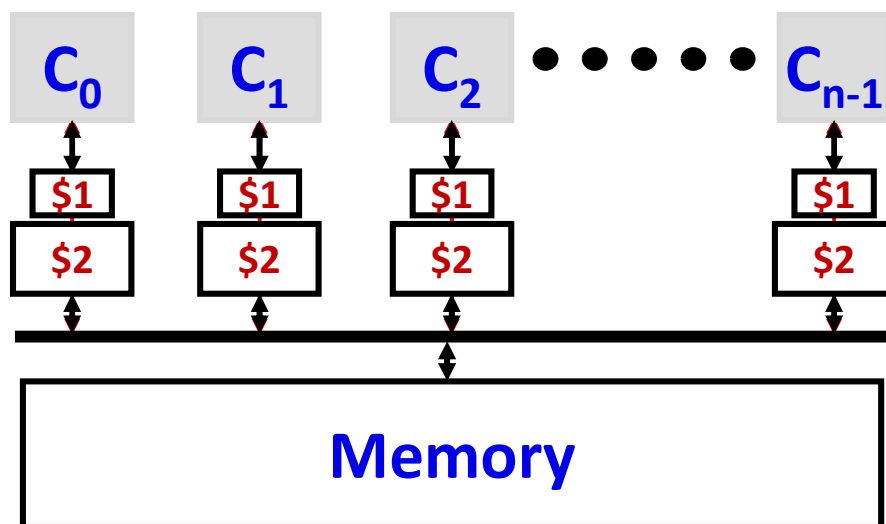
*how does one build a **BARRIER** () ?*

- is this actually faster than if all nodes sent to node-0?

*What if **p** is small? What if **p** is very large?*

Real answer is a combination of techniques

LogP applies to shared memory too



```
do {
    pthread_barrier_wait(...);

    half=(remain+1)/2;
    if (id<(remain/2))
        psum[id]=psum[id]+
            psum[id+half];
    remain=half;
} while (remain>1);
```

- When C_0 is reading $\text{psum}[0+\text{half}]$, the value originates in the cache of C_{half}
 - **L**: time from C_0 's cache miss to when data retrieved from the cache of C_{half} (*via cache coherence*)
 - **g**: there is a finite bandwidth between C_0 and C_{half}
 - **o**: as low as a LW instruction but also pay for stalls

Implications of Communication Cost

- Large **g**—*can't exchange a large amount of data*
 - must have lots of work per byte communicated
 - only scalable for applications with high **AI**
- Large **o**—*can't communicate frequently*
 - can only exploit coarse-grain parallelism
 - if DMA, amount of data not necessarily limited
- Large **L**—*can't send data at the last minute*
 - must have high average parallelism (*more work/time between production and use of data*)
- High cost in each category limits
 - the kind of applications that can speed up, and
 - how much they can speed up

Parallelization not just for Performance

- Ideal parallelization over N CPUs
 - $T = \text{Work} / (k_{\text{perf}} \cdot N)$
 - $E = (k_{\text{switch}} + k_{\text{static}} / k_{\text{perf}}) \cdot \text{Work}$
 N -times static power, but N -times faster runtime
 - $P = N (k_{\text{switch}} \cdot k_{\text{perf}} + k_{\text{static}})$
 - Alternatively, forfeit speedup for power and energy reduction by $S_{\text{freq}} = 1/N$ (assume $S_{\text{voltage}} \approx S_{\text{freq}}$ below)
 - $T = \text{Work} / k_{\text{perf}}$
 - $E'' = (k_{\text{switch}} / N^2 + k_{\text{static}} / (k_{\text{perf}} N)) \cdot \text{Work}$
 - $P'' = k_{\text{switch}} \cdot k_{\text{perf}} / N^2 + k_{\text{static}} / N$
- Also works with using N slower-simpler CPUs

Don't forget