I: Performance metrics (cont’d) II: Parallel programming models and mechanics

Prof. Richard Vuduc

Georgia Institute of Technology

CSE/CS 8803 PNA, Spring 2008

[L.05] Tuesday, January 22, 2008

Algorithms for 2-D (3-D) Poisson, $N=n^2$ ($=n^3$)

Algorithm	Serial	PRAM	Memory	# procs
Dense LU	N^3	N	N^2	N^2
Band LU	N^2 ($N^{7/3}$)	N	$N^{3/2}$ ($N^{5/3}$)	N ($N^{4/3}$)
Jacobi	**N^2 ($N^{5/3}$)**	**N ($N^{2/3}$)**	**N**	**N**
Explicit inverse	N^2	log N	N^2	N^2
Conj. grad.	**$N^{3/2}$ ($N^{4/3}$)**	**$N^{1/2(1/3)}$ log N**	**N**	**N**
RB SOR	**$N^{3/2}$ ($N^{4/3}$)**	**$N^{1/2}$ ($N^{1/3}$)**	**N**	**N**
Sparse LU	$N^{3/2}$ (N^2)	$N^{1/2}$	N log N ($N^{4/3}$)	N
FFT	N log N	log N	N	N
Multigrid	N	$\log^2$ N	N	N
Lower bound	**N**	**log N**	**N**	

PRAM = idealized parallel model with zero communication cost.

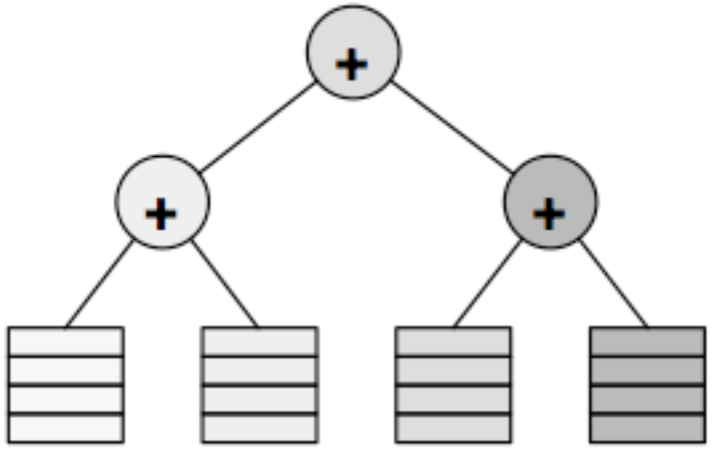
Source: Demmel (1997)

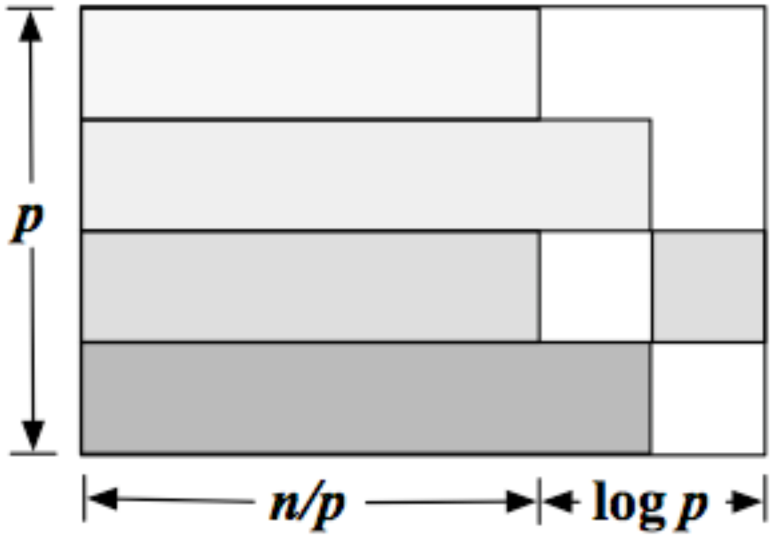
Sources for today’s material

- Mike Heath at UIUC
- CS 267 (Yelick & Demmel, UCB)

Efficiency and scalability metrics (wrap-up)

Example: Summation using a tree algorithm





- **Efficiency**

$$E_p \equiv \frac{C_1}{C_p} \approx \frac{n}{n + p\log p} = \frac{1}{1 + \frac{p}{n}\log p}$$

Basic definitions

M	**Memory complexity**	Storage for given problem (*e.g.*, words)
W	**Computational complexity**	Amount of work for given problem (*e.g.*, flops)
V	**Processor speed**	Ops / time (*e.g.*, flop/s)
T	**Execution time**	Elapsed wallclock (*e.g.*, secs)
C	**Computational cost**	(No. procs) * (exec. time) [*e.g.*, processor-hours]

Parallel scalability

- Algorithm is **scalable** if

$$E_p \equiv \frac{C_1}{C_p} = \Theta(1) \text{ as } p \to \infty$$

- Why use more processors?
  - Solve fixed problem in less time
  - Solve larger problem in same time (or any time)
  - Obtain sufficient aggregate memory
  - Tolerate latency and/or use all available bandwidth (Little’s Law)

Is this algorithm scalable?

- No, for fixed **problem size**, **exec. time**, and **work / proc.**
- Determine **isoefficiency function** for which efficiency is constant

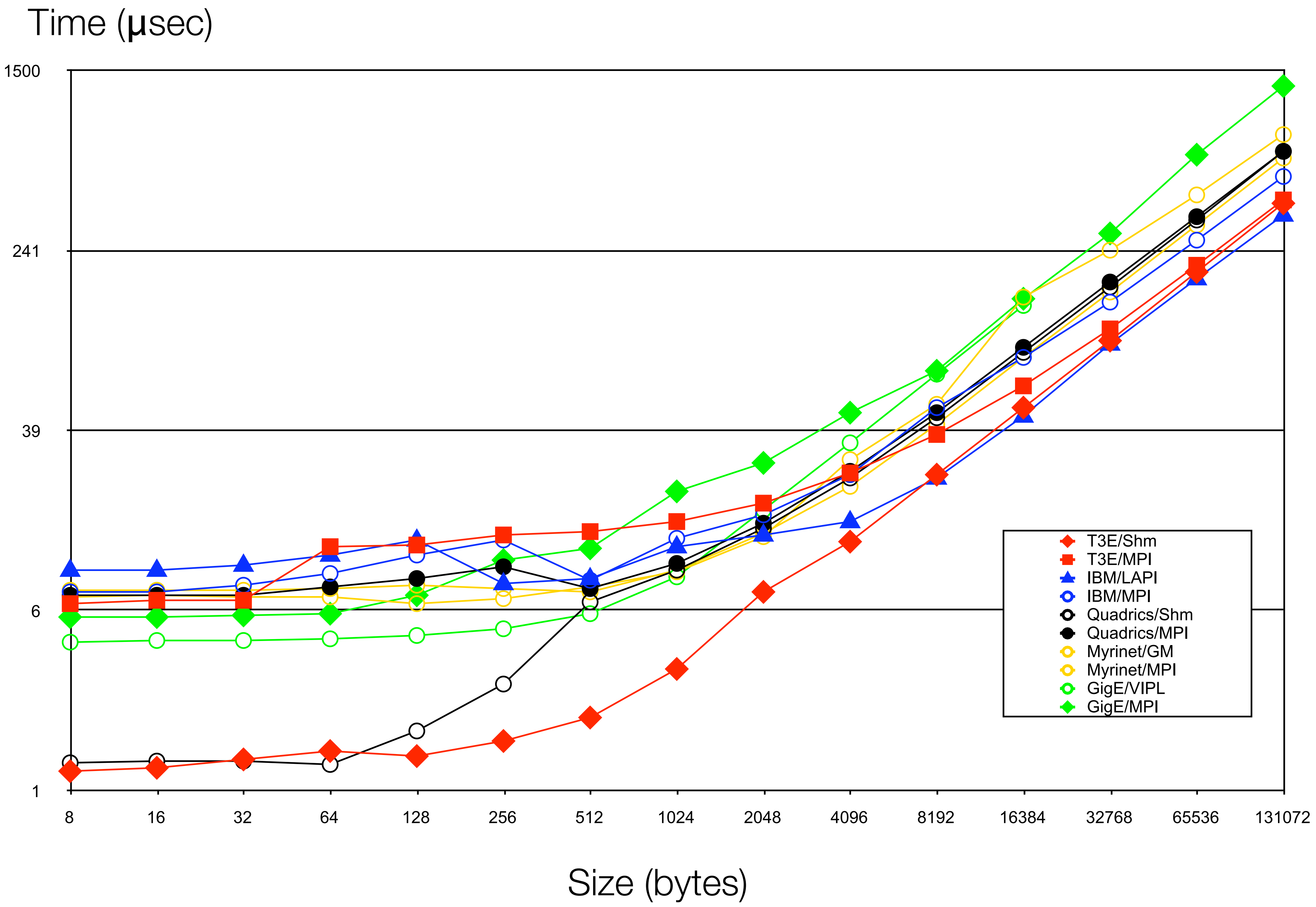
$$
\begin{aligned}
E_p \equiv \frac{C_1}{C_p} &\approx \frac{n}{n + p\log p} = \frac{1}{1 + \frac{p}{n}\log p} = E \text{ (const.)} \\
&\Longrightarrow \\
n(p) &= \Theta(p\log p)
\end{aligned}
$$

- But then execution time grows with p:

$$T_p = \frac{n}{p} + \log p = \Theta(\log p)$$

A simple model of communication performance

Time to Send a Message



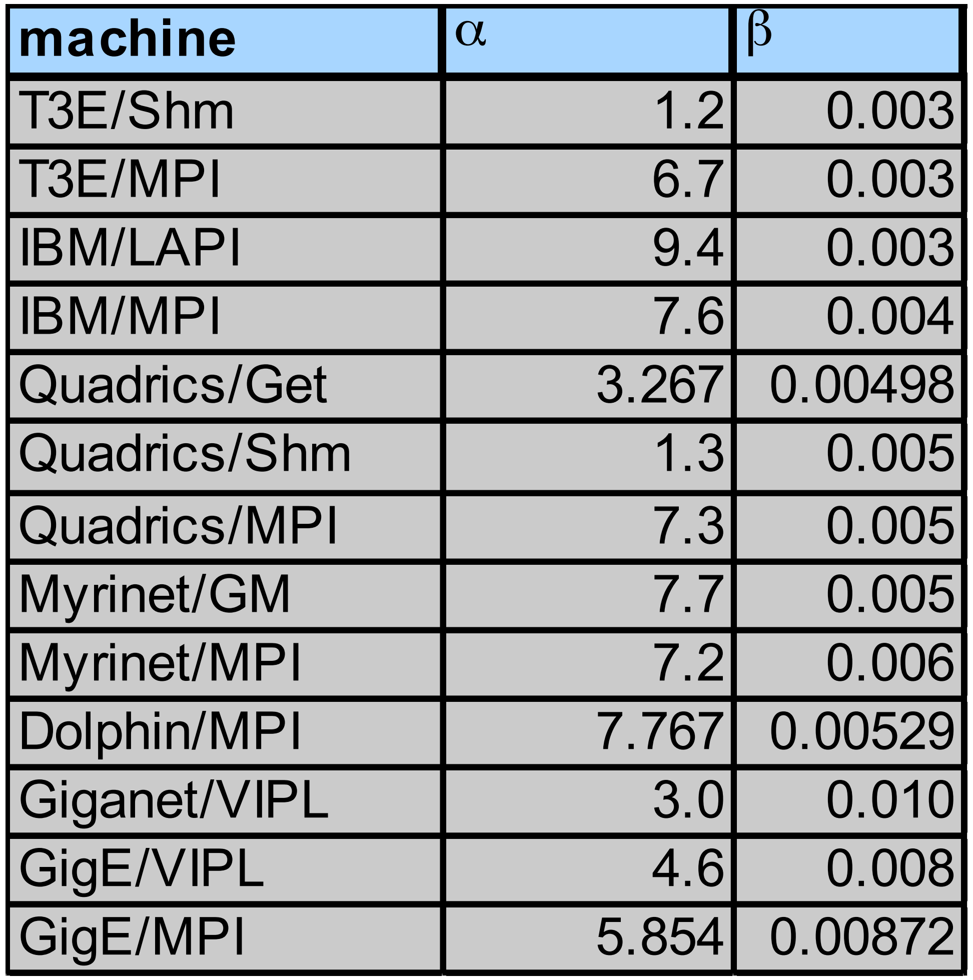
Latency and bandwidth model

- Model time to send a message in terms of latency and bandwidth

$$t(n) \quad = \quad \alpha + \frac{n}{\beta}$$

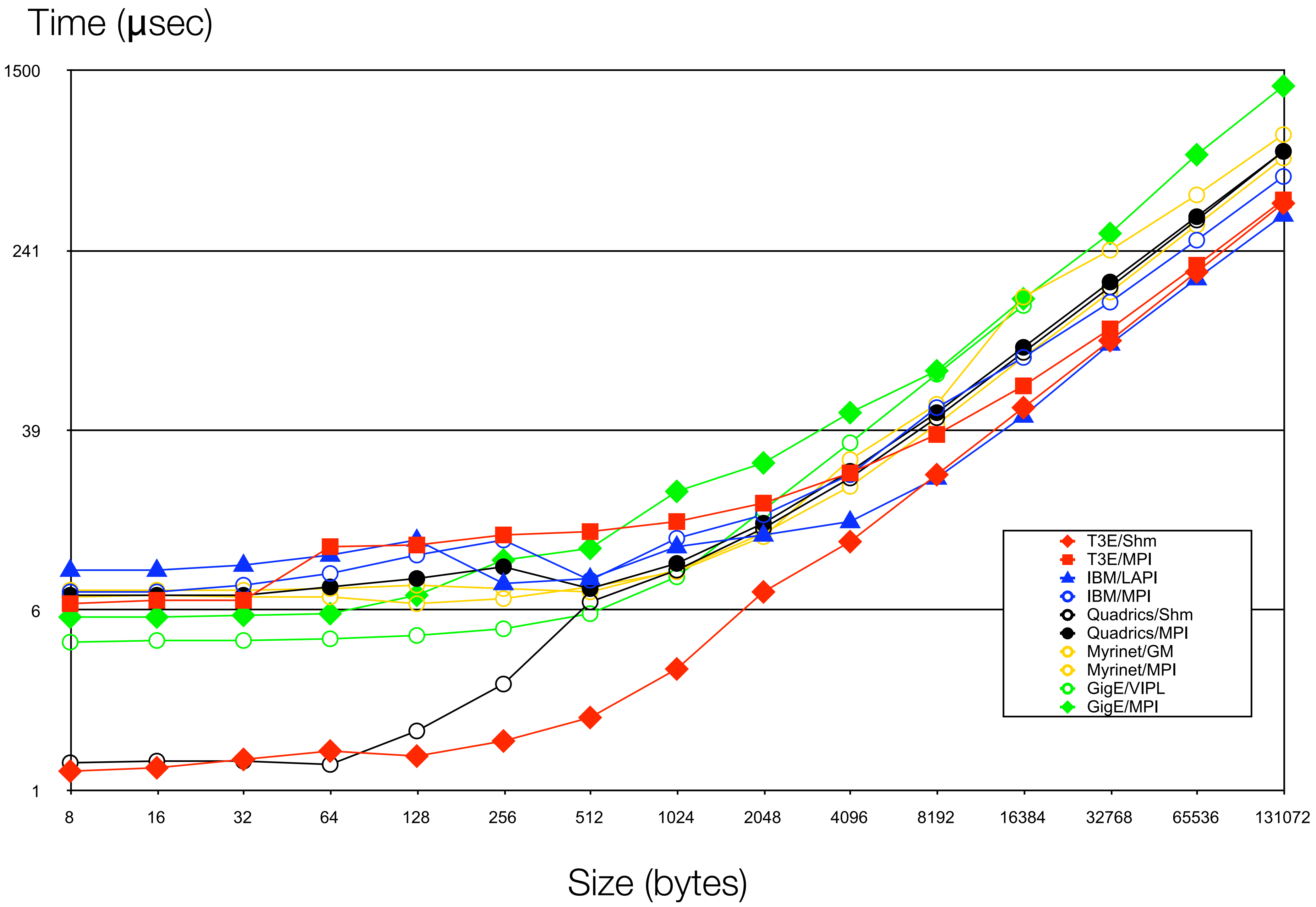
- Usually have cost(flop) << 1/**β** << **α**
  - One long message cheaper than many short ones
  - Can do hundreds or thousands of flops for each message
- Efficiency demands large computation-to-communication ratio

Empirical latency and (inverse) bandwidth (μsec) on real machines

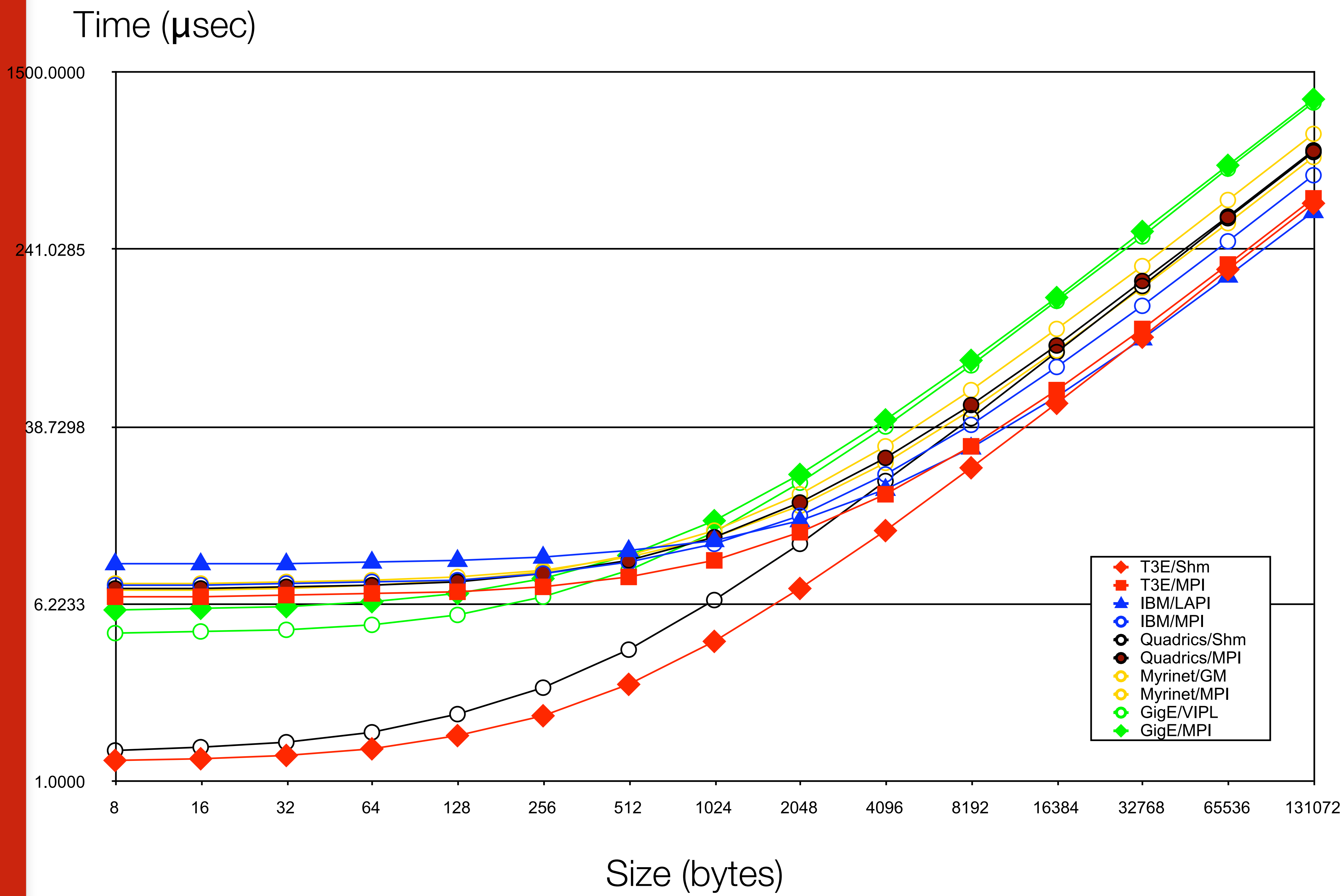


machine	α	β
T3E/Shm	1.2	0.003
T3E/MPI	6.7	0.003
IBM/LAPI	9.4	0.003
IBM/MPI	7.6	0.004
Quadrics/Get	3.267	0.00498
Quadrics/Shm	1.3	0.005
Quadrics/MPI	7.3	0.005
Myrinet/GM	7.7	0.005
Myrinet/MPI	7.2	0.006
Dolphin/MPI	7.767	0.00529
Giganet/VIPL	3.0	0.010
GigE/VIPL	4.6	0.008
GigE/MPI	5.854	0.00872

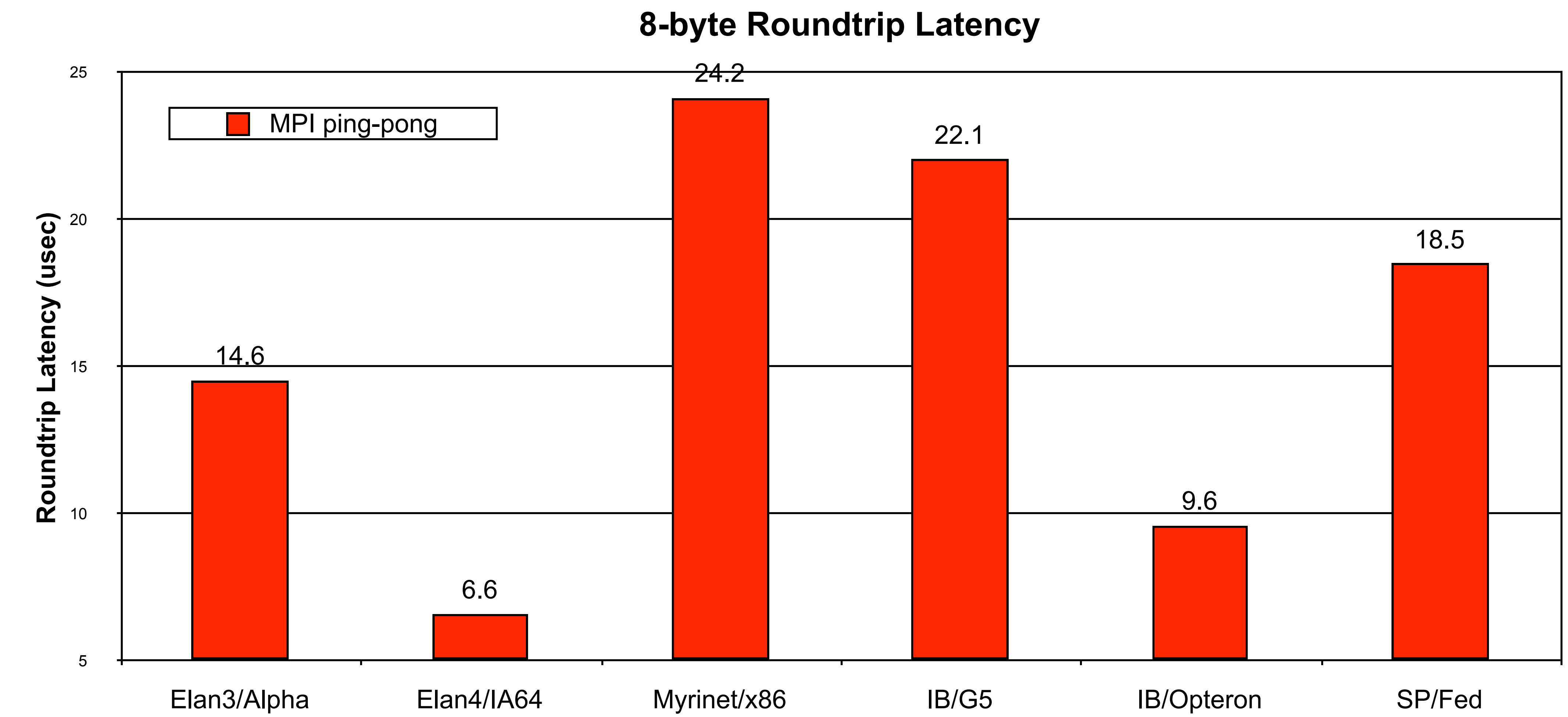
Time to Send a Message



Time to Send a Message (Model)

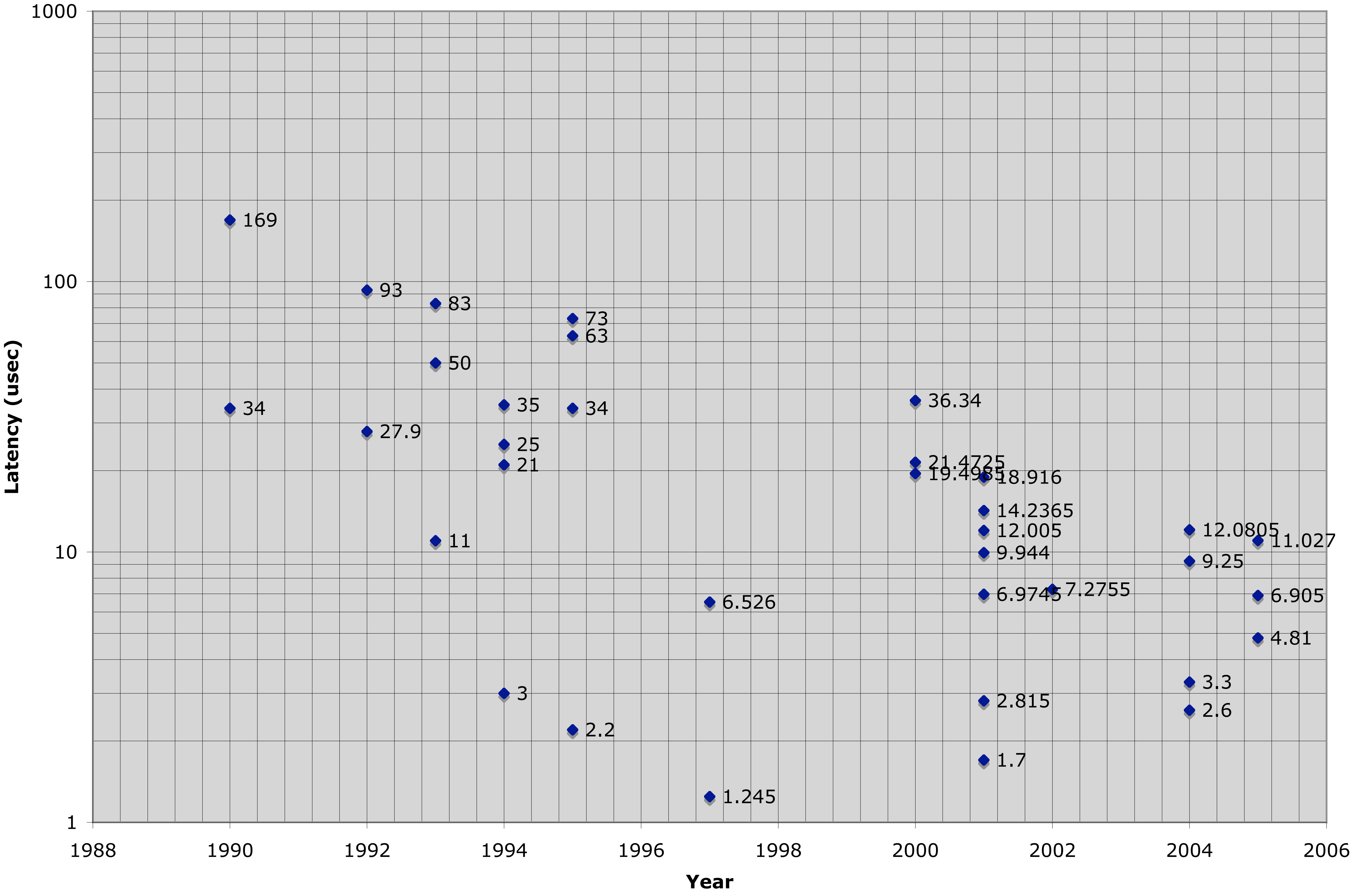


Latency on some current machines (MPI round-trip)

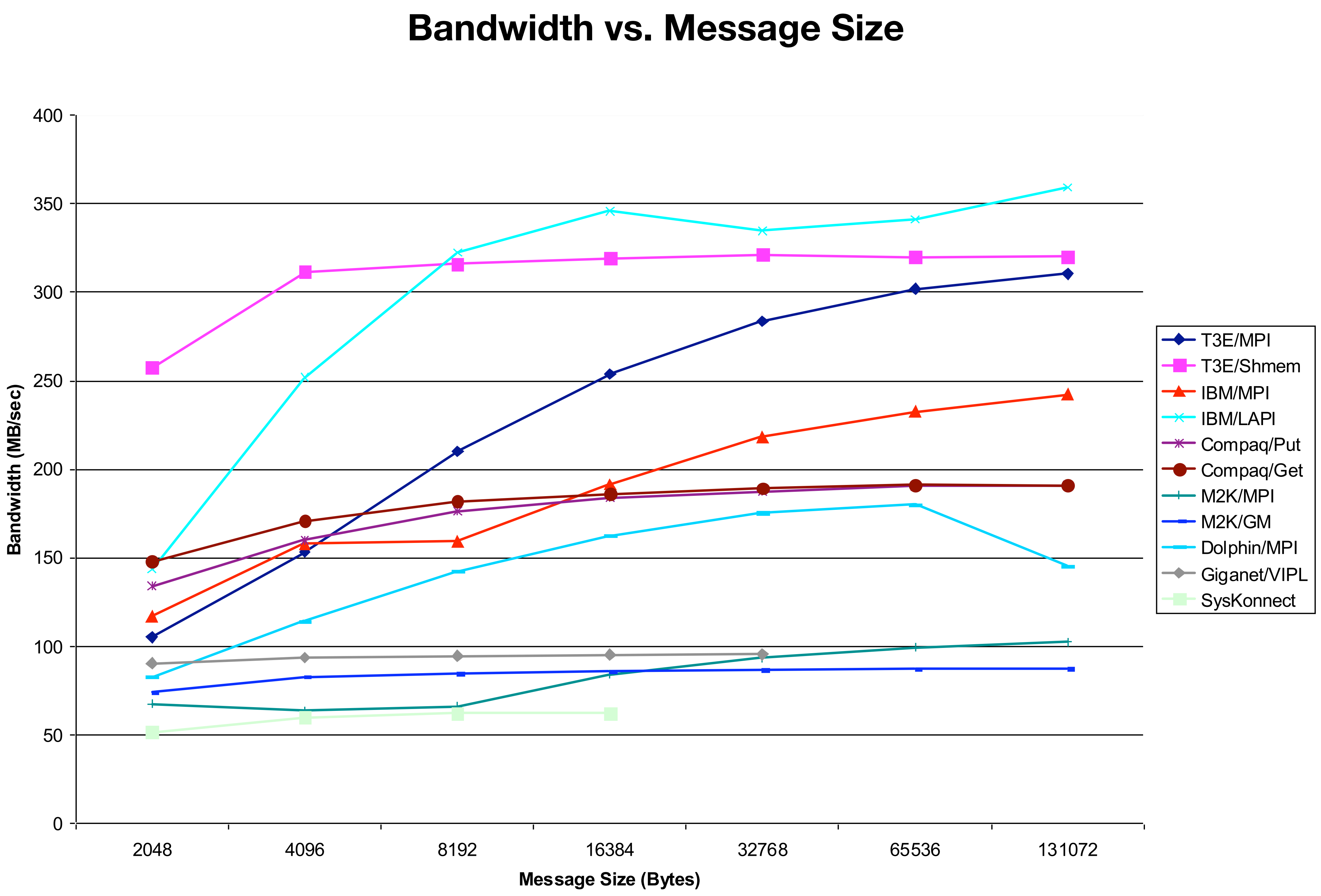


Source: Yelick (UCB/LBNL)

End-to-end latency (1/2 round-trip) over time



Source: Yelick (UCB/LBNL)

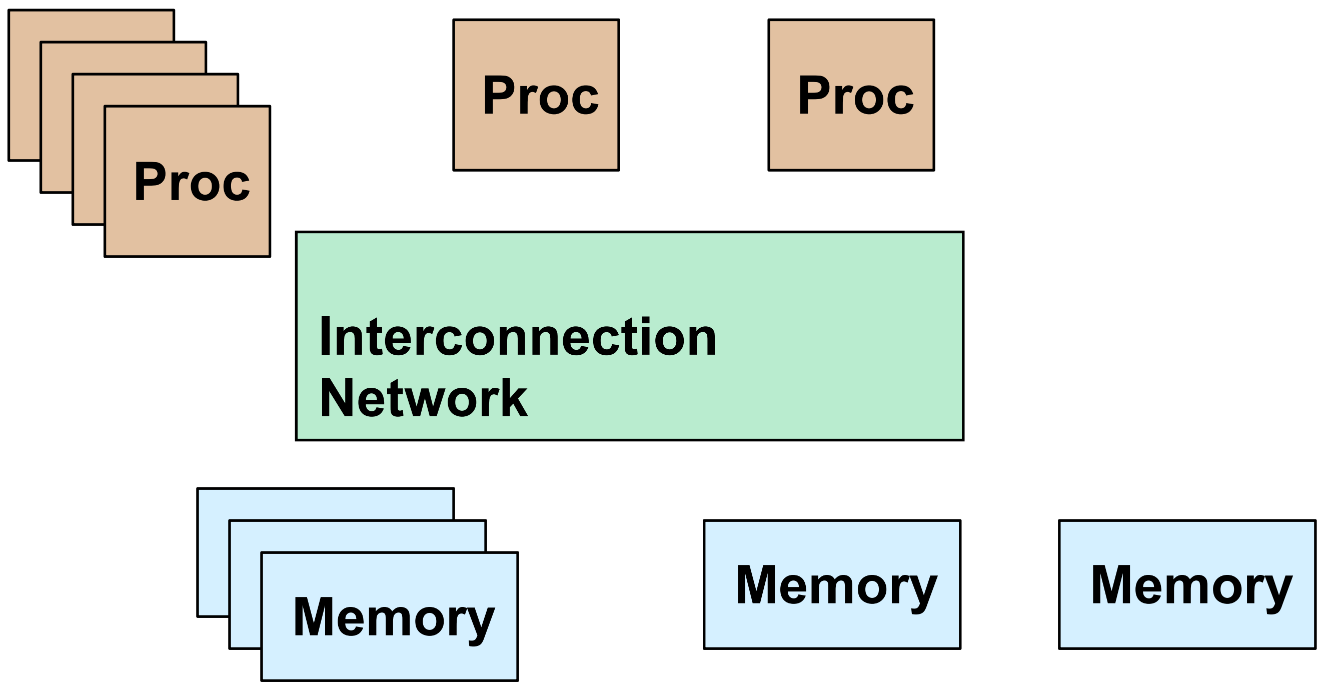


Source: Mike Welcome (NERSC)

Parallel programming models

A generic parallel architecture

- Physical location of memories, processors? Connectivity?



What is a “parallel programming model?”

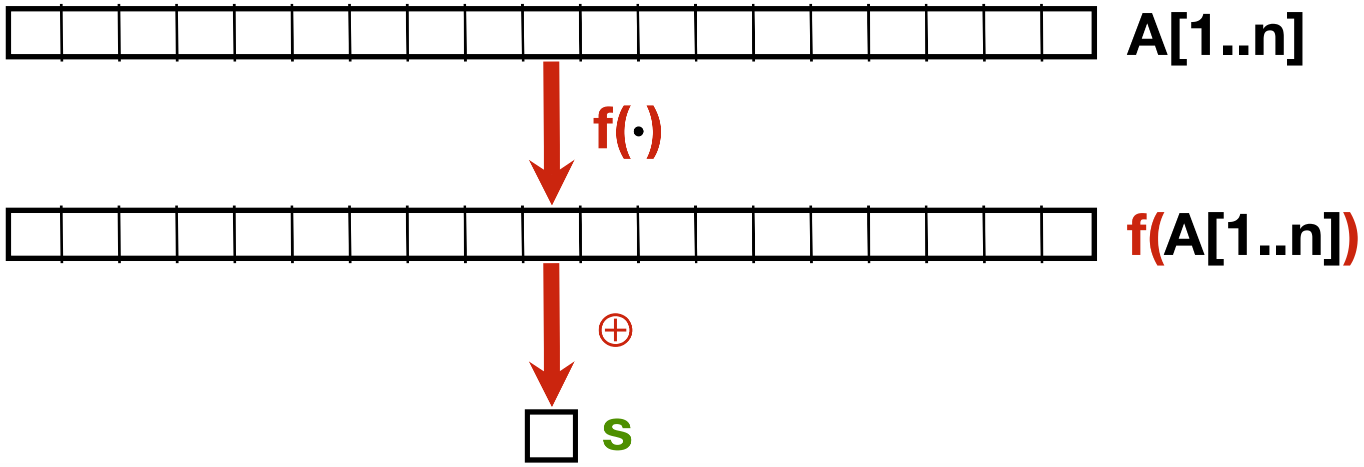
- **Languages + libraries composing abstract view of machine**
- Major constructs
  - **Control**: Create parallelism? Execution model?
  - **Data**: Private vs. shared?
  - **Synchronization**: Coordinating tasks? Atomicity?
- Variations in models
  - Reflect diversity of machine architectures
  - Imply variations in cost

Running example: Summation

- Compute the sum,

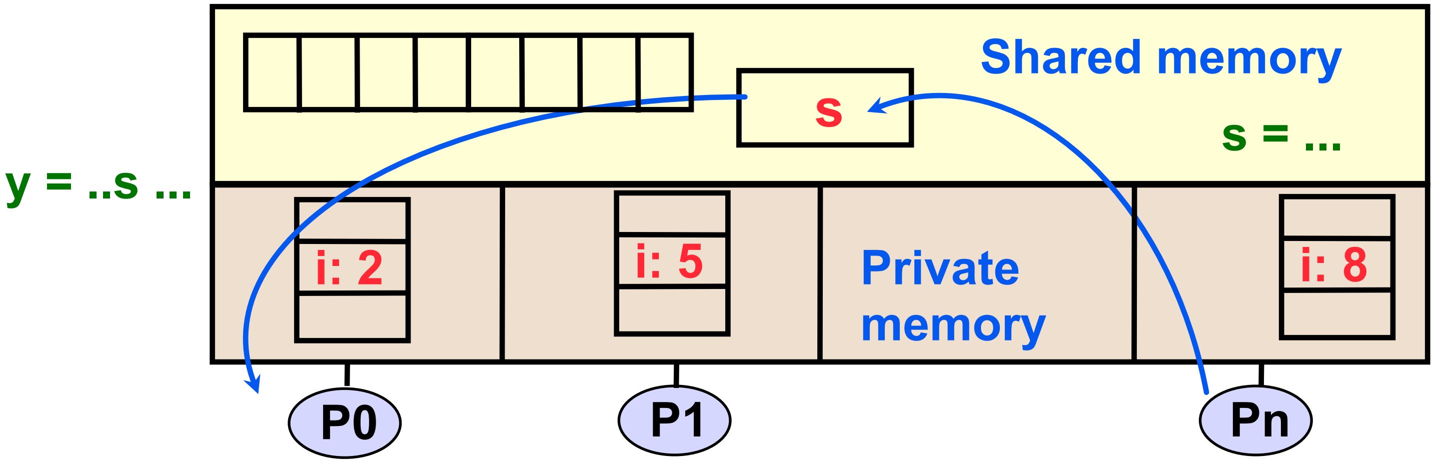
$$s = \sum_{i=1}^{n} f(a_i)$$

- Questions: Where is “A”? Which processors do what? How to combine?



Programming model 1: Shared memory

- **Program** = collection of **threads** of control
- Each thread has **private** variables
- May access **shared** variables, for communicating implicitly and synchronizing



Need to avoid race conditions: Use locks

- **Race condition** (**data race**): Two threads access a variable, with at least one writing and concurrent accesses

```
shared int s = 0;
```

```
Thread 1

  for i = 0, n/2-1
      s = s + f(A[i])
```

```
Thread 2

  for i = n/2, n-1
      s = s + f(A[i])
```

Need to avoid race conditions: Use locks

- **Explicitly lock** to guarantee atomic operations

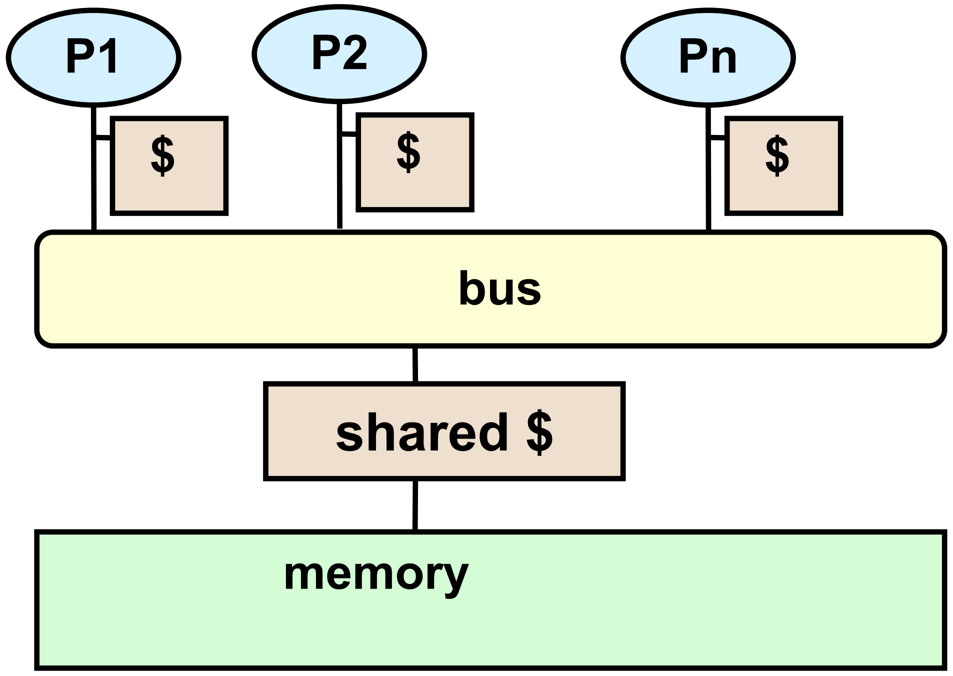
```
shared int s = 0;
shared lock lk;
```

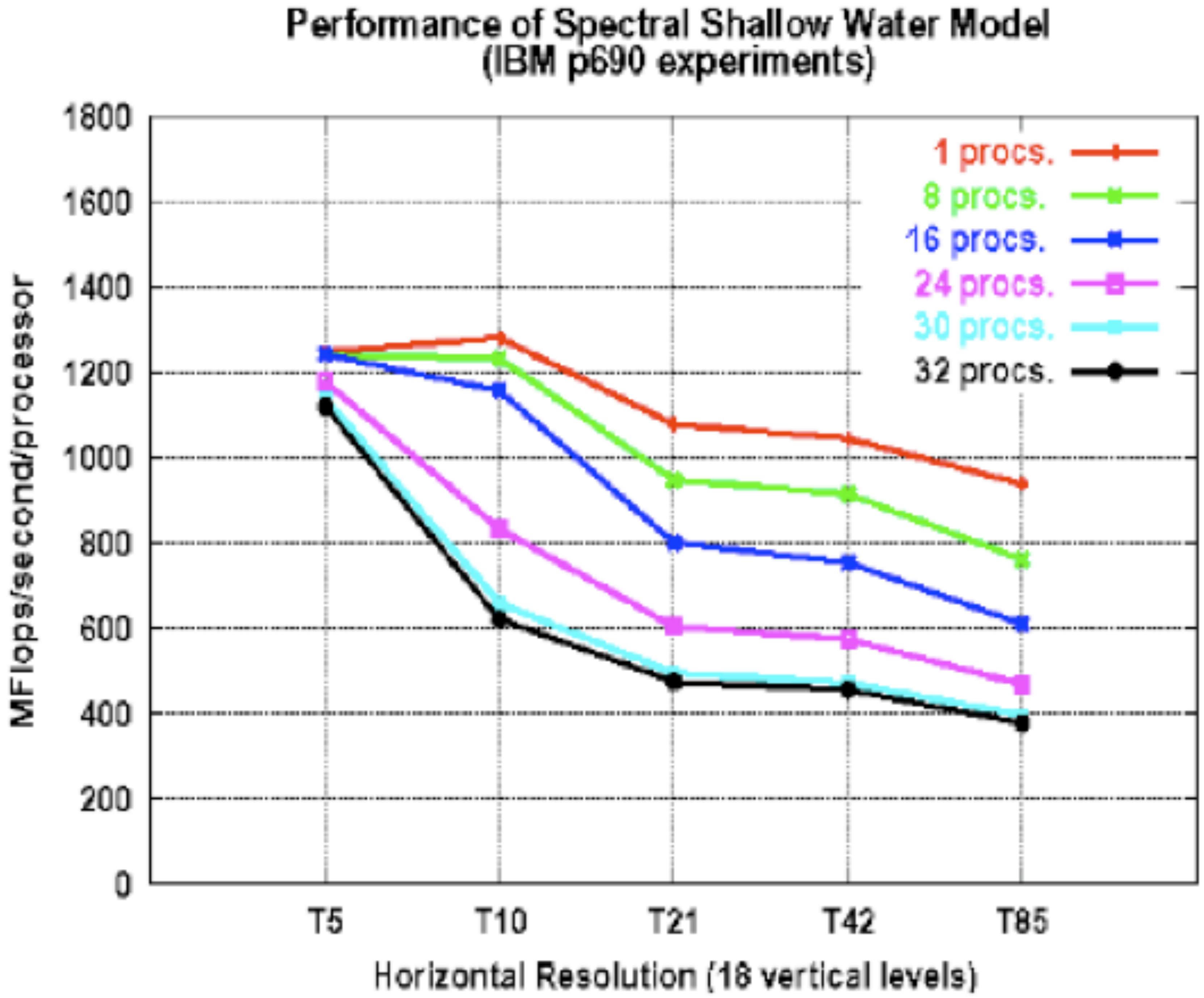
```
Thread 1

   local_s1= 0
   for i = 0, n/2-1
      local_s1 = local_s1 + f(A[i])
   lock(lk);
   s = s + local_s1
   unlock(lk);
```

```
Thread 2

   local_s2 = 0
   for i = n/2, n-1
      local_s2= local_s2 + f(A[i])
   lock(lk);
   s = s +local_s2
   unlock(lk);
```

Machine model 1a:
Symmetric multiprocessors (SMPs)

- All processors connect to **large shared memory**
- Challenging to scale both hardware & software > 32 procs

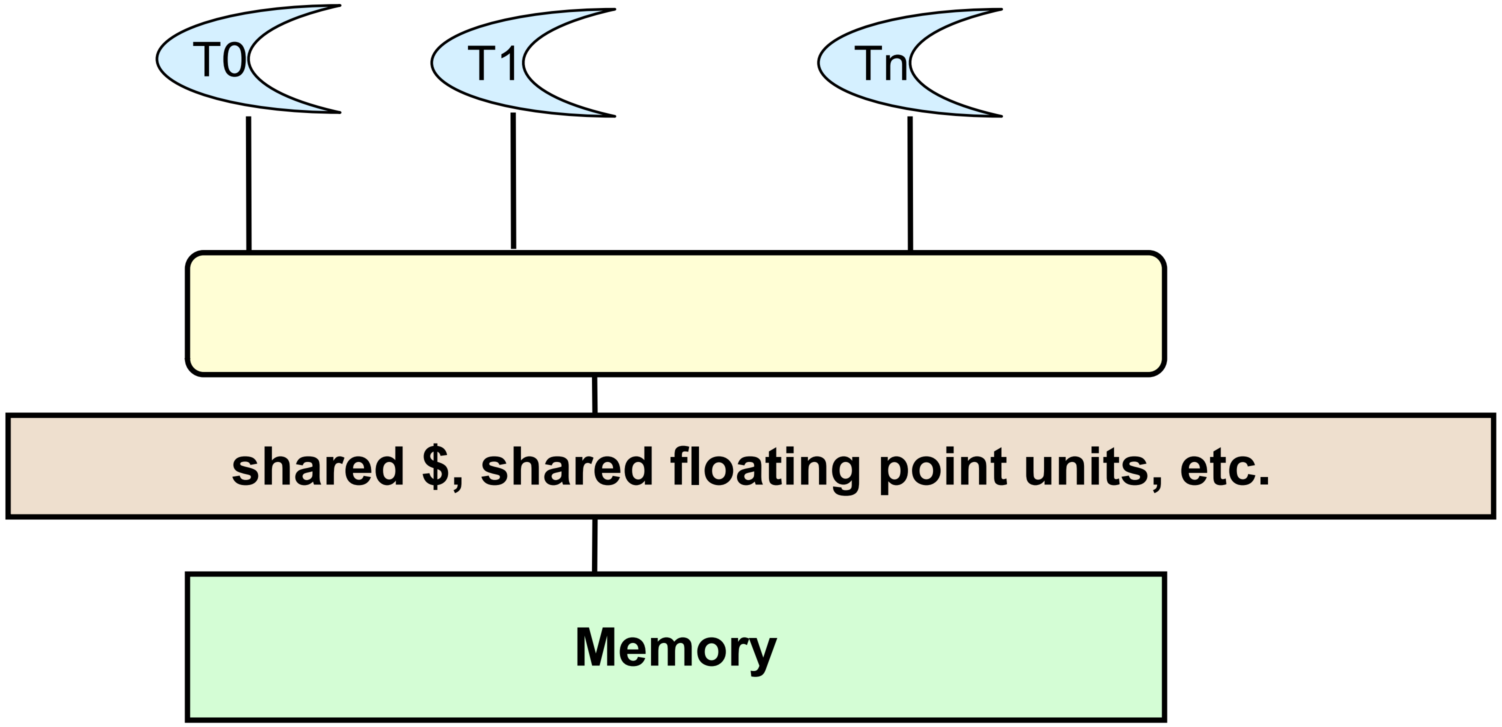


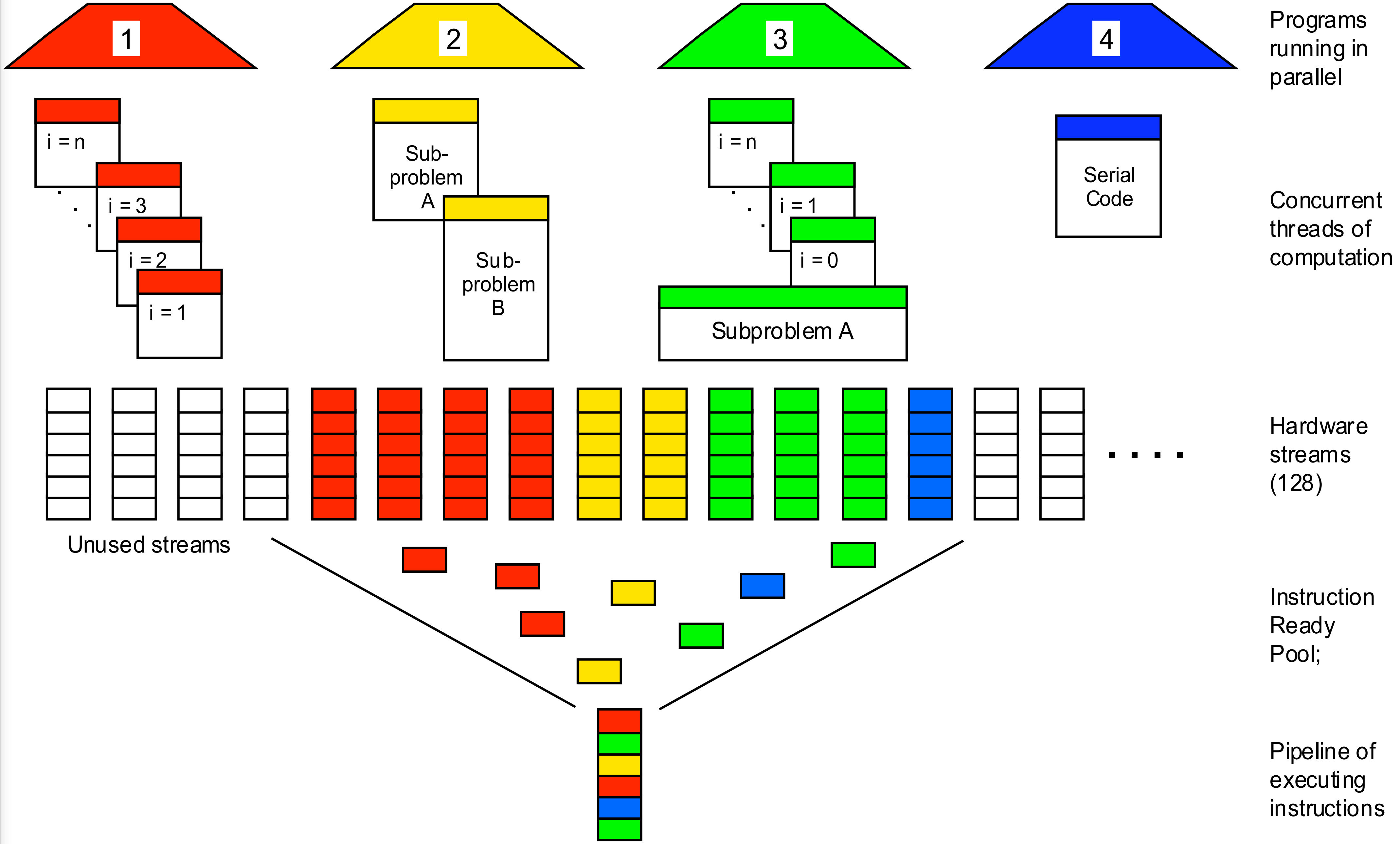


Source: Pat Worley (ORNL)

Machine model 1b: Simultaneous multithreaded processor (SMT)

- Multiple thread contexts share memory and functional units
- Switch among threads during long-latency memory ops





Programs running in parallel

Concurrent threads of computation

Hardware streams (128)

Instruction Ready Pool;

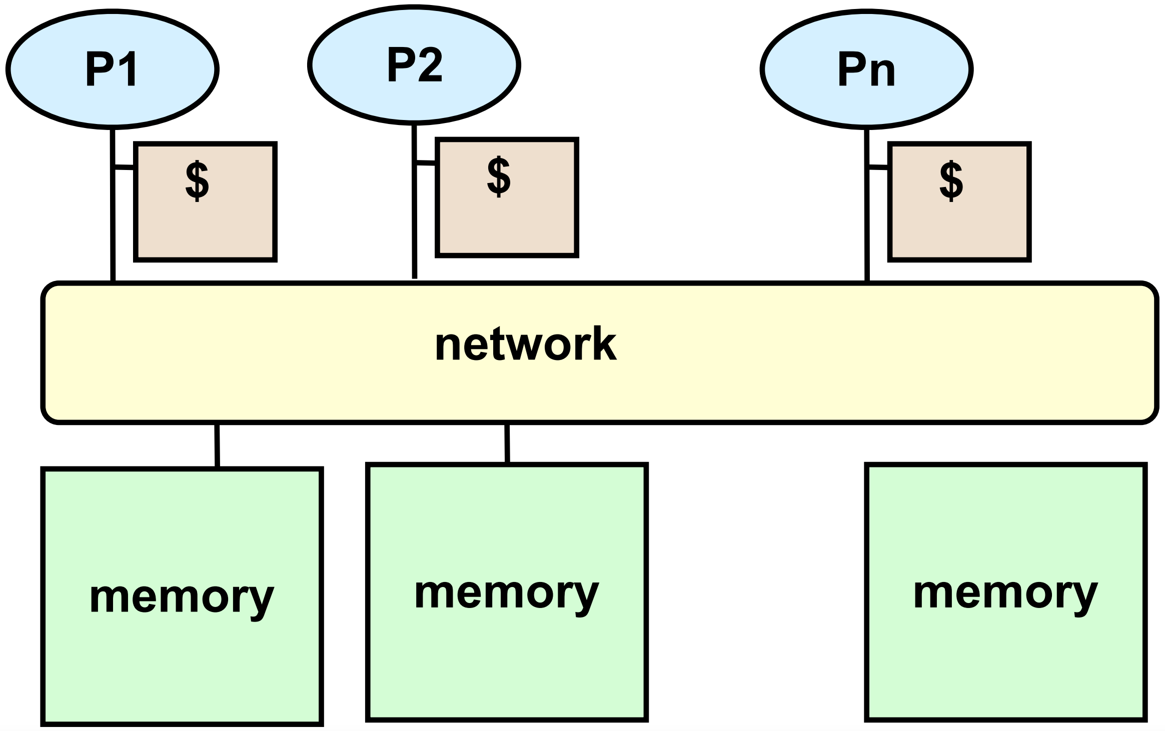
Pipeline of executing instructions

Cray El Dorado processor

Source: John Feo (Cray)

Machine model 1c: Distributed shared memory

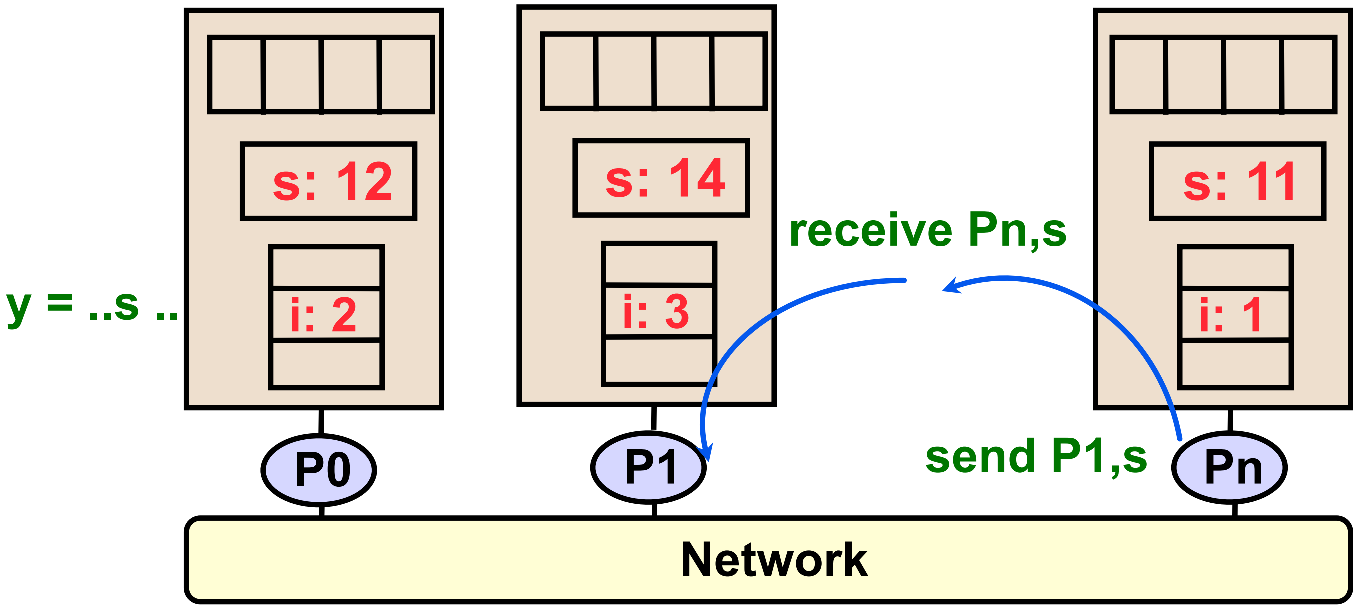
- Memory **logically shared**, but **physically distributed**
- Challenge to scale **cache coherency protocols** > 512 procs



Cache lines (pages) must be large to amortize overhead ➔ locality is critical to performance

Programming model 2: Message passing

- Program = **named** processes
- **No shared** address space
- Processes communicate via **explicit send/receive** operations



Example: Computing A[1]+A[2]

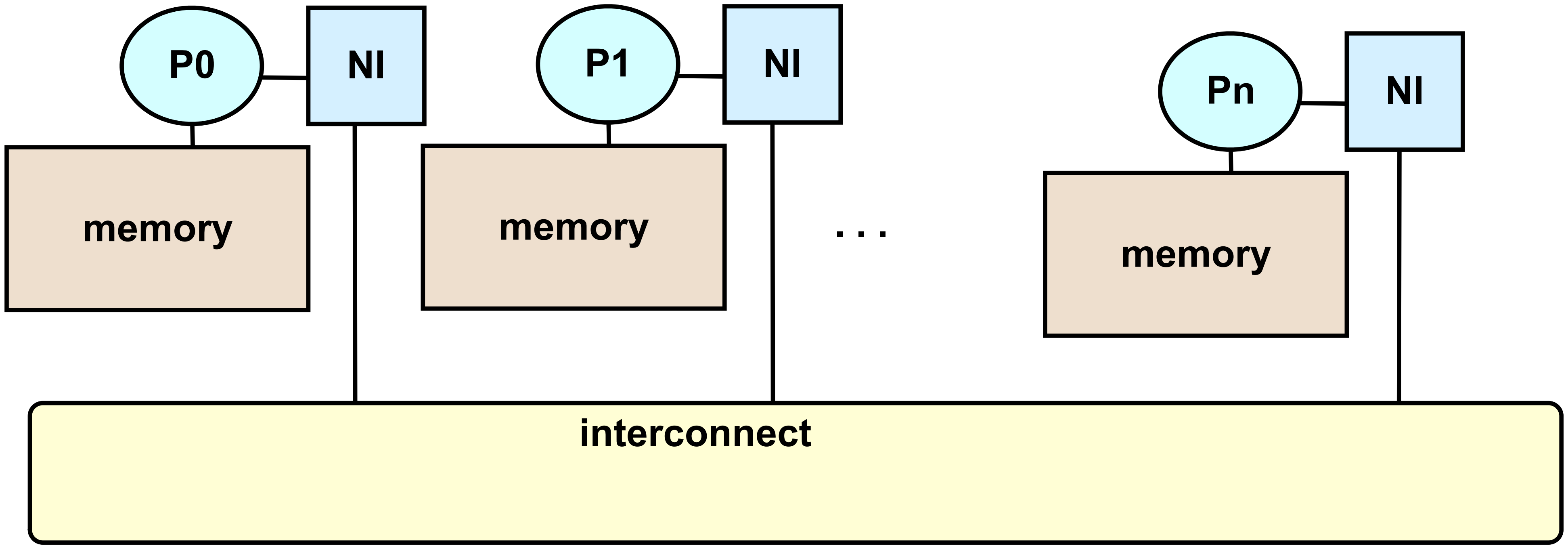
```
Processor 1:
    x = A[1]
    SEND x → Proc. 2
    RECEIVE y ← Proc. 2
    s = x + y
```

```
Processor 2:
    x = A[2]
    SEND x → Proc. 1
    RECEIVE y ← Proc. 1
    s = x + y
```

- What could go wrong in the following code?
  - Scenario A: Send/receive is like the **telephone** system
  - Scenario B: Send/receive is like the **post office**

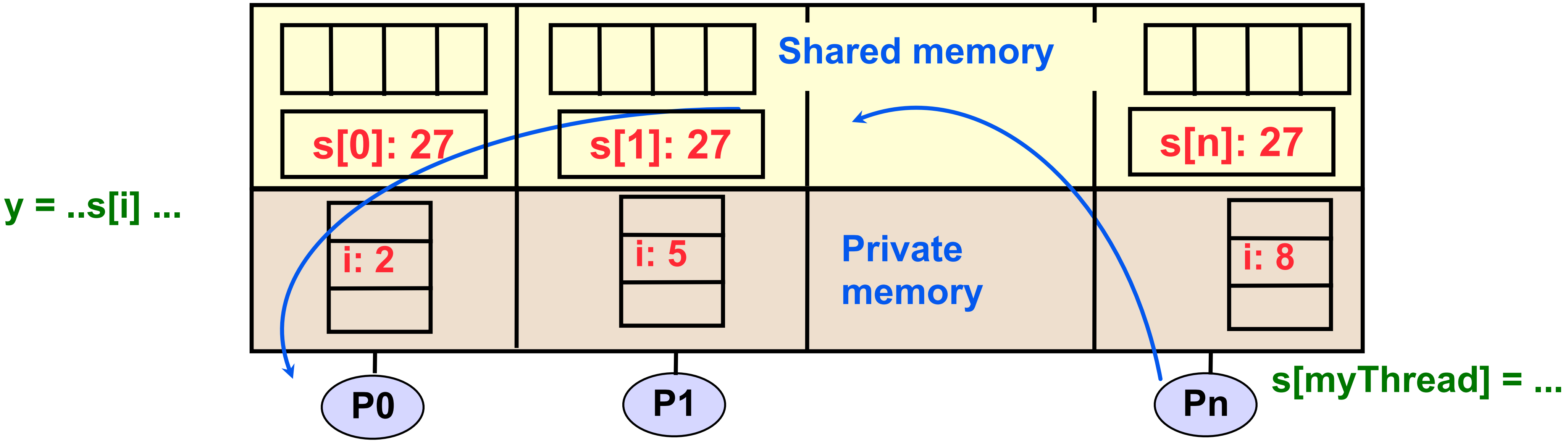
Machine model 2a: Distributed memory

- **Separate** processing nodes, memory
- Communicate through **network interface** over interconnect



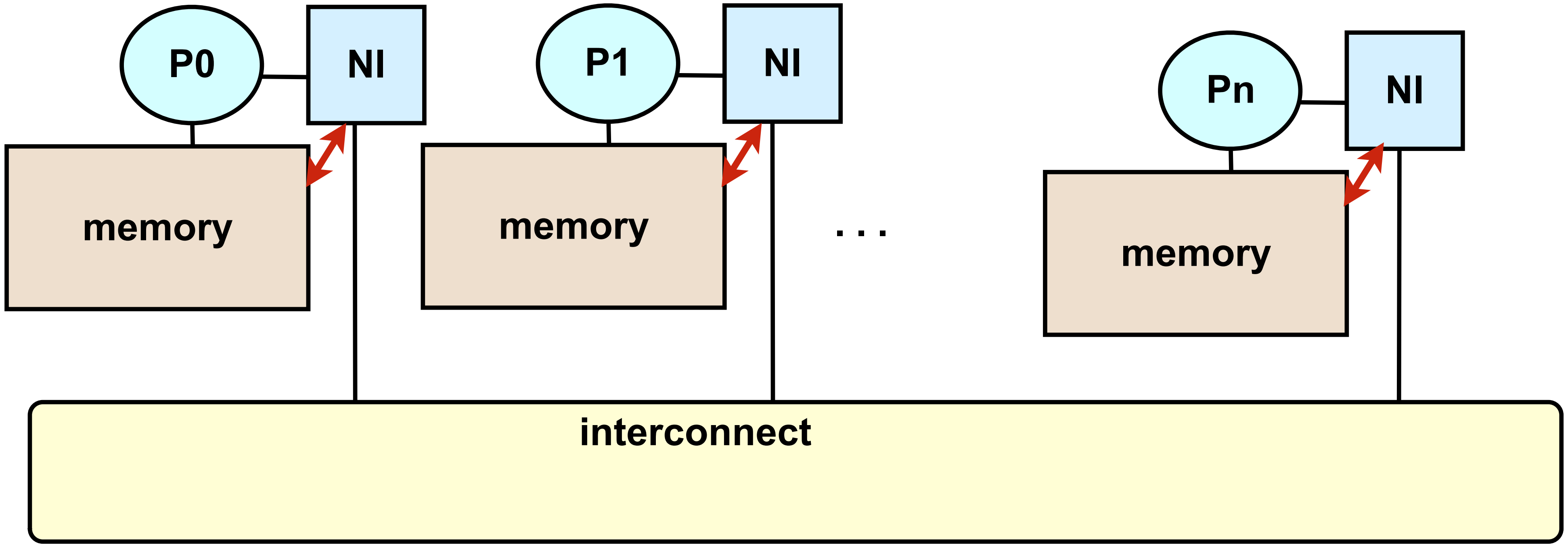
Programming model 2b: Global address space (GAS)

- Program = **named** threads
- Shared data, but **partitioned** over local processes
- Implied cost model: **remote accesses cost more**



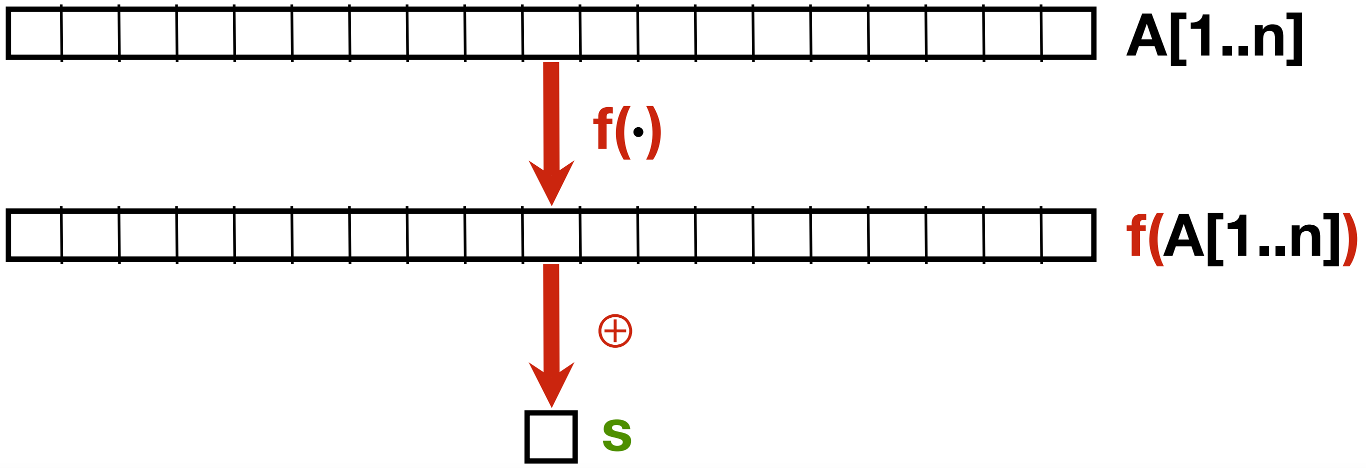
Machine model 2b:
Global address space

- Same as distributed, but NI can access memory **w/o interrupting CPU**
- **One-sided** communication; remote direct memory access (RDMA)



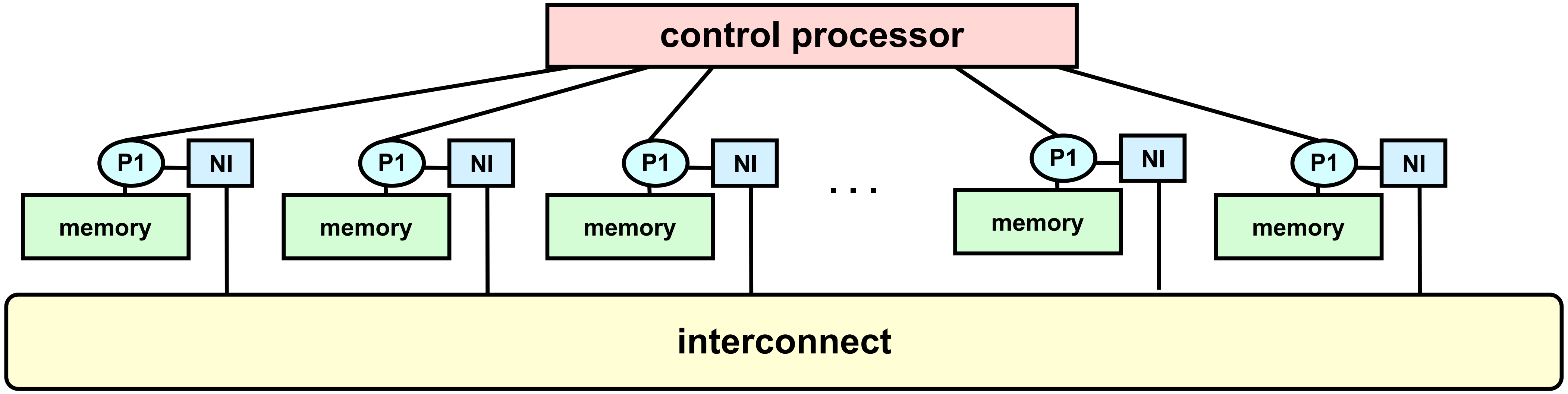
Programming model 3: Data parallel

- Program = **single thread** performing **parallel operations** on data
- **Implicit** communication and coordination; easy to understand
- *Drawback*: Not always applicable
- Examples: HPF, MATLAB/StarP



Machine model 3a: Single instruction, multiple data (SIMD)

- Control processor issues instruction, (usually) simpler processors execute
- May “turn off” some processors
- Examples: CM2, Maspar

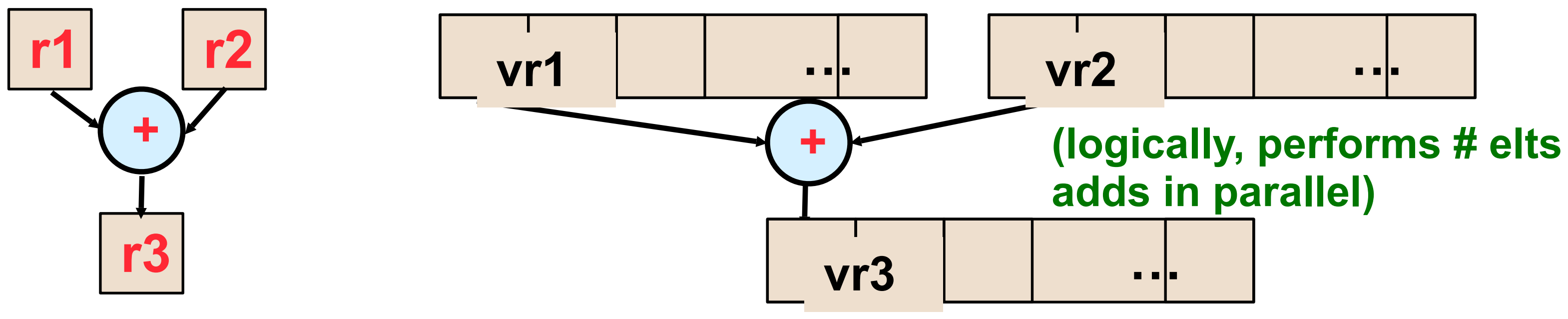


Machine model 3b: Vector processors

- Single processor with multiple functional units
  - Perform same operation
  - Instruction specifies large amount of parallelism, hardware executes on a subset
- **Rely on compiler** to find parallelism
- Resurgent interest
  - Large scale: Earth Simulator, Cray X1
  - Small scale: SIMD units (*e.g.*, SSE, Altivec, VIS)

Vector hardware

- Operations on vector registers, O(10-100) elements / register



(logically, performs # elts adds in parallel)

- Actual hardware has 2-4 vector **pipes** or **lanes**

Programming model 4: Hybrid

- May mix any combination of preceeding models
  - MPI + threads
  - DARPA HPCS languages mix threads and data parallel in global address space

Machine model 4: Clusters of SMPs (CLUMPs)

- Use SMPs as building block nodes
- Many clusters (*e.g.*, GT “warp” cluster)
- Best programming model?
  - “Flat” MPI
  - Shared mem in SMP, MPI between nodes

Administrivia

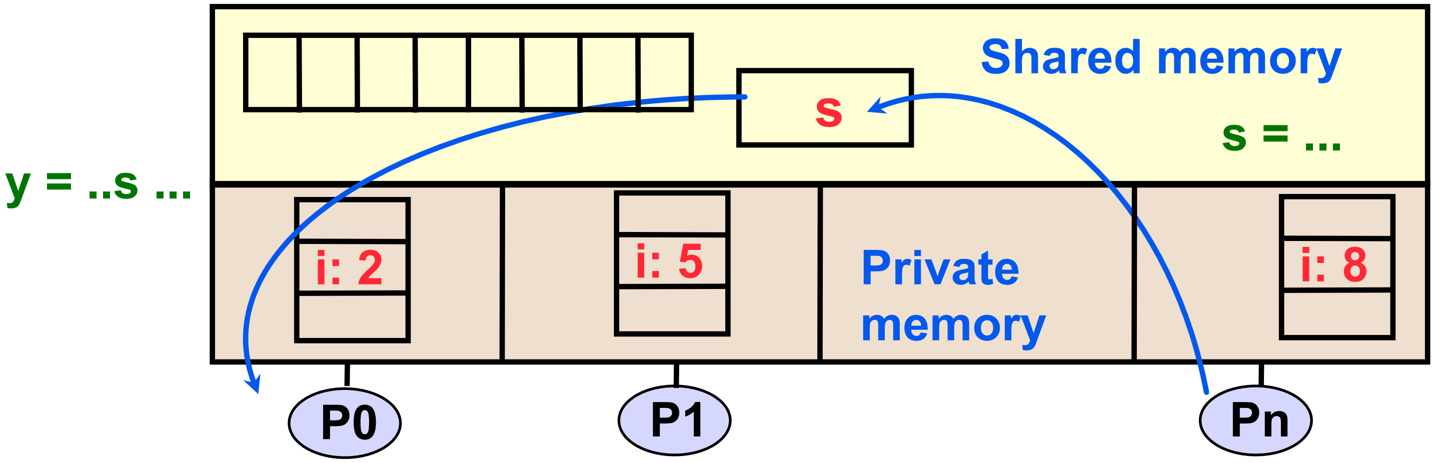
Administrative stuff

- No office hours today (maybe “virtual” only—AIM:**VuducOfficeHours**)
- **Accounts**: Apparently, you already have them or will soon (!)
  - Try logging into ‘warp1’ with your UNIX account password
  - If it doesn’t work, go see TSO Help Desk (and **good luck!**)
    - CCB 148 / M-F 7a-5p / 404.894.7065 / AIM:tsohlpdsk
  - **IHPCL mailing list:**
    - https://mailman.cc.gatech.edu/mailman/listinfo/ihpc-lab

Shared memory programming: POSIX Threads and OpenMP

Programming model 1: Shared memory

- **Program** = collection of **threads** of control
- Each thread has **private** variables
- May access **shared** variables, for communicating implicitly and synchronizing



Shared memory programming

- Libraries for existing languages
  - POSIX Threads (PThreads), Solaris Threads: Portable, low-level library
  - OpenMP: Pragma-based, targets scientific computing apps
  - Intel Thread Building Blocks (TBB): pThreads + OpenMP
- Language extensions

Common notions of thread creation

```
cobegin
   task1 (a1);
   task2 (a2);
coend
```

```
id = fork (task1, a1);
task2 (a2);
join (id);
```

```
v = future (task1 (a1));
…
… = … v …
```

POSIX Threads (PThreads)

- Portable system call interface for creating and synchronizing threads
- Threads share all global variables
- Fork/join style

```
errcode = pthread_create (&thread_id,
                          &thread_attribute,
                          &thread_fun,
                          &fun_arg)
  …
errcode = pthread_join (thread_id, NULL);
```

- *Reference*: https://computing.llnl.gov/tutorials/pthreads/

Loop-level parallelism

- May fork threads at any time, *e.g.*, within a loop

```
… A[n];

for (i = 0; i < n; ++i)
   pthread_create (…, &task, &i);
…
```

- Must have sufficient granularity to mask thread-creation overhead

Low-level policy control

- Detached state: Avoid pthread_join calls
- Scheduling parameters: priority, policy (FIFO vs. round-robin)
- Contention scope: With what thread does this thread compete for CPU

Barriers for global synchronization (Optional extension)

- Usage outline

```
pthread_barrier_t b;
pthread_barrier_init (&b, NULL, 3);  // 3 threads
…
pthread_barrier_wait (&b);   // All threads wait
…
pthread_barrier_destroy (&b);
```

Mutual exclusion locks (mutexes)

- Basic usage

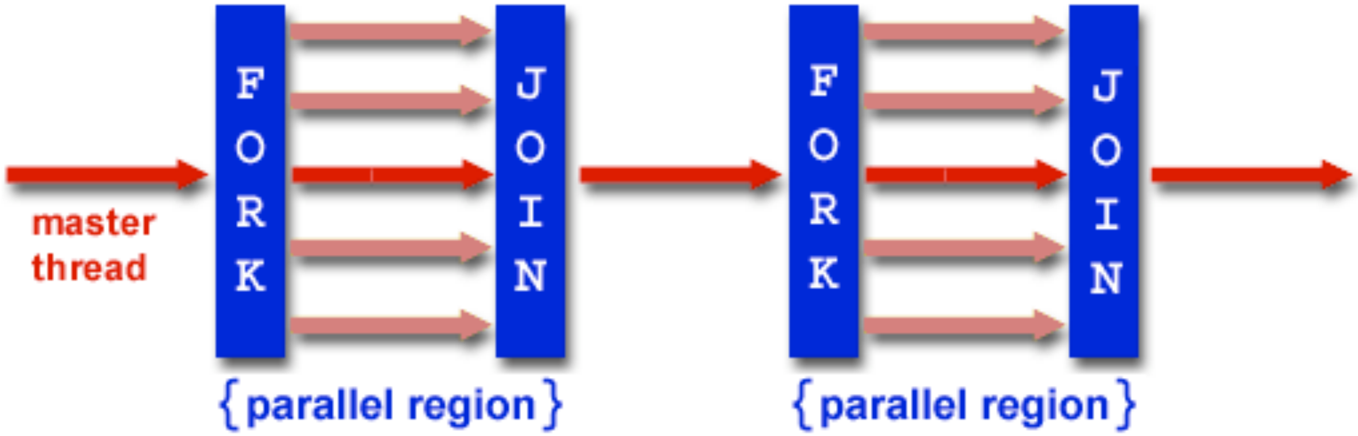
```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init (&lock, NULL);
…
pthread_mutex_lock (&lock);
   // … do critical work …
pthread_mutex_unlock (&lock);
```

- Beware of **deadlock**

Thread 1	Thread 2
lock (a);	**lock** (b);
lock (b);	**lock** (a);

OpenMP: An API for multithreaded shared-memory programming

- Programmer identifies **serial** and **parallel regions**, not threads



- Library + directives (requires compiler support)
- Official website: http://www.openmp.org
  - Also: https://computing.llnl.gov/tutorials/openMP/

Simple example

```
int main()
{



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

   return 0;
}
```

Simple example

```
int main()
{
   omp_set_num_threads (16);

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

Concurrent loops

- May parallelize a loop, but **you must check dependencies**

```
s = 0;
for (i = 0; i < n; ++i)
  s += x[i];
```

```
#pragma omp parallel for \
        shared (s)
for (i = 0; i < n; ++i)
  #pragma omp critical
  s += x[i];
```

```
#pragma omp parallel for \
        reduction(+: s)
for (i = 0; i < n; ++i)
  s += x[i];
```

Loop scheduling

- Use “schedule” clause to partition loop iterations
- **Static**: ***k*** iterations per thread, assigned statically

```
#pragma omp parallel for schedule static(k) …
```

- **Dynamic**: ***k*** iterations per thread, using logical work queue

```
#pragma omp parallel for schedule dynamic(k) …
```

- **Guided**: ***k*** iterations per thread initially, reduced with each allocation

```
#pragma omp parallel for schedule guided(k) …
```

- **Run-time**: Use value of environment variable, **OMP_SCHEDULE**

Synchronization primitives

Critical sections	No explicit locks	`#pragma omp critical`
		`{ … }`

Barriers		`#pragma omp barrier`

Explicit locks	May require flushing	`omp_set_lock (l);`
		…
		`omp_unset_lock (l);`

Single-thread regions	Inside parallel regions	`#pragma omp single`
		`{ /* executed once */ }`

“In conclusion…”

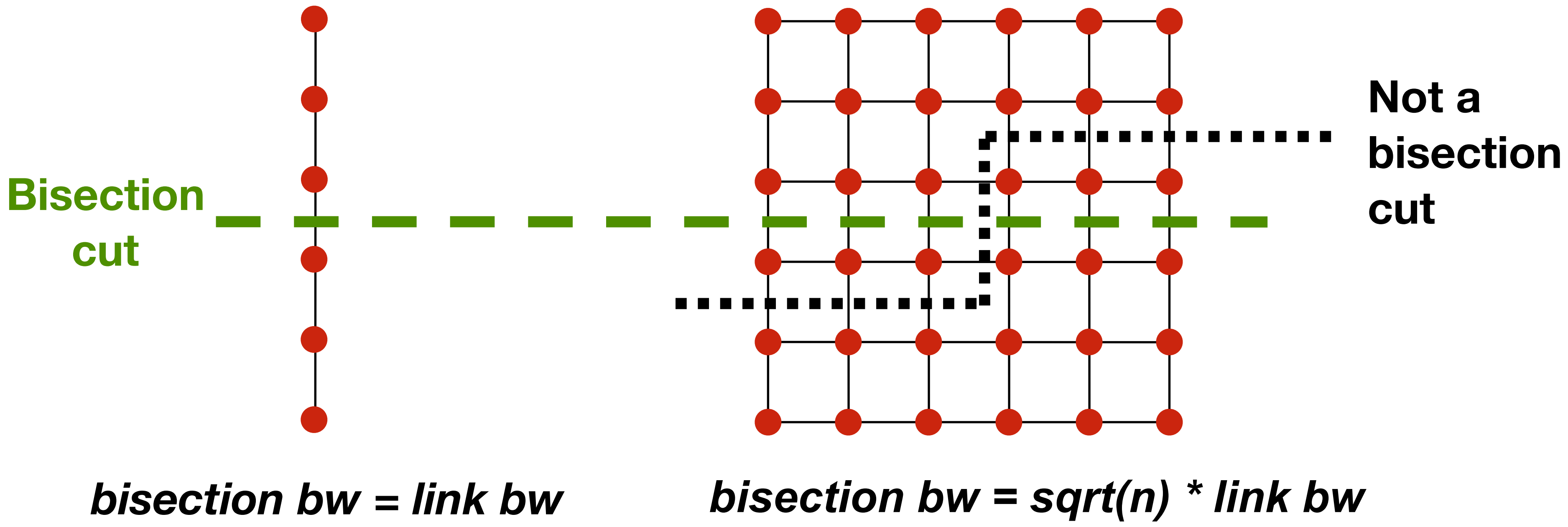
Backup slides

Network topology

- Of great interest historically, particularly in mapping algorithms to networks
  - Key metric: Minimize hops
  - Modern networks hide hop cost, so topology less important
- Large gap in hardware/software latency: On IBM SP, *cf*. 1.5 usec to 36 usec
- Topology affects bisection bandwidth, so still relevant

Bisection bandwidth

- Bandwidth across smallest cut that divides network in two equal halves
- Important for all-to-all communication patterns

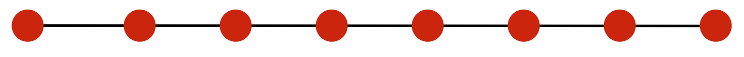


Linear and ring networks

Linear

Diameter ~ *n*/3

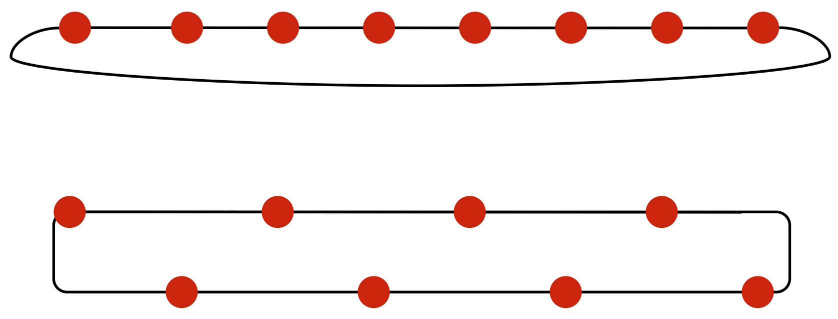
Bisection = 1



Ring/Torus

Diameter ~ *n*/4

Bisection = 2



Multidimensional meshes and tori

2-D mesh

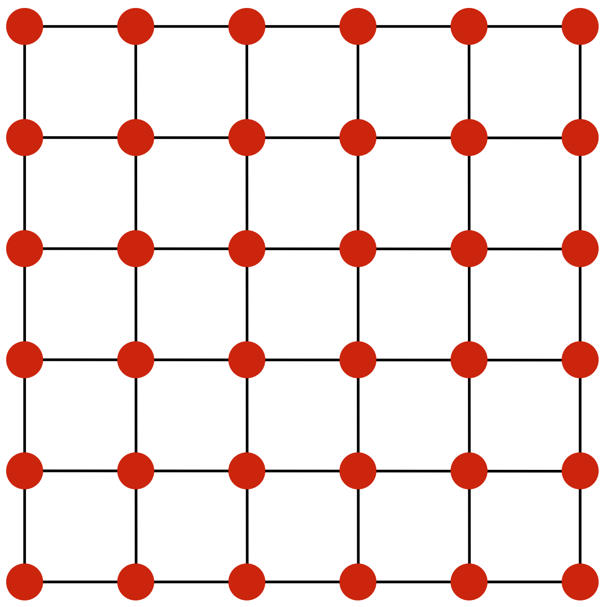
Diameter ~ 2*sqrt(*n*)

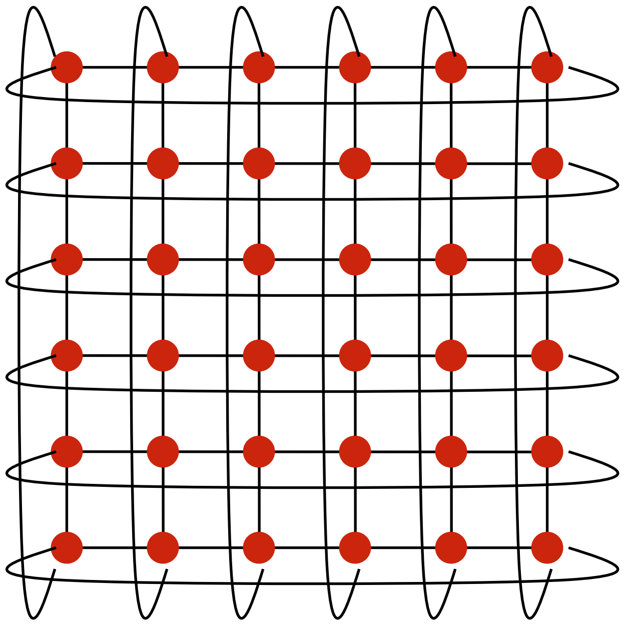
Bisection = sqrt(*n*)

2-D torus

Diameter ~ sqrt(*n*)

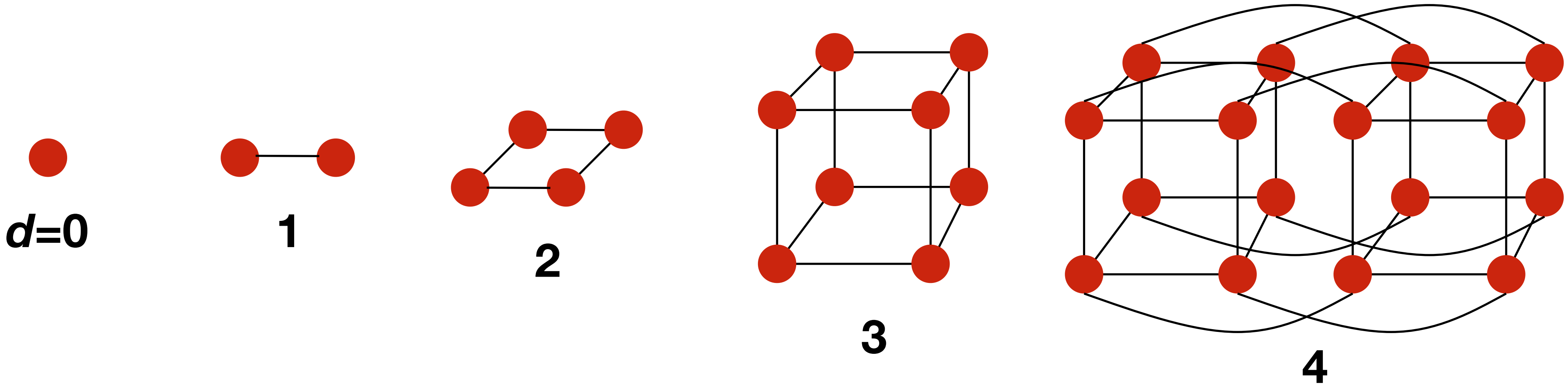
Bisection = 2*sqrt(*n*)





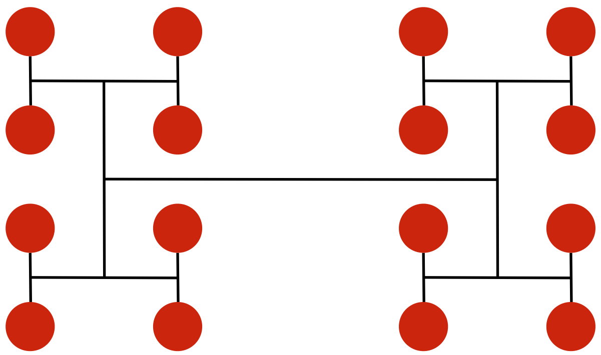
Hypercubes

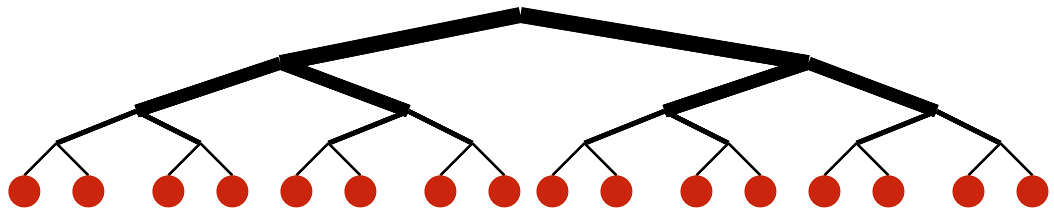
- No. of nodes = 2^d for dimension *d*
  - Diameter = *d*
  - Bisection = *n/2*

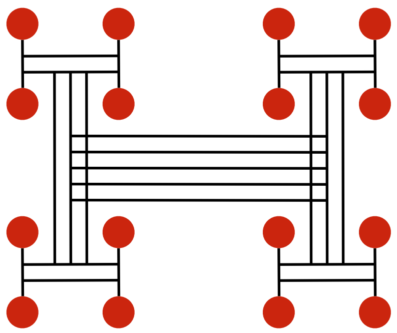


Trees

- Diameter = log *n*
- Bisection bandwidth = 1
- **Fat trees**: Avoid bisection problem using fatter links at top

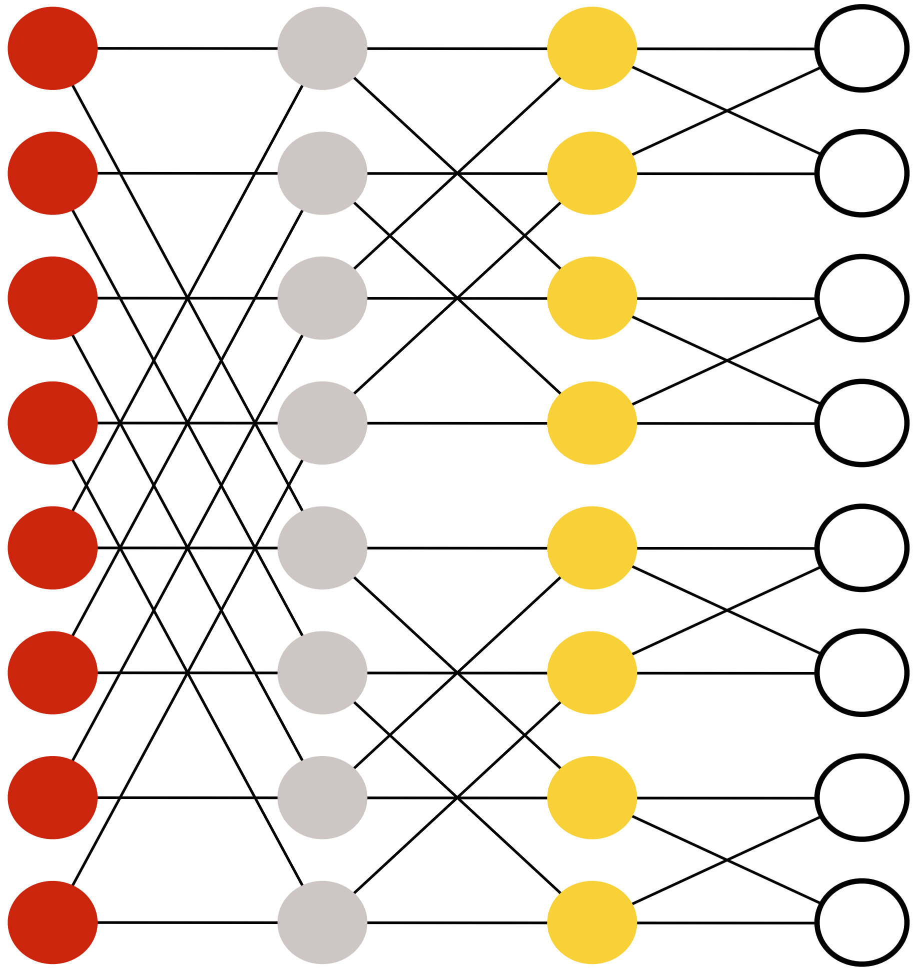






Butterfly networks

- Diameter = log *n*
- Bisection = *n*
- Cost: Wiring



Topologies in real machines

Newer

Older

Machine	Network
Cray XT3, XT4	3D torus
BG/L	3D torus
SGI Altix	Fat tree
Cray X1	4D hypercube*
Millennium (UCB, Myricom)	Arbitrary*
HP Alphaserver (Quadrics)	Fat tree
IBM SP	~ Fat tree
SGI Origin	Hypercube
Intel Paragon	2D mesh
BBN Butterfly	Butterfly

Evolution of distributed memory machine networks

- Message queues replaced by direct memory access (**DMA**)
- **Wormhole** routing: Processor packs/copies, initiates transfer, then goes on
- Message passing libraries provide store-and-forward abstraction
  - May send/receive between any pair of nodes
  - Time proportional to distance since each processor along path participates