Large Scale Multiprocessors and Scientific Applications

By

Pushkar Ratnalikar

Namrata Lele

Agenda

- Introduction
- Interprocessor Communication
- Characteristics of Scientific Applications
- Synchronization: Scaling up
- Performance of Scientific Applications on Shared Memory Multiprocessors
- Performance of Scientific Applications on Distributed Memory Multiprocessors
- Implementing Cache Coherence
- Custom Cluster Approach: Blue Gene/L

Introduction

- Primary application of large-scale multiprocessors is for true parallel programming
- Characteristics of parallel programs:
  - Amount of parallelism
  - Size of parallel tasks
  - Frequency and nature of intertask communication
  - Frequency and nature of synchronization

Interprocessor Communication

- Performance metrics:

1. Communication bandwidth
2. Communication latency = Sender overhead + Time of Flight + Transmission time + Receiver overhead
3. Communication latency hiding

Advantages of Shared Memory Communication Mechanism

- Ease of programming for complex and dynamic communication patterns among processors
- Lower overhead for communication and better use of communication bandwidth when communicating small items
- Ability to use hardware controlled caching to reduce the frequency of remote communication by supporting automatic caching of all data.

Advantages of Message Passing Communication Mechanism

- Hardware can be simpler compared to shared memory
- Communication is explicit which means its simple to understand
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization

Characteristics of Scientific Applications

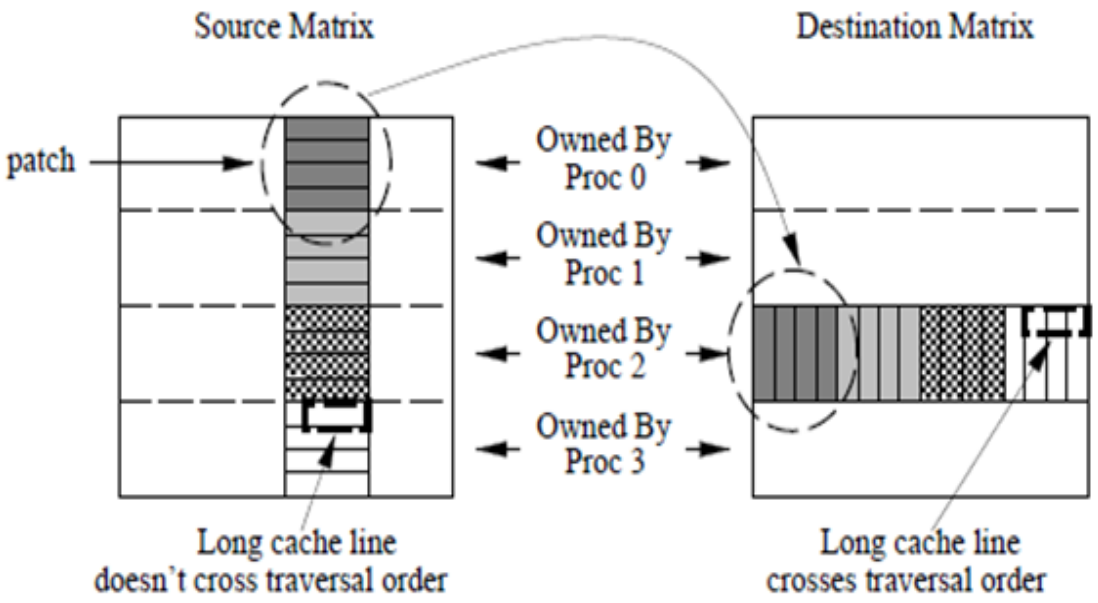
- FFT kernel : Used in fields ranging from signal processing to fluid flow to climate modeling
- LU kernel : LU factorization of a dense matrix
- Barnes Application : Solves a problem in galaxy evolution
- Oceans Application : Simulates the influence of eddy and boundary currents on large scale flow in the ocean

FFT Kernel

The Six–Step FFT Algorithm

1. Transpose data matrix
2. Perform 1–D FFT on each row of data matrix
3. Apply roots of unity to data matrix
4. Transpose data matrix
5. Perform 1–D FFT on each row of data matrix
6. Transpose data matrix

(a) Algorithm Steps



(b) FFT Transpose Phase

FFT Algorithm and Transpose Phase.

Sequential time for n data points O(n log n)

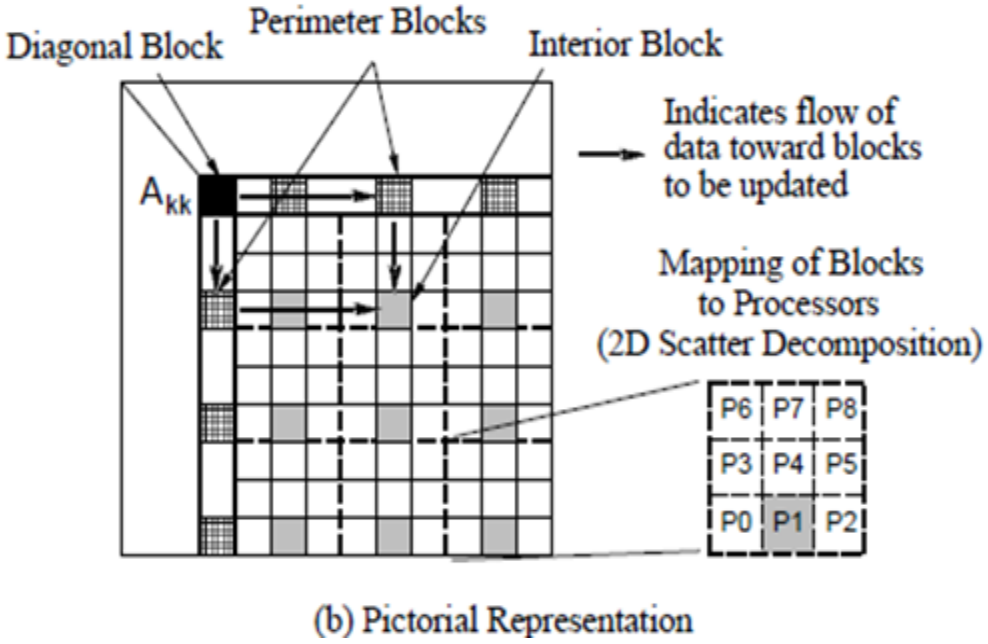
LU Kernel

Blocked Dense LU Factorization

1. For k=0 to N−1 do
2. Factor diagonal block A_{kk}
3. Update all perimeter blocks in column k and row k using A_{kk}
4. For j=k+1 to N−1 do
5. For i=k+1 to N−1 do
6. $A_{ij} = A_{ij} - A_{ik}^{*} A_{kj}$

/* Update interior blocks using corresponding perimeter blocks */

(a) Algorithm Steps

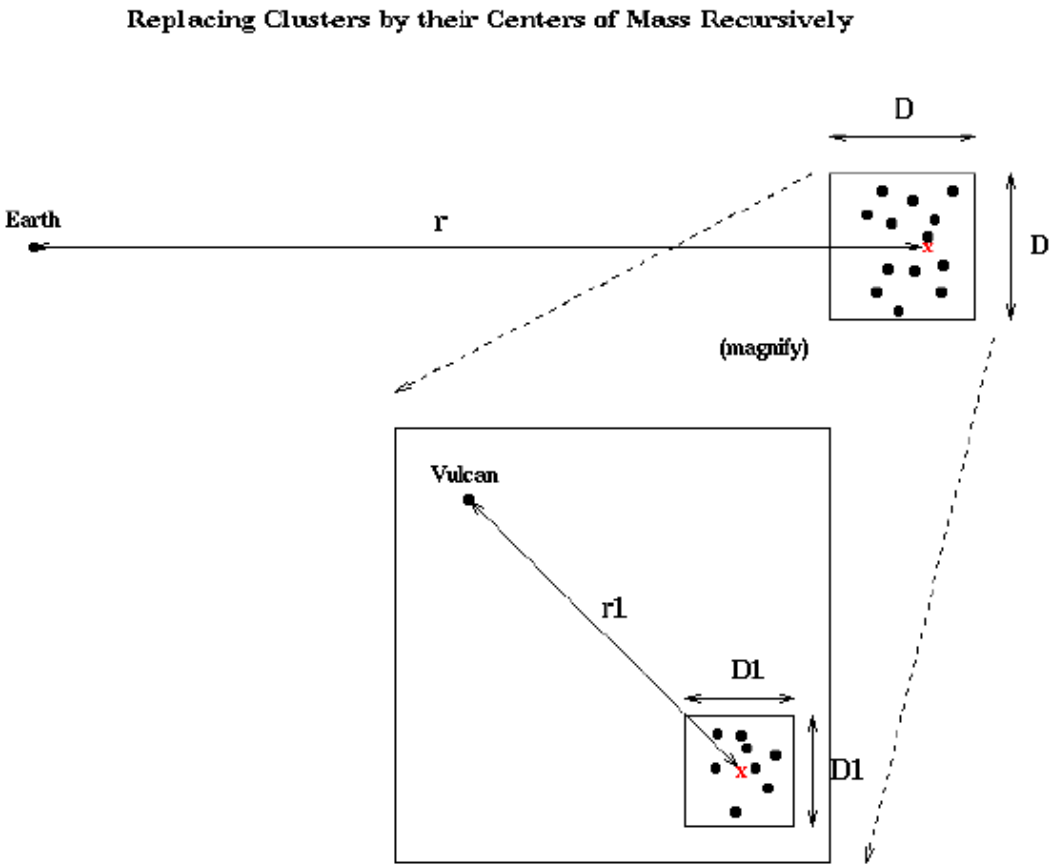


(b) Pictorial Representation

Blocked Dense LU Factorization.

Sequential time for n x n matrix is $o(n^3)$

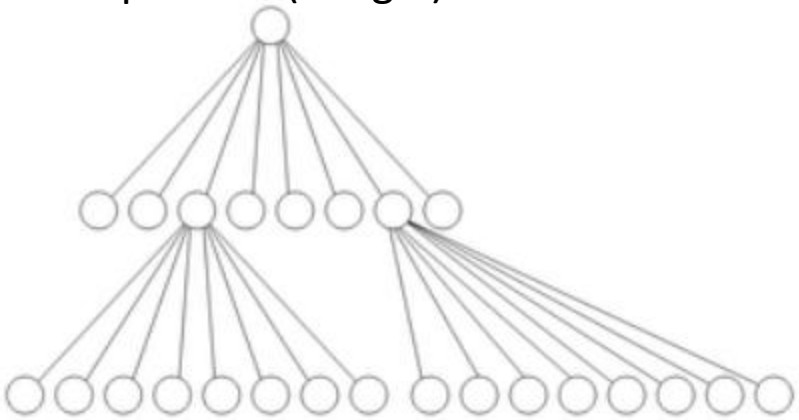
Barnes Application



Applying the idea of force calculation recursively

Barnes Application

Sequential time for n data points O(n log n)



Octree

- Each processor is allocated a subtree
- Size of subtree allocated to a processor is based on some measure of work it has to do(how many other cells it needs to visit) rather than just on the number of nodes in the subtree

Oceans Application

- Red-black Gauss-Seidel colors points in grid to consistently update points based on previous values of adjacent neighbors.
- Each grid in hierarchy has fewer points than the grid below and is an approximation to the lower grid.
- The entire ocean basin is partitioned into square subgrids that are allocated to the portion of the address space corresponding to the local memory of individual processors
- Communication occurs when boundary points of a subgrid are accessed by adjacent subgrid
- Sequential time for nxn grid: $O(n^2)$

Computation/Communication for Parallel Programs

- If the ratio of computation to communication is high, it means the application has lots of computation for each datum communicated.
- Knowing how the ratio changes as we increase the processor count sheds light on how well the application can be sped up.

Computation/Communication for Scientific applications

Application	Scaling of computation	Scaling of communication	Scaling of computation-to-communication
FFT	$\frac{n \log n}{p}$	$\frac{n}{p}$	$\log n$
LU	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$
Barnes	$\frac{n \log n}{p}$	approximately $\frac{\sqrt{n}(\log n)}{\sqrt{p}}$	approximately $\frac{\sqrt{n}}{\sqrt{p}}$
Ocean	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$

Computation/Communication Example: Ocean Application

Processors (p)	Size(n)	Time (min)	Computation	Communication
1	1	58	1	0
2	1	40	½	sqrt(1)/sqrt(2)
32	2	?	2/32 = 1/16	sqrt(2)/sqrt(32) =1/4
128	16	?	16/128 = 1/8	sqrt(16)/sqrt(128) = 1/(2 * sqrt(2))
128	32	?	32/128 = 1/4	sqrt(32)/sqrt(128) = 1/2
256	128	?	128/256 = 1/2	sqrt(128)/sqrt(256) = 1/2

From the table, if we call the base time of computation x, and the base time of communication y then we get (using minutes to be consistent):

$$(1) \qquad 58 = 1x + 0y$$

$$(2) \qquad 40 = .5x + 1/(sqrt(2))y$$

From the first equation, we now know that x=58min. From this, we can solve for y in equation 2, which is y=12 * sqrt(2).

Computation/Communication Example: Ocean Application

From this we can solve for each of the times (Note: $T_{p,n}$ where p is #processors and n is size):

$T_{32,2} = 1/16x + 1/4y$
$= 1/16(58) + 1/4(12*sqrt(2))$
$= 3.625 + 4.24$
$= 8min\ 26sec = \underline{8\ min}$

$T_{128,16} = 1/8x + 1/(2*sqrt(2))y$
$= 1/8(58) + 1/(2*sqrt(2))(12*sqrt(2))$
$= 7.25 + 6$
$= 13\ min\ 25\ sec = \underline{13\ min}$

Processors (p)	Size(n)	Time (min)	Computation	Communication
1	1	58	1	0
2	1	40	½	sqrt(1)/sqrt(2)
32	2	8	2/32 = 1/16	sqrt(2)/sqrt(32) =1/4
128	16	13	16/128 = 1/8	sqrt(16)/sqrt(128) = 1/(2 * sqrt(2))
128	32	23	32/128 = 1/4	sqrt(32)/sqrt(128) = 1/2
256	128	37	128/256 = 1/2	sqrt(128)/sqrt(256) = 1/2

Synchronization: Scaling Up

- Synchronization performance challenges:

  Example: Suppose 10 processors on a bus try to lock a variable simultaneously. Assume each bus transaction read/write miss is 100 clock cycles long. Determine the time required for all 10 processors to acquire the lock, assuming they are all spinning when the lock is released at time 0.

Example continued

```
lockit:   LL      R2,0(R1)      ;load linked
          BNEZ    R2,lockit     ;not available-spin
          DADDUI  R2,R0,#1      ;locked value
          SC      R2,0(R1)      ;store
          BEQZ    R2,lockit     ;branch if store fails
```

When i processes are contending for the lock, they perform the following sequence of actions, each of which generates a bus transaction:

i load linked operations to access the lock

i store conditional operations to try to lock the lock

1 store (to release the lock)

Thus for i processes, there are a total of $2i + 1$ bus transactions.

Thus, for n processes, the total number of bus operations is:

$$\sum_{i=1}^{n} (2i + 1) = n(n + 1) + n = n^2 + 2n$$

For 10 processes there are 120 bus transactions requiring 12,000 clock cycles or 120 clock cycles per lock acquisition!

Barrier Synchronization

- A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes
- A typical implementation of barrier can be done with two spin locks (lock and unlock notation):

  - One to protect a counter that tallies the processes arriving at the barrier

  - One to hold the processes until the last process arrives at the barrier
- Barrier uses the ability to spin on a variable until it satisfies a test; we use the spin(condition) notation

Simple Barrier Code

```
lock (counterlock);/* ensure update atomic */
if (count==0) release=0;/* first=>reset release */
count = count + 1;/* count arrivals */
unlock(counterlock);/* release lock */
if (count==total) {/* all arrived */
        count=0;/* reset counter */
        release=1;/* release processes */
}
else {/* more to come */

        spin (release==1);/* wait for arrivals */
}
```

- total = No of processes that must reach the barrier
- count = Tally of how many processes have reached the barrier
- Lock and unlock are basic spin locks
- Release is used to hold the processes until the last one reaches the barrier

Sense Reversing Barrier Code

```
local_sense =! local_sense; /* toggle local_sense */
lock (counterlock);/* ensure update atomic */
count=count+1;/* count arrivals */
if (count==total) {/* all arrived */
       count=0;/* reset counter */
       release=local_sense;/* release processes */
}
unlock (counterlock);/* unlock */
spin (release==local_sense);/* wait for signal */
}
```

If a process races ahead to the next instance of this barrier while some other processes are still in the barrier, the fast process cannot trap the other processes, since it does not reset the value of release as it did before.

Synchronization Mechanisms for Large-Scale Multiprocessors

- Software Implementations:

  -Spin Lock with exponential back-off

```
            DADDUI   R3,R0,#1     ;R3 = initial delay
lockit:     LL       R2,0(R1)     ;load linked
            BNEZ     R2,lockit    ;not available-spin
            DADDUI   R2,R2,#1     ;get locked value
            SC       R2,0(R1)     ;store conditional
            BNEZ     R2,gotit     ;branch if store succeeds
            DSLL     R3,R3,#1     ;increase delay by factor of 2
            PAUSE    R3           ;delays by value in R3
            J        lockit
gotit:      use data protected by lock
```

  -Queuing Locks: Construct a queue of waiting processors; whenever processor frees up the lock it causes the next processor in the queue to attempt access

Synchronization Mechanisms for Large-Scale Multiprocessors

- Hardware Implementation:

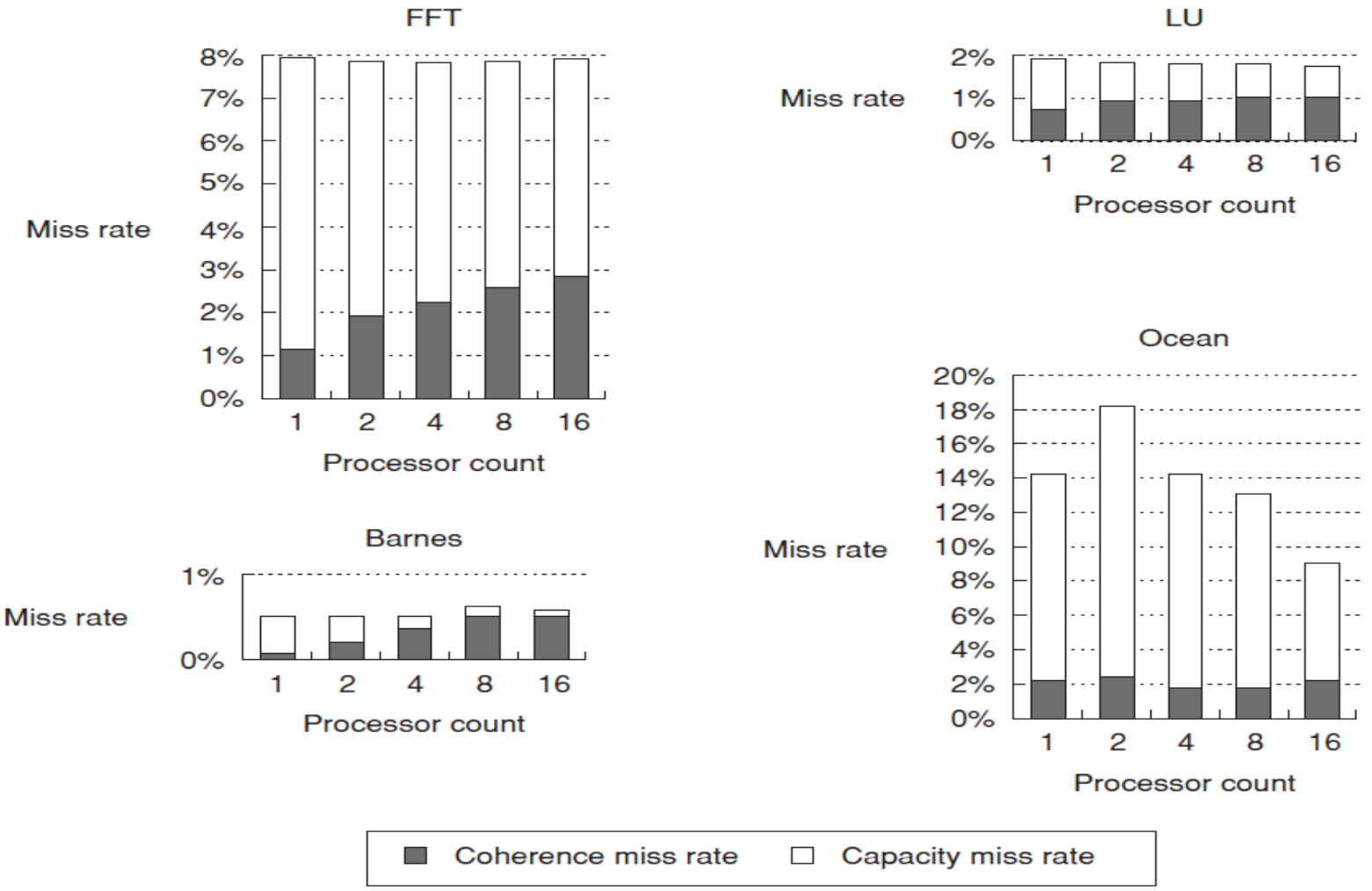
  -Queuing Lock :

  Has a synchronization controller. If the lock is free, it is simply returned to the processor. If the lock is unavailable the controller creates a record of the node's request and sends the processor back a locked value for the variable which the processor then spins on. When lock is freed, controller selects a processor to go ahead from the list of waiting processors. It can then either update the lock variable in the selected processors cache or invalidate the copy, causing a miss and then fetch the available copy of the lock.

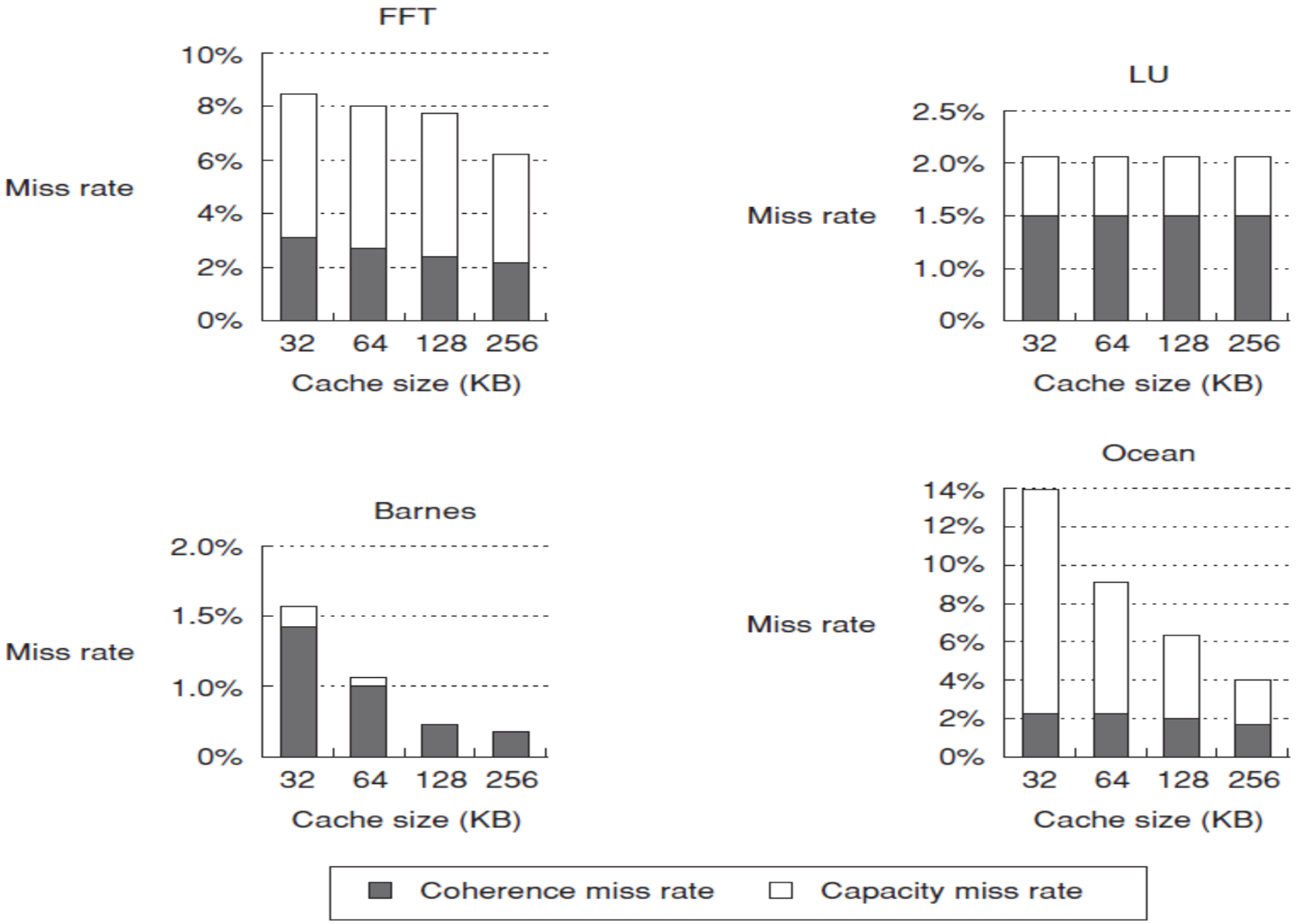
Performance of Scientific Workload on Shared-Memory Multiprocessors

- Variables
  - Processor Count
  - Cache Size
  - Block Size
- Metrics
  - Coherence Misses
  - Uniprocessor Misses
    - Capacity Misses
    - Conflict Misses
    - Compulsory Misses

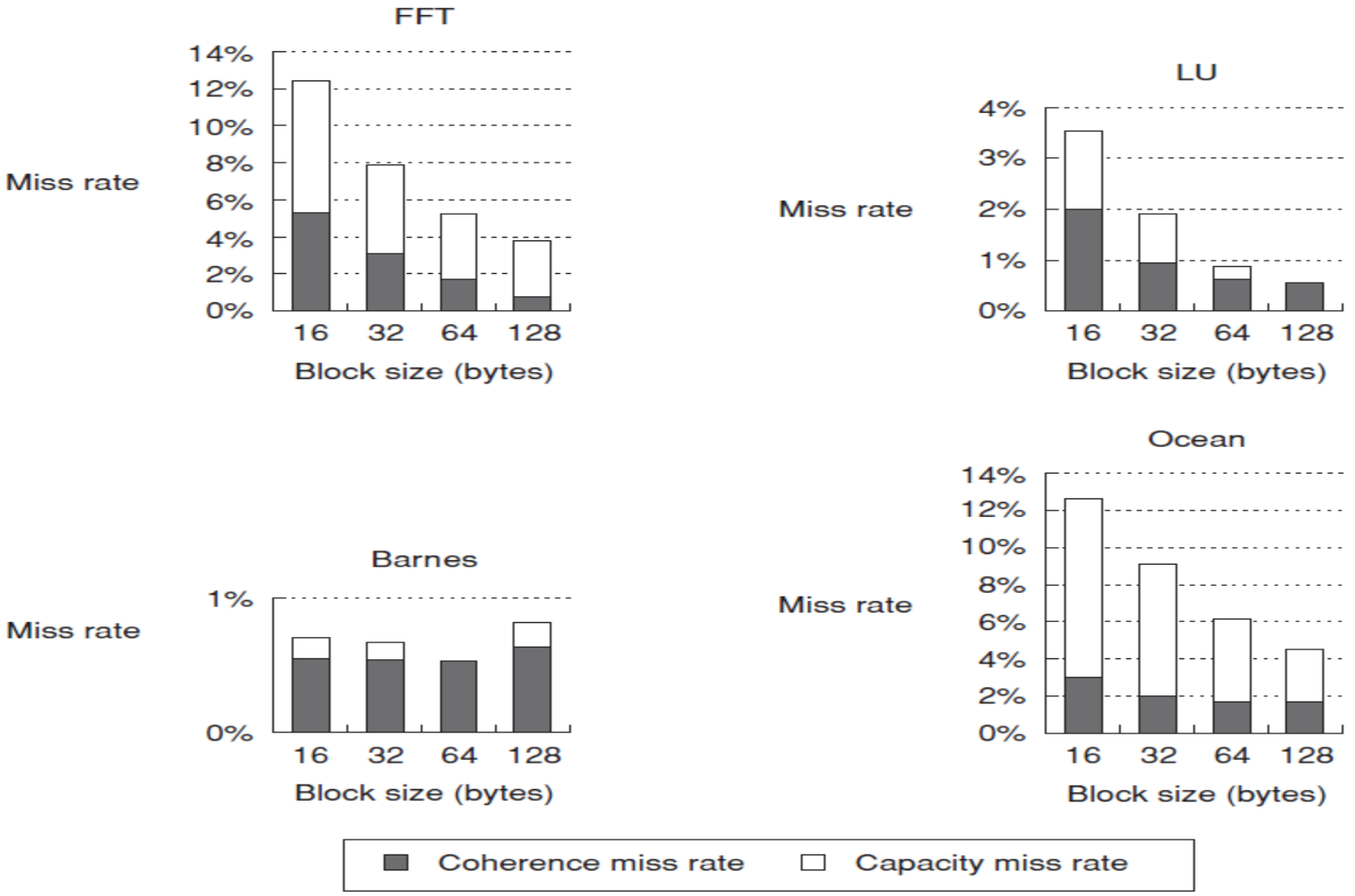
Varying Processor Count



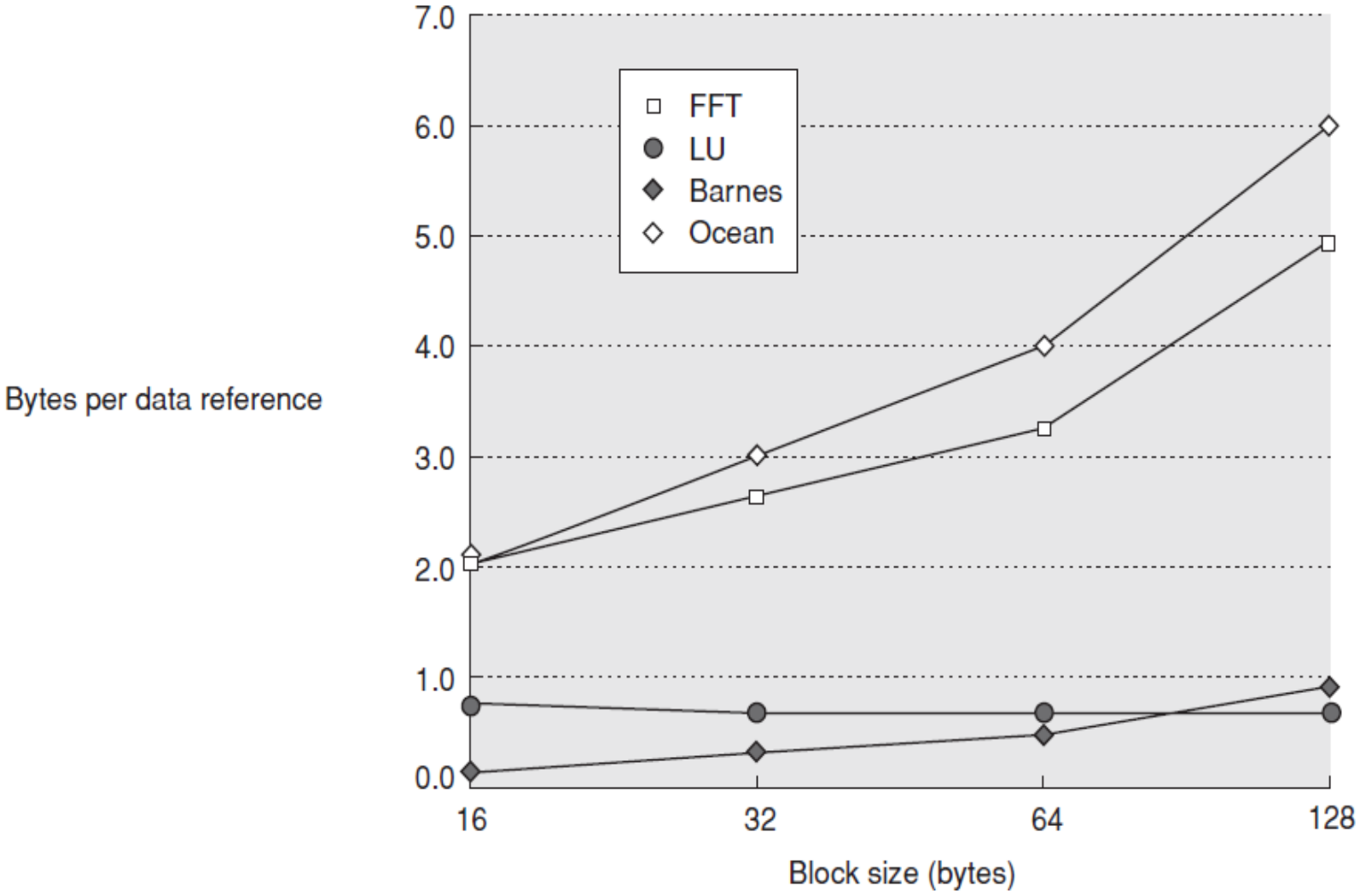
Varying Cache Size



Varying Block Size



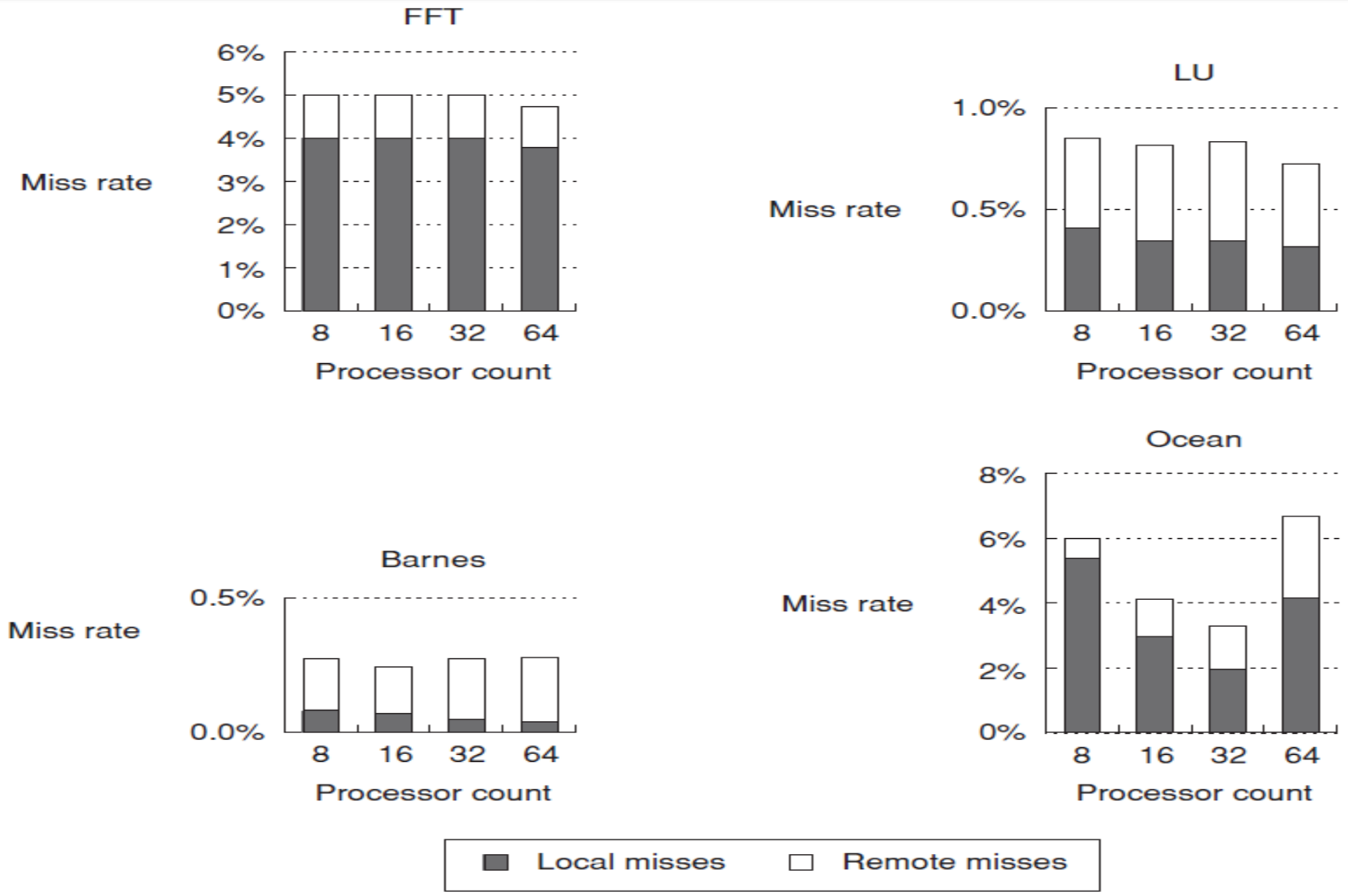
Bus Traffic for data misses



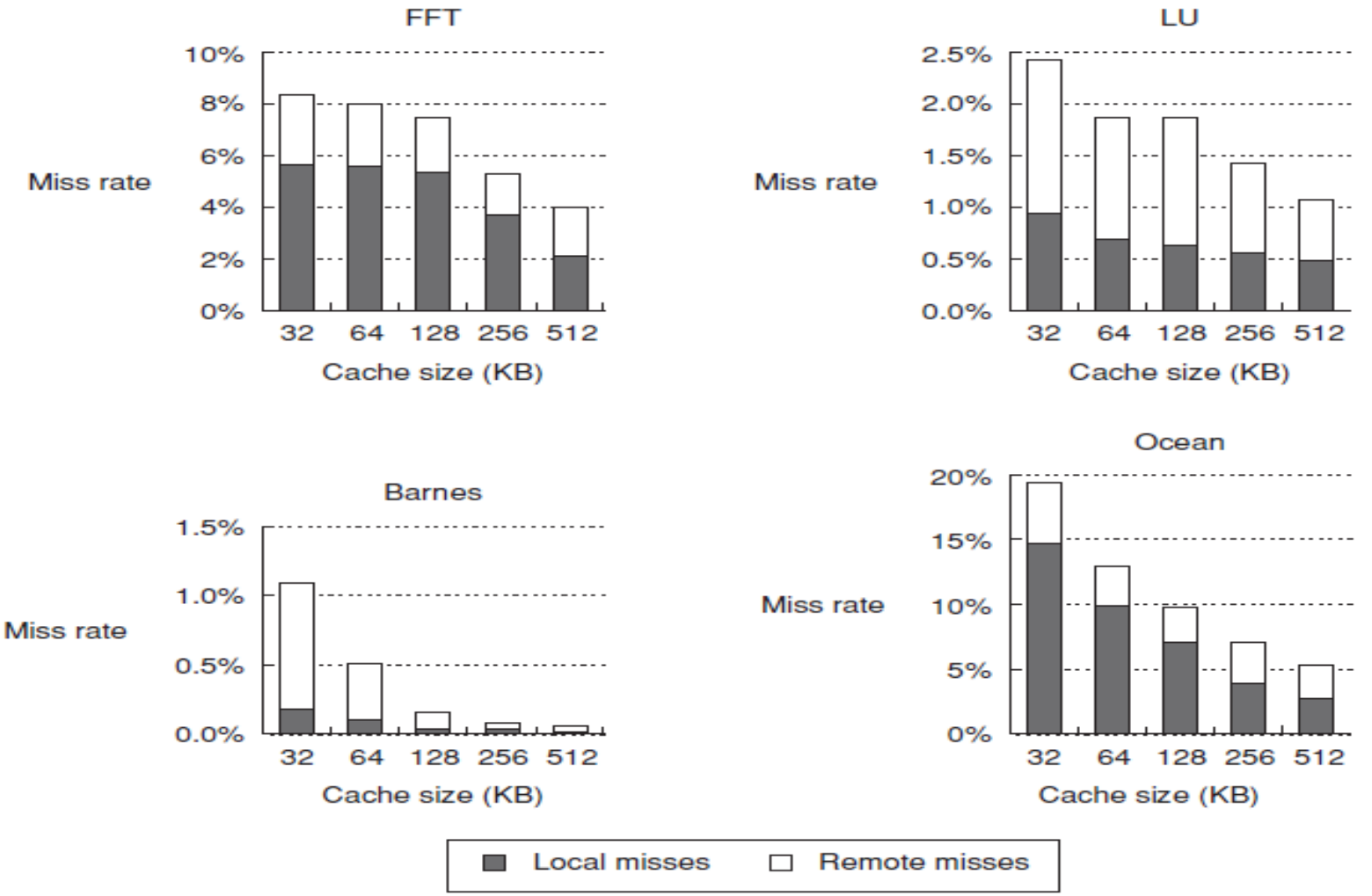
Performance of Scientific Workload on Distributed-Memory Multiprocessors

- Variables
  - Processor Count
  - Cache Size
  - Block Size
- Metrics
  - Local Misses
  - Remote Misses

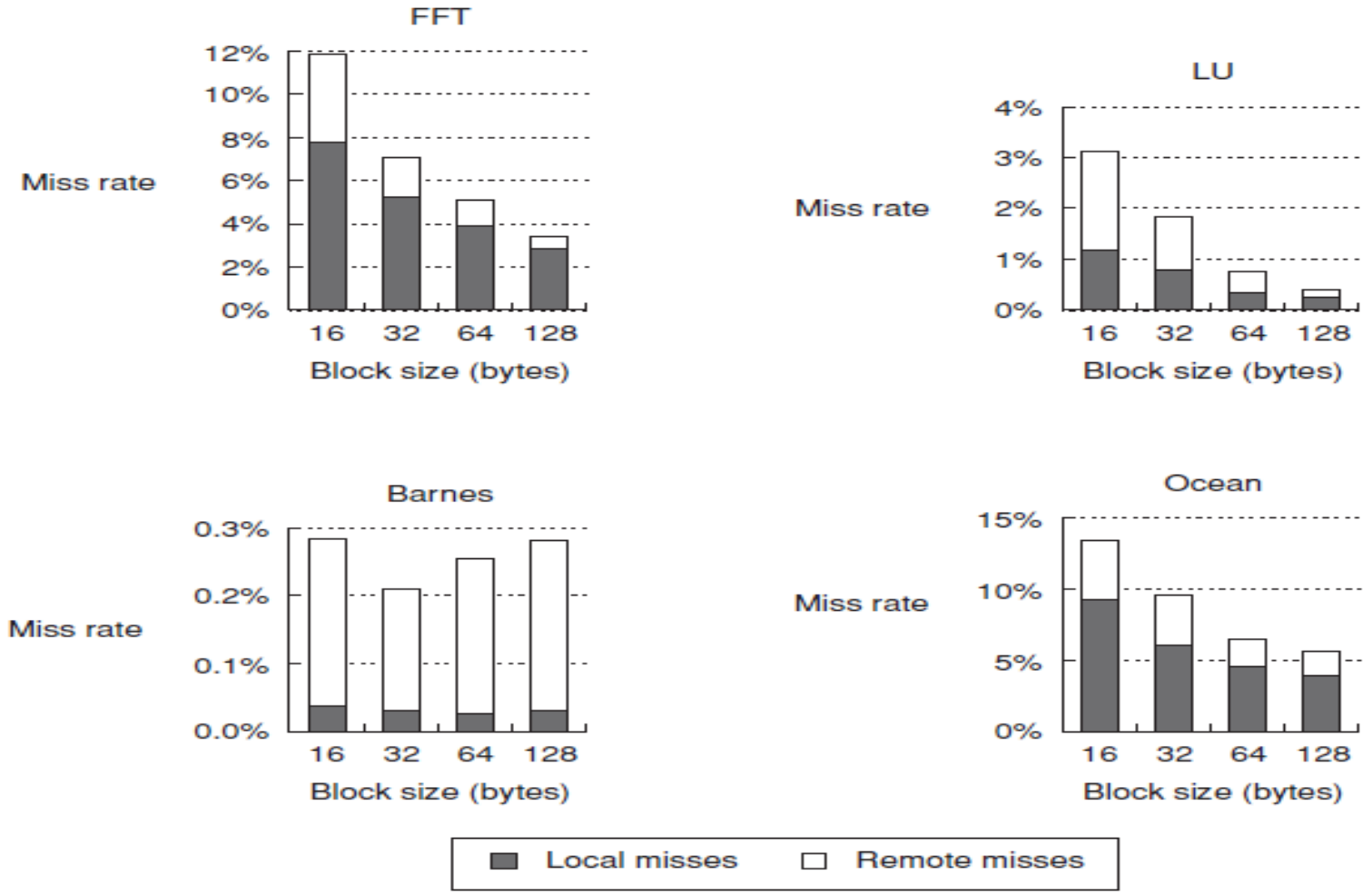
Varying Processor Count



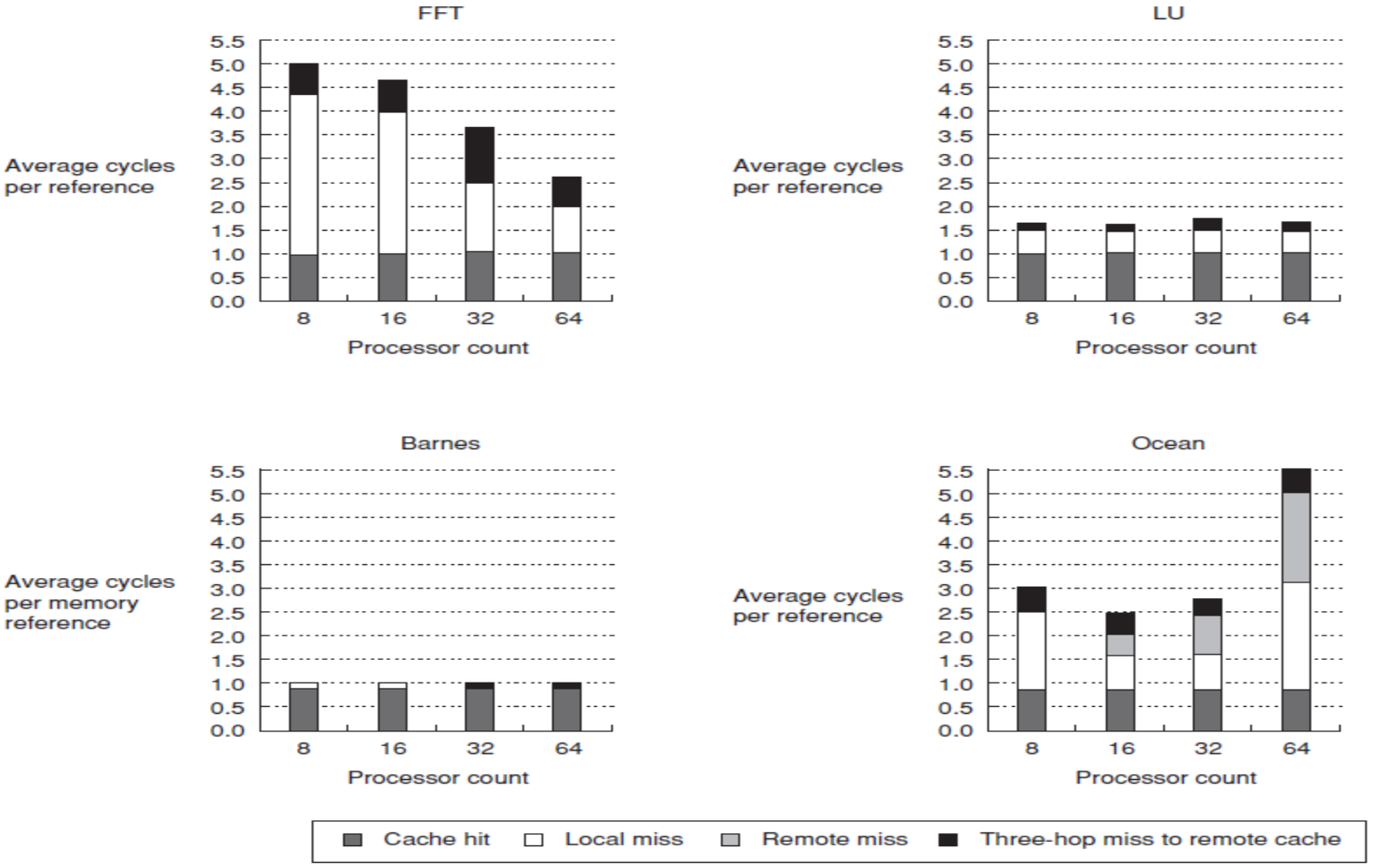
Varying Cache Size



Varying Block Size

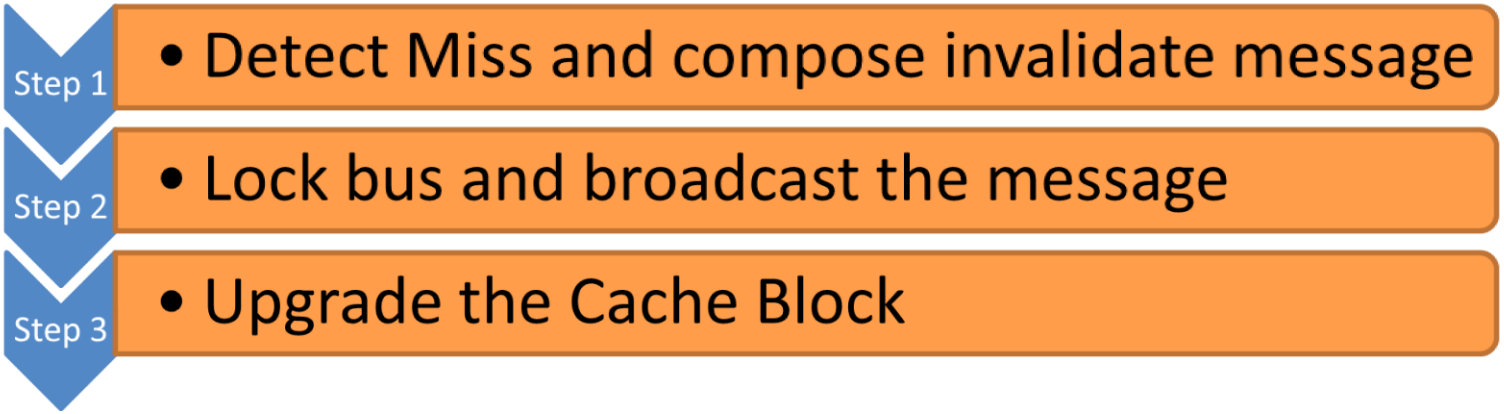


Effective Latency



Implementing Cache Coherence

- Snoopy protocol(handling of upgrade miss)



- Contention to send invalidate message !
- e.g. processors P1 and P2 are attempting to upgrade the same cache block at the same time!

Implementing Cache Coherence

- Race winner is decided by the ordering imposed by the medium.
- What about the looser?
- Need to impose the condition that the looser(or any other processor contending) must handle any pending invalidates before it can generate its own invalidate i.e. P2 should handle any pending invalidates.

Implementing Cache Coherence in DSM-M's

- No broadcast medium
- Need to overcome problems related to non-atomic actions in snoopy protocols(W/O broadcast medium).
- Write requests are easily serviced as the unique directory which holds the block processes requests and informs the requester about success
- NAK is sent to the loser's in the race. This causes the loser to regenerate the request.

Implementing Directory Controller

- Need to have the same capabilities as that in the snoopy case
- Need to handle requests for independent blocks while awaiting response to request from a local processor
- Process requests in order.
- Directory should be multithreaded i.e. handle requests for multiple blocks independently.

Implementing Directory Controller

- Directory controller should be reentrant i.e. capable of suspending its execution while waiting for reply and accept another transaction.
- The major implementation difficulty is to handle NAK's(keep track of outstanding transactions and the corresponding NAK's)
- Usually there is a reply slot for each request. It can hold ACK or a NAK.

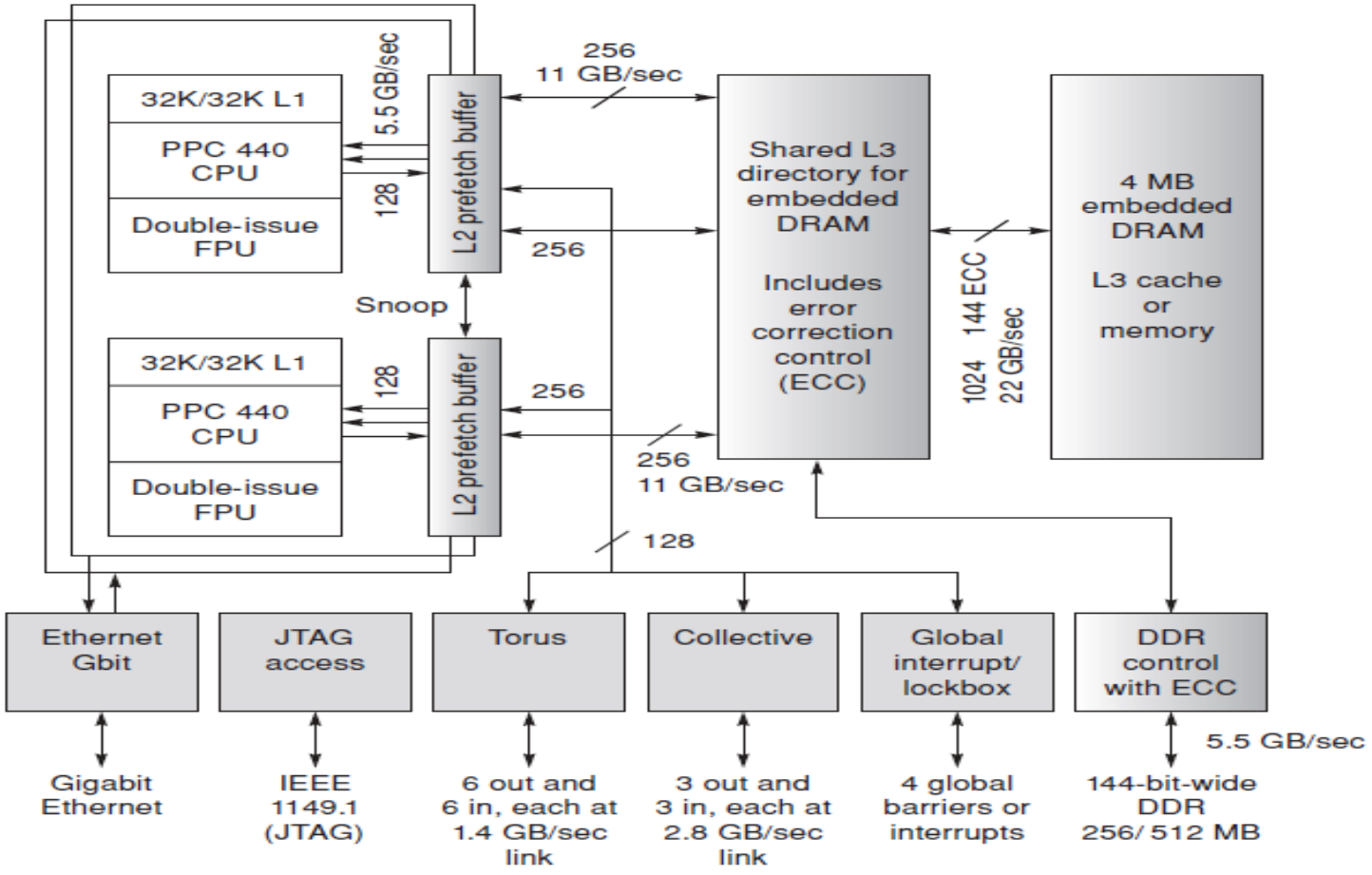
Blue Gene/L

- Scalable distributed memory message-passing supercomputer.
- Focus on Power consumption. Has the highest throughput/cubic foot.
- The first Super Computer to give 100 TFLOPS sustained on real-world applications like 3-D molecular dynamics code, simulating solidification of molten metal under high pressure and temperature conditions.

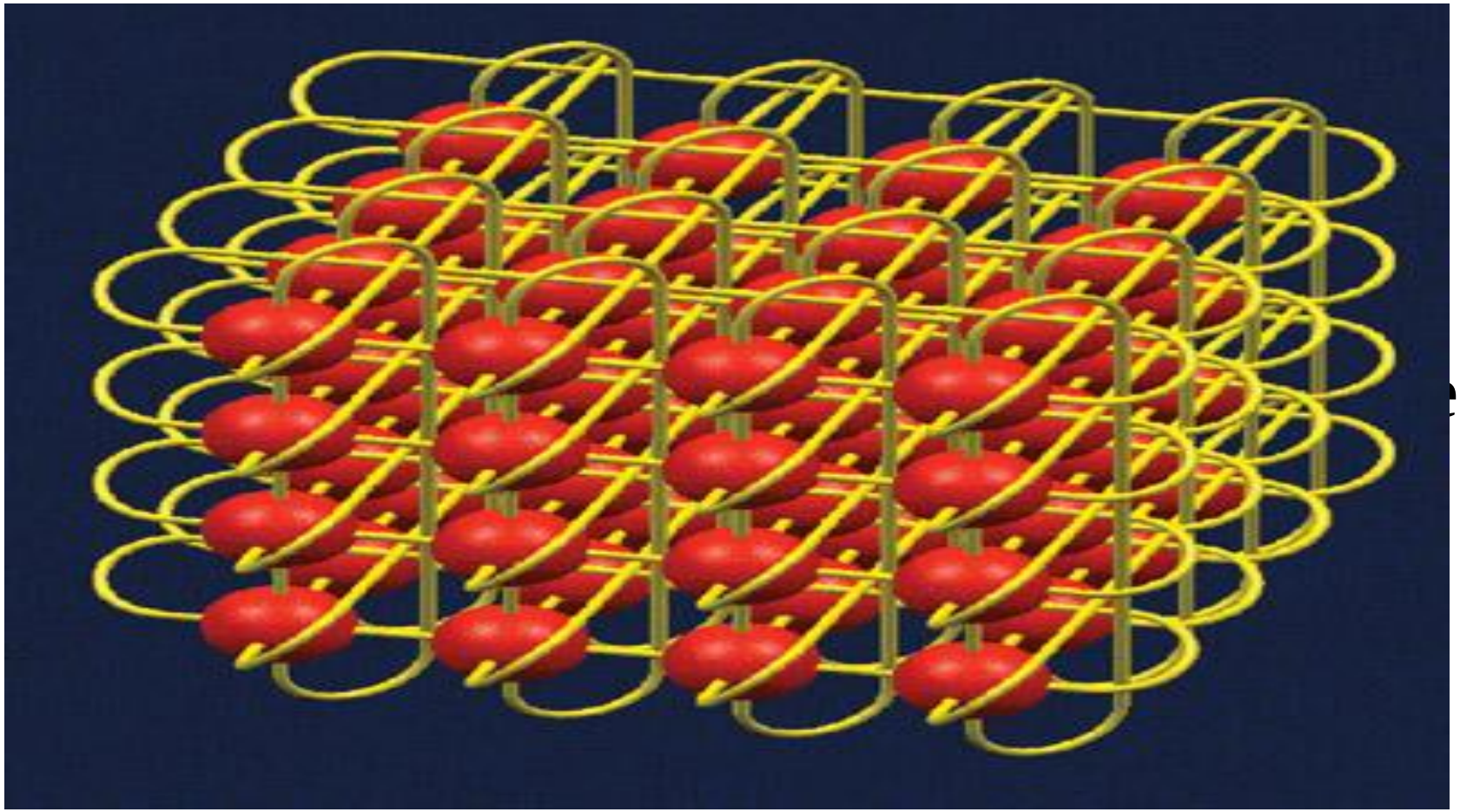
Blue Gene/L node

- Customized processing node, each containing two PowerPC 4000 chips.
- Each modestly clocked at 700 MHz(to reduce power consumption)
- 2-issue superscalar and 7 stage pipeline.
- Consists of up to 64 K nodes, organized in 32 racks each taking approx. 50 cubic feet.

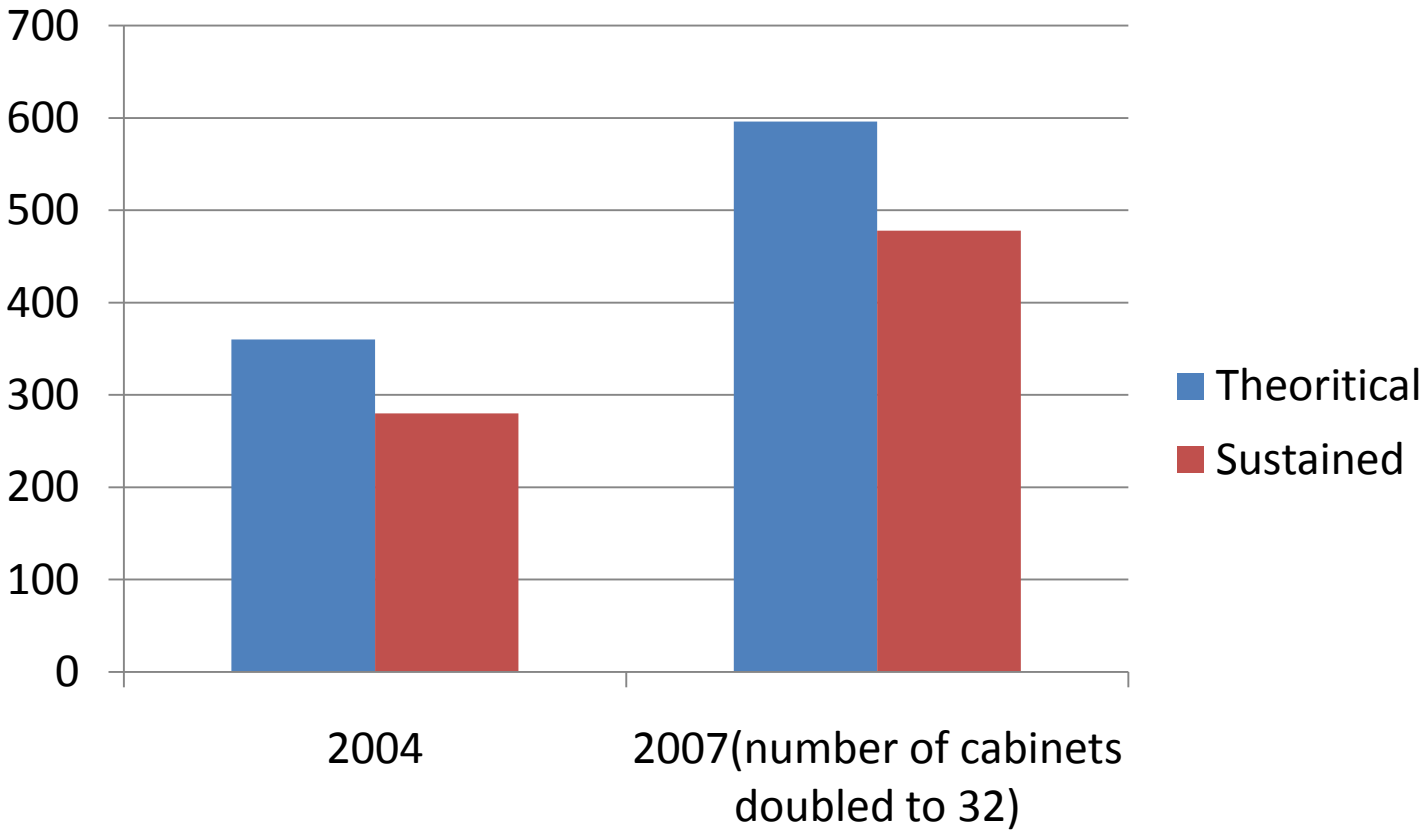
Blue Gene/L node



Blue Gene/L inter-connection network



Blue Gene/L Performance on LINPACK



References

- Computer Architecture, A Quantitative Approach, 4th edition, J.L. Hennessy and D.A. Patterson: Appendix H
- Into Wide Blue Yonder with BlueGene/L (https://www.llnl.gov/str/April05/Seager.html)
- BlueGene/L torus interconnection network(http://www.research.ibm.com/journal/rd/492/adiga.html)
- The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors By-Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy https://eprints.kfupm.edu.sa/70813/1/70813.pdf
- CSUSB, K. E. Schubert http://ftp.csci.csusb.edu/schubert/tutorials/csci610/w05/DTD_Ocean.pdf
- http://charm.cs.uiuc.edu/~bhatele/academics/uiuc/cs498lvk_report_bhatele.pdf

Thank You!