Appendix I
Large-Scale Multiprocessors and 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	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$

Figure I.1 Scaling of computation, of communication, and of the ratio are critical factors in determining performance on parallel multiprocessors. In this table, *p* is the increased processor count and *n* is the increased dataset size. Scaling is on a per-processor basis. The computation scales up with *n* at the rate given by *O*( ) analysis and scales down linearly as *p* is increased. Communication scaling is more complex. In FFT, all data points must interact, so communication increases with *n* and decreases with *p*. In LU and Ocean, communication is proportional to the boundary of a block, so it scales with dataset size at a rate proportional to the side of a square with *n* points, namely, $\sqrt{n}$; for the same reason communication in these two applications scales inversely to $\sqrt{p}$. Barnes has the most complex scaling properties. Because of the fall-off of interaction between bodies, the basic number of interactions among bodies that require communication scales as $\sqrt{n}$. An additional factor of log *n* is needed to maintain the relationships among the bodies. As processor count is increased, communication scales inversely to $\sqrt{p}$.

```
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 */
}
```

Figure I.2 Code for a simple barrier. The lock `counterlock` protects the counter so that it can be atomically incremented. The variable `count` keeps the tally of how many processes have reached the barrier. The variable release is used to hold the processes until the last one reaches the barrier. The operation `spin(release==1)` causes a process to wait until all processes reach the barrier.

```
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 */
}
```

Figure I.3 Code for a sense-reversing barrier. The key to making the barrier reusable is the use of an alternating pattern of values for the flag `release`, which controls the exit from the barrier. 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 in Figure I.2.

Event	Number of times for process *i*	Corresponding source line	Comment
LL `counterlock`	*i*	`lock (counterlock);`	All processes try for lock.
Store conditional	*i*	`lock (counterlock);`	All processes try for lock.
LD `count`	1	`count = count + 1;`	Successful process.
Load linked	$i-1$	`lock (counterlock);`	Unsuccessful process; try again.
SD `count`	1	`count = count + 1;`	Miss to get exclusive access.
SD `counterlock`	1	`unlock(counterlock);`	Miss to get the lock.
LD `release`	2	`spin (release==local_sense);/`	Read release: misses initially and when finally written.

Figure I.4 Here are the actions, which require a bus transaction, taken when the *i*th process reaches the barrier. The last process to reach the barrier requires one less bus transaction, since its read of release for the spin will hit in the cache!

```
          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
```

Figure I.5 A spin lock with exponential back-off. When the store conditional fails, the process delays itself by the value in R3. The delay can be implemented by decrementing a copy of the value in R3 until it reaches 0. The exact timing of the delay is multiprocessor dependent, although it should start with a value that is approximately the time to perform the critical section and release the lock. The statement pause R3 should cause a delay of R3 of these time units. The value in R3 is increased by a factor of 2 every time the store conditional fails, which causes the process to wait twice as long before trying to acquire the lock again. The small variations in the rate at which competing processors execute instructions are usually sufficient to ensure that processes will not continually collide. If the natural perturbation in execution time was insufficient, R3 could be initialized with a small random value, increasing the variance in the successive delays and reducing the probability of successive collisions.

```
struct node{/* a node in the combining tree */
      int counterlock; /* lock for this node */
      int count; /* counter for this node */
      int parent; /* parent in the tree = 0..P-1 except for root */
};
struct node tree [0..P-1]; /* the tree of nodes */
int local_sense; /* private per processor */
int release; /* global release flag */

/* function to implement barrier */
barrier (int mynode, int local_sense) {
      lock (tree[mynode].counterlock); /* protect count */
      tree[mynode].count=tree[mynode].count+1;
             /* increment count */
      if (tree[mynode].count==k) {/* all arrived at mynode */
             if (tree[mynode].parent >=0) {
                    barrier(tree[mynode].parent);
             } else{
                    release = local_sense;
             };
             tree[mynode].count = 0; /* reset for the next time */
      unlock (tree[mynode].counterlock); /* unlock */
      spin (release==local_sense); /* wait */
};
/* code executed by a processor to join barrier */
local_sense =! local_sense;
barrier (mynode);
```

Figure I.6 An implementation of a tree-based barrier reduces contention considerably. The tree is assumed to be prebuilt statically using the nodes in the array tree. Each node in the tree combines *k* processes and provides a separate counter and lock, so that at most *k* processes contend at each node. When the *k*th process reaches a node in the tree, it goes up to the parent, incrementing the count at the parent. When the count in the parent node reaches *k*, the release flag is set. The count in each node is reset by the last process to arrive. Sense-reversing is used to avoid races as in the simple barrier. The value of `tree[root].parent` should be set to − 1 when the tree is initially built.

```
local_sense =! local_sense; /* toggle local_sense */
fetch_and_increment(count);/* atomic update */
if (count==total) {/* all arrived */
        count=0;/* reset counter */
        release=local_sense;/* release processes */
}
else {/* more to come */
        spin (release==local_sense);/* wait for signal */
}
```

Figure I.7 Code for a sense-reversing barrier using fetch-and-increment to do the counting.

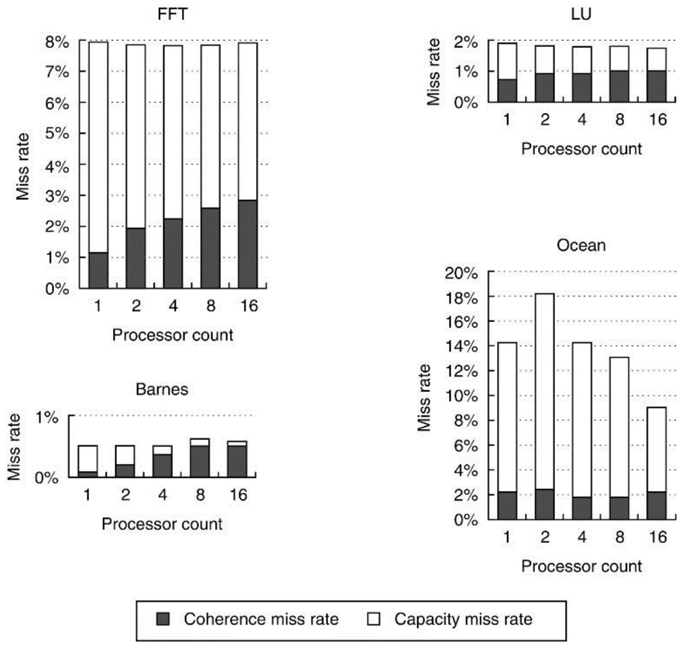


Figure I.8 Data miss rates can vary in nonobvious ways as the processor count is increased from 1 to 16. The miss rates include both coherence and capacity miss rates. The compulsory misses in these benchmarks are all very small and are included in the capacity misses. Most of the misses in these applications are generated by accesses to data that are potentially shared, although in the applications with larger miss rates (FFT and Ocean), it is the capacity misses rather than the coherence misses that comprise the majority of the miss rate. Data are potentially shared if they are allocated in a portion of the address space used for shared data. In all except Ocean, the potentially shared data are heavily shared, while in Ocean only the boundaries of the subgrids are actually shared, although the entire grid is treated as a potentially shared data object. Of course, since the boundaries change as we increase the processor count (for a fixed-size problem), different amounts of the grid become shared. The anomalous increase in capacity miss rate for Ocean in moving from 1 to 2 processors arises because of conflict misses in accessing the subgrids. In all cases except Ocean, the fraction of the cache misses caused by coherence transactions rises when a fixed-size problem is run on an increasing number of processors. In Ocean, the coherence misses initially fall as we add processors due to a large number of misses that are write ownership misses to data that are potentially, but not actually, shared. As the subgrids begin to fit in the aggregate cache (around 16 processors), this effect lessens. The single-processor numbers include write upgrade misses, which occur in this protocol even if the data are not actually shared, since they are in the shared state. For all these runs, the cache size is 64 KB, two-way set associative, with 32-byte blocks. Notice that the scale on the *y*-axis for each benchmark is different, so that the behavior of the individual benchmarks can be seen clearly.

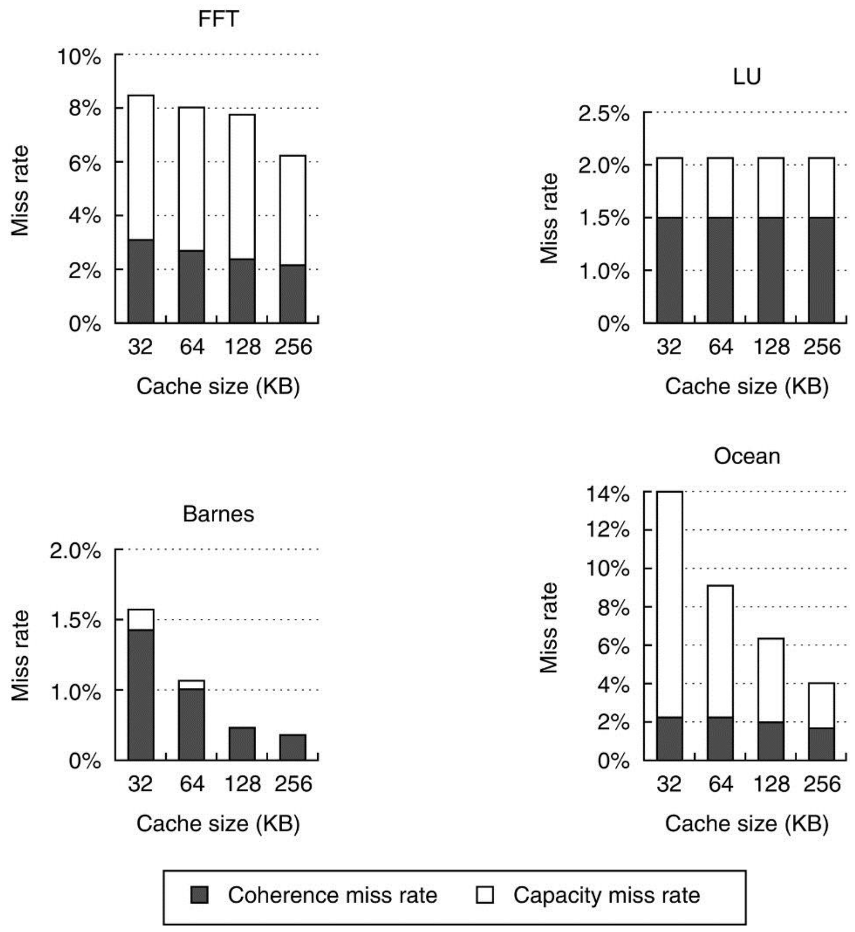


Figure I.9 The miss rate usually drops as the cache size is increased, although coherence misses dampen the effect. The block size is 32 bytes and the cache is two-way set associative. The processor count is fixed at 16 processors. Observe that the scale for each graph is different.

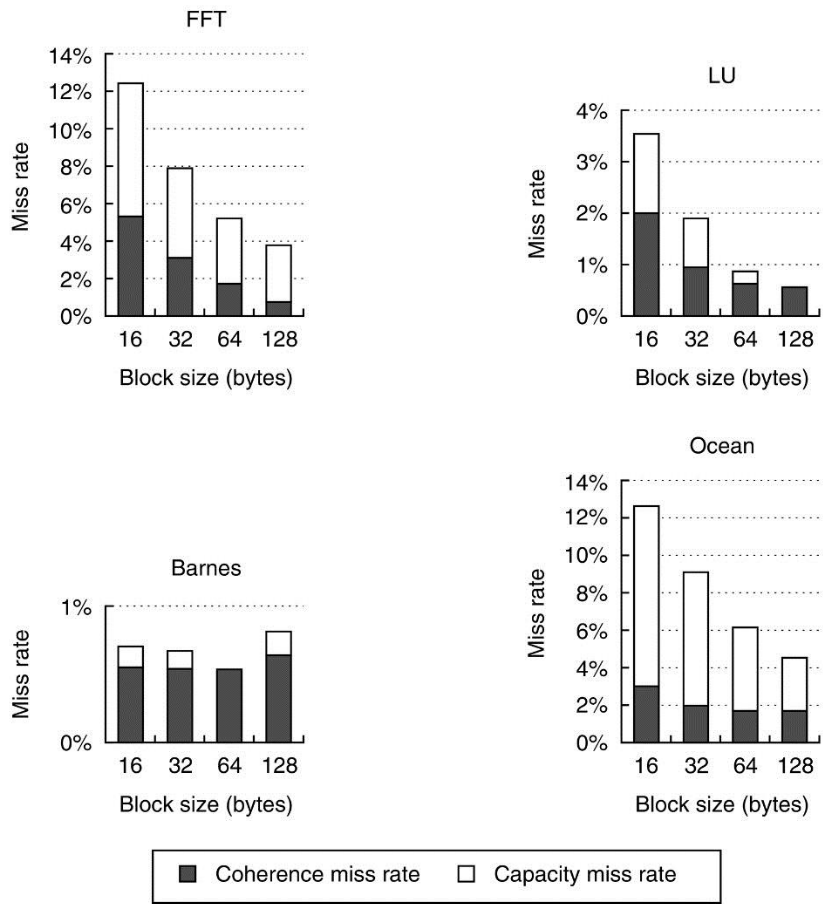


Figure I.10 The data miss rate drops as the cache block size is increased. All these results are for a 16-processor run with a 64 KB cache and two-way set associativity. Once again we use different scales for each benchmark.

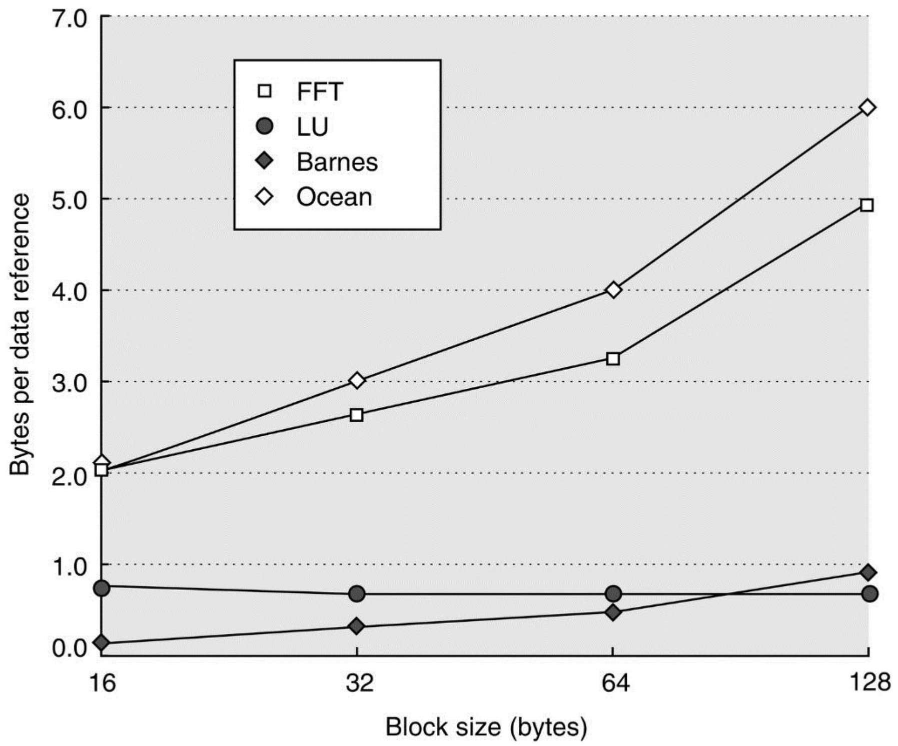


Figure I.11 Bus traffic for data misses climbs steadily as the block size in the data cache is increased. The factor of 3 increase in traffic for Ocean is the best argument against larger block sizes. Remember that our protocol treats ownership or upgrade misses the same as other misses, slightly increasing the penalty for large cache blocks; in both Ocean and FFT, this simplification accounts for less than 10% of the traffic.

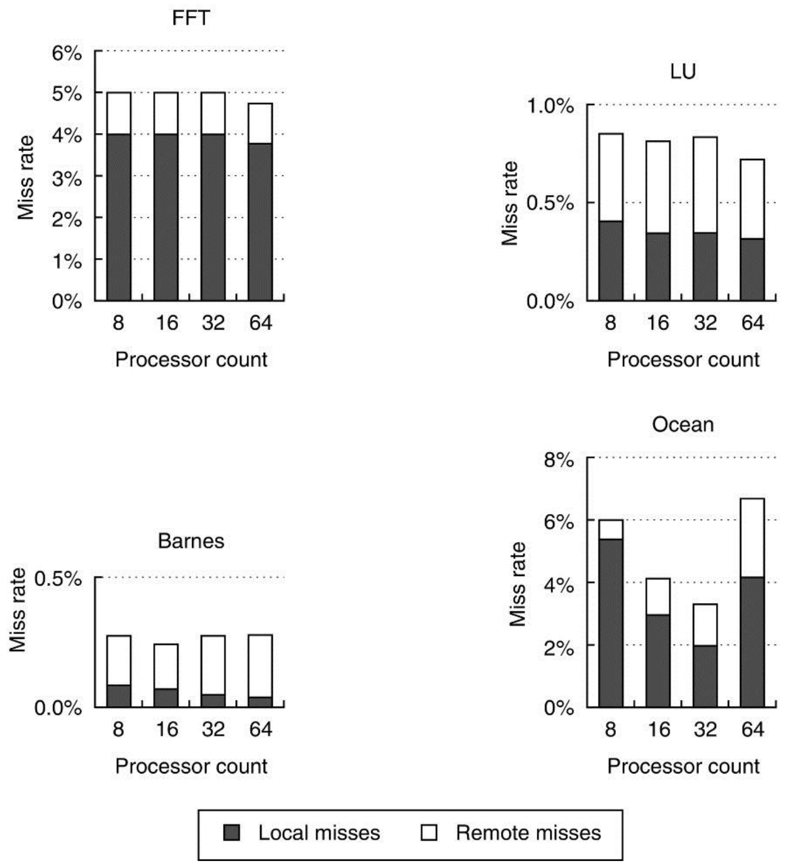


Figure I.12 The data miss rate is often steady as processors are added for these benchmarks. Because of its grid structure, Ocean has an initially decreasing miss rate, which rises when there are 64 processors. For Ocean, the local miss rate drops from 5% at 8 processors to 2% at 32, before rising to 4% at 64. The remote miss rate in Ocean, driven primarily by communication, rises monotonically from 1% to 2.5%. Note that, to show the detailed behavior of each benchmark, different scales are used on the *y*-axis. The cache for all these runs is 128 KB, two-way set associative, with 64-byte blocks. Remote misses include any misses that require communication with another node, whether to fetch the data or to deliver an invalidate. In particular, in this figure and other data in this section, the measurement of remote misses includes write upgrade misses where the data are up to date in the local memory but cached elsewhere and, therefore, require invalidations to be sent. Such invalidations do indeed generate remote traffic, but may or may not delay the write, depending on the consistency model.

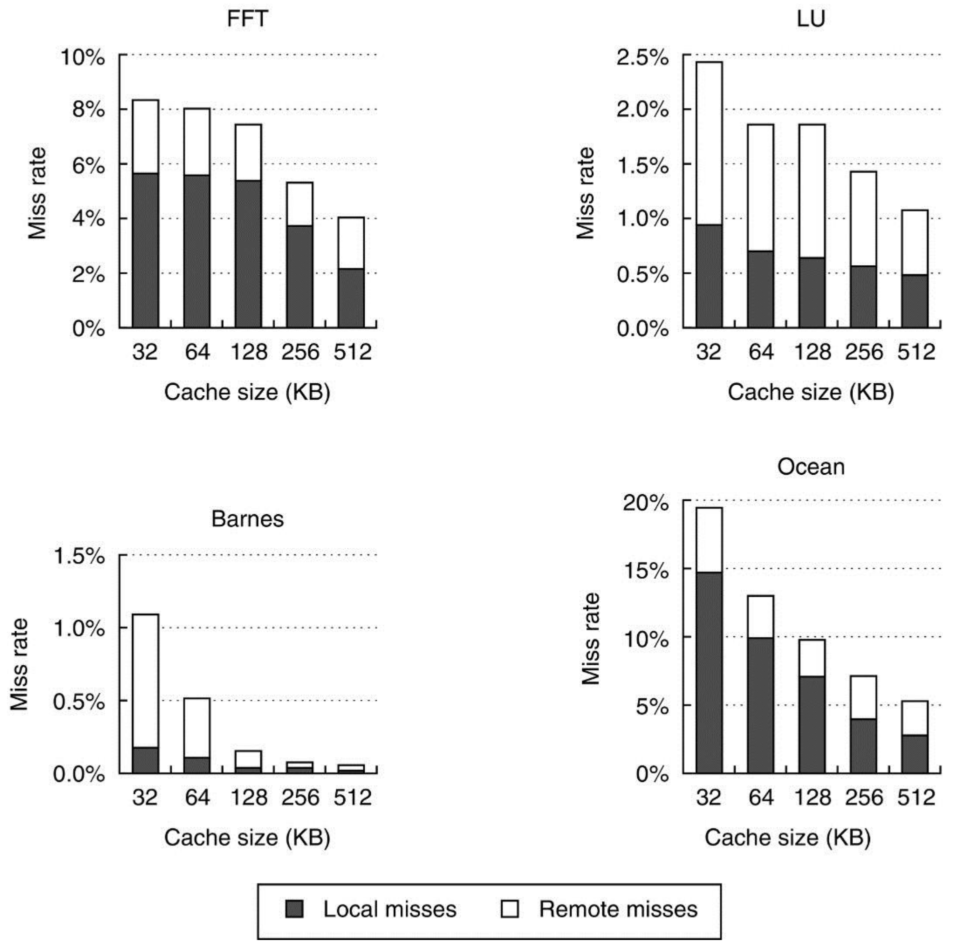


Figure I.13 Miss rates decrease as cache sizes grow. Steady decreases are seen in the local miss rate, while the remote miss rate declines to varying degrees, depending on whether the remote miss rate had a large capacity component or was driven primarily by communication misses. In all cases, the decrease in the local miss rate is larger than the decrease in the remote miss rate. The plateau in the miss rate of FFT, which we mentioned in the last section, ends once the cache exceeds 128 KB. These runs were done with 64 processors and 64-byte cache blocks.

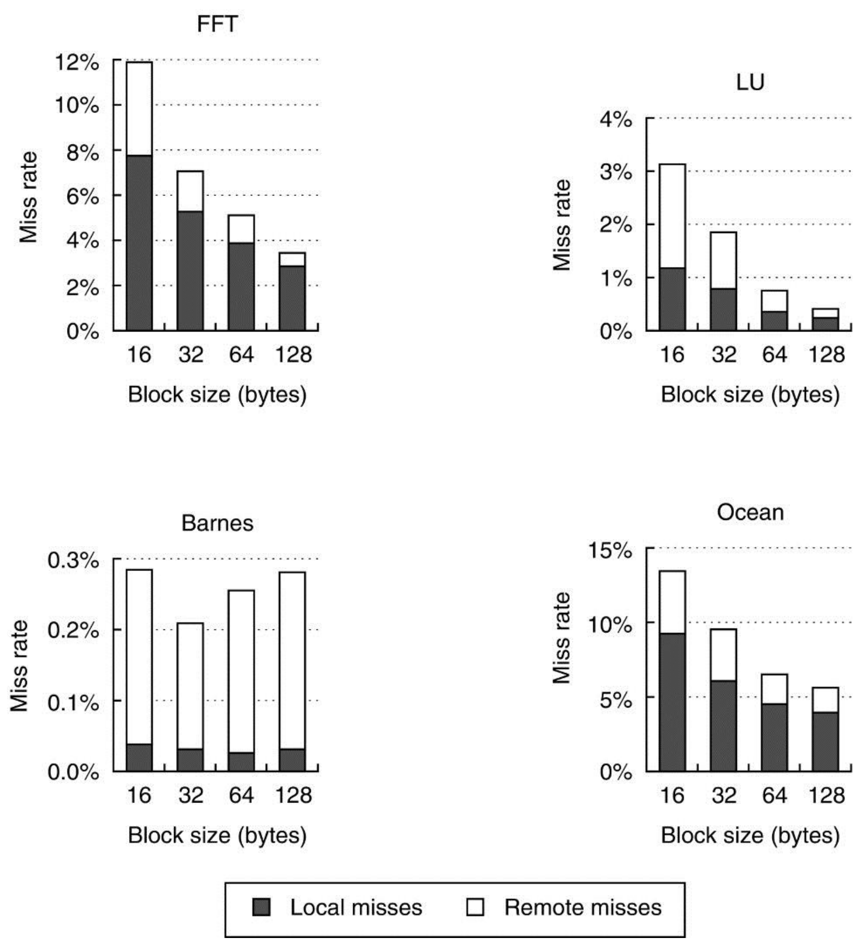


Figure I.14 Data miss rate versus block size assuming a 128 KB cache and 64 processors in total. Although difficult to see, the coherence miss rate in Barnes actually rises for the largest block size, just as in the last section.

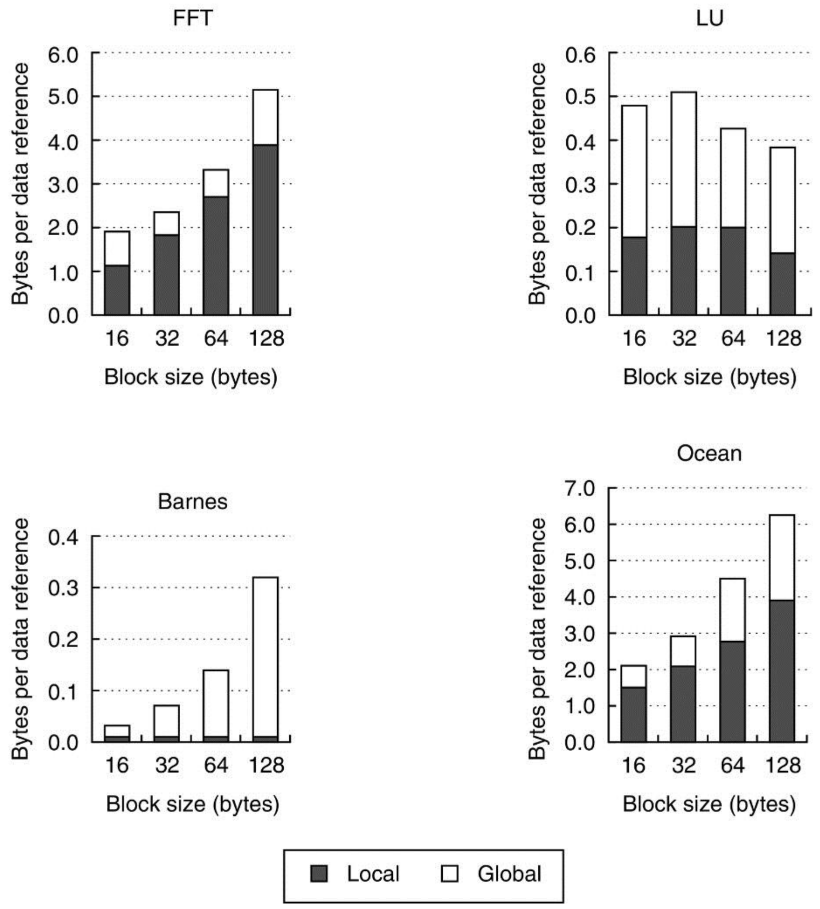


Figure I.15 The number of bytes per data reference climbs steadily as block size is increased. These data can be used to determine the bandwidth required per node both internally and globally. The data assume a 128 KB cache for each of 64 processors.

Characteristic	Processor clock cycles ≤16 processors	Processor clock cycles 17–64 processors
Cache hit	1	1
Cache miss to local memory	85	85
Cache miss to remote home directory	125	150
Cache miss to remotely cached data (three-hop miss)	140	170

Figure I.16 Characteristics of the example directory-based multiprocessor. Misses can be serviced locally (including from the local directory), at a remote home node, or using the services of both the home node and another remote node that is caching an exclusive copy. This last case is called a three-hop miss and has a higher cost because it requires interrogating both the home directory and a remote cache. Note that this simple model does not account for invalidation time but does include some factor for increasing interconnect time. These remote access latencies are based on those in an SGI Origin 3000, the fastest scalable interconnect system in 2001, and assume a 500 MHz processor.

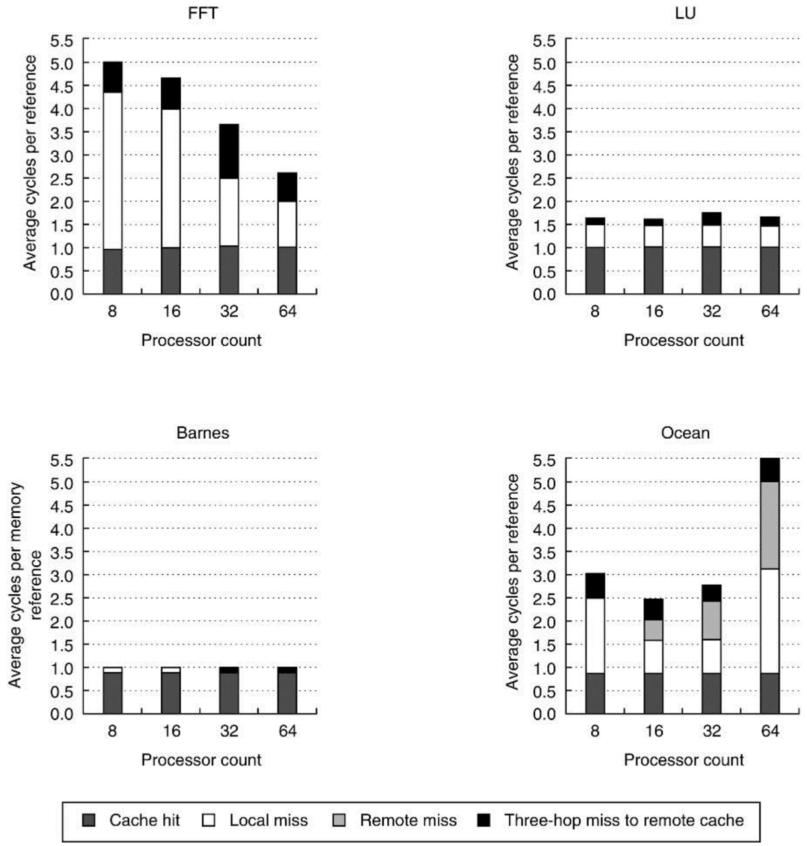


Figure I.17 The effective latency of memory references in a DSM multiprocessor depends both on the relative frequency of cache misses and on the location of the memory where the accesses are served. These plots show the memory access cost (a metric called average memory access time in Chapter 2) for each of the benchmarks for 8, 16, 32, and 64 processors, assuming a 512 KB data cache that is two-way set associative with 64-byte blocks. The average memory access cost is composed of four different types of accesses, with the cost of each type given in Figure I.16. For the Barnes and LU benchmarks, the low miss rates lead to low overall access times. In FFT, the higher access cost is determined by a higher local miss rate (1–4%) and a significant three-hop miss rate (1%). The improvement in FFT comes from the reduction in local miss rate from 4% to 1%, as the aggregate cache increases. Ocean shows the biggest change in the cost of memory accesses, and the highest overall cost at 64 processors. The high cost is driven primarily by a high local miss rate (average 1.6%). The memory access cost drops from 8 to 16 processors as the grids more easily fit in the individual caches. At 64 processors, the dataset size is too small to map properly and both local misses and coherence misses rise, as we saw in Figure I.12.

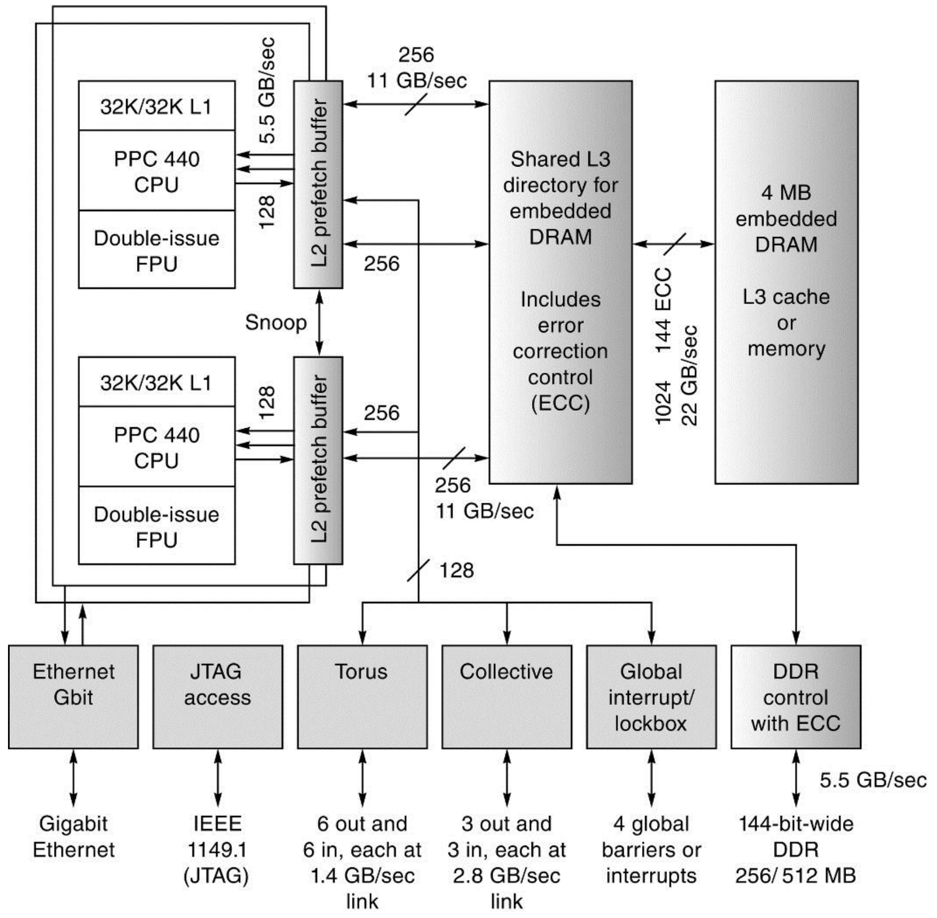


Figure I.18 The BG/L processing node. The unfilled boxes are the PowerPC processors with added floating-point units. The solid gray boxes are network interfaces, and the shaded lighter gray boxes are part of the memory system, which is supplemented by DDR RAMS.

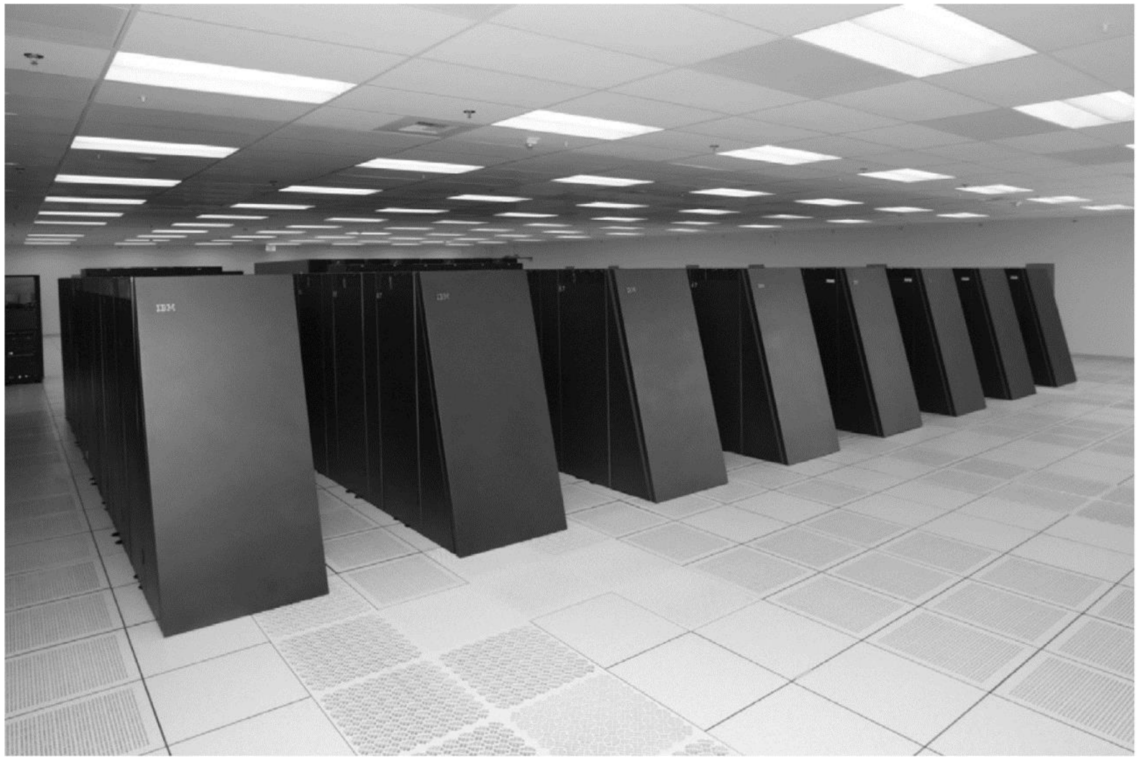
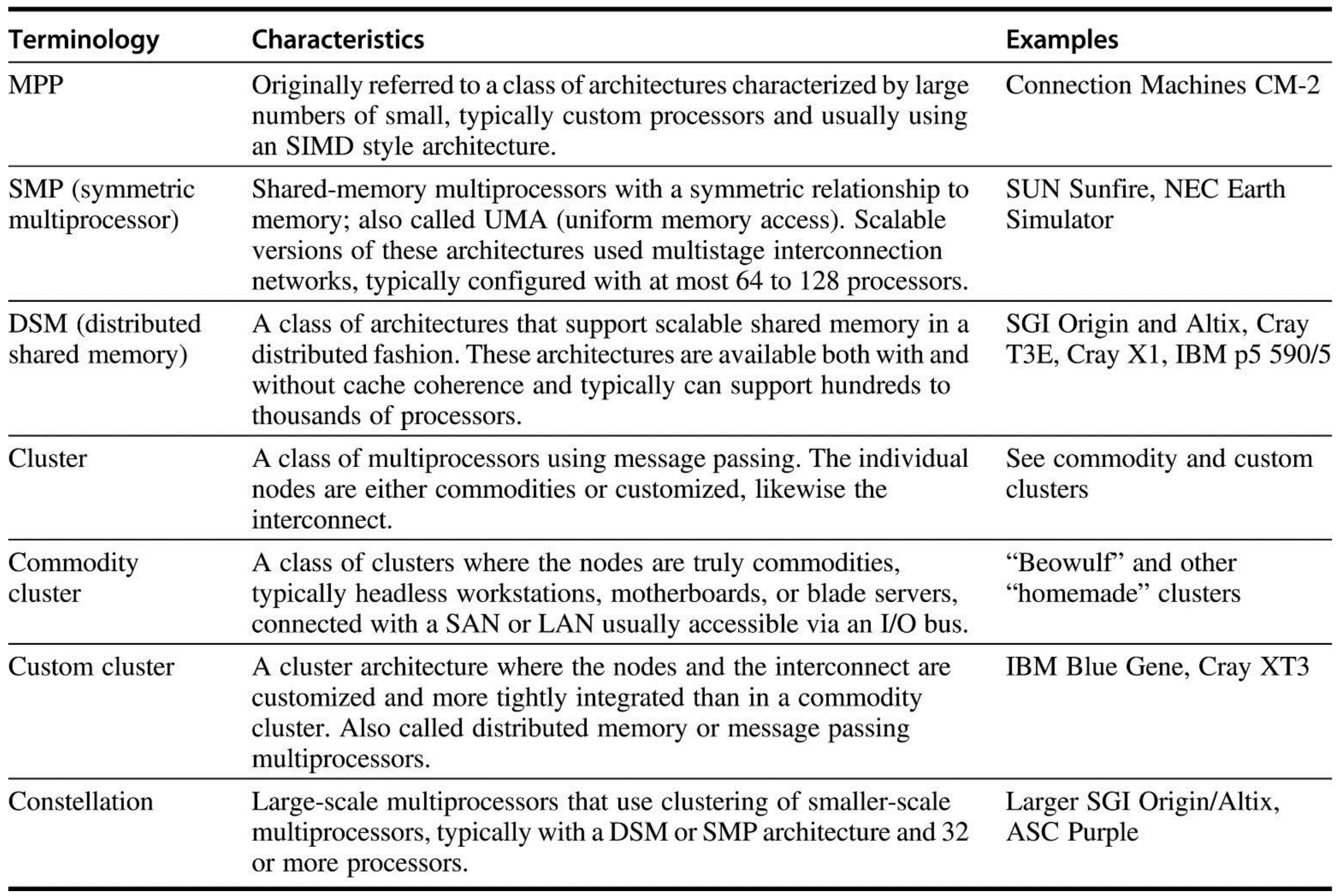
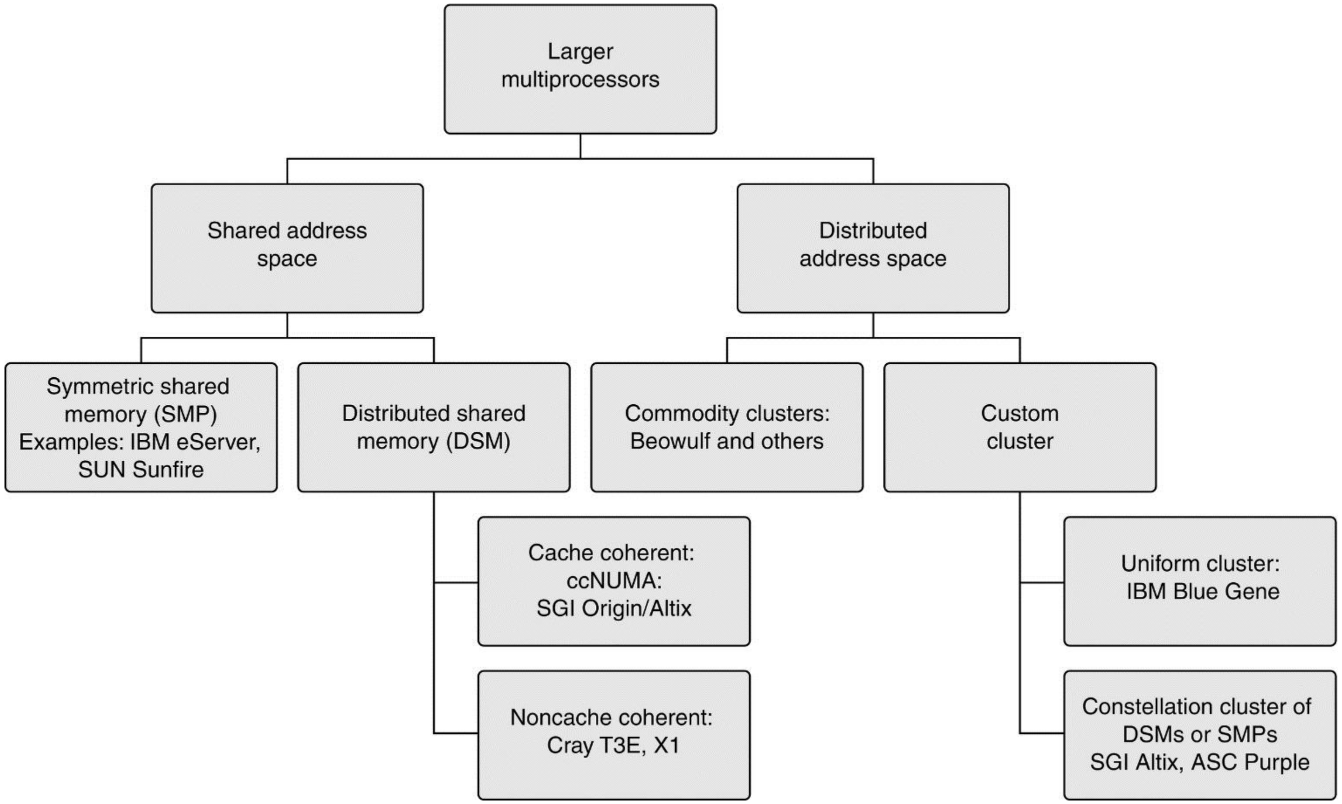


Figure I.19 The 64 K-processor Blue Gene/L system.



Terminology	Characteristics	Examples
MPP	Originally referred to a class of architectures characterized by large numbers of small, typically custom processors and usually using an SIMD style architecture.	Connection Machines CM-2
SMP (symmetric multiprocessor)	Shared-memory multiprocessors with a symmetric relationship to memory; also called UMA (uniform memory access). Scalable versions of these architectures used multistage interconnection networks, typically configured with at most 64 to 128 processors.	SUN Sunfire, NEC Earth Simulator
DSM (distributed shared memory)	A class of architectures that support scalable shared memory in a distributed fashion. These architectures are available both with and without cache coherence and typically can support hundreds to thousands of processors.	SGI Origin and Altix, Cray T3E, Cray X1, IBM p5 590/5
Cluster	A class of multiprocessors using message passing. The individual nodes are either commodities or customized, likewise the interconnect.	See commodity and custom clusters
Commodity cluster	A class of clusters where the nodes are truly commodities, typically headless workstations, motherboards, or blade servers, connected with a SAN or LAN usually accessible via an I/O bus.	"Beowulf" and other "homemade" clusters
Custom cluster	A cluster architecture where the nodes and the interconnect are customized and more tightly integrated than in a commodity cluster. Also called distributed memory or message passing multiprocessors.	IBM Blue Gene, Cray XT3
Constellation	Large-scale multiprocessors that use clustering of smaller-scale multiprocessors, typically with a DSM or SMP architecture and 32 or more processors.	Larger SGI Origin/Altix, ASC Purple

Figure I.20 A classification of large-scale multiprocessors. The term *MPP,* which had the original meaning described above, has been used more recently, and less precisely, to refer to all large-scale multiprocessors. None of the commercial shipping multiprocessors is a true MPP in the original sense of the word, but such an approach may make sense in the future. Both the SMP and DSM class includes multiprocessors with vector support. The term *constellation* has been used in different ways; the above usage seems both intuitive and precise [Dongarra et al. 2005].



Larger multiprocessors
Shared address space
Distributed address space
Symmetric shared memory (SMP) Examples: IBM eServer, SUN Sunfire
Distributed shared memory (DSM)
Commodity clusters: Beowulf and others
Custom cluster
Cache coherent: ccNUMA: SGI Origin/Altix
Noncache coherent: Cray T3E, X1
Uniform cluster: IBM Blue Gene
Constellation cluster of DSMs or SMPs SGI Altix, ASC Purple

Figure I.21 The space of large-scale multiprocessors and the relation of different classes.