5.6 IMPLICATIONS FOR SOFTWARE

So far, we have looked at high-level architectural issues for bus-based cache-coherent multiprocessors and at how architectural and protocol trade-offs are affected by workload characteristics. Let us now come full circle and examine how the architectural characteristics of these small-scale machines influence parallel software. That is, instead of keeping the workload fixed and improving the machine or its protocols, we keep the machine fixed and examine how to improve parallel programs. Improving synchronization algorithms to reduce traffic and latency was an example of this, but let us look at the parallel programming process more generally.

The general techniques for load balance and inherent communication discussed in Chapter 3 also apply to cache-coherent machines. In addition, one general partitioning principle that is applicable across a wide range of computations on these machines is to try to assign computation such that only one processor writes a given set of data, at least during a single computational phase. In many computations, processors read one large shared data structure and write another. In Raytrace, for example, processors read a scene and write an image. A choice is available of whether to partition the computation so the processors write disjoint pieces of the destination structure and read share the source structure, or read disjoint pieces of the source structure and write share the same memory locations in the destination. All other considerations being equal (such as load balance and programming complexity), it is usually advisable to avoid write sharing in these situations. Write sharing not only causes invalidations and, hence, cache misses and traffic, but if different processes write the same words, it is very likely that the writes must be protected by synchronization such as locks, which are even more expensive.

The structure of communication is not much of a variable: with a single centralized memory, little incentive exists to use explicit memory-to-memory data transfers, so all communication is implicit through loads and stores that lead to the transfer of cache blocks. Mapping is not an issue (other than to try to ensure that processes migrate from one processor to another as little as possible) and is invariably left to the operating system. The most interesting issues are managing data locality and artifactual communication in the orchestration step, and in particular, addressing temporal and spatial locality to reduce the number of cache misses and hence reduce latency, traffic, and contention on the shared bus.

With main memory being centralized, temporal locality is exploited in the processor caches. The specialization of the working set curve introduced in Chapter 3 for bus-based machines is shown in Figure 5.32. All capacity-related misses go to the same bus and memory and are about as expensive as coherence misses. The other three kinds of misses will occur and generate bus traffic even with an infinite cache. The major goal for temporal locality is to have working sets fit in the cache hierarchy, and the techniques are the same as those discussed in Chapter 3.

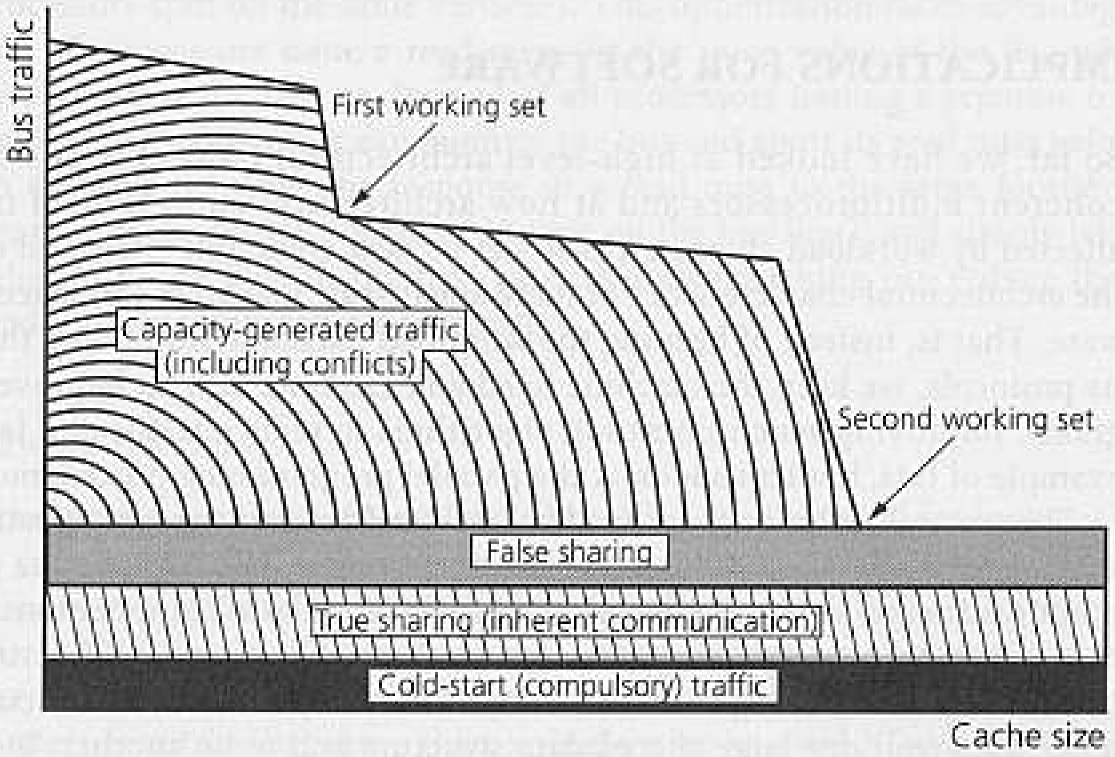

FIGURE 5.32 Data traffic on the shared bus and its components as a function of cache size. The points of inflection indicate the working sets of the program.

For spatial locality, a centralized memory makes data distribution and the granularity of allocation in main memory irrelevant (only interleaving data among memory banks to reduce contention may be an issue, just as in uniprocessors). The ill effects of poor spatial locality are *fragmentation* (i.e., fetching unnecessary data on a cache block) and false sharing. The reasons are that the granularity of communication and the granularity of coherence are both cache blocks, which are larger than a word. The former causes fragmentation, and the latter causes false sharing. (We assume here that techniques to eliminate false sharing like subblock dirty bits are not used since they are not found in most real machines.) Let us examine some techniques to alleviate these problems and effectively exploit the prefetching effects of long cache blocks, as well as techniques to alleviate cache conflicts by better spatial organization of data. Many such techniques can be found in a programmer's "bag of tricks." The following provides only a sampling of the most general ones.

- *Assign tasks to reduce spatial interleaving of access patterns.* It is desirable to assign tasks such that each processor tends to access large contiguous chunks of data. For example, if an array computation with *n* elements is to be divided among *p* processors, it is better to divide it so that each processor accesses *n/p* contiguous elements rather than to use a finely interleaved assignment of elements. This increases spatial locality and reduces false sharing of cache blocks. Of course, load balancing or other constraints may force us to do otherwise.
- *Structure data to reduce spatial interleaving of access patterns.* We saw an example of this in the equation solver kernel in Chapter 3, when we used higher-dimensional arrays to keep a processor's partition of an array contiguous in the

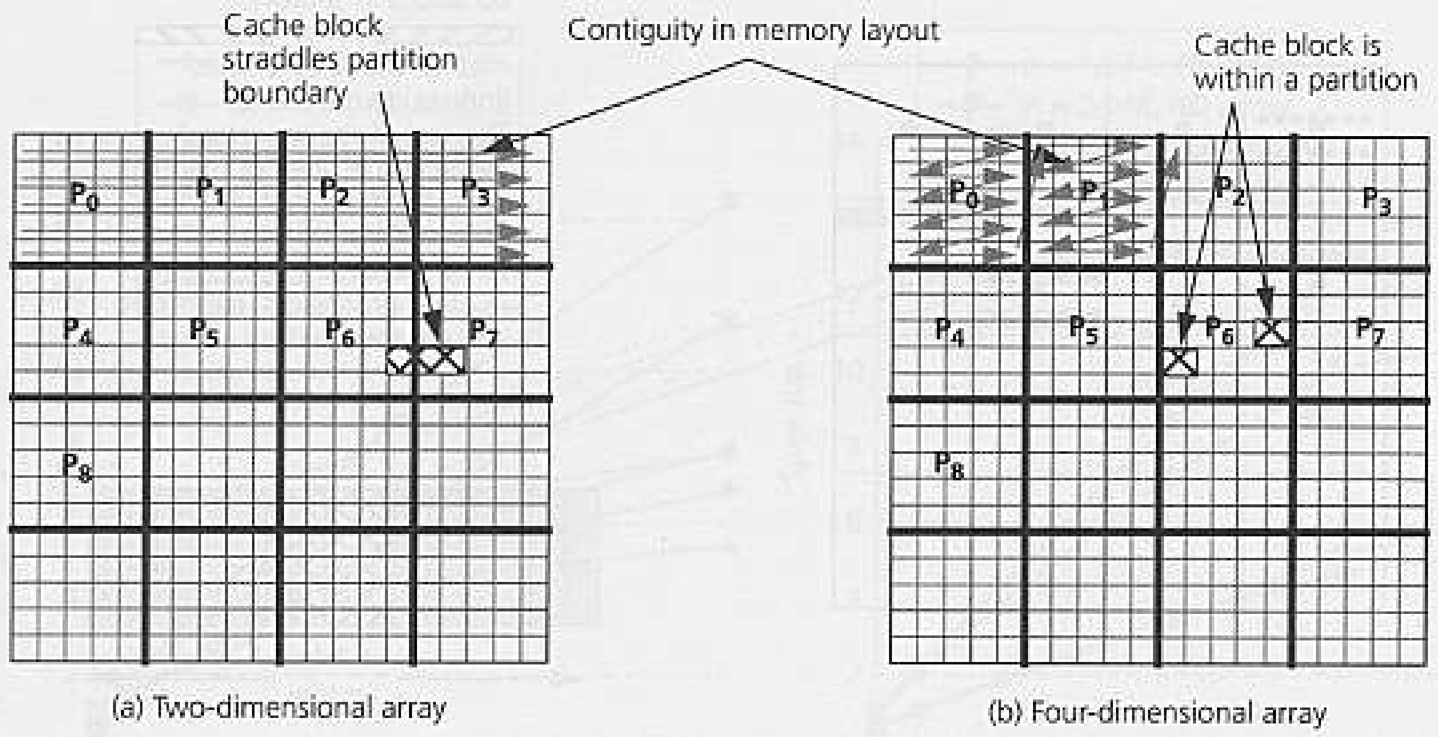
Cache block straddles partition boundary
Contiguity in memory layout
Cache block is within a partition
P_0
P_1
P_2
P_3
P_4
P_5
P_6
P_7
P_8
(a) Two-dimensional array
(b) Four-dimensional array

FIGURE 5.33 Reducing false sharing and fragmentation by using higher-dimensional arrays to keep partitions contiguous in the address space. In the two-dimensional array case, cache blocks straddling partition boundaries cause both fragmentation (a miss brings in useless data from the other processor's partition) as well as false sharing. The four-dimensional array representation makes partitions contiguous and alleviates these problems.

address space in order to allocate partitions locally at page granularity in physically distributed memory. This technique also helps reduce false sharing, fragmentation of data transfer, and conflict misses, as shown in Figures 5.33 and 5.34, all of which cause misses and traffic on the bus. A cache block larger than a single grid element may straddle a column-oriented partition boundary, as shown in Figure 5.33(a). If the block is larger than two grid elements, it can cause communication due to false sharing. This is easiest to see if we assume for a moment that there is no inherent communication in the algorithm; for example, suppose in each sweep a process simply adds a constant value to each of its assigned grid elements instead of performing a nearest-neighbor computation. Now, even a two-element (or larger) cache block straddling a partition boundary would be false-shared as different processors wrote different words on it. This would also cause fragmentation in communication, since a process reading its own boundary element and missing on it would also fetch other elements in the other processor's partition that are on the same cache block but that it does not need. The conflict-misses problem is explained in Figure 5.34. The issue in all these cases is noncontiguity of partitions. Thus, a single data structure transformation (as in Figure 5.33[b]) helps us solve all our spatial locality–related problems in the equation solver kernel. Figure 5.35 illustrates the performance impact of using higher-dimensional arrays to represent grids or blocked matrices in the Ocean and LU applications on the SGI

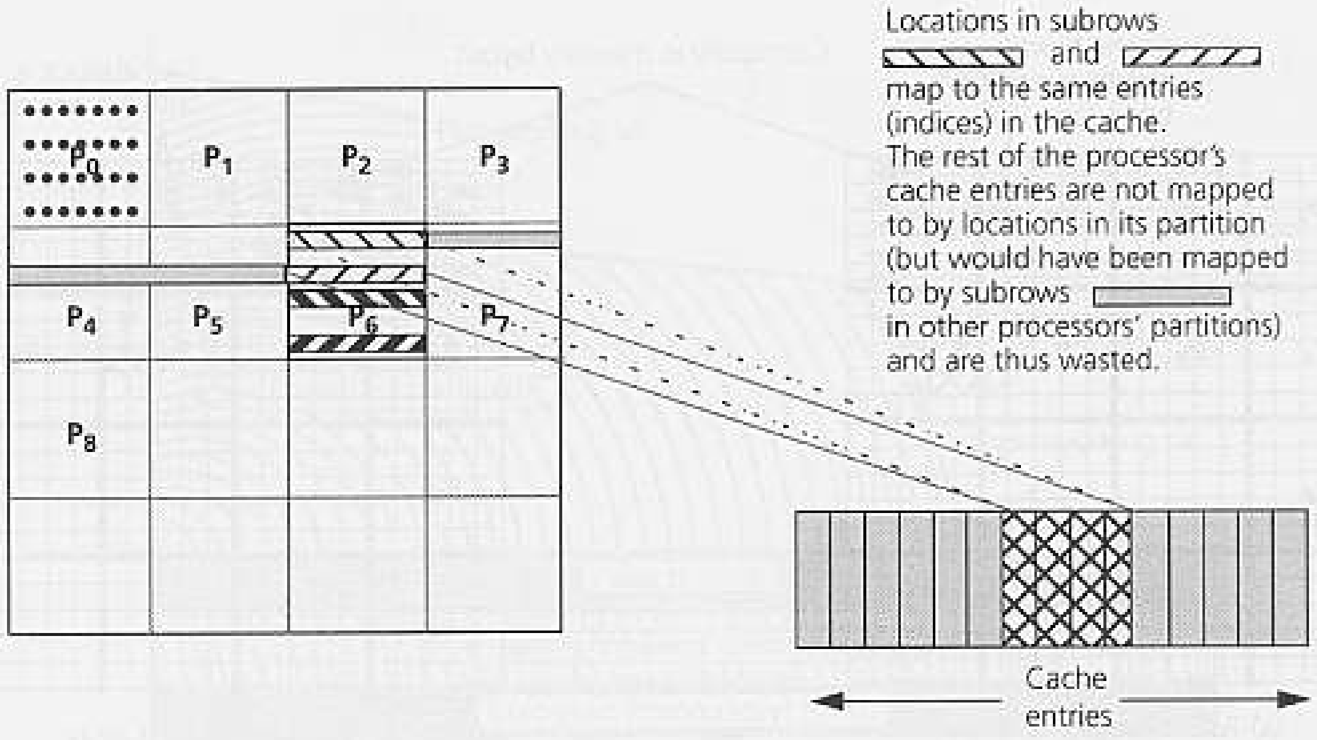
Locations in subrows ▧ and ▨ map to the same entries (indices) in the cache.
The rest of the processor's cache entries are not mapped to by locations in its partition (but would have been mapped to by subrows ▒ in other processors' partitions) and are thus wasted.
P0
P1
P2
P3
P4
P5
P6
P7
P8
Cache entries

FIGURE 5.34 Cache mapping conflicts caused by a two-dimensional array representation in a direct-mapped cache. The figure shows the worst case, in which the separation between successive subrows in a process's partition (i.e., the size of a full row of the 2D array) is exactly equal to the size of the cache, so consecutive subrows map directly on top of one another in the cache. Every subrow accessed knocks the previous subrow out of the cache. In the next sweep over its partition, the processor will miss on every cache block it references, even if the cache as a whole is large enough to fit a whole partition. Many intermediately poor cases may be encountered depending on grid size, number of processors, and cache size. Since the cache size in bytes is a power of two, sizing the dimensions of allocated arrays to be powers of two is discouraged.

Challenge. The impact of conflicts and false sharing on uniprocessor and multiprocessor performance is clear.

- *Beware of conflict misses.* In illustrating conflict misses in the grid solver, Figure 5.34 shows how allocating power-of-two-sized arrays can cause pathological cache conflict problems since the cache size is also a power of two. Even if the logical size of the array that the application needs is a power of two, it is often useful to allocate a larger array that is not a power of two and then access only the amount needed. However, this strategy can interfere with allocating data at page granularity (also a power of two) in machines with physically distributed memory, so we may have to be careful. The cache mapping conflicts in this example are within a single data structure that is accessed in a predictable manner and can thus be alleviated in a structured way. Mapping conflicts are more difficult to avoid when they happen across different major data structures (e.g., across different grids used by the Ocean application), where they may have to be alleviated by ad hoc padding and alignment. However, in a shared address space they are particularly insidious when they occur on seemingly harmless shared variables or data structures that a programmer is not inclined to think about. For example, a frequently accessed pointer to an important data structure may conflict in a direct-mapped cache

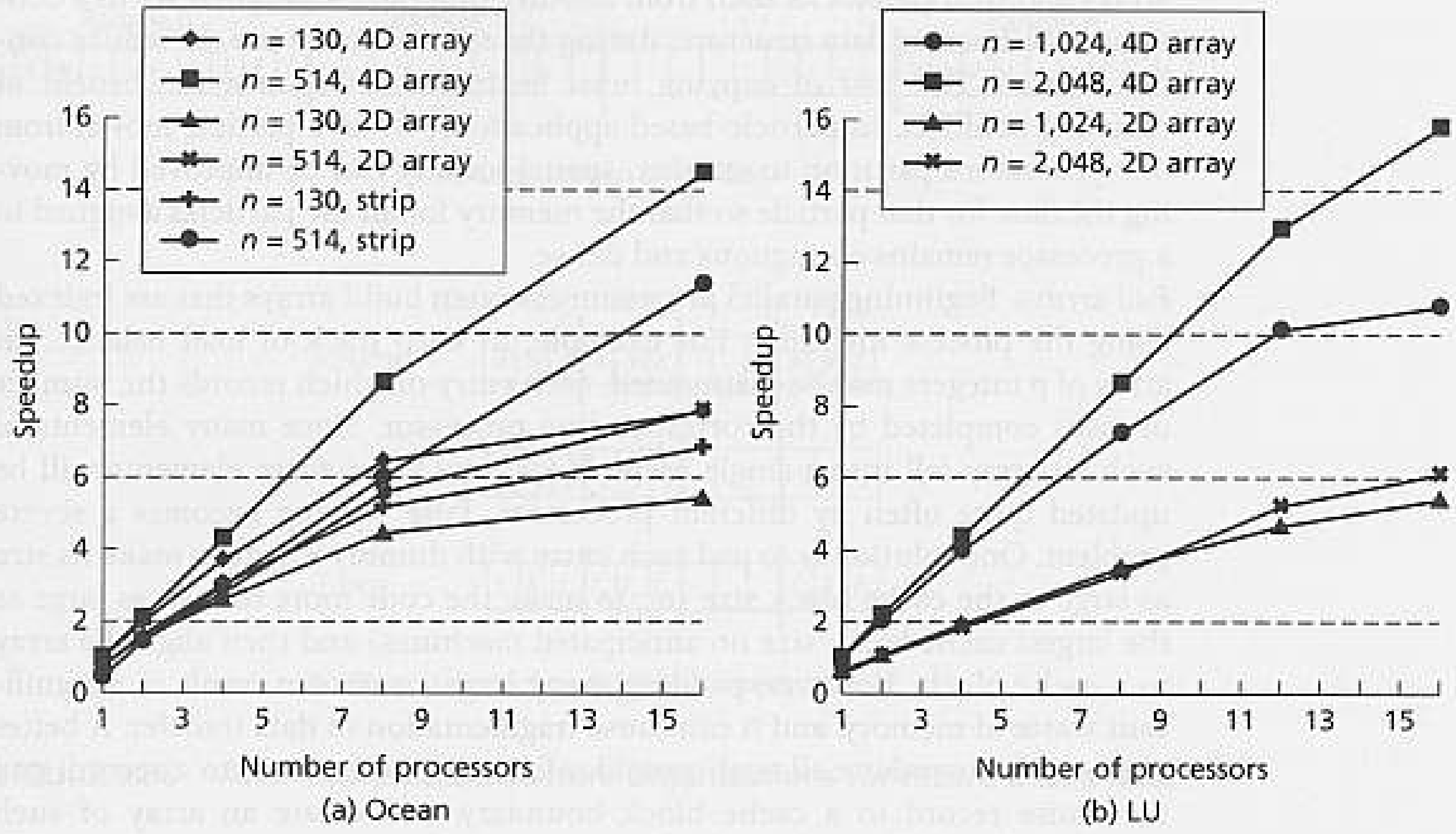

FIGURE 5.35 Performance impact of using 4D versus 2D arrays to represent two-dimensional grid or matrix data structures on the SGI Challenge. Results are shown for different problem sizes for the Ocean and LU applications. For Ocean, "strip" indicates partitioning into strips of contiguous rows (in which 2D or 4D arrays don't matter), while all other cases assume partitioning into squarelike blocks.

with a scalar variable that is also frequently accessed during the same computation, causing a lot of traffic. Fortunately, such problems tend to be infrequent in modern (large and set-associative) second-level caches. In general, efforts to exploit locality can be wasted if attention is not paid to reducing conflict misses.

- *Use per-processor heaps.* It is desirable to have separate heap regions for each processor (or process) from which it allocates data dynamically. Otherwise, if a program performs a lot of very small memory allocations, data used by different processors may fall on the same cache block.
- *Copy data to increase spatial locality.* If a processor is going to reuse a set of data that is otherwise allocated noncontiguously in the address space, it is often desirable to make a contiguous copy of the data for that period to improve spatial locality and reduce cache conflicts. Copying requires memory accesses and has a cost, and it is not useful if the data is likely to reside in the cache anyway. For example, in blocked matrix factorization or multiplication, with a 2D array representation of the matrix a block is not contiguous in the address space (just like a partition in the equation solver kernel). However, a 2D representation makes programming easier. It is therefore not uncommon to use 2D

arrays and to copy blocks used from another processor's assigned set to a contiguous temporary data structure, during the time of active use, to reduce conflict misses. The cost of copying must be traded off against the benefit of reducing conflicts. In particle-based applications, when a particle moves from one processor's partition to another, spatial locality can be improved by moving the data for that particle so that the memory for all the particles assigned to a processor remains contiguous and dense.

- *Pad arrays.* Beginning parallel programmers often build arrays that are indexed using the process identifier. For example, to keep track of load balance, an array of p integers may be maintained, each entry of which records the number of tasks completed by the corresponding processor. Since many elements of such an array fall into a single cache block, and since these elements will be updated quite often by different processors, false sharing becomes a severe problem. One solution is to pad each entry with dummy words to make its size as large as the cache block size (or, to make the code more robust, as large as the largest cache block size on anticipated machines) and then align the array to a cache block. However, padding many large arrays can result in a significant waste of memory, and it can cause fragmentation in data transfer. A better strategy is to combine all such variables for a given process into a record, pad the entire record to a cache block boundary, and create an array of such records indexed by process identifier.
- *Determine how to organize arrays of records.* Suppose we have a number of logical records to represent, such as the particles in the Barnes-Hut gravitational simulation. Should we represent them as a single array of n particles, each entry being a record with fields like position, velocity, force, mass, and so on, as in Figure 5.36(a)? Or should we represent them as separate arrays of size n, one per field, as in Figure 5.36(b)? Programs written for vector machines such as traditional CRAY computers tend to use a separate array (vector) for each property or field of an object—in fact, even one per field per physical dimension (x, y, or z). When data is accessed by field, for example, the velocity of all particles, this increases the performance of vector operations by making accesses to memory unit stride and hence reducing memory bank conflicts. In cache-coherent multiprocessors, however, new trade-offs arise, and the best way to organize data depends on the access patterns.

  An interesting tension is illustrated by the particle update and force calculation phases of the Barnes-Hut application. Consider the update phase first. A processor reads and writes only the position and velocity fields of all its assigned particles in this phase. However, its assigned particles are not contiguous in the shared particle array. Suppose there is one array of size n (number of particles) per field or property. A double-precision three-dimensional position (or velocity) is 24 bytes of data, so several of these may fit on a cache block. Since adjacent particles in the array may be read and written by different processors, false sharing can result. For this phase, it is better to have a single array of particle records, where each record holds all information about that particle; that is, to organize data by particle rather than by field.

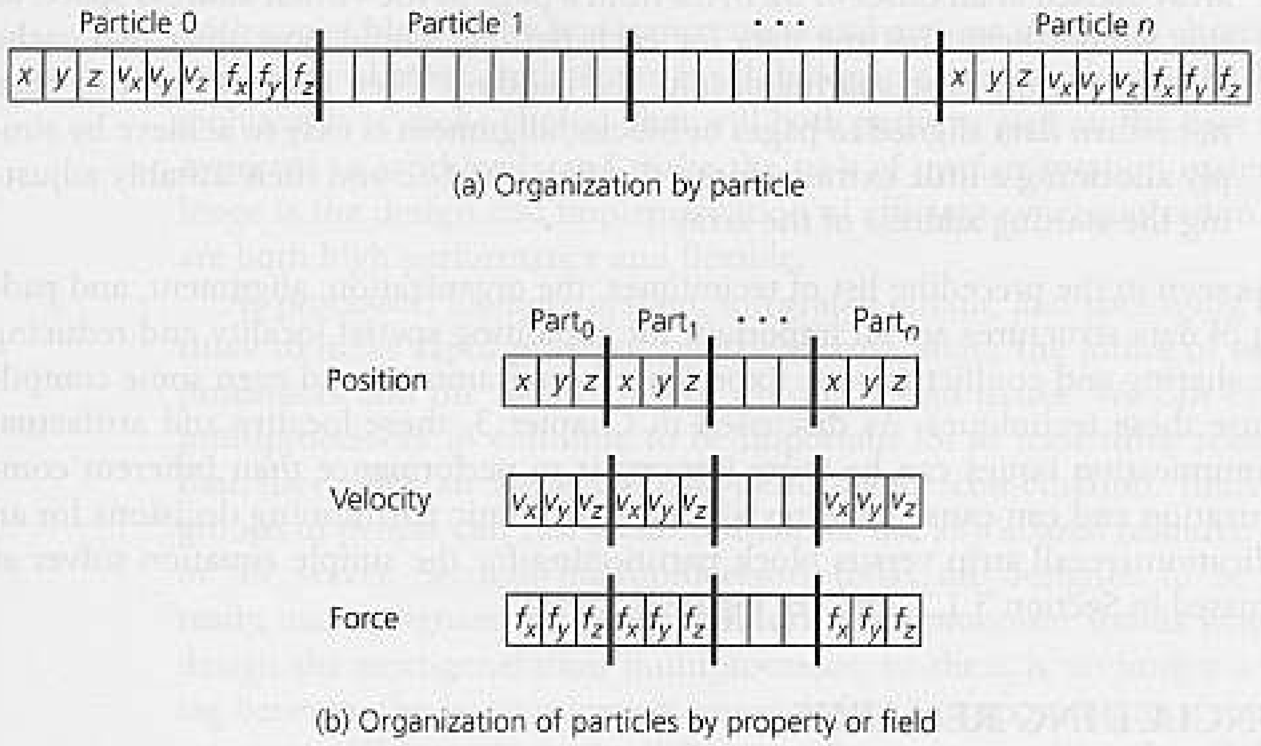

FIGURE 5.36 Alternative data structure organizations for record-based data

Now consider the force calculation phase of the same application. Suppose we use an organization by particle rather than by field as above. To compute the force on a particle, a processor reads the position values of many other particles and cells; it then updates the force components of its own particle. However, the force and position components of a particle may fall on the same cache block. In updating force components, it may therefore invalidate the position values of this particle from the caches of other processors that are using and reusing them as a result of false sharing within a particle record, even though the position values themselves are not being modified in this phase of computation. In this case, it would probably be better if we were to split the single array of particle records into two arrays of size *n* each, one for positions (and perhaps other properties) and one for forces. The entries of the force array themselves could be padded to reduce cross-particle false sharing. In general, it is often beneficial to split arrays of records to separate fields that are used in a read-only manner in a phase from the fields whose values are updated in the same phase. Different situations or phases may dictate different organizations for a data structure, and the ultimate decision depends on which pattern or phase dominates performance.

- *Align arrays.* In conjunction with the preceding techniques, it is often necessary to align arrays to cache block boundaries to achieve the full benefits. For example, given a cache block size of 64 bytes and 8-byte fields, we may have decided to maintain a single array of particle records with *x, y, z, fx, fy,* and *fz*. To avoid cross-particle false sharing, we pad each 48-byte record with two dummy 8-byte fields to fill a cache block. However, this wouldn't help if the array started at an offset of 32 bytes from a page in the virtual address space, as this would mean that the data for each particle would now span two cache blocks, causing false sharing despite the padding. Even if a `malloc` call does not return data aligned to pages or blocks, alignment is easy to achieve by simply allocating a little extra memory through `malloc` and then suitably adjusting the starting address of the array.

As seen in the preceding list of techniques, the organization, alignment, and padding of data structures are all important for exploiting spatial locality and reducing false sharing and conflict misses. Experienced programmers and even some compilers use these techniques. As discussed in Chapter 3, these locality and artifactual communication issues can be more important to performance than inherent communication and can cause us to revisit our algorithmic partitioning decisions for an application (recall strip versus block partitioning for the simple equation solver as discussed in Section 3.1.2, and see Figure 5.35[a]).

5.7 CONCLUDING REMARKS

Symmetric shared memory multiprocessors are a natural extension of workstations and personal computers. A sequential application can run totally unchanged and yet benefit in performance by obtaining a larger fraction of a processor's time and by taking advantage of the large amount of shared main memory and I/O capacity typically available on such machines. Parallel applications are also relatively easy to bring up, as all shared data is directly accessible from all processors using ordinary loads and stores. Gradual parallelization is possible by selectively parallelizing computationally intensive portions of a sequential application, subject to the dictates of Amdahl's Law. For multiprogrammed workloads, a key advantage is the fine granularity at which resources can be shared among application processes and by the operating system, which can thus easily export a familiar, single-system image to each application. This is true both temporally, in that processors and/or main memory pages can frequently be reallocated among different application processes, and physically, in that main memory may be split among applications at the granularity of individual pages. Because of these appealing features, all major vendors of computer systems, from workstation suppliers like Sun, Silicon Graphics, Hewlett-Packard, Digital, and IBM to personal computer suppliers like Intel and Compaq, are producing and selling such machines. In fact, for some of the large workstation vendors, these multiprocessors constitute a substantial fraction of their revenue stream and a still larger fraction of their net profits because of the higher margins on these higher-end machines.

The key technical challenge in the design of symmetric multiprocessors is the organization and implementation of the shared memory system, which is used for communication between processors in addition to handling all regular memory accesses. Most small-scale parallel machines found today use the system bus as the interconnect for communication, and the challenge then becomes how to maintain coherency of the shared data in the private caches of the processors. A large variety of options are available to the system architect, including the set of states associated with cache blocks, the bus transactions and actions used, the choice of cache block size, and whether updates or invalidations are used. The key task of the system architect is to make choices that will both perform well on the data sharing patterns expected in workloads and make the task of implementation easier. Another challenge is the design and implementation of efficient synchronization techniques that are both high performance and flexible.

As processor, memory system, integrated circuit, and packaging technology continue to make rapid progress, questions arise about the future of small-scale multiprocessors and the importance of various design issues. We can expect small-scale multiprocessors to continue to be important for at least three reasons. The first is that they offer an attractive cost-performance combination. Individuals or small groups of people can easily afford them for use as a shared resource or as a compute or file server. Second, microprocessors today are designed to be multiprocessor-ready, and designers are aware of future microprocessor trends when they begin to design the next-generation multiprocessor, so there is no longer a significant time lag between the latest microprocessor and its incorporation in a multiprocessor. As we saw in Chapter 1, the Intel Pentium Pro processor line plugs "gluelessly" into a shared bus. The third reason is that the essential software technology for parallel machines (compilers, operating systems, programming languages) is maturing rapidly for small-scale shared memory machines. For example, most computer system vendors have efficient parallel versions of their operating systems ready for their bus-based multiprocessors. As levels of integration increase, multiple processors on a chip become attractive. While the optimal design points may change, the design issues that we have explored in this chapter are fundamental and will remain important with progress in technology.

This chapter has explored many of the key design aspects of bus-based multiprocessors at the "logical" level, involving cache block state transitions and complete (atomic) bus transactions. At this level, the design and implementation appears to be a rather simple extension of traditional cache controllers. However, much of the difficulty in such designs and many of the opportunities for optimization and innovation occur at the next lower level of protocol design and at the more detailed "physical" level. The next chapter goes down a level deeper into the design and organization of bus-based cache-coherent multiprocessors and some of their natural generalizations.

5.8 EXERCISES

5.1 Is the cache coherence problem an issue with processor registers? Given that registers are not kept consistent in hardware, how do current systems guarantee the desired semantics of a program?

5.2 Consider the following graph indicating the miss rate of an application as a function of cache block size on a multiprocessor. As might be expected, the curve has a U-shaped appearance. Consider the three points A, B, and C on the curve. Indicate