AIX

SMP Overview

By Debora Blakely-Fogel

AIX 4.1.1 operating system, a new release of AIX, supports Symmetric Multiprocessor (SMP) systems based on the PowerPC family of processors. This article defines and provides an overview of SMP systems.

Many types of multiprocessors are being used today. A Symmetric Multiprocessor (SMP) is unique in that the system looks exactly the same to each processor in the system. The following is a brief discussion of three Multiprocessor (MP) technologies supported by the AIX operating system.

Shared Memory MP: Figure 1 shows a symmetric multiprocessor configuration. A symmetric multiprocessor, also known as a *shared memory* multiprocessor, has multiple processors that can each address all memory and all devices. User processes running on any processor see the full machine. If two or more processors access the same word in memory, the hardware keeps the caches consistent—invisible to application processes.

Compared to other types, the advantage of SMPs is their use of the same programming model as uniprocessors. Most existing applications written for uniprocessor POWER platforms will run unchanged on an SMP running AIX 4.1.1.

Shared Nothing MP: Figure 2 shows a *shared nothing* multiprocessor or distributed memory configuration. All processors have their own memory and disks. Uniprocessor programs must be changed to run on this configuration because they must pass messages across an interconnect in order to use the multiple processors. SP2™ is an example of a shared nothing multiprocessor that runs AIX.

Shared nothing MPs generally scale better than SMPs because they have no memory bus contention and no cache coherency problems among the processors. However, the changed programming model often outweighs the advantages of this type of MP.

Shared Disk MP: Figure 3 shows a *shared disk* multiprocessor configuration in which processors share only disks. Unlike the SMP, each processor on a shared disk multiprocessor has its own memory. The Clustered Multiprocessor (CMP) portion of HACMP/6000 software allows users to configure RISC System/6000® machines in a shared disk multiprocessor configuration.

The shared disk MP, like the shared nothing MP, has no memory bus contention or cache coherency problem among the processors. However, a centralized locking scheme is used to control access to the disks. This locking scheme requires change to some applications (such as databases) and generally offsets the performance advantages of no memory bus contention or the cache coherency problem.

Scaling

One of the most important metrics of MP performance is scaling. When a processor is added to the system, how much additional performance is obtained on a given workload? Scaling is workload-dependent; some workloads will scale better than others. In addition, workloads will scale differently on different styles of multiprocessors. For example, a workload that shares a lot of data is likely to scale better on an SMP than on a shared nothing MP. This is because all processes on an SMP have a consistent view of the data, and processes on a shared nothing cluster must do message passing to share data.

To help explain this, Figure 4 shows a hypothetical graph of SMP scaling. In a perfect world,

Debora Blakely-Fogel

one would expect performance to increase linearly as processors are added. For example, if a processor is added to an existing two-processor complex, one might expect a 50% increase in performance. However, this does not happen because of the overhead required to maintain a consistent view of the memory and other shared resources for each of the processors. Generally, each additional processor increases performance by slightly less than the previously added processor. In fact, on all SMPs for all real workloads, adding more processors after some critical number ceases to boost performance, and actually decreases throughput. This happens at the point where the cost to maintain a consistent view of memory and other shared resources becomes greater than the processing power provided by a processor.

The greatest strength of an SMP is that it looks just like a uniprocessor to each process running on each processor. Maintaining this illusion is difficult for both the hardware and the Operating System (OS), and the costs of doing so are the root of all limitations of SMPs. This is because the system must do a significant amount of work to maintain a consistent view of memory for all the processors without having a significant impact on performance. There are usually two critical factors limiting SMP scaling:

- **Contention for the bus or switch that connects all the processors to the shared memory**. This may cause workloads with high cache miss rates to scale poorly.
- **Contention for shared words in memory**. This is viewed as contention for the locks that serialize access to these words and help keep the data coherent. Poor scaling may result for workloads that share data between processes or that spend much time in the kernel.

The SMP Cache Consistency Problem

The most basic problem that all SMPs must deal with is SMP cache consistency. Figure 5 shows an example of the problem. Suppose process p1 is running on processor 1 and process p2 is running on processor 2. Suppose also that processes p1 and p2 are working together on a problem and sharing some memory. Consider the following sequence of events:

1. Process p1 loads address 123, which contains the character “a”.

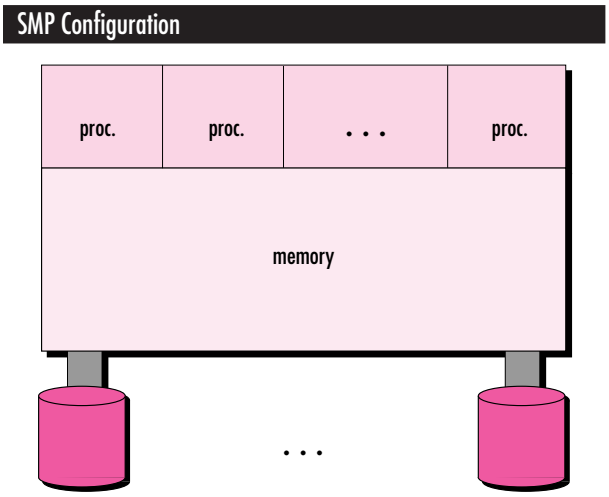

Figure 1. Symmetric multiprocessor configuration

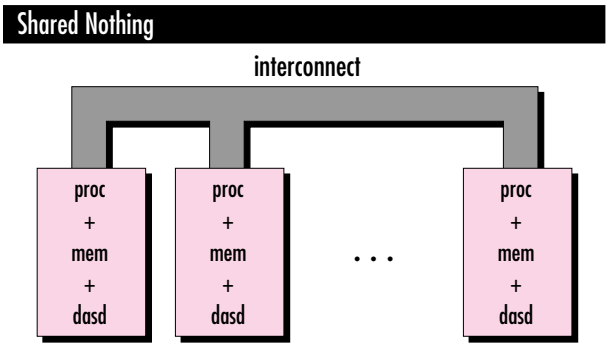

Figure 2. Shared nothing MP cluster

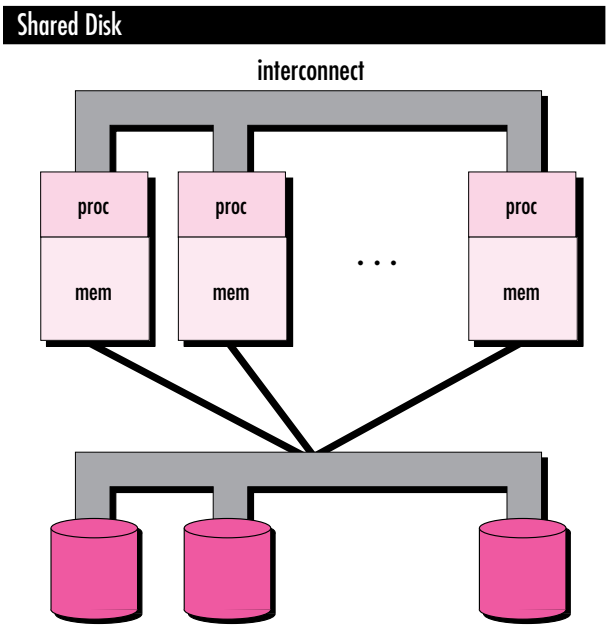

Figure 3. Shared disk MP cluster

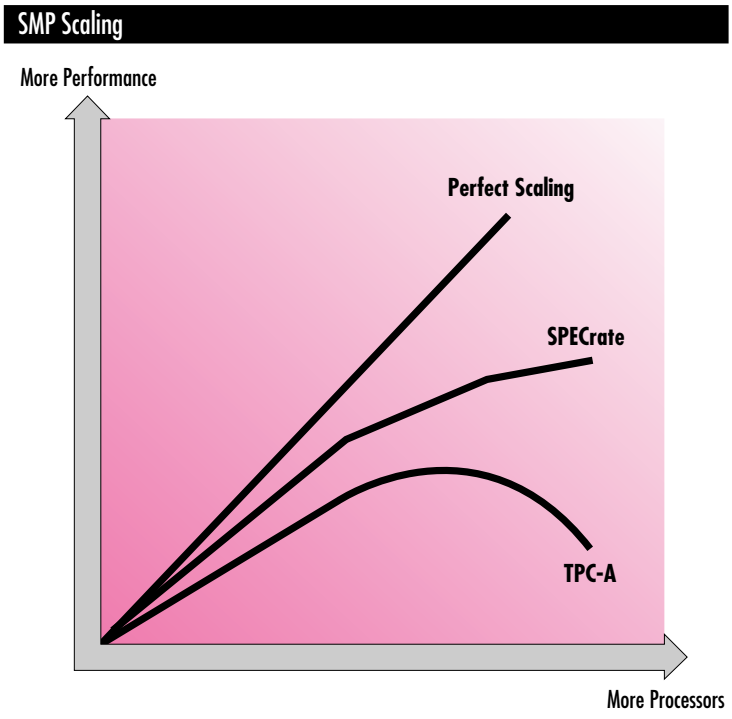

Figure 4. Hypothetical SMP scaling

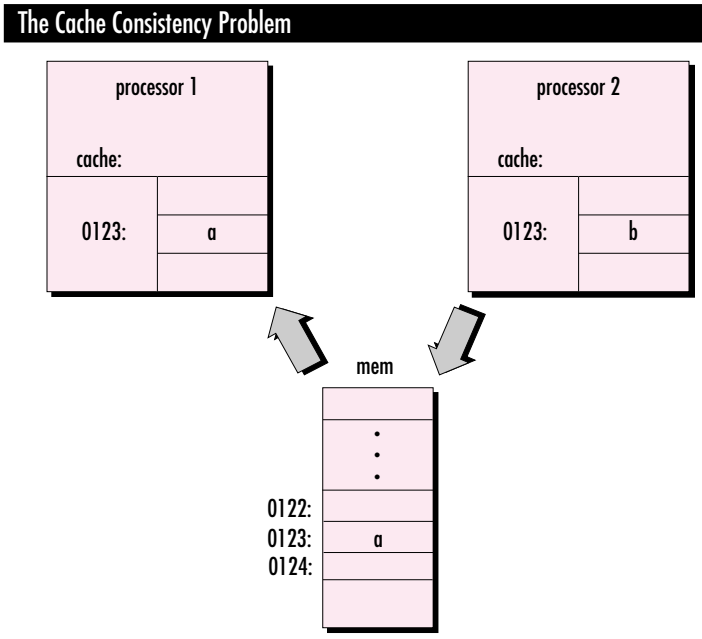

Figure 5. SMP cache consistency problem

2. Process p2 stores the character "b" into address 123.

3. Process p1 loads address 123 again.

The value seen by process p1 at step 3 is very important. With a naive implementation, p1 sees "a" because it has a copy of address 123 in its cache, process p2's load request never goes out to memory, and p1 does not see the new value "b" that process p2 placed there.

As shown in Figure 6, logic at each processor-bus interface broadcasts a message over the bus each time a word in its cache is changed. The logic also snoops on the bus for such messages from other processors. Whenever it detects that another processor changed the value at an address that is copied in its own cache, the snooping logic invalidates that entry in its cache. This *cross invalidate* reminds the processor that the value in that location in the cache is invalid, and it must look somewhere else for the correct value.

In this example, when process p1 reads address 123 the second time, it gets a cache miss and must look elsewhere for the value. The extra snooping logic determines where process p1 should look to get the proper value for address 123. If the new value has been written to memory, process p1 will obtain the value from memory. If the new value has not yet been written to memory, process p1 will get it from processor 2's cache.

Since cross invalidates increase cache misses and the snooping protocol adds to the bus traffic, solving the cache consistency problem reduces the performance and scalability of all SMPs.

False Sharing

The unit of access in a cache is called a *line*. A typical cache line on RISC System/6000 machines is 32 bytes or eight words. It is possible for two processes to reference two different portions of data that fall in the same cache line because they lie close to each other in memory.

In the example in Figure 7, if the process on processor 1 changes the value of d1, the cache consistency logic will invalidate processor 2's cache line causing a cache miss when d2 is accessed, even though the two processes were not sharing any data. This is called *false sharing*.

False sharing increases cache misses and bus traffic, further reducing SMP throughput and scaling.

Locks

One of the well-known problems of SMPs is maintaining data consistency. Consider the classic example of a doubly linked list. Removing an element from the list requires updating two pointers.

- The forward link of the element preceding the element that is being removed
- The backward link of the element following the element that is being removed

When only one of the pointers has been updated, the list is in an inconsistent state. If two processors in a multiprocessor system add or remove elements from the same list simultaneously, the result can be unpredictable.

The Problem: Data Integrity

If multiple processes share a piece of data and if two or more of them try to update it almost simultaneously, the result can be incorrect or incoherent. For example:

- Two processes each add 1 to a shared counter. If the beginning value is x, the correct final value must be x+2. However, if both processes increment the counter simultaneously, the result will be x+1.
- Two processes each try to add an element to a linked list. If both processes try to add an element simultaneously, the result is unpredictable.

A *critical section* is a section of code that modifies shared data, and therefore must not be executed by more than one process at a time. One process must be made to wait until the other process has finished its critical section. The two critical sections must be serialized.

The problem of serializing access to shared data is generic to parallelized code. It occurs both in the kernel and in parallelized user applications. It is especially critical in the kernel because most kernel data must be protected. The kernel performs services on demand, so any processor can be executing any kernel component at any time.

This problem also arises in a milder form in current, uniprocessor, multiprogrammed versions of AIX because of interrupts and explicitly shared data. A kernel component could be in the middle of updating some data when an interrupt occurs. A second instance of the same component might run on behalf of a higher priority thread or process and try to update the same data. On a uniprocessor, this problem can be solved by carefully enabling and disabling interrupts. However, disabling interrupts is not sufficient on an SMP, because interrupts can only be disabled on the processor on which the thread or process is running. If a kernel component is updating some

The Cache Consistency Solution

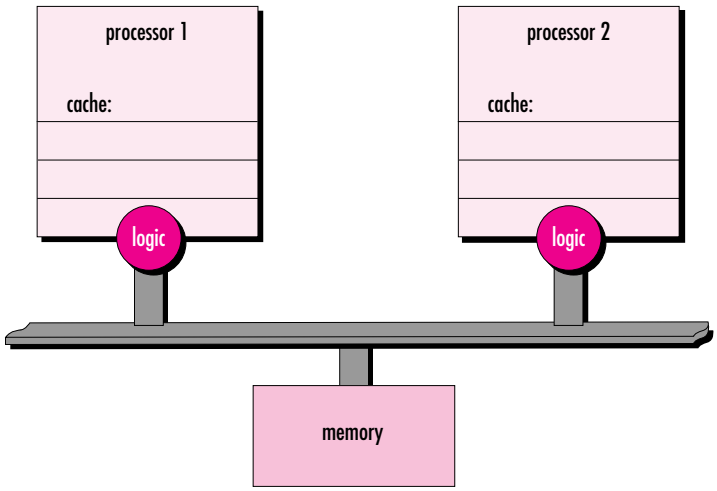

Figure 6. SMP cache consistency solution

False Sharing

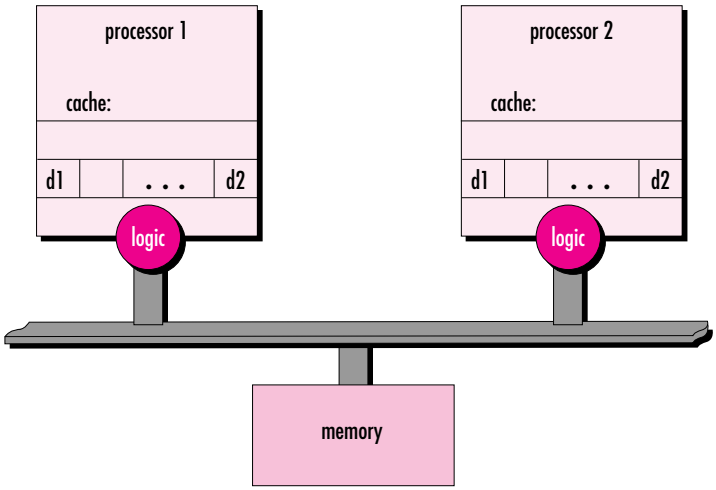

Figure 7. False sharing

data with interrupts disabled on its processor, a second instance of the same component can run on another processor and try to update the same data.

The Solution: Locks

Conceptually, a lock is just a bit in memory that processes use to regulate their entry into critical sections. If a process wants to enter a critical section, it examines the corresponding lock. If the bit is off, the process turns it on, then enters the critical section. If the bit is on, the process waits until it is off before entering. Each process must reset the bit when leaving the critical section.

Locks are not quite so simple to implement because two or more processes could test the same lock simultaneously, determine that the lock is available, and enter the critical section. Because taking a lock requires several operations (read, test, and set the lock bit), this operation is itself a critical section. Therefore, multiprocessor hardware must provide a way to perform this test-and-set operation atomically with respect to the other processors. This means that if more than one processor is trying to obtain the same lock simultaneously, exactly one of them will succeed. All MP locking primitives use this kind of atomic operation as their basic building block.

AIX developers can choose between two types of locks: mutually exclusive simple locks and complex read-write locks.

Fortunately, application developers do not have to implement their own locks. Locking services are provided by AIX for the following types of synchronization:

- **Kernel thread <—> kernel thread**: AIX provides the `simple_lock()`, `simple_unlock()`, `lock_read()`, `lock_write()`, and `lock_done()` services.
- **User process <—> user process**: AIX provides the `msem_lock()` and `msem_unlock()` services.
- **User thread <—> user thread**: The pthreads library provides the `sem_wait()`, `sem_post()`, `pthread_mutex_lock()`, and `pthread_mutex_unlock()` services.

Different Types of Locks

To execute correctly on an SMP, the system must be Multiprocessor Safe (MP Safe). This means that every subsystem and application that accesses shared or global data must have a locking strategy in place that maintains data consistency.

Mutex Versus Read-Write Locks

The locks that have been discussed so far are mutually exclusive or *mutex locks*. They allow one process or thread at a time into a critical section.

If a piece of shared data is read-mostly, it makes sense to distinguish between the many processes that only want to look at or read the data but not change it, and the few processes that want to change or write the data. A read-write lock allows multiple readers into the critical section at once, but guarantees mutual exclusion for writers.

AIX has both types, and developers must choose the type most appropriate for the situation.

Spin Versus Blocking

When a process wants a lock already owned by another process, the process has to wait. A *spin lock* allows the waiting process to keep its processor, repeatedly checking the lock bit in a tight loop (spin) until the lock becomes available. Spin locks are useful for locks that are held only for very short times. A *blocking lock* suspends the process until the lock is free and then puts it back on the run queue. This is suitable for locks that may be held for longer periods of time.

AIX developers can choose between two types of locks: mutually exclusive simple locks that allow the process to spin while waiting for the lock to become available, and complex read-write locks that block the process while waiting for the lock to become available.

The rules about using locks are strictly conventions. Neither hardware nor software has an enforcement or checking mechanism. Although using locks has made AIX 4.1 MP Safe, it is the developer's responsibility to define and implement an appropriate locking strategy to protect their own global data.

Waiting for Locks

Figure 8 shows an application that causes the system to spend 10% of its time in a certain kernel component. Suppose that because of the complexity of that component, the developer decides to make the whole component one large critical section. That is, there is only one mutex lock for the whole component, and it is requested at all entry points in the component and released at all exit points. On a 4-way SMP, this mutex lock will be busy 40% (4 x 10%) of the time.

According to queuing theory, the busier a resource, the longer the average wait to get it. The relationship is non-linear—if the use of a lock is doubled, the average wait time for that lock more than doubles. In addition, if the use of the same lock were halved, the average wait time for that lock will be reduced by more than half.

Waiting always decreases system performance no matter how it is done. If a spin lock is used, the processor is busy but not doing useful work (not contributing to throughput). If a blocking lock is used, the overhead of context switching and dispatching and the consequent increase in cache misses is incurred.

Lock Granularity

The amount of time a given lock is busy is a function of how often it is requested and how long it is held once acquired. One of the most effective ways to reduce lock wait time is to reduce the size of what the lock is protecting: its granularity.

Figure 9 shows that lock holding time can be reduced by ensuring that it encompasses only the code that accesses shared data instead of all the code in a component. Locks should always be associated with specific data items or structures, not with components or routines.

When serializing access to large data structures, consider an array of locks, one for each element of the structure instead of one lock for the whole structure. This will reduce the frequency for which any one of the locks is requested.

Unfortunately, there is a counter-effect to reducing lock granularity. In the example in Figure 9, we suggest requesting and releasing a lock three times instead of once. In the best case, when a simple lock is free and we do not have to spin, each lock-unlock pair costs approximately 20 instructions. Therefore, in the example, we have added a minimum of approximately 40 instructions to our path length. Complex locks are even more expensive. Each uncontested lock-unlock pair costs approximately 125 instructions, and using them in our example would add approximately 250 instructions to our path length.

Figure 10 shows that if lock granularity is too fine, too many instructions are used to request and release the locks. If lock granularity is too coarse, too much time is spent waiting for the locks. The proper balance in each component must be discovered empirically. As an initial guideline for AIX 4.1, try to hold locks for no more than approximately 300 instructions.

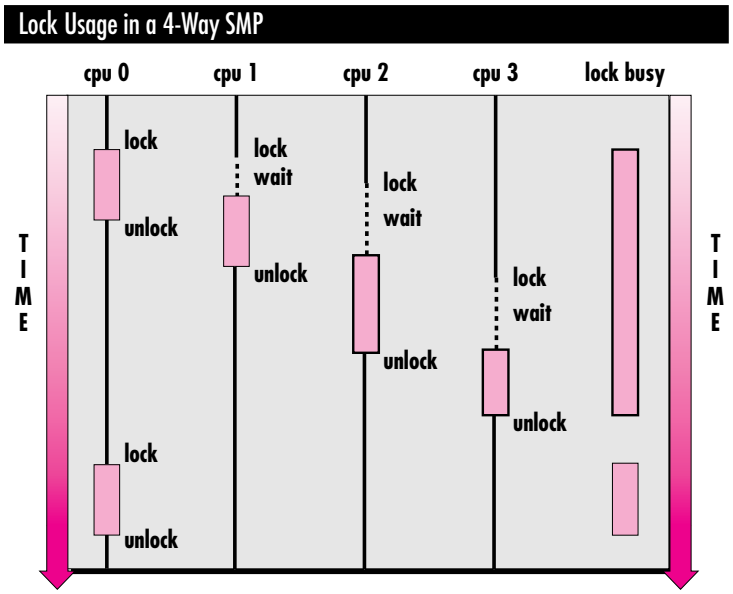

Figure 8. Lock usage in a 4-way SMP

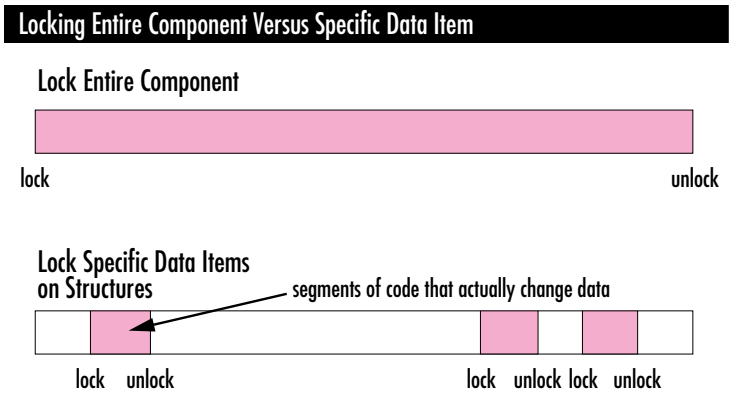

Figure 9. Comparison of locking an entire component versus a specific data item or structure

Performance Tips for Locks

The following are some performance tips for locks:

- Never perform synchronous I/O or any other blocking activity while holding a lock.
- Move all unnecessary instructions (those not directly related to reading or modifying the protected data) outside the critical section.
- If there is more than one access to the same data in a given component, try to move the accesses together so they can be covered by one lock-unlock pair (provided the combined code is under the 300 instruction limit).

Lock Granularity Effect on System Throughput

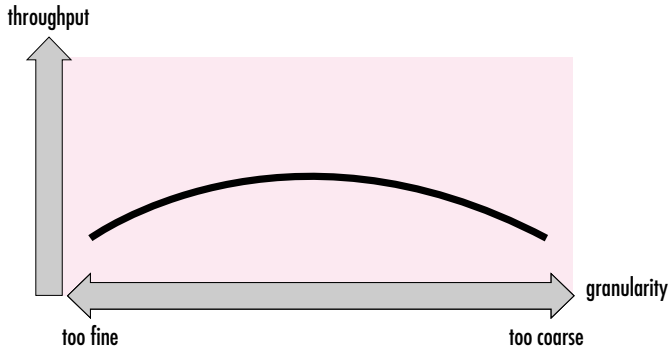

Figure 10. The effect of lock granularity on system throughput

- Do not put two locks in the same cache line. In fact, to reduce false sharing, do not put anything in the same cache line with a lock except data that it is protecting.
- If more than one lock must be held simultaneously, request the busiest one last (if allowed by the hierarchy).
- Avoid double wakeup. If some data must be modified under a lock and another process must be notified when the modification is completed, release the lock and then post the wakeup.
- If the protected data is read-mostly, consider using a complex lock instead of a simple one. However, be sure the extra path length in the normal, uncontested case will be justified.
- Start out with medium- to coarse-grained locks and reduce granularity only if improved performance is needed.

Lock Tuning in an Application

MP performance tuning is very difficult. It should be attempted only if the routine is used frequently and its performance is not satisfactory. Here are the steps to follow when tuning the locks in an application:

1. Use profiling to find where the routine is spending its time.

2. Within these areas, find the locks with the longest wait times. Both kinds of kernel locks are instrumented. When lock instrumentation is turned on, two counters are kept for each lock. The counters indicate how often the lock was used and the lock's wait times.

3. For each of these locks, apply the tuning tips above.

4. Repeat these steps until performance is satisfactory.

Processor Affinity

There is a single, common run queue in AIX 4.1 accessed by all processors. When the thread currently running on a processor blocks or its quantum expires, the highest priority thread that can be run will be dispatched on that processor.

One consequence of this dispatching policy is that over its lifetime a thread bounces around, running first on one processor and then another. Each time it starts executing on a different processor, it suffers cache misses until it has brought its own instructions and data into the cache. If the thread has not been blocked long, some of its instructions and data may still be in the cache of the last processor on which it ran. In this case, it would be more efficient to run the thread there again instead of on a different processor.

The policy of trying to run a thread on the same processor as it ran last is called *processor affinity*. AIX 4.1 has only a very weak form of processor affinity: if two or more threads on the run queue are tied for highest priority, a processor will choose the one that ran on it most recently.

Binding

The strongest form of processor affinity is to "bind" a process or thread to one processor and never run it on any other processor, even if its processor is busy and another is idle. The `bindprocess()` system call allows an application to do this. This might be useful for a process that seldom blocks for long periods and whose response time is important. Binding a process to one processor may help that process, but it can hurt overall system throughput because it sometimes forces a processor to remain idle when it could be doing useful work. Binding is appropriate only in special circumstances, such as on a system that is dedicated to a single application.

Two Ways to Parallelize

MPs are useful only if performance improvements can be realized by processes using an MP instead of a uniprocessor. In an environment where many applications or processors execute independently of each other, performance improvements are realized by allowing the multiple applications to execute simultaneously on separate processors. However, in computing environments dominated by a single application,

parallelizing the application can often provide the improved performance being sought. *Parallelizing* an application means breaking it up into pieces to allow useful work to be done on more than one processor simultaneously.

Processes

The traditional way to parallelize an application has been to break it into multiple processes. These processes communicate using either an Interprocess Communication (IPC) mechanism, such as a pipe, or shared memory. The processes must be able to block waiting for events such as messages from other processes, and they must coordinate access to shared objects with something like locks.

Threads

We can now do much the same thing with threads, which have exactly the same coordination problems and similar mechanisms to deal with them. Using the common POSIX™ interface makes threaded applications portable.

Threads are separate loci of control within one process. A thread is a dispatchable unit with its own program counter, register save area, and stack that shares with its sibling threads most other resources belonging to the process, including memory and open files.

Threads are the only dispatchable unit in AIX 4.1. When a process is created, one thread is automatically created within it. Additional threads are created by explicit calls made by existing threads, such as `pthread_create()` in the pthread library.

A single process can now have any number of its threads running simultaneously on different processors. Coordinating them and serializing access to shared data are the developer's responsibility.

Threads and processes each have their own advantages to be considered when determining which method to use for parallelizing an application. Threads may be faster than processes, and memory sharing is easier with threads than with processes. Because memory sharing must be done explicitly when using processes, the process model provides help with data protection. In addition, a process implementation will distribute more easily to multiple machines (such as client/server) or clusters and will run on AIX 3.2.5.

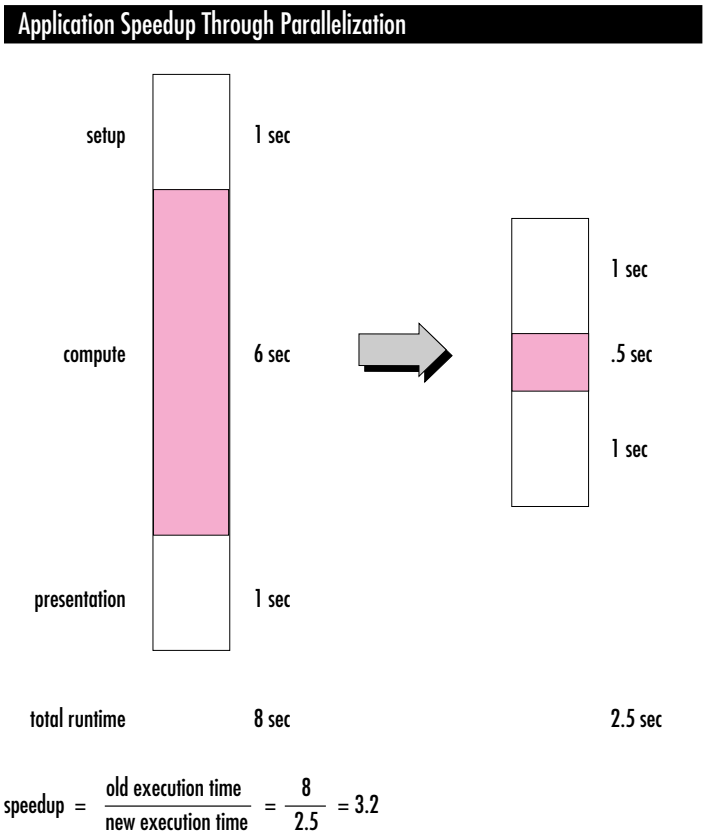

Figure 11. Application speedup through parallelization

Amdahl's Law

Amdahl's Law[1] quantifies the fact that if only part of a program is sped up, the part that was not sped up still runs as slowly as ever. Suppose the program shown in Figure 11 spends one second reading data and doing setup, six seconds computing, and one second producing graphs. Suppose also that the computing part can be parallelized so that it runs 12 times faster.

Making the main part of the program run 12 times faster sounds good, but only makes the program run 3.2 times faster. In fact, if the compute phase of this program could be made to run infinitely fast so that it took zero seconds, the program would still only run 8/2 or 4 times faster. This is an upper bound on the speedup that can be achieved by parallelizing the compute phase of this application.

[1] Schutzer, Daniel. *Parallel Processing and the Future Data Center.* New York: Van Nostrand Reinhold. 1994.

Amdahl's Law

$$\text{Speedup} = \frac{\text{Execution time}_{old}}{\text{Execution time}_{new}}$$

$$= \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

Figure 12. Amdahl's law

Application Speedup Prediction

$$\text{Speedup} = \frac{1}{(1 - .75) + \frac{.75}{12}}$$

$$= \frac{1}{.25 + .0625}$$

$$= 3.2$$

Figure 13. Application speedup prediction

The formula for Amdahl's Law is shown in Figure 12 where $\text{Fraction}_{enhanced}$ is the fraction of the original path length we managed to speed up, and $\text{Speedup}_{enhanced}$ is the amount by which we managed to speed up that fraction.

Amdahl's formula can be used to predict overall speedup before a change is implemented and $\text{Execution time}_{new}$ can be measured. Figure 13 illustrates how overall speedup for the previous example could have been predicted.

Should I Parallelize My Application?

Many developers will ask the question "Should I parallelize my application?" There is no easy answer to this question. Parallelizing a substantial application is not easy and should be undertaken only if the following three criteria are met.

1. The application really needs to be speeded up and it cannot be done another way. Some likely candidates are as follows:

- Current response time is too slow. For example, a simulation or decision-support program takes hours, but the answers are only useful if they can be delivered in less than one hour. On Wall Street, evaluating certain financial instruments is only useful if it can be done in a few seconds.
- Current throughput is inadequate and needs to be increased, such as in an Online Transaction Processing (OLTP) environment
- An application needs to be enhanced, but still run in the same amount of time (such as a more accurate simulation or searching longer lists)

2. There is enough inherent parallelism in the problem. Finding and capitalizing on this parallelism may require redesigning and reimplementing its solution. A significant portion of the problem must be parallelizable. Remember Amdahl's Law!

- Data parallelism (good applications include OLTP, image processing, searching, and so on)
- Algorithm parallelism (good environments include client/server, pipelining, simulating different cases in parallel, and others)

3. You will need to learn parallel programming, invest the necessary resources, and suffer the inevitable frustration. Parallelizing a substantial application is not easy, but it may provide developers the needed performance improvements.

Debora Blakely-Fogel, *IBM Corporation, RISC System/6000 Division, 11400 Burnet Road, Austin, TX 78758. Ms. Blakely-Fogel is an advisory programmer responsible for providing technical assistance to software vendors. She has a BS in Mathematics from the University of Massachusetts at Lowell and an MS in Computer Science from New Mexico State University.*