Selecting Locking Designs for Parallel Programs

Paul E. McKenney
Sequent Computer Systems, Inc.

Abstract

Parallelizing a program can greatly speed it up. However, the synchronization primitives needed to protect shared resources result in contention, overhead, and added complexity. These costs can overwhelm the performance benefits of parallel execution. Since the only reason to go to the trouble of parallelizing a program is to gain performance, it is necessary to understand the performance implications of synchronization primitives *in addition to* their correctness, liveness, and safety properties.

This paper presents a pattern language to assist you in selecting appropriate locking designs for parallel programs.

Section 1 presents the example that is used throughout the paper to demonstrate use of the patterns. Section 2 overviews contexts in which these patterns are useful. Section 3 describes the forces common to all of the patterns. Section 4 presents several indexes to the patterns. Section 5 presents the patterns themselves.

1 Example Algorithm

A simple hashed lookup table is used to illustrate the patterns in this paper. This example searches for a specified key in a list of elements and performs operations on those elements. Both the individual elements and the list itself may require mutual exclusion.

The data structure for a monoprocessor implementation is shown in Figure 1. The `lt_next` field links

```
typedef struct looktab {
        struct looktab_t *lt_next;
        int     lt_key;
        int     lt_data;
} looktab_t;
```

Figure 1: Lookup-Table Element

the individual elements together, the `lt_key` field contains the search key, and the `lt_data` field contains the data corresponding to that key.

Again, this structure will be embellished as needed for each of the example uses with synchronization.

A search for the element with a given key might be implemented as shown in Figure 2.

```
/* Header for list of looktab_t's. */

looktab_t *looktab_head[LOOKTAB_NHASH] =
        { NULL };
#define LOOKTAB_HASH(key) \
        ((key) % LOOKTAB_NHASH)

/*
 * Return a pointer to the element of the
 * table with the specified key, or return
 * NULL if no such element exists.
 */

looktab_t *
looktab_search(int key)
{
        looktab_t *p;

        p = looktab_head[LOOKTAB_HASH(key)];
        while (p != NULL) {
                if (p->lt_key == key) {
                        return (p);
                }
                p = p->lt_next;
        }
        return (NULL);
}
```

Figure 2: Lookup-Table Search

2 Overview of Context

Use the patterns in this paper for development or maintenance efforts when parallelization is a central

issue, for example, when you **Share the Load** [Mes95] among several processors in order to improve capacity of reactive systems.

For new development, use the following steps:

1. Analyze architecture/design:

   (a) Identify activities that are good candidates for parallelization.

   (b) Identify shared resources.

   (c) Identify communications and synchronization requirements.

   These items combined will define the critical sections. You must guard all critical sections with synchronization primitives in order for the program to run correctly on a parallel processor.

2. Use the patterns in this paper to select a locking design.[1]

3. Measure the results. If they are satisfactory, you are done! Otherwise, proceed with the maintenance process below.

Use prototypes as needed to help identify candidate activities. Relying on the intuition of architects and designers is risky, especially when they are solving unfamiliar problems. If the prototype cannot be scaled up to production levels, use differential profiling [McK95] to help predict which activities are most prone to scaling problems.

For maintenance, use the following steps:

1. Measure current system.

2. Analyze measurements to identify bottleneck activities and resources.

3. Use the patterns in this paper to select a locking design.

4. Measure the results. If they are satisfactory, you are done! Otherwise, repeat this process.

Use the pattern language presented in this paper for step 2 of the development procedure and step 3 of the maintenance procedure. The other steps of both procedures are the subject of other pattern languages.

[1] It may also be necessary to select designs for communication, notification, and versioning. Pattern languages for these aspects of design are the subject of another paper.

3 Forces

The forces acting on the performance of parallel programs are speedup, contention, overhead, read-to-write ratio, economics, and complexity:

Speedup: Getting a program to run faster is the only reason to go to all of the time and trouble required to parallelize it. Speedup is defined to be the ratio of the time required to run a sequential version of the program to the time required to run a parallel version.

Contention: If more CPUs are applied to a parallel program than can be kept busy given that program, the excess CPUs are prevented from doing useful work by contention.

Overhead: A monoprocessor version of a given parallel program would not need synchronization primitives. Therefore, any time consumed by these primitives is overhead that does not contribute directly to the useful work that the program is intended to accomplish. Note that the important measure is the relationship between the synchronization overhead and the serial overhead–critical sections with greater overhead may tolerate synchronization primitives with greater overhead.

Read-to-Write Ratio: A data structure that is mostly rarely updated may often be protected with lower-overhead synchronization primitives than may a data structure with a high update rate.

Economics: Budgetary constraints can limit the number of CPUs available regardless of the potential speedup.

Complexity: A parallel program is more complex than an equivalent sequential program because the parallel program has a much larger state space than does the sequential program. A parallel programmer must consider synchronization primitives, locking design, critical-section identification, and deadlock in the context of this larger state space.

This greater complexity often (but not always!) translates to higher development and maintenance costs. Therefore, budgetary constraints can limit the number and types of modifications made to an existing program – a given degree of speedup is worth only so much time and trouble.

These forces will act together to enforce a maximum speedup. The first three forces are deeply interrelated, so the remainder of this section analyzes these interrelationships.[2]

Note that these forces may also appear as part of the context. For example, economics may act as a force ("cheaper is better") or as part of the context ("the cost must not exceed $100").

An understanding of the relationships between these forces can be very helpful when resolving the forces acting on an existing parallel program.

1. The less time a program spends in critical sections, the greater the potential speedup.

2. The fraction of time that the program spends in a given critical section must be much less than the reciprocal of the number of CPUs for the actual speedup to approach the number of CPUs. For example, a program running on 10 CPUs must spend much less than one tenth of its time in the critical section if it is to scale well.

3. Contention effects will consume the excess CPU and/or wallclock time should the actual speedup be less than the number of available CPUs. The larger the gap between the number of CPUs and the actual speedup, the less efficiently the CPUs will be used. Similarly, the greater the desired efficiency, the smaller the achievable speedup.

4. If the available synchronization primitives have high overhead compared to the critical sections that they guard, the best way to improve speedup is to reduce the number of times that the primitives are invoked (perhaps by fusing critical sections, using data ownership, or by moving toward a more coarse-grained parallelism such as code locking).

5. If the critical sections have high overhead compared to the primitives guarding them, the best way to improve speedup is to increase parallelism by moving to reader/writer locking, data locking, or data ownership.

6. If the critical sections have high overhead compared to the primitives guarding them and the data structure being guarded is read much more often than modified, the best way to increase parallelism is to move to reader/writer locking.

[2] A real-world parallel system will have many additional forces acting on it, such as data-structure layout, memory size, memory-hierarchy latencies, and bandwidth limitations.

4 Index to Patterns for Selecting Locking Patterns

This section contains indexes based on relationships between the patterns (Section 4.1), forces resolved by the patterns (Section 4.2), and problems commonly encountered in parallel programs (Section 4.3).

4.1 Pattern Relationships

Section 5 presents the following patterns:

1. Sequential Program (5.1)
2. Code Locking (5.2)
3. Data Locking (5.3)
4. Data Ownership (5.4)
5. Parallel Fastpath (5.5)
6. Reader/Writer Locking (5.6)
7. Hierarchical Locking (5.7)
8. Allocator Caches (5.8)
9. Critical-Section Fusing (5.9)
10. Critical-Section Partitioning (5.10)

Relationships between these patterns are shown in Figure 3 and are described in the following paragraphs.

Parallel Fastpath, Critical-Section Fusing, and Critical-Section Partitioning are meta-patterns.

Reader/Writer Locking, Hierarchical Locking, and Allocator Caches are instances of the Parallel Fastpath meta-pattern. Reader/Writer Locking and Hierarchical Locking are themselves meta-patterns; they can be thought of a modifiers to the Code Locking and Data Locking patterns. Parallel Fastpath, Hierarchical Locking, and Allocator Caches are ways of combining other patterns and thus are template patterns.

Critical-Section Partitioning transforms Sequential Program into Code Locking and Code Locking into Data Locking. It also transforms conservative Code Locking and Data Locking into more aggressively parallel forms.

Critical-Section Partitioning transforms Data Locking into Code Locking and Code Locking into Sequential Program. It also transforms aggressive Code Locking and Data Locking into more conservative forms.

Assigning a particular CPU or process to each partition of a data-locked data structure results in Data Ownership. A similar assignment of a particular CPU,

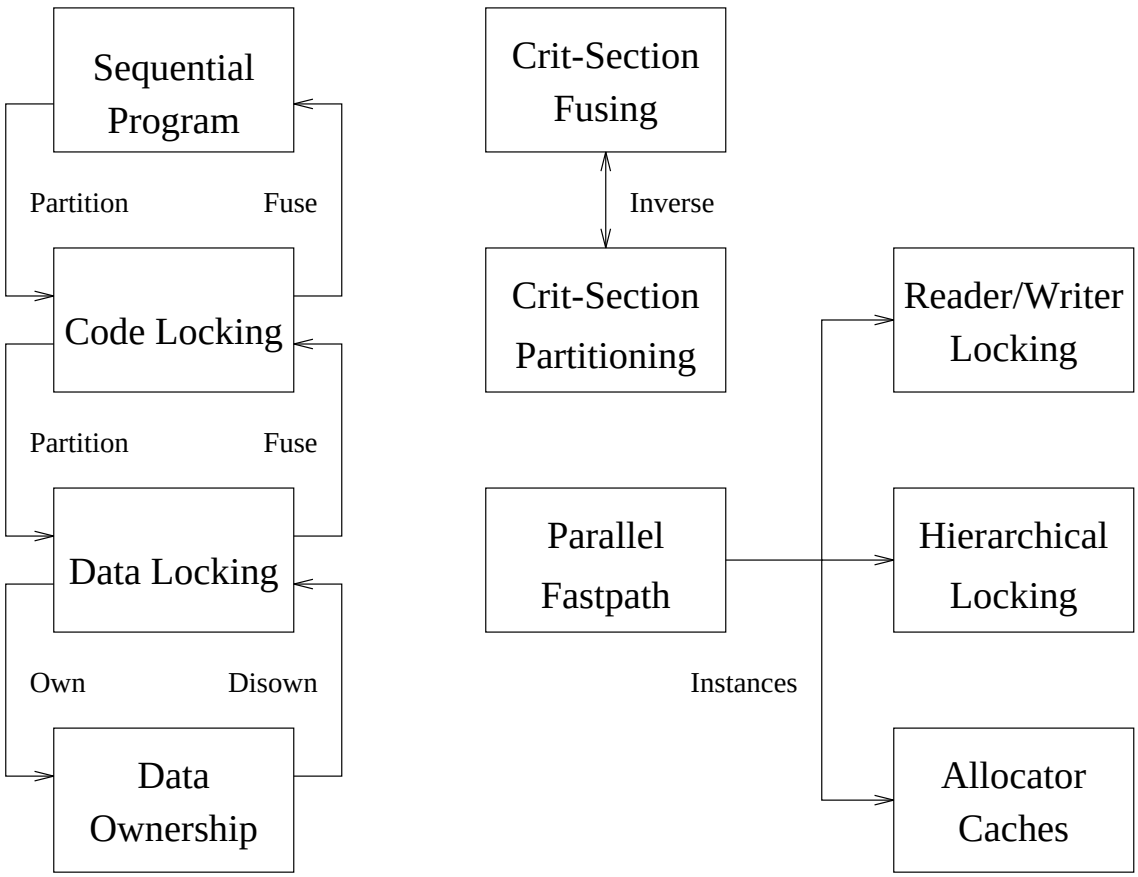

Figure 3: Relationship Between Pattern

process, or computer system to each critical section of a code-locked program results in Client/Server (used heavily in distributed systems but not described in this paper).

4.2 Force Resolution

Table 4.2 compares how each of the patterns resolves each of the forces. Plus signs indicate that a pattern resolves a force well. For example, Sequential Program resolves Contention and Overhead perfectly due to lack of synchronization primitives, Budget perfectly since sequential programs run on cheap single-CPU machines, and Complexity perfectly because sequential implementations of programs are better understood and more readily available than are parallel versions.

Minus signs indicate that a pattern resolves a force poorly. Again, Sequential Program provides extreme examples with Speedup since a sequential program allows no speedup[3] and with Read-to-Write Ratio because multiple readers cannot proceed in parallel in a sequential program.

Question marks indicate that the quality of resolution is quite variable. Programs based on Data Ownership can be extremely complex if CPUs must access each other's data. If no such access if needed, the programs can be as trivial as a script running multiple instances of a sequential program in parallel.

See the individual patterns for more information on how they resolve the forces.

4.3 Fault Table

Use Table 4.3 to locate a replacement pattern for a pattern that is causing more problems than it is solving.

5 Patterns for Selecting Locking Designs

5.1 Sequential Program

Problem How can you eliminate the complexity of parallelization?

Context An existing parallel program with excessive complexity that runs fast enough on a single processor.

[3]If you run multiple instances of a sequential program in parallel, you have used Data Locking or Data Ownership instead of Sequential Program.

Forces

- Contention (+ + +): There is no parallel execution, so there is absolutely no contention.
- Overhead (+ + +): There are no synchronization primitives, so there is no synchronization overhead.
- Economics (+ + +): The program runs on a single CPU. Single-CPU systems are usually cheaper than parallel systems.
- Complexity (+ + +): Pure sequential execution eliminates all parallel complexity. In fact, many sequential programs are available as freeware. You need only understand how to install such programs to run them sequentially. In contrast, you would need to intimately understand the program, your machine, and parallelism in order to produce a parallel version.
- Speedup (− − −): There is no parallel execution, so there is absolutely no speedup.
- Read-to-Write Ratio (− − −): There is no parallel execution, so there is absolutely no opportunity to allow readers to proceed in parallel.

Solution Construct an entirely sequential program. Eliminate all use of synchronization primitives, thereby eliminating the overhead and complexity associated with them.

Resulting Context A completely sequential program with no complexity, overhead, contention, or speedup from parallelization.

Design Rationale If the program runs fast enough on a single processor, remove the synchronization primitives and spare yourself their overhead and complexity.

5.2 Code Locking

Problem How can you parallelize and maintain existing third-party code in an inexpensive and efficient way?

Context An existing sequential program that has only one resource, whose critical sections are very small compared to the parallel portion of the code, of which only modest speedups are required, or whose development- or maintenance-cost constraints rule out more complex and effective parallelization techniques.

Speedup	Contention	Overhead	R/W	$	Complexity	Pattern
− − −	+ + +	+ + +	− − −	+ + +	+ + +	Sequential Program (5.1)
0	−−	0	− − −	0	++	Code Locking (5.2)
+	+	0	+	−	−−	Data Locking (5.3)
+ + +	+ + +	+?	+	− − −	?	Data Ownership (5.4)
++	++	++	+	−−	−−	Parallel Fastpath (5.5)
++	+	+	+ + +	−−	−	Reader/Writer Locking (5.6)
+	++	−	0	−	− − −	Hierarchical Locking (5.7)
++	++	++	N/A	−−	−	Allocator Caches (5.8)
0	−	+	0	0	−−	Critical-Section Fusing (5.9)
0	+	−	0	0	+	Critical-Section Partitioning (5.10)

Forces

- Complexity (++): The monitor-like structure of code-locked programs is more easily understood.
- Speedup (0): Code locking usually permits only modest speedups.
- Overhead (0): Code locking uses a modest number of synchronization primitives and so suffers only modest overhead.
- Economics (0): Code locking scales modestly, so requires a system with few CPUs.
- Contention (−−): The simple locking pattern usually results in high contention.
- Read-to-Write Ratio (− − −): There is no parallel execution of critical sections sharing a specific resource, so there is absolutely no opportunity to allow readers to proceed in parallel.

The required speedup may be modest, or the contention and overhead present may be negligible, thereby allowing adequate speedups.

Economics may prohibit applying more than a small number of CPUs to a problem, thereby rendering high speedups irrelevant.

Highly parallel code can be prohibitively complex. In many cases, the less-complex code-locking approach is preferable.

Solution Parallelize and maintain existing third-party code inexpensively and efficiently using code locking when the code spends a very small fraction of its execution in critical sections or when you require only modest scaling.

See Figure 4 for an example. Note that calls to the `looktab_search()` function and subsequent uses of the return value must be surrounded by a synchronization primitive as shown in the figure. Do not try to bury the locks in the `looktab_search()`, since this would allow multiple CPUs to update the element simultaneously, which would corrupt the data value. Instead, hide the locking within higher level functions (member functions in an object-oriented implementation) that call `looktab_search()` as part of specific, complete operations on the table.

```
/*
 * Global lock for looktab_t
 * manipulations.
 */

slock_t looktab_mutex;

. . .

/*
 * Look up a looktab element and
 * examine it.
 */

S_LOCK(&looktab_mutex);
p = looktab_search(mykey);

/*
 * insert code here to examine or
 * update the element.
 */

S_UNLOCK(&looktab_mutex);
```

Figure 4: Code-Locking Lookup Table

It is relatively easy to create and maintain a code-locking version of a program. No restructuring is needed, only the (admittedly non-trivial) task of inserting the locking operations.

Old Pattern	Problem	Pattern to use or apply
Sequential Program (5.1)	Need faster execution.	Code Locking (5.2) Data Locking (5.3) Data Ownership (5.4) Parallel Fastpath (5.5) Reader/Writer Locking (5.6) Hierarchical Locking (5.7) Allocator Caches (5.8)
Code Locking (5.2)	Speedup limited by contention.	Data Locking (5.3) Data Ownership (5.4) Parallel Fastpath (5.5) Reader/Writer Locking (5.6) Hierarchical Locking (5.7) Allocator Caches (5.8)
Code Locking (5.2) Data Locking (5.3) Hierarchical Locking (5.7)	Speedup limited by synchronization overhead or by both contention and synchronization overhead.	Data Ownership (5.4) Parallel Fastpath (5.5) Reader/Writer Locking (5.6) Allocator Caches (5.8)
Sequential Program (5.1) Code Locking (5.2) Data Locking (5.3) Reader/Writer Locking (5.6)	Speedup limited by contention and synchronization cheap compared to non-critical-section code in critical sections.	Critical-Section Partitioning (5.10)
Code Locking (5.2) Data Locking (5.3) Reader/Writer Locking (5.6)	Speedup limited by synchronization overhead and contention is low.	Critical-Section Fusing (5.9)
Code Locking (5.2) Data Locking (5.3)	Speedup limited by contention and readers could proceed in parallel.	Reader/Writer Locking (5.6)
Data Locking (5.3) Data Ownership (5.4) Parallel Fastpath (5.5) Reader/Writer Locking (5.6) Hierarchical Locking (5.7) Allocator Caches (5.8)	Speedup greater than necessary and complexity is too high (e.g., takes too long to merge changes for new version of program).	Sequential Program (5.1) Code Locking (5.2)
Code Locking (5.2)	Speedup greater than necessary and complexity is too high (e.g., takes too long to merge changes for new version of program).	Sequential Program (5.1)
Data Ownership (5.4)	Complexity of passing operations to the data is too high (e.g., takes too long to merge changes for new version of program).	Data Locking (5.3)

Table 1: Locking-Design Fault Table

This program might scale well if the table search and update was a very small part of the program's execution time.

Resulting Context A moderately parallel program that is very similar to its sequential counterpart, resulting in relatively added complexity.

Design Rationale Code locking is the simplest locking design. It is especially easy to retrofit an existing program to use code locking in order to run it on an multiprocessor. If the program has only a single shared resource, code locking will even give optimal performance. However, most programs of any size and complexity require much of the execution to occur in critical sections, which in turn sharply limits the scaling as specified by Amdahl's Law.

Therefore, use code locking on programs that spend only a small fraction of their run time in critical sections or from which only modest scaling is required. In these cases, code locking will provide a relatively simple program that is very similar to its sequential counterpart.

5.3 Data Locking

Problem How can you obtain better speedups than straightforward parallelizations such as code locking can provide?

Context An existing sequential or parallel program requiring greater speedup than can be obtained via code locking, and whose data structures may be split up into independent partitions, so that different partitions may be operated on in parallel.

Forces

- Speedup (+): Data locking associates different locks with different data structures or with different parts of data structures. Contention and overhead still limit speedups.
- Contention (+): Accesses to different data structures proceed in parallel. However, accesses to data guarded by the same lock still result in contention.
- Read-to-Write Ratio (+): Readers accessing different data structures proceed in parallel. However, readers attempting to access structures guarded by the same lock are still serialized.
- Overhead (0): A data locked program usually uses about the same number of synchronization primitives to perform a given task as a code-locked program does.
- Economics (−): Greater speedups require bigger machines.
- Complexity (−−): Data locked programs can be extremely complex and subtle, particularly when critical sections must be nested, leading to deadlock problems.

A greater speedup than can be obtained via code locking must be needed badly enough to invest the development, maintenance, and machine resources required for more aggressive parallelization.

The program must also be partitionable so as to allow data locking to be used to reduce contention and overhead.

Solution Partition the data structures to allow each of the resulting portions to be processed in parallel. Data locking often results in better speedups than straightforward parallelizations such as code locking can provide. The following paragraphs present two examples of data locking.

The first example is trivial but important: lock on instances of data structures rather than on the code operating on those data structures.[4] For example, if there were several independent lookup tables in a program, each could have its own critical section, as shown in Figure 5. This figure assumes that `looktab_search()` has been modified to take an additional pointer to the table to be searched. Structuring the code in this manner allows searches of different tables to proceed in parallel, although searches of a particular table will still be serialized.

In the second example, allocate a lock for each hash line in the hash-line-header data structure,[5] as shown in Figure 6. This may be done if the individual elements in a table are independent from one another, and allows searches of a single table to proceed in parallel if the items being searched for do not collide after being hashed.

A major difference between these two examples is the degree to which locking considerations have been allowed to constrain the design. You can change the first example's design to use, say, a linked binary search tree, without changing the locking design. A

[4] Or, in object-oriented terminology, lock on instances of a class rather than the class itself.

[5] In object-oriented programs, this level of data locking results in many independent critical sections per instance.

```
/* Global lock for looktab_t manipulations. */

slock_t my_looktab_mutex;
looktab_t *my_looktab[LOOKTAB_NHASH];

. . .

/* Look up a looktab element and examine it. */

S_LOCK(&my_looktab_mutex);
p = looktab_search(my_looktab, mykey);

/* insert code here to examine or update the element. */

S_UNLOCK(&my_looktab_mutex);
```

Figure 5: Partitioned Lookup Table

single lock still guards a single table regardless of that table's implementation.

However, the second example's locking design forces you to use a hashing scheme. The only degree of freedom is your choice of design for the overflow chains. The benefit of this constraint is unlimited speedup–you can increase speedup simply by increasing the size of the hash table.

In addition, you can combine these two techniques to allow fully parallel searching within and between lookup tables.

Resulting Context A more heavily parallel program that is less similar to its sequential counterpart, but which exhibits much lower contention and overhead and thus higher speedups.

Design Rationale Many algorithms and data structures may be partitioned into independent parts, with each part having its own independent critical section. Then the critical sections for each part can execute in parallel (although only one instance of the critical section for a given part could be executing at a given time). Use data locking when critical-section overhead must be reduced, and where synchronization overhead is not limiting speedups. Data locking reduces this overhead by distributing the instances of the overly-large critical section into multiple critical sections.

5.4 Data Ownership

Problem How can you make programs with frequent, small critical sections achieve high speedups on machines with high synchronization overheads?

Context An existing sequential or parallel program that can be partitioned so that only one process accesses a given data item at a time, thereby greatly reducing or eliminating the need for synchronization.

Forces

- Speedup (+ + +): Since each CPU operates on its data independently of the other CPUs, Data Ownership provides excellent speedups.
- Contention (+ + +): No CPU accesses another CPU's data, so there is absolutely no contention.
- Read-to-Write Ratio (+): Each CPU can read its own data independently of the other CPUs, but since only one CPU owns a given piece of data, only one CPU may read a given piece of data at a time.
- Overhead (+?): In the best case, the CPUs operate independently, free of overhead. However, if CPUs must share information, this sharing will exact some sort of communications or synchronization overhead.
- Complexity (???): In the best case, a program constructed using Resource Ownership is simply

```
/*
 * Type definition for looktab header.
 * All fields are protected by lth_mutex.
 */

typedef struct looktab_head {
        looktab_t *lth_head;
        slock_t *lth_mutex;
} looktab_head_t;

/* Header for list of looktab_t's. */

looktab_head_t *looktab_head[LOOKTAB_NHASH] = { NULL };
#define LOOKTAB_HASH(key) ((key) % LOOKTAB_NHASH)

/*
 * Return a pointer to the element of the table with the
 * specified key, or return NULL if no such element exists.
 */

looktab_t *
looktab_search(int key)
{
        looktab_t *p;

        p = looktab_head[LOOKTAB_HASH(key)].lth_head;
        while (p != NULL) {
                if (p->lt_key > key) {
                        return (NULL);
                }
                if (p->lt_key == key) {
                        return (p);
                }
                p = p->lt_next;
        }
        return (NULL);
}

/* . . . */

        /* Look up a looktab element and examine it. */

        S_LOCK(&looktab_head[LOOKTAB_HASH(mykey)].lth_mutex);
        p = looktab_search(mykey);

        /* insert code here to examine or update the element. */

        S_UNLOCK(&looktab_head[LOOKTAB_HASH(mykey)].lth_mutex);
```

Figure 6: Data Locking

a set of sequential programs run independently in parallel. The main issue here is how to balance the load among these independent programs. For example, if 90% of the work is assigned to one CPU, then there will be at most a 10% speedup.

In more complex cases, the CPUs must access each other's data. In these cases, arbitrarily complex sharing mechanisms must be designed and implemented. For data ownership to be useful, CPUs must access their own data almost all the time. Otherwise, the overhead of sharing can overwhelm the benefit of ownership.

- Economics (– – –): Excellent speedups require lots of equipment. This means lots of money, even if the equipment is a bunch of cheap PCs connected to a cheap LAN.

A high speedup must be needed badly enough to invest the time and effort needed to develop, maintain, and run a highly parallel version.

The program must be perfectly partitionable, eliminating contention and synchronization overhead.

Solution Use data ownership to partition data used by programs with frequent, small critical sections running on machines with high synchronization overheads. This partitioning can eliminate mutual-exclusion overhead, thereby greatly increasing speedups.

The last example of the previous section splits a lookup table into multiple independent classes of keys, where the key classes are defined by a hash function.

If each key class can be assigned to a separate process, we have a special case of partitioning known as data ownership. Data ownership is a very powerful pattern, as it can entirely eliminate synchronization overhead. This is particularly useful in programs that have small critical sections whose overhead is dominated by synchronization overhead. If the key classes are well balanced, data ownership can allow virtually unlimited speedups.

Partitioning the lookup-table example over separate processes would require a convention or mechanism to force lookups for keys of a given class to be performed by the proper process. For example, Figure 7 shows how a simple modulo mapping from key to CPU. The `on_cpu()` function is similar to an RPC call; the function is invoked with the specified arguments on the specified CPU.

Note that the form of the algorithm has been changed considerably. The original serial form passed back a pointer that the caller could use as it saw fit. However, data ownership requires that a specific CPU perform all operations on a particular data structure. We therefore cannot pass a reference to data owned by one CPU to the CPU that wants to operate on that data. We instead must pass the operation to the CPU that owns the data structure.[6] This operation passing usually requires synchronization, so we have simply traded one form of overhead for another. This tradeoff may neverthless be worthwhile if each CPU processes its own data most of the time, particularly if each data structure is large so that the overhead of passing the operation is small compared to that of moving the data. For example, a version of OSF/1 for massively-parallel processors used a `vproc` layer to direct operations (such as UNIX signals) to processes running on other nodes [ZRB+93].

Data ownership might seem arcane, but it is used very frequently:

1. Any variables accessible by only one CPU or process (such as `auto` variables in C and C++) are owned by that CPU or process.

2. An instance of a user interface owns the corresponding user's context. It is very common for applications interacting with parallel database engines to be written as if they were entirely sequential programs. Such applications own the user interface and his current action. Explicit parallelism is thus confined to the database engine itself.

3. Parametric simulations are often trivially parallelized by granting each CPU ownership of its own region of the parameter space.

This is the paradox of data ownership. The more thoroughly you apply it to a program, the more complex the program will be. However, if you structure a program to use *only* data ownership, as in these three examples, the resulting program can be *identical* to its sequential counterpart.

Resulting Context A more heavily parallel program that is often even less similar to its sequential counterpart, but which exhibits much lower contention and overhead and thus higher speedups.

[6] The Active Object pattern [Sch95] describes an object-oriented approach to this sort of operation passing. More complex operations that atomicly update data owned by many CPUs must use a more complex operation-passing approach such as two-phase commit [Tay87].

```
/* Header for list of looktab_t's. */

looktab_t *looktab_head[LOOKTAB_NHASH] = { NULL };
#define LOOKTAB_HASH(key) \ ((key) % LOOKTAB_NHASH)

/*
 * Look up the specified entry and invoke the specified
 * function on it.  The key must be one of this CPU's keys.
 */
looktab_t *
looktab_srch_me(int key, int (*func)(looktab_t *ent))
{
        looktab_t *p;

        p = looktab_head[LOOKTAB_HASH(key)];
        while (p != NULL) {
                if (p->lt_key == key) {
                        return (*func)(p);
                }
                p = p->lt_next;
        }
        return (func(NULL));
}

/*
 * Look up the specified entry and invoke the specified
 * function on it.  Force this to happen on the CPU that
 * corresponds to the specified key.
 */

int
looktab_search(int key, int (*func)(looktab_t *ent))
{
        int which_cpu = key % N_CPUS;

        if (which_cpu == my_cpu) {
            return (looktab_srch_me(key, func));
        }
        return (on_cpu(which_cpu, key, func));
}
```

Figure 7: Partitioning Key Ranges

Design Rationale Data ownership allows each CPU to access its data without incurring synchronization overhead. This can result in perfectly linear speedups. Some programs will still require one CPU to operate on another's CPU's data, and these programs must pass the operation to the owning CPU. The overhead of this operation passing will limit the speedup, but may be better than directly sharing the data.

5.5 Parallel Fastpath

Problem How can you achieve high speedups in programs that cannot use aggressive locking patterns throughout?

Context An existing sequential or parallel program that can use an aggressive locking patterns for the majority of its workload (the fastpath), but that must use a more conservative patterns for a small part of its workload.

Forces

- Speedup (++): Parallel fastpaths have very good speedups.
- Contention (++): Since the common-case fastpath code uses an aggressive locking design, contention is very low.
- Overhead (++): Since the common-case fastpath uses either lightweight synchronization primitives or omits synchronization primitives altogether, overhead is very low.
- Read-to-Write Ratio (+): The parallel fastpath may allow readers to proceed in parallel.
- Economics (−−): Higher speedups require larger, more expensive systems with more CPUs.
- Complexity (−−): Although the fastpath itself is often very straightforward, the off-fastpath code must handle complex recovery in cases that the fastpath cannot handle.

A high speedup must be needed badly enough to invest the resources needed to develop, maintain, and run a highly parallel version.

The program must be highly partitionable and must have its speedup limited by synchronization overhead and contention. The fraction of the execution time that cannot be partitioned limits the speedup. For example, if the off-fastpath code uses code locking and 10% of the execution time occurs off the fastpath, then the maximum achievable speedup will be 10. Either the off-fastpath code must not be executed very frequently or it must itself use a more aggressive locking pattern.

Solution Use parallel fastpaths to aggressively parallelize the common case without incurring the complexity required to aggressively parallelize the entire algorithm.

You must understand not only the specific algorithm you wish to parallelize, but also the workload that the algorithm will be subjected to. Great creativity and design effort is often required to construct a parallel fastpath.

Resulting Context A more heavily parallel program that is even less similar to its sequential counterpart, but which exhibits much lower contention and overhead and thus higher speedups.

Design Rationale Parallel fastpath allows the common case to be fully partitioned without requiring that the entire algorithm be fully partitioned. This allows scarce design and coding effort to be focused where it will do the most good.

Parallel fastpath combines different patterns (one for the fastpath, one elsewhere) and is therefore a template pattern. The following instances of parallel fastpath occur often enough to warrant their own patterns:

1. Reader/Writer Locking (5.6).
2. Hierarchical Locking (5.7).
3. Resource Allocator Caches (5.8).

5.6 Reader/Writer Locking

Problem How can programs that rarely modify shared data improve their speedups?

Context An existing program with much of the code contained in critical sections, but where the read-to-write ratio is large.

Forces

- Read-to-Write Ratio (+ + +): Reader/Writer Locking takes full advantage of favorable read-to-write ratios.
- Speedup (++): Since readers proceed in parallel, very high speedups are possible.

- Contention (++): Since readers do not contend, contention is low.
- Overhead (+): The number of synchronization primitives required is about the same as for Code Locking, but the reader-side primitives can be cheaper in many cases.
- Complexity (−): The reader/writer concept adds a modest amount of complexity.
- Economics (−−): More CPUs are required to achieve higher speedups.

Synchronization overhead must not dominate, contention must be high, and read-to-write ratio must be high. Low synchronization overhead is especially important, as most implementations of reader/writer locking incur greater synchronization overhead than does normal locking.

Specialized forms of reader/writer locking may be used when synchronization overhead dominates [And91, Tay87].

Solution Use reader/writer locking to greatly improve speedups of programs that rarely modify shared data.

The lookup-table example uses read-side primitives to search the table and write-size primitives to modify it. Figure 8 shows locking for search, and Figure 9 shows locking for update. Since this example demonstrates reader/writer locking applied to a code-locked program, the locks must surround the calls to `looktab_search()` as well as the code that examines or modifies the selected element.

Reader/writer locking can easily be adapted to data-locked programs as well.

Resulting Context A program that allows CPUs that are not modifying data to proceed in parallel, thereby increasing speedup.

Design Rationale If synchronization overhead is negligible (i.e., the program uses coarse-grained parallelism), and only a small fraction of the critical sections modify data, then allowing multiple readers to proceed in parallel can greatly increase speedup.

The reader/writer locking pattern can be thought of as a modifier to the Code Locking (5.2) and Data Locking (5.3) patterns, with the reader/writer locks being assigned to code paths and data structures, respectively. It also is an instance of Parallel Fastpath (5.5).

```
/*
 * Global lock for looktab_t
 * manipulations.
 */

srwlock_t looktab_mutex;

. . .

/*
 * Look up a looktab element and
 * examine it.
 */

S_RDLOCK(&looktab_mutex);
p = looktab_search(mykey);

/*
 * insert code here to examine
 * the element.
 */

S_UNLOCK(&looktab_mutex);
```

Figure 8: Read-Side Locking

```
/*
 * Global lock for looktab_t
 * manipulations.
 */

srwlock_t looktab_mutex;

. . .

/*
 * Look up a looktab element and
 * examine it.
 */

S_WRLOCK(&looktab_mutex);
p = looktab_search(mykey);

/*
 * insert code here to update
 * the element.
 */

S_UNLOCK(&looktab_mutex);
```

Figure 9: Write-Side Locking

Reader/writer locking is a simple instance of asymmetric locking. Snaman [ST87] describes a more ornate six-mode asymmetric locking design used in several clustered systems. Asymmetric locking primitives can be used to implement a very simple form of the **Observer Pattern** [GHJV95]–when a writer releases the lock, all readers are notified of the change in state.

5.7 Hierarchical Locking

Problem How can you obtain better speedups when updates are complex and expensive, but where coarse-grained locking is needed for infrequent operations such as insertion and deletion?

Context An existing sequential or parallel program suffering high contention due to coarse-grained locking combined with frequent, high-overhead updates.

Forces

- Contention (++): Updates proceed in parallel. However, coarsely-locked operations are still serialized and can result in contention.
- Speedup (+): Hierarchical locking allows coarsely-locked operations to proceed in parallel with frequent, high-overhead operations. Contention still limits speedups.
- Read-to-Write Ratio (0): Readers accessing a given finely-locked element proceed in parallel. However, readers traversing coarsely-locked structures are still serialized.
- Overhead (−): Hierarchical-locked programs executes more synchronization primitives than do code-locked or data-locked programs, resulting in higher synchronization overhead.
- Economics (−): Greater speedups require bigger machines.
- Complexity (− − −): Hierarchically-locked programs can be extremely complex and subtle, since they are extremely prone to deadlock problems.

A large speedup must be be needed badly enough to invest the development, maintenance, and machine resources required for more aggressive parallelization.

The program must also have hierarchical data structures so that hierarchical locking may be used to reduce contention and overhead.

Solution Partition the data structures into coarse- and fine-grained portions. For example, use a single lock for the internal nodes and links of a search tree, but maintain a separate lock for each of the leaves. If the updates to the leaves is expensive compared to searches and to synchronization primitives, hierarchical locking can result in better speedups than can code or data locking.

For example, allocate a lock for each hash line in the hash-line-header data structure,[7] as shown in Figure 10. This may be done if the individual elements in a table are independent from one another, and allows searches of a single table to proceed in parallel if the items being searched for do not collide after being hashed.

In this example, hierarchical locking is an instance of Parallel Fastpath (5.5) that uses data locking for the fastpath and code locking elsewhere.

You can combine hierarchical locking with data locking (e.g., by changing the search structure from a single linked list to a hashed table with per-hash-line locks) to further reduce contention. This results in an instance of Parallel Fastpath (5.5) that uses data locking for both the fastpath and the search structure. Since there can be many elements per hash line, the fastpath is using a more aggressive form of data locking than is the search structure.

Resulting Context A more heavily parallel program that is less similar to its sequential counterpart, but which exhibits much lower contention and thus higher speedups.

Design Rationale If updates have high overhead, then contention will be reduced by allowing each element to have its own lock.

Since hierarchical locking can make use of different types of locking for the different levels of the hierarchy, it is a template that combines other patterns. It also is an instance of Parallel Fastpath (5.5).

5.8 Allocator Caches

Problem How can you make achieve high speedups in global memory allocators?

Context An existing sequential or parallel program that spends much of its time allocating or deallocating data structures.[8]

Forces

- Speedup (++): Speedups are limited only by the allowable size of the caches.
- Contention (++): The common-case access that hits the per-CPU cache causes no contention.
- Overhead (++): The common-case access that hits the per-CPU cache causes no synchronization overhead.
- Complexity (−): The per-CPU caches make the allocator more complex.
- Economics (−−): High speedups translate to more expensive machines with more CPUs.
- Read-to-Write Ratio (N/A): Allocators normally do not have a notion of reading or writing. If allocation and free operations are considered to be reads, then reads proceed in parallel.[9]

A high speedup must be needed badly enough to invest the resources needed to develop, maintain, and run an allocator with a per-CPU cache.

Solution Create a per-CPU (or per-process) cache of data-structure instances. A given CPU owns the instances in its cache (Data Ownership (5.4)), and therefore need not incur overhead and contention penalties to allocate and free them. Fall back to a global allocator with a less aggressive locking pattern (Code Locking (5.2)) when the per-CPU cache either overflows or underflows. The global allocator is needed to support arbitrary-duration producer-consumer relationships among the CPUs, which make it impossible to fully partition the memory among the CPUs.

Resulting Context A more heavily parallel program that contains an allocator that is quite different than its sequential counterpart, but which exhibits much lower contention and overhead and thus higher speedups.

You do not need to modify the code calling the memory allocator. This pattern confines the complexity to the allocator itself.

[7] In object-oriented programs, this level of data locking results in many independent critical sections per instance.

[8] Programs that allocate resources other than data structures (e.g., I/O devices) use data structures to represent these resources. The allocator-cache pattern is therefore general enough to include general resource allocation. For ease of exposition but without loss of generality, this section focuses on memory allocation.

[9] Perhaps changing the size of the caches or some other attribute of the allocator is considered a write.

```
slock_t looktab_mutex;
looktab_t *looktab_head;
typedef struct looktab {
        struct looktab_t *lt_next;
        int     lt_key;
        int     lt_data;
        slock_t lt_mutex;
} looktab_t;

/*
 * Return a pointer to the element of the table with the
 * specified key holding lt_mutex, or return NULL if no
 * such element exists.
 */

looktab_t *
looktab_search(int key)
{
        looktab_t *p;

        S_LOCK(&looktab_mutex);
        p = looktab_head;
        while (p != NULL) {
                if (p->lt_key > key) {
                        S_UNLOCK(&looktab_mutex);
                        return (NULL);
                }
                if (p->lt_key == key) {
                        S_LOCK(&p->lt_mutex);
                        S_UNLOCK(&looktab_mutex);
                        return (p);
                }
                p = p->lt_next;
        }
        return (NULL);
}

/* . . . */

        /* Look up a looktab element. */

        p = looktab_search(mykey);

        /* insert code here to examine or update the element. */

        S_LOCK(&p->lt_mutex);
```

Figure 10: Hierarchical Locking

Design Rationale Many programs allocate a structure, use it for a short time, then free it. These programs' requests can often be satified from per-CPU/process caches of structures, eliminating the overhead of locking in the common case. If the cache hit rate is too low, increase it by increasing the size of the cache. If the cache is large enough, the reduced allocation overhead on cache hits more than makes up for the slight increase incurred on cache misses.

McKenney and Slingwine [MS93] describe the design and performance of memory allocator that uses caches. This allocator applies the Parallel Fastpath (5.5) pattern twice, using Code Locking (5.2) to allocate pages of memory, Data Locking (5.3) to coalesce blocks of a given size into pages, and Data Ownership (5.4) for the per-CPU caches. Accesses to the per-CPU caches is free of synchronization overhead and of contention. In fact, the overhead of allocations and deallocations that hit the cache are several times cheaper than the synchronization primitives. Cache-hit rates often exceed 95%, so the performance of the allocator is very close to that of its cache.

Other choices of patterns for the caches, coalescing, and page-allocation make sense in other situations. Therefore, the allocator-cache pattern is a template that combines other patterns to create an allocator. It also is an instance of Parallel Fastpath (5.5).

5.9 Critical-Section Fusing

Problem How can you get high speedups from programs with frequent, small critical sections on machines with high synchronization overheads?

Context An existing program with many small critical sections so that synchronization overhead dominates, but which is not easily partitionable.

Forces

- Overhead (+): Fusing critical sections decreases overhead by reducing the number of synchronization primitives.
- Speedup (0): Fusing critical sections improves speedup only when speedup is limited primarily by overhead.
- Read-to-Write Ratio (0): Fusing critical sections usually has no effect on the ability to run readers in parallel.
- Economics (0): Fusing critical sections requires more CPUs if speedup increases.
- Contention (−): Fusing critical sections increases contention.
- Complexity (−−): Fusing otherwise-unrelated critical sections can add confusion and thus complexity to the program.

Additional speedup must be needed badly enough to justify the cost of creating, maintaining, and running a highly parallel program.

Synchronization overhead must dominate, and contention must be low enough to allow increasing the size of critical sections.

Solution Fuse small critical sections into larger ones to get high speedups from programs with frequent, small critical sections on machines with high synchronization overheads.

For example, imagine a program containing back-to-back searches of a code-locked lookup table. A straightforward implementation would acquire the lock, do the first search, do the first update, release the lock, acquire the lock once more, do the second search, do the second update, and finally release the lock. If this sequence of code was dominated by synchronization overhead, eliminating the first release and second acquisition as shown in Figure 11 would increase speedup.

In this case, the critical sections being fused used the same lock. Fuse critical sections that use different locks by combining the two locks into one, and make all critical sections that used either of the two original locks use the single new lock instead.

Resulting Context A program with fewer but larger critical sections, thus less subject to synchronization overheads.

Design Rationale If the overhead of the code between two critical sections is less than the overhead of the synchronization primitives, fusing the two critical sections will decrease overhead and increase speedups.

Critical-section fusing is a meta-pattern than transforms Data Locking (5.3) into Code Locking (5.2) and Code Locking (5.2) into Sequential Program (5.1). In addition, it transforms more-aggressive variants of Code Locking (5.2) and Data Locking (5.3) into less-aggressive variants.

Critical-section fusing is the inverse of Critical-Section Partitioning (5.10).

```
/*
 * Global lock for looktab_t
 * manipulations.
 */

slock_t looktab_mutex;

. . .

/*
 * Look up a looktab element and
 * examine it.
 */

S_LOCK(&looktab_mutex);
p = looktab_search(mykey);

/*
 * insert code here to examine or
 * update the element.
 */

p = looktab_search(myotherkey);

/*
 * insert code here to examine or
 * update the element.
 */

S_UNLOCK(&looktab_mutex);
```

Figure 11: Critical-Section Fusing

5.10 Critical-Section Partitioning

Problem How can you get high speedups from programs with infrequent, large critical sections on machines with low synchronization overheads?

Context An existing program with a few large critical sections so that contention dominates, where the critical sections might contain code that is not relevant to the data structures that they protect, but that is not easily partitionable.

This situation is rather rare in highly parallel code, since the cost of synchronization overhead has been decreasing more slowly than instruction or memory-access overhead. The increasing relative cost of synchronization makes it less likely that contention effects will dominate. You can find it in cases where Critical-Section Fusing (5.9) has been applied too liberally and in code that is not highly parallel.

If the program is large or unfamiliar but is not being maintained in sequential form by another organization, you should consider applying other patterns (such as Data Locking (5.3)) because the effort of applying the more difficult and effective patterns can be small compared to the effort required to analyze and understand the program.

Forces

- Contention (+): Partitioning critical sections decreases contention.
- Complexity (+): Partitioning "thrown-together" critical sections can clarify the intent of the code.
- Speedup (0): Partitioning critical sections improves speedup only when speedup is limited primarily by overhead.
- Read-to-Write Ratio (0): Splitting critical sections usually has no effect on the ability to run readers in parallel.
- Economics (0): Splitting critical sections requires more CPUs if speedup increases.
- Overhead (−): Splitting critical sections can increase overhead by increasing the number of synchronization primitives. There are some special cases where overhead is unchanged, such as when a code-locked program's critical sections are partitioned by data structure, resulting in a data-locked program.

Contention must dominate, and synchronization overhead must be low enough to allow decreasing the size of critical sections.

Solution Split large critical sections into smaller ones to get high speedups from programs with infrequent, large critical sections on machines with low synchronization overheads.

For example, imagine a program parallelized using "huge locks" that covered entire subsystems. These subsystems are likely to contain a fair amount of code that does not need to be in a critical section. Splitting the critical sections to allow this code to run in parallel might increase speedup.

If you think of a sequential program as being a code-locked program with a single critical section containing the whole program, then you would apply critical-section splitting to move from a sequential to a code-locked program.

Resulting Context A program with more but smaller critical sections, thus less subject to contention.

Design Rationale If the overhead of the non-critical-section code inside a single critical sections is greater than the overhead of the synchronization primitives, splitting the critical section can decrease overhead and increase speedups.

Critical-section partitioning is a meta-pattern that transforms Sequential Program (5.1) into Code Locking (5.2) and Code Locking (5.2) into Data Locking (5.3). It also transforms less-aggressive variants of Code Locking (5.2) and Data Locking (5.3) into more-aggressive variants.

Critical-section partitioning is the inverse of Critical-Section Fusing (5.9).

6 Acknowledgments

I owe thanks to Ward Cunningham and Steve Peterson for encouraging me to set these ideas down and for many valuable conversations, to my PLoP'95 shepard, Erich Gamma, for much coaching on how to set forth patterns, to the members of PLoP'95 Working Group 4 for their insightful comments and discussion, to Ralph Johnson for his tireless championing of use of active voice in patterns, and to Dale Goebel for his consistent support.

References

[And91] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys:*, 1991.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[McK95] Paul E. McKenney. Differential profiling. In *MASCOTS'95*, Toronto, Canada, January 1995.

[Mes95] Gerard Meszaros. A pattern language for improving capacity of reactive systems. In *Pattern Languages of Program Design*, September 1995.

[MS93] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Conference Proceedings*, Berkeley CA, February 1993.

[Sch95] Douglas C. Schmidt. Active object. In *Pattern Languages of Program Design*, September 1995.

[ST87] William E. Snaman and David W. Thiel. The VAX/VMS distributed lock manager. *Digital Technical Journal*, September 1987.

[Tay87] Y. C. Tay. *Locking Performance in Centralized Databases*. Academic Press, 1987.

[ZRB+93] Roman Zajcew, Paul Roy, David Black, Chris
Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, and Durriya Netterwala. An OSF/1 UNIX for massively parallel multicomputers. In *1993 Winter USENIX*, pages 449–468, January 93.