The one line of C code that will make your program ten times faster

Ben Klemens

19 Feb 2014

[This blog post eventually became Chapter 12 of *21st Century C*[1], published by O'Reilly Media.]

It might be a link-bait title, but it's true: you can cut the time it takes for your compute-intensive program to run in half, a quarter, or a tenth with one line of code.

My claim is not too good to be true if you can modify your single-threaded program to use parallel threads, because just about all the computers sold in the last few years—even many telephones—are multicore. If you are reading this on a keyboard-and-monitor computer, you may be able to find out how many cores your computer has via:

- Cygwin: `env | grep NUMBER_OF_PROCESSORS`
- Linux: `grep cores /proc/cpuinfo`
- Mac: `sysctl hw.logicalcpu`

When I talk about doing amazing things with a single line of code, I must be talking about writing in C, which is what the entirety of this piece will be about. Other languages have their own mechanisms to parallelize code, which are sometimes much more cumbersome than the one-line additions to the C code shown here.

Here, I will cover:

- A quick overview of the several standards and specifications that exist for writing concurrent C code.
- The one line of OpenMP code that will make your `for` loops ten times faster. To kill the suspense, it's `#pragma omp parallel for`.
- Notes on the compiler flags you'll need to compile with OpenMP or pthreads.
- Some considerations of when it's safe to use that one magic line. If you do decide to read this, please don't stop reading until after you finish this section.
- Implementing map-reduce, which requires extending that one line by another clause.
- The syntax for running a handful of distinct tasks in parallel, like the UI and back-end of a GUI-based program.

[1] `http://www.amazon.com/exec/obidos/redirect?link_code=ur2&camp=1789&tag=caltechdivini-20&creative=9325&path=tg/detail/-/1491903899/qid=1120157199/sr=8-1/ref=pd_bbs_ur_1`

- C's `_Thread_local` keyword, which makes thread-private copies of global static variables.
- Critical regions and mutexes.
- Atomic variables in OpenMP.
- A quick note on sequential consistency and why you want it.
- POSIX threads, and how they differ from OpenMP.
- Atomic scalar variables via C atoms.
- Atomic structs via C atoms.

Why did I write this? Although there are a lot of books devoted to the topic of threading, I did a survey of general C textbooks, and could not find even one that covered OpenMP. So here I will give you enough to get started—maybe even enough that you may never need to refer to the full books on threading theory.

I am aiming this at readers who want to know enough to safely speed up their code, not readers who want to become experts in concurrency. I will use the default scheduling and the safest form of synchronization throughout, even though there exist cases where you can do better by fine-tuning those things, will not wade into details of cache optimization, and will not give you an exhaustive list of useful OpenMP pragmas (which your Internet search engine will do just fine). If you want to know all the details, see Breshears [2009], Grama et al. [2003], or Gove [2010].

If you have comments, corrections, or bug fixes, please let me know. [Unless you want to write to me about how Visual Studio can't compile the code here, whose syntax conforms to the ANSI/ISO standard. I get a lot of people angry at me about that, but dude, write to Microsoft about it.]

The environment It wasn't until the December 2011 revision that a threading mechanism was a part of standard C. That is late to the party, and others have already provided mechanisms. So you've got several options:

- POSIX threads. The pthreads standard was defined in POSIX v1, circa 1995. The `pthread_create` function works by assigning a function of a certain form to each thread, so you have to write an appropriate function interface, and typically an accompanying struct.
- OpenMP is a specification that uses `#pragmas` (and a handful of library functions) to tell the compiler when and how to thread. This is what you can use to turn a serial-running `for` loop into a threaded `for` loop with one line of code. The first OpenMP spec for C came out in 1998.
- The C standard library now includes headers that define functions for threading and atomic variable operations.
- Windows also has its own threading system, which works similarly to pthreads. For example, the `CreateThread` function takes in a function and a pointer to parameters much like `pthread_create` does.

Which to use depends on your target environment, and your own goals and preferences. If your goal is to have as many C compilers as possible reliably compile your code, then the best choice at the moment from the list above may be the one that largely steps outside of C: OpenMP. It is supported by all major compilers—even Visual Studio! [footnote: Visual Studio supports version 2.0, and OpenMP is at version 4.0, but the basic pragmas covered here are not new.] As I write this today, few users have compilers and standard libraries that support standard C threading. I have been testing code using release candidates for standard libraries; they may or may not have become the current release by the time you read this, but a large share of users remain consistently several years behind. If you are unable to rely on `#pragma`s, then pthreads are available for any POSIX-conformant host (even MinGW).

There are other options, such as the MPI (message passing interface, for talking across networked nodes) or OpenCL (especially useful for GPU processing). On POSIX systems, you can use the `fork` system call to effectively produce two clones of your program that operate independently.

The ingredients Our syntactic needs are not great. In all of the cases, we will need:

- A means of telling the compiler to set off several threads at once. To give an early example, Nabokov [1962] includes this instruction on line 404: *[And here time forked.]* The remainder of the section vacillates between two threads.
- A means of marking a point where the new threads cease to exist, and the main thread continues alone. In the above early example, the barrier is implicit at the end of the segment, but some of the options will have an explicit marker.
- A means of marking parts of the code that should not be threaded, because they can not be made thread-safe. For example, what would happen if thread one resized `array` to size 20 at the same time that thread two is resizing it to size 30? If we could slow down time, we would see that even a simple increment like `x++` requires a series of finite-time operations that another thread could conflict with. Using OpenMP pragmas, these unshareable segments will be marked as *critical regions*; in pthread-influenced systems these will be marked via *mutexes* (a crunching-together of *mutual exclusion*).
- Means of dealing with variables shared across threads: how can we guarantee that a change to a shared variable in one thread is visible in another, and what should be done when two threads want to handle a variable at the same time? Strategies include taking a global variable and making a thread-local copy, and syntax to put a mini-mutex around each use of a variable.

OpenMP As an example, let us thread a word-counting program. For the sake of focusing on the threading part, the function itself will just call the POSIX-standard `wc` program, and doesn't try very hard to be safe. So the `wc` function is nothing special, and `main` is basically just a `for` loop calling `wc` going over the list of input files. After the loop, the program sums up the individual counts to a grand total.

```
#define _GNU_SOURCE //ask stdio.h to include asprintf. Or include it yourself.
#include <stdio.h>
#include <stdlib.h>

int wc(char *docname){
    char *cmd;
    asprintf(&cmd, "cat %s | wc", docname);
    FILE *wc = popen(cmd, "r");
    char out[100];
    fread(out, 1, 100, wc);
    return atoi(out);
}

int main(int argc, char **argv){
    argc−−;
    argv++;
    if(!argc){
        fprintf(stderr, "Please give some file names on the command line.");
        return 1;
    }
    int count[argc];
    #pragma omp parallel for
    for (int i=0; i< argc; i++){
        count[i] = wc(argv[i]);
        printf("%s:\t%i\n", argv[i], count[i]);
    }

    long int sum=0;
    for (int i=0; i< argc; i++) sum+=count[i];
    printf("$\Sigma$:\t%li\n", sum);

}
```

Did you see the line that makes this a threaded program? The OpenMP instruction is this line:

```
#pragma omp parallel for
```

indicating that the `for` loop immediately following should be broken down into segments and run across as many threads as the compiler deems optimal. In this case, I've lived up to my promise, and turned a not-parallel program into a parallel program with one line of code.

OpenMP works out how many threads your system can run and splits up the work accordingly. In cases where you need to set the number of threads to N manually, you can do so either by setting an environment variable before the program runs:

```
export OMP_NUM_THREADS=N
```

or by using a C function in your program:

```
#include <omp.h>
omp_set_num_threads(<replaceable>N</replaceable>);
```

These facilities are probably most useful for fixing the thread count to $N = 1$. If you want to return to the default of requesting as many threads as your computer has processors, use:

```
#include <omp.h>
omp_set_num_threads(omp_get_num_procs());
```

Compiling OpenMP For gcc and clang (where clang support for OpenMP is still in progress on some platforms), compiling this requires adding an `-fopenmp` flag for the compilation. If you need a separate linking step, add `-fopenmp` to the link flags as well (and the compiler will know if any libraries need to be linked to, and will do what you want). For pthreads, you will need to add `-pthread`.

If you are using gcc or clang, you can paste the following three lines into a file named `makefile` and then compile the above by just typing `make`. It includes both `-fopenmp` and `-pthread` so you can recycle it for the other examples below; just change `openmp_wc` to the program name you are compiling at the moment (or add the new program as another target).

```
CFLAGS=-g -Wall -std=gnu99 -O3 -fopenmp -pthread
LDLIBS=-fopenmp
openmp_wc:
```

If you are using Autoconf, you will need to add a line to your existing `configure.ac` script:

```
AC_OPENMP
```

which generates an `$OPENMP_CFLAGS` variable, which you will then need to add to existing flags in `Makefile.am`. For example,

```
AM_CFLAGS = $(OPENMP_CFLAGS) -g -Wall -O3 ...
AM_LDFLAGS = $(OPENMP_CFLAGS) $(SQLITE_LDFLAGS) $(MYSQL_LDFLAGS)
```

So that took three lines of code, but now Autoconf will correctly compile your code on every known platform that supports OpenMP.

Autoconf usually defines a variable like `HAVE_OPENMP` so you can put `#ifdef` blocks into your code as needed. In this case, Autoconf doesn't have to do this, because the OpenMP standard already requires that an `_OpenMP` variable be defined if the compiler accepts OpenMP pragmas.

Once you have the program compiled as `threaded_wc`, try
`./threaded_wc 'find ~ -type f'`
to start a word-count of every file in your home directory. You can open `top` in another window and see if multiple instances of `wc` crop up.

Interference Now that we have the needed line of syntax to make the program multithreaded, are we guaranteed that it works? For easy cases where every iteration of the loop is guaranteed to not interact with any other iteration, yes. But for other cases, we'll need to be more cautious.

To verify whether a team of threads will work, we need to know what happens with each variable, and the effects of any side-effects.

- If a variable is private to a thread, then we are guaranteed that it will behave as if in a single-threaded program, without interference. The iterator in the loop, named `i` in the above example, is made private to each thread (see the OpenMP 4.0 standard §2.6). Variables declared inside of the loop are private to the given loop.

- If a variable is being read by a number of threads and is never written by any of them at any point, then you are still safe. This isn't quantum physics: reading a variable never changes its state (and I'm not covering C atomic flags, which actually can't be read without setting them).
- If a variable is shared, and it is written to by one thread an never read by any other thread, there is still no competition—the variable is effectively private.
- if a variable is shared across threads, and it is written to by one thread, and it is read from or written to by any other thread, now you've got real problems, and the rest of this piece is basically about this case.

The first implication is that, where possible, we should avoid sharing written-to variables wherever possible. You can go back to the example and see one way of doing this: all of the threads use the `count` array, but iteration `i` touches only element `i` in the array, so each array element is effectively thread-private. Further, the `count` array itself is not resized, freed, or otherwise changed during the loop, and similarly with `argv`. We'll even get rid of the `count` array below.

We don't know what internal variables `printf` uses, but we can check the C standard to verify that all of the operations in the standard library that operate on input and output streams (almost everything in `stdio.h`) are thread-safe, so we can call `printf` without worry about multiple calls stepping on each other (C11 §7.21.2(7) and (8)).

When I wrote this sample, it took some rewriting to make sure that those conditions were met. However, some of the considerations, such as avoiding global variables, are good advice even in the single-threaded world. Also, the post-C99 style of declaring variables at their first use is paying off, because a variable declared inside a segment to be threaded is unambiguously private to that thread.

As an aside, OpenMP's `omp parallel for` pragma understands only simple loops: the iterator is of integer type, it is incremented or decremented by a fixed (loop-invariant) amount every step, and the ending condition compares the iterator to a loop-invariant value or variable. Anything that involves applying the same routine to each element of a fixed array fits this form.

Map-reduce The word count program has a very common form: each thread does some independent task producing an output, but we are really interested in an aggregate of those outputs. OpenMP supports this sort of map-reduce workflow via an addendum to the above pragma. In this version, I replace the `count` array with a single variable, `total_wc`, and add `reduction(+:total_wc)` to the OpenMP pragma. From here, the compiler does the work to efficiently combine each thread's `total_wc` to a grand total.

```
#define _GNU_SOURCE //enable asprintf
#include <stdio.h>
#include <stdlib.h>

int wc(char *docname){ //unchanged from before
    char *cmd;
    asprintf(&cmd, "cat %s | wc", docname);
    FILE *wc = popen(cmd, "r");
    char out[100];
    fread(out, 1, 100, wc);
    return atoi(out);
}

int main(int argc, char **argv){
    argc--;
    argv++;
    if(!argc){
        fprintf(stderr, "Please give some file names on the command line.");
        return 0;
    }
    long int total_wc=0;
    #pragma omp parallel for \
    reduction(+:total_wc)
    for (int i=0; i< argc; i++){
        int ct = wc(argv[i]);
        printf("%s:\t%i\n", argv[i], ct);
        total_wc += ct;
    }
    printf("$\Sigma$:\t%li\n", total_wc);
}
```

Again, there are restrictions: the operator in `reduction(+:variable)` can only be a basic arithmetic (`+`, `*`, `-`), bitwise (`&`, `|`, `^`), or logical (`&&`, `||`) operation. Otherwise, you'll have to go back to something like the `count` array above and writing your own post-thread reduction. Also, don't forget to initialize the reduction variable before the team of threads runs.

Multiple tasks Instead of having an array and applying the same operation to every array element, you may have two entirely distinct operations, and they are independent and could run in parallel. For example, programs with a user interface often put the UI on one thread and the back-end processing on another, so that slowdown in processing doesn't make the UI seize up. Naturally, the pragma for this is the `parallel sections` pragma:

```
#pragma omp parallel sections
{
    #pragma omp section
    {
    //Everything in this block happens only in one thread
    UI_starting_fn();
    }

    #pragma omp section
    {
    //Everything in this block happens only in one other thread
    backend_starting_fn();
    }
}
```

Here are a few more features of OpenMP that I didn't mention but that you may enjoy:

- simd: Single instruction, multiple data. Some processors have a facility to apply the same operation to every element of a vector. This is distinct from multiple threads, which run on multiple cores, and is not available on all processors. See `#pragma omp simd`.
- When the number of tasks is not known ahead of time, you can use `#pragma omp task` to set off a new thread. For example, you may be running through a tree structure with a single thread, and at each terminal node, use `#pragma omp task` to start up a new thread to process the leaf.
- You may be searching for something with multiple threads, and when one thread finds the goal, there is no point having the other threads continue. Use `#pragma omp cancel` or `pthread_cancel` to call off the other threads.

Also, I must add one more caveat, lest some readers go out and put a `#pragma` over every single `for` loop in everything: there is overhead to generating a thread. This code:

```
int x =0;
#pragma omp parallel for
for (int i=0; i< 10; i++){
    x++;
}
```

will spend more time generating thread info than incrementing `x`, and would almost certainly be faster un-threaded. Use threading liberally, but don't go crazy with it.

If you have verified that none of your threaded segments write to a shared variable, and all functions called are also thread-safe, then you can stop reading now. Insert `#pragma omp parallel for` or `parallel sections` at the appropriate points, and enjoy your speed gains. The rest of this discussion, and in fact the majority of writing on threading, is about modifying shared resources.

Thread local Static variables—even those declared inside of your `#pragma omp parallel` region—are shared across all threads by default. You can generate a separate private copy for each thread by adding a `threadprivate` clause to the pragma. E.g.,

```
static int state;
#pragma omp parallel for threadprivate(state)
for (int i=0; i< 100; i++)
    ...
```

With some commonsense caveats, the system retains your set of threadprivate variables, so if `static_x` was 2.7 in thread 4 at the end of one parallel region, it will still be 2.7 in thread 4 at the start of the next parallel region with four threads (OpenMP §2.14.2). There is always a master thread running; outside of the parallel region, the master thread retrains its copy of the static variable.

C's `_Thread_local` keyword splits off static variables in a similar manner.

In C, a thread local static variable's "lifetime is the entire execution of the thread for which it is created, and its stored value is initialized when the thread is started." (C11 §6.2.4(4)) If we read thread 4 in the first parallel region to be the same thread as thread 4 in another parallel region, then this behavior is identical to the OpenMP behavior; if they are read to be separate threads, then the C standard specifies that thread local storage is re-initialized at every parallel region. [Footnote: Jens Gustedt enumerated lots of little differences between C threads and pthreads[2], and suggests that many are due to errors or loose language in the C spec. On the linked page, he also proposes a Cthreads-pthreads translation chart.]

There is still a master thread that exists outside of any parallel regions (it's not stated explicitly, but C11 §5.1.2.4(1) implies this), so a thread-private static variable in the master thread looks a lot like a traditional lifetime-of-the-program static variable.

The keyword in the C standard seems to be an emulation of the gcc-specific `__thread` keyword. Within a function, you can use either of:

```
static __thread int i; //GCC−specific; works today.
// or
static _Thread_local int i; //C11, when your compiler implements it.
```

Outside of a function, the `static` keyword is optional, because it is the default. The standard requires a `threads.h` header that defines `thread_local`, so you don't need the annoying underscore-capital combination (much like `stdbool.h` defines `bool` to be `_Bool`). But this header isn't yet implemented by any major compilers.

You can check for which to use via a block of preprocessor conditions, like this one, which sets the string `threadlocal` to the right thing for the given situation.

```
#undef threadlocal
#ifdef _ISOC11_SOURCE
    #define threadlocal _Thread_local
#elif defined(__APPLE__)
    #define threadlocal //not yet implemented.
#elif defined(__GNUC__) && !defined(threadlocal)
    #define threadlocal __thread
#else
    #define threadlocal
#endif
```

Non-static variables declared outside of the parallel region are shared by default. You can make private copies of `localvar` available to each thread via a `firstprivate(localvar)` clause. A copy is made for each thread, and initialized with the value of the variable at the start of the thread. At the end, they are all destroyed, and the original variable is untouched; add `lastprivate(localvar)` to copy the last thread's value back to the outside-the-region variable.

There is also a plain `private()` clause, that sets up private variables for the threads that have no interaction with the variable declared outside the parallel block.

[2]`https://gustedt.wordpress.com/2012/10/14/c11-defects-c-threads-are-not-realizable-with-posix-thr`

There isn't a whole lot you can do with this form, given that you can reduce confusion (and typing) by declaring such a private variable inside the block where it is clearly understood by both readers and the OpenMP system to be private and destroyed on thread exit. You might need it in a `for` loop for variables that shouldn't be re-initialized on every iteration.

Shared resources To this point, I've stressed the value of using private variables, and a means of multiplexing a single static variable into a set of thread-local private variables. But sometimes, a resource really does need to be shared, and the *critical region* is the simplest means of protecting it. It marks a segment of code that should only be executed by a single thread at a time. As with most other OpenMP constructs, it operates on the subsequent block:

```
#pragma omp critical (a_private_block)
{
    interesting code here.
}
```

We are guaranteed that this block will be entered by only one thread at a time. If another thread gets to this point while there is a thread in the critical region, then the recently-arrived thread waits at the head of the region until the thread currently executing the critical region exits.

This is called *blocking*, and a blocked thread is inactive for some period of time. This is time-inefficient, but it is far better to have inefficient code than incorrect code. It is as easy to set critical regions as it is to set parallel regions, so you can use them liberally.

The `(a_private_block)`, with the parens, is a name that allows you to link together critical regions, such as to protect a single resource used at different points in the code. Say that you do not want a structure to be read while another is writing to the structure, for example:

```
#pragma omp critical (delicate_struct)
{
    delicate_struct_update(ds);
}

[other code here]

#pragma omp critical (delicate_struct)
{
    delicate_struct_read(ds);
}
```

We are guaranteed that there will be at most one thread in the overall critical region that is the sum of the two segments, and so there will never be a call to `delicate_struct_update` simultaneous to `delicate_struct_read`. The intermediate code will thread as usual.

A friendly warning: the name is technically optional, but all unnamed critical regions are treated as part of the same group. This is a common form for small sample programs (like most of what you find on the Internet) but probably not what you want in nontrivial code. By naming every critical region, you can prevent accidentally linking two segments together.

Protecting individual elements The number 18 is evenly divisible by six positive integers: 1, 2, 3, 6, 9, and 18. The number 13 has two factors, 1 and 13, meaning that it is a prime number.

It is easy to find prime numbers—there are fully 664,579 of them under ten million. But there are only 446 numbers under ten million with exactly three factors, six with seven factors, and one with 17 factors. Other patterns are easier to find: there are 2,228,418 numbers under ten million with exactly eight factors.

Here is a program to find those factor counts, threaded via OpenMP. The basic storyline involves two arrays. The first is a ten million element array, `factor_ct`. Start with two, and add one to every slot in the array divisible by two (i.e., every even number). Then add one to every slot divisible by three, and so on, up to five million (which would only add a tally to slot ten million, if it were in the array). At the end of that procedure, we know how many factors every number has. You can insert a `for` loop to `fprintf` this array to a file if so desired.

Then, set up another array to tally how many numbers have 1, 2, ... factors. Before doing this, we have to find the maximum count, so we know how big of an array to set up. Then, go through the first array listing each number's factor count, and add one to the appropriate tally.

Each step is clearly a candidate for parallelization via `#pragma omp parallel for`, but conflicts may easily arise. The thread marking multiples of 5 and the thread marking multiples of 7 may both want to increment `factor_ct[35]` at the same time. To prevent a write conflict, say that we mark the line where we add one to the count of factors for item `i` as a critical region:

```
#pragma omp critical (factor)
factor_ct[i]++;
```

When one thread wants to increment `factor_ct[30]`, it blocks the other thread that wants to increment `factor_ct[33]`. Critical regions are about certain blocks of code, and make sense if some blocks are associated with one resource, but we are really trying to protect ten million separate resources, which brings us to *mutexes* and *atomic variables*.

Mutex is short for *mutual exclusion*, and is used to block threads much like the multi-part critical regions above. However, the mutex is a struct like any other. We can have an array of ten million of them, and then lock mutex i before writing to item i, and release mutex i after the write is complete. In code, it would look something like this:

```
omp_lock_t locks[1e7];
for (long int i=0; i< lock_ct; i++)
    omp_init_lock(&locks[i]);

for (long int scale=2; scale*i < max; scale++) {
        omp_set_lock(&locks[scale*i]);
        factor_ct[scale*i]++;
        omp_unset_lock(&locks[scale*i]);
    }
```

The `omp_set_lock` function is really a wait-and-set function: if the mutex is unlocked, then lock it and continue; if the mutex is already locked, block the thread and wait, then continue when the other thread that has the lock reaches `omp_unset_lock` and gives the all-clear.

As desired, we have effectively generated ten million distinct critical regions. The only problem is that the mutex struct itself takes up space, and allocating ten million of them may be more work than the basic math itself. The solution I present in the code is to use only 128 mutexes, and lock mutex `i` mod 128. That means any two threads working with two different numbers have about a one in 128 chance of needlessly locking each other. That's not terrible, and on my test box here is a major speedup from allocating and using ten million mutexes.

Pragmas are baked into a compiler that understands them, but these mutexes are are plain C structs and functions, so this example needs to `#include <omp.h>`.

Enough preface. Here is the code; the part about finding the largest number of factors is in a separate listing below:

```
/* Suggested makefile:
----------
P=openmp_factors
CFLAGS=`pkg-config --cflags glib-2.0` -g -Wall -std=gnu99 -O3 -fopenmp
LDLIBS=`pkg-config --libs glib-2.0` -fopenmp

$(P):
----------
*/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h> //malloc
#include <string.h> //memset

#include "openmp_getmax.c"

int main(){
    long int max = 1e7;
    int *factor_ct = malloc(sizeof(int)*max);

    int lock_ct = 128;
    omp_lock_t locks[lock_ct];
    for (long int i=0; i< lock_ct; i++)
        omp_init_lock(&locks[i]);

    factor_ct[0] = 0;
    factor_ct[1] = 1;
    for (long int i=2; i< max; i++)
        factor_ct[i] = 2;
    #pragma omp parallel for
    for (long int i=2; i<= max/2; i++)
        for (long int scale=2; scale*i < max; scale++) {
                omp_set_lock(&locks[scale*i % lock_ct]);
                factor_ct[scale*i]++;
                omp_unset_lock(&locks[scale*i % lock_ct]);
            }

    int max_factors = get_max(factor_ct, max);
    long int tally[max_factors+1];
    memset(tally, 0, sizeof(long int)*(max_factors+1));

    #pragma omp parallel for
    for (long int i=0; i< max; i++){
        int factors = factor_ct[i];
        omp_set_lock(&locks[factors % lock_ct]);
        tally[factors]++;
        omp_unset_lock(&locks[factors % lock_ct]);
    }

    for (int i=0; i<=max_factors; i++)
        printf("%i\t%li\n", i, tally[i]);
}
```

Here is `openmp_getmax.c`, which finds the maximum value within the `factor_ct` list. Because OpenMP doesn't provide a `max` reduction, we have to maintain an array of maxes and then find the max among those. The array is `omp_get_max_threads()` long, A thread can use `omp_get_thread_num()` to find its index.

```
/* See compilation notes in atomic_factors.c, openmp_atoms.c, or pthread_factors.c*/

int get_max(int *array, long int max){
    int thread_ct = omp_get_max_threads();
    int maxes[thread_ct];
    memset(maxes, 0, sizeof(int)*thread_ct);

    #pragma omp parallel for
    for (long int i=0; i< max; i++){
        int this_thread = omp_get_thread_num();
        if (array[i] > maxes[this_thread])
            maxes[this_thread] = array[i];
    }

    int global_max=0;
    for (int i=0; i< thread_ct; i++)
        if (maxes[i] > global_max)
            global_max = maxes[i];
    return global_max;
}
```

As with the pair of critical regions above, you could use one mutex to protect a resource at several points in a code base.

```
omp_set_lock(&delicate_lock);
delicate_struct_update(ds);
omp_unset_lock(&delicate_lock);

[other code here]

omp_set_lock(&delicate_lock);
delicate_struct_read(ds);
omp_unset_lock(&delicate_lock);
```

Atoms An atom is a small, indivisible element. Atomic operations often work via features of the processor, and OpenMP limits them to acting on scalars: almost always an integer or floating-point number, or sometimes a pointer (i.e., a memory address). C will provide atomic structs, but even then you will typically need to use a mutex to protect the struct.

However, the case of simple operations on a scalar is a common one, and in that case we can dispense with mutexes and use atomic operations to effectively put an implicit mutex around every use of a variable.

You'll have to tell OpenMP what you want to do with your atom:

```
#pragma omp atomic read
out = atom;

#pragma omp atomic write seq_cst
atom = out;

#pragma omp atomic update seq_cst
atom ++; //or atom--;

#pragma omp atomic update
//or any binary operation: atom *= x, atom /=x, ...
atom -= x;

#pragma omp atomic capture seq_cst
//an update-then-read
out = atom *= 2;
```

The `seq_cst` is optional but recommended (if your compiler supports it); I'll get to it in a moment.

From there, it is up to the compiler to build the right instructions to make sure that a read to an atom in one part of the code is unaffected by a write in an atom in another part of the code.

In the case of the factor-counter, all of the resources protected by mutexes are scalars, so we actually didn't need to use mutexes. We can make the code shorter and more readable using atoms:

```
/* Suggested makefile:
----------
P=openmp_atoms
CFLAGS=-g -Wall -std=gnu99 -O3 -fopenmp

$(P):
----------
*/
#include <omp.h>
#include <stdio.h>
#include <string.h> //memset

#include "openmp_getmax.c"

int main(){
    long int max = 1e7;
    int *factor_ct = malloc(sizeof(int)*max);

    factor_ct[0] = 0;
    factor_ct[1] = 1;
    for (long int i=2; i< max; i++)
        factor_ct[i] = 2;

    #pragma omp parallel for
    for (long int i=2; i<= max/2; i++)
        for (long int scale=2; scale*i < max; scale++) {
                #pragma omp atomic update
                factor_ct[scale*i]++;
            }

    int max_factors = get_max(factor_ct, max);
    long int tally[max_factors+1];
    memset(tally, 0, sizeof(long int)*(max_factors+1));

    #pragma omp parallel for
    for (long int i=0; i< max; i++){
        #pragma omp atomic update
        tally[factor_ct[i]]++;
    }

    for (int i=0; i<=max_factors; i++)
        printf("%i\t%li\n", i, tally[i]);
}
```

Sequential consistency A good compiler will reorder the sequence of operations in a manner that is mathematically equivalent to the code you wrote, but which runs faster. If a variable is initialized on line ten but first used on line twenty, maybe it's faster to do an initialize-and-use on line twenty than to execute two separate steps. Here is a two-threaded example, taken from C11 §7.17.3(15) and reduced to more readable pseudocode:

```
x = y = 0;

// Thread 1:
r1 = load(y);
store(x, r1);

// Thread 2:
r2 = load(x);
store(y, 42);
```

Reading the page, it seems like `r2` can't be 42, because the assignment of 42 is happening on the line subsequent to the one where `r2` is assigned. If thread 1 executed entirely before thread 2, between the two lines of thread 2, or entirely afterward, then `r2` can't be 42. But the compiler could swap the two lines of thread 2, because one line is about `r2` and `x`, and the other is about `y`, so there is no dependency that requires one to happen before the other. So this sequence is valid:

```
x = y = 0;
store(y, 42); //thread 2
r1 = load(y); //thread 1
store(x, r1); //thread 1
r2 = load(x); //thread 2
```

Now all of `y`, `x`, `r1`, and `r2` are 42.

The C standard goes on with even more perverse cases, even commenting that one of them "is not useful behavior, and implementations should not allow it."

So that's what the `seq_cst` clause is about: it tells the compiler that atomic operations in a given thread should occur in the order listed in the code file. It was added in OpenMP 4.0, to take advantage of C's sequentially consistent atoms, and your compiler probably doesn't support it yet. In the mean time, keep an eye out for the sort of subtleties that could happen when the compiler shuffles your within-a-thread independent lines of code.

Pthreads Now let us translate the above example to use pthreads. We have similar elements: a means of dispersing threads and gathering them, and mutexes. Pthreads don't give you atomic variables, but plain C does; see below.

The big difference in the code is that the `pthread_create` function to set a new thread running takes in (among other elements) a single function of the form `void *fn(void *in)`, and because that function takes in one void pointer, we have to write a function-specific struct to take in the data. The function also returns another struct, though if you are defining and ad hoc typedef for a function, it is usually easier to include output elements in the typedef for the input struct than to typedef a special input struct and a special output struct.

Let me cut out one of the key sections:

```
tally_s thread_info[thread_ct];
for (int i=0; i< thread_ct; i++){
    thread_info[i] = (tally_s){.this_thread=i, .thread_ct=thread_ct,
                    .tally=tally, .max=max, .factor_ct=factor_ct,
                    .mutexes=mutexes, .mutex_ct =mutex_ct};
    pthread_create(&threads[i], NULL, add_tally, thread_info+i);
}
for (int t=0; t< thread_ct; t++)
    pthread_join(threads[t], NULL);
```

The first `for` loop generates a fixed number of threads (so it is hard to have pthreads dynamically generate a thread count appropriate to different situations). It first sets up the needed struct, and then it calls `pthread_create` to call the `add_tally` function, sending in the purpose-built struct. At the end of that loop, there are `thread_ct` threads at work.

The next `for` loop is the gather step. The `pthread_join` function blocks until the given thread has concluded its work. Thus, we can't go past the `for` loop until all threads are done, at which point the program is back to the single main thread.

The pthread library provides mutexes that behave very much like OpenMP mutexes. For our limited purposes here, changing to pthread mutexes is merely a question of changing names.

Here's the program rewritten with pthreads. Breaking each subroutine into a separate thread, defining a function-specific struct, and the disperse-and-gather routines add a lot of lines of code (and I'm still recycling the OpenMP `max_factor` function).

```
/* Compile with:
export CFLAGS="-g -Wall -O3 --std=c99 -pthread -fopenmp"
make pthread_factors
*/
#include <omp.h> //get_max is still OpenMP
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h> //malloc
#include <string.h> //memset

#include "openmp_getmax.c"

typedef struct {
    long int *tally;
    int *factor_ct;
    int max, thread_ct, this_thread, mutex_ct;
    pthread_mutex_t *mutexes;
} tally_s;

void *add_tally(void *vin){
    tally_s *in = vin;
    for (long int i=in->this_thread; i < in->max; i += in->thread_ct){
        int factors = in->factor_ct[i];
        pthread_mutex_lock(&in->mutexes[factors % in->mutex_ct]);
        in->tally[factors]++;
        pthread_mutex_unlock(&in->mutexes[factors % in->mutex_ct]);
    }
    return NULL;
}

typedef struct {
    long int i, max, mutex_ct;
    int *factor_ct;
    pthread_mutex_t *mutexes ;
} one_factor_s;

void *mark_factors(void *vin){
    one_factor_s *in = vin;
    long int si = 2*in->i;
    for (long int scale=2; si < in->max; scale++, si=scale*in->i) {
        pthread_mutex_lock(&in->mutexes[si % in->mutex_ct]);
        in->factor_ct[si]++;
        pthread_mutex_unlock(&in->mutexes[si % in->mutex_ct]);
    }
    return NULL;
}

int main(){
    long int max = 1e7;
    int *factor_ct = malloc(sizeof(int)*max);

    int thread_ct = 4, mutex_ct = 128;
    pthread_t threads[thread_ct];
    pthread_mutex_t mutexes[mutex_ct];
    for (long int i=0; i< mutex_ct; i++)
        pthread_mutex_init(&mutexes[i], NULL);

    factor_ct[0] = 0;
    factor_ct[1] = 1;
    for (long int i=2; i< max; i++)
        factor_ct[i] = 2;

    one_factor_s x[thread_ct];
    for (long int i=2; i<= max/2; i+=thread_ct){
        for (int t=0; t < thread_ct && t+i <= max/2; t++){//extra threads do no harm.
            x[t] = (one_factor_s){.i=i+t, .max=max,
                            .factor_ct=factor_ct, .mutexes=mutexes, .mutex_ct=mutex_ct};
            pthread_create(&threads[t], NULL, mark_factors, &x[t]);
        }
        for (int t=0; t< thread_ct; t++)
```

```
            pthread_join(threads[t], NULL);
    }
    FILE *o=fopen("xpt", "w");
    for (long int i=0; i < max; i ++){
        int factors = factor_ct[i];
        fprintf(o, "%i %li\n", factors, i);
    }
    fclose(o);

    int max_factors = get_max(factor_ct, max);
    long int tally[max_factors+1];
    memset(tally, 0, sizeof(long int)*(max_factors+1));

    tally_s thread_info[thread_ct];
    for (int i=0; i< thread_ct; i++){
        thread_info[i] = (tally_s){.this_thread=i, .thread_ct=thread_ct,
                                   .tally=tally, .max=max, .factor_ct=factor_ct,
                                   .mutexes=mutexes, .mutex_ct =mutex_ct};
        pthread_create(&threads[i], NULL, add_tally, &thread_info[i]);
    }
    for (int t=0; t< thread_ct; t++)
        pthread_join(threads[t], NULL);

    for (int i=0; i<=max_factors; i++)
        printf("%i\t%li\n", i, tally[i]);
}
```

A warning: the curly brace at the end of a `for` loop ends the scope, so any locally-declared variables are tossed out. Normally, we don't get to the end of scope until all called functions have returned, but the entire point of `pthread_create` is that the main function continues while the thread runs. Thus, this code fails:

```
for (int i=0; i< 10; i++){
    tally_s thread_info = {...};
    pthread_create(&threads[i], NULL, add_tally, &thread_info);
}
```

because the location `add_tally` is reading will disappear by the time it is put to use. Moving the declaration outside the loop—

```
tally_s thread_info;
for (int i=0; i< 10; i++){
    thread_info = (tally_s) {...};
    pthread_create(&threads[i], NULL, add_tally, &thread_info);
}
```

—also doesn't work, because what is stored at `thread_info` changes on the second iteration of the loop, even while the first iteration is looking at that location. Thus, the example sets up an array of function inputs, which guarantees that one thread's info will persist and not be changed while the next thread is being set up.

What does pthreads give us for all that extra work? There are more options. For example, the `pthread_rwlock_t` is a mutex that blocks if any thread is writing to the thread, but does not block multiple simultaneous reads. The `pthread_cont_t` is a semaphore that can be used to block and unblock multiple threads at once on a signal, and could be used to implement the special cases of read-write locks or general mutexes. But with great power come great ways to screw things up. It is easy to write fine-tuned pthreaded code that runs better than OpenMP on the test computer and is all wrong for next year's computer.

The OpenMP spec makes no mention of pthreads, and the POSIX spec makes no mention of OpenMP, so there is no pseudo-legal document that requires the meaning of *thread* used by OpenMP match the meaning of *thread* in POSIX. However, the authors of your compiler had to find some means of implementing OpenMP, POSIX or Windows, and C thread libraries, and they were working too hard by far if they wrote separate threading procedures for each specification. The few C thread libraries I have seen to date have documentation explicitly stating that they are built on pthreads. Further, your computer's processor does not have a pthreads core and a separate OpenMP core: it has one set of machine instructions to control threads, and it is up to the compiler to reduce the syntax of all the standards into that single set of instructions. Therefore, it is not unreasonable to mix and match these standards, generating threads via an easy OpenMP `#pragma` and using pthread mutexes or C atoms to protect resources, or starting with OpenMP and then adopting one segment to pthreads as needed.

C atoms The C standard includes two headers, `stdatomic.h` and `threads.h`, which specify functions and types for atomic variables and threads. Here, I will give an example with pthreads to do the threading and C atoms to protect the variables.

There are two reasons why I'm not using C threads. First, outside of little in-text snippets, I don't publish code samples unless I've tested them first, and as of this writing, I couldn't get a compiler/standard library pair that implements `threads.h`. This is understandable, because of the second reason: C threads are modeled on C++ threads, which are modeled on a least-common-denominator between Windows and POSIX threads, and so C threads are largely a relabeling without the addition of many especially exciting features. C atoms do bring new things to the table, though.

Given a type `my_type`, be it a struct, a scalar, or whatever, declare it to be atomic via:

_Atomic(my_type) x

In the near future,

_Atomic my_type x

will work, but I don't have a compiler on hand yet that will read it. For the integer types defined in the standard, you can reduce this to `atomic_int x`, `atomic_bool x`, et cetera.

Simply declaring the variable as atomic gives you a few things for free: `x++`, `--x`, `x *= y`, and other simple binary operation/assignment steps work in a thread-safe manner (C11 §6.5.2.4(2) and §6.5.16.2(3)). These operations, and all of the thread-safe operations below, are all `seq_cst`, as discussed in the context of the OpenMP atoms above (in fact, OpenMP v4 p 132 says OpenMP atoms and C11 atoms should have similar behavior). However, other operations will have to happen via atom-specific functions:

- Initialize with `atomic_init(&your_var, starting_val)`, which sets the starting value "while also initializing any additional state that the implementation might need to carry for the atomic object." (C11 §7.17.2.2(2)). This is not thread-safe, so do it before you disperse all your threads.
- Use `atomic_store(&your_var, x)` to assign `x` to `your_var` thread-safely.
- Use `x = atomic_load(&your_var)` to thread-safely read the value of `your_var` and assign it to `x`.
- Use `x = atomic_exchange(&your_var, y)` to write `y` to `your_var` and copy the previous value to `x`.
- Use `x = atomic_fetch_add(&your_var, 7)` to add seven to `your_var` and set `x` to the pre-addition value; `atomic_fetch_sub` subtracts (but there is no `atomic_fetch_mul` or `atomic_fetch_div`).

There is a lot more to atomic variables, partly because the C committee is hoping that future implementations of threading libraries will use these atomic variables to produce mutexes and other such constructs within standard C. Because I assume that you are not designing your own mutexes, I won't cover those facilities (such as the `atomic_compare_exchange_weak` and `_strong` functions; see Wikipedia[3] for details).

Here is the example rewritten with atomic variables. I use pthreads for the threading, so it is still pretty verbose, but the verbiage about mutexes is eliminated.

```
/* Compile with:
export CFLAGS="-g -Wall -O3 --std=c11 -pthread -latomic"
make c_factors
*/
#include <pthread.h>
#include <stdatomic.h>
#include <stdlib.h> //malloc
#include <string.h> //memset
#include <stdio.h>

int get_max_factors(_Atomic(int) *factor_ct, long int max){
    //single-threading to save verbiage.
    int global_max=0;
    for (long int i=0; i< max; i++){
        if (factor_ct[i] > global_max)
            global_max = factor_ct[i];
    }
    return global_max;
}
```

[3] `https://en.wikipedia.org/wiki/Compare_and_swap`

```
typedef struct {
    _Atomic(long int) *tally;
    _Atomic(int) *factor_ct;
    int max, thread_ct, this_thread;
} tally_s;

void *add_tally(void *vin){
    tally_s *in = vin;
    for (long int i=in->this_thread; i < in->max; i += in->thread_ct){
        int factors = in->factor_ct[i];
        in->tally[factors]++;
    }
    return NULL;
}

typedef struct {
    long int i, max;
    _Atomic(int) *factor_ct;
} one_factor_s;

void *mark_factors(void *vin){
    one_factor_s *in = vin;
    long int si = 2*in->i;
    for (long int scale=2; si < in->max; scale++, si=scale*in->i) {
        in->factor_ct[si]++;
    }
    return NULL;
}

int main(){
    long int max = 1e7;
    _Atomic(int) *factor_ct = malloc(sizeof(_Atomic(int))*max);

    int thread_ct = 4;
    pthread_t threads[thread_ct];

    atomic_init(factor_ct, 0);
    atomic_init(factor_ct+1, 1);
    for (long int i=2; i< max; i++)
        atomic_init(factor_ct+i, 2);

    one_factor_s x[thread_ct];
    for (long int i=2; i<= max/2; i+=thread_ct){
        for (int t=0; t < thread_ct && t+i <= max/2; t++){
            x[t] = (one_factor_s){.i=i+t, .max=max,
                                  .factor_ct=factor_ct};
            pthread_create(&threads[t], NULL, mark_factors, x+t);
        }
        for (int t=0; t< thread_ct && t+i <=max/2; t++)
```

```
            pthread_join(threads[t], NULL);
    }

    int max_factors = get_max_factors(factor_ct, max);
    _Atomic(long int) tally[max_factors+1];
    memset(tally, 0, sizeof(long int)*(max_factors+1));

    tally_s thread_info[thread_ct];
    for (int i=0; i< thread_ct; i++){
        thread_info[i] = (tally_s){.this_thread=i, .thread_ct=thread_ct,
                        .tally=tally, .max=max,
                        .factor_ct=factor_ct};
        pthread_create(&threads[i], NULL, add_tally, thread_info+i);
    }
    for (int t=0; t< thread_ct; t++)
        pthread_join(threads[t], NULL);

    for (int i=0; i<max_factors+1; i++)
        printf("%i\t%li\n", i, tally[i]);
}
```

Atomic structs Structs can be atomic. However, "Accessing a member of an atomic structure or union object results in undefined behavior." (C11 §6.5.2.3(5)) This does not make them useless, but it does dictate a certain procedure for working with them:

- Copy the shared atomic struct to a not-atomic private struct of the same base type: `struct_t private_struct = atomic_load(&shared_struct).`
- Mess around with the private copy.
- Copy the modified private copy back to the atomic struct: `atomic_store(&shared_struct, private_s`

If there are two threads that could modify the same struct, you still have no guarantee that your structs won't change between the read in step one and the write in step three. So you will probably still need to ensure that only one thread is writing at a time, either by design or with mutexes. But you no longer need a mutex for reading a struct.

Here is one more prime finder. The knock-out method above has proven to be much faster on my tests, but this version nicely demonstrates an atomic struct.

I want to check that a candidate is not evenly divisible by any number less than itself. But if a candidate number is not divisible by 3 and not divisible by 5, then I know it is not divisible by 15, so I need only check whether a number is divisible by smaller primes. Further, there is no point checking past half of the candidate, because the largest possible factor is the one that satisfies 2 * factor = candidate. So, in pseudocode:

```
for (candidate in 2 to a million){
    is_prime = true
    for (test in (the primes less than candidate/2))
        if (candidates/test has no remainder)
            is_prime = false
}
```

The only problem now is to keep that list of (the primes less than candidate/2). We need a size-modifiable list, which means that a `realloc` will be necessary. I am going to use a raw array with no end-marker, so I also need to keep the length. This is a perfect candidate for an atomic struct, because the array itself and the length must be kept in sync.

In the code snippet below, `prime_list` is a struct to be shared across all threads. You can see that its address is passed as a function argument a few times, but all other uses are in `atomic_init`, `atomic_store`, or `atomic_load`. The `add_a_prime` function is the only place where it is modified, and it uses the above workflow of copying to a local struct and working with the local. It is wrapped by a mutex, because two simultaneous `realloc`s would be a disaster.

The `test_a_number` function has one other trick: it waits until the `prime_list` has primes up to candidate/2 before proceeding, lest some factor be missed. After that, the algorithm is as per the pseudocode above. Note that there are no mutexes anywhere in this part of the code, because it only uses `atomic_load` to read the struct.

As a final note, I init the list three times: once using a macro defined by the C standard for this purpose, once using the function defined for this purpose, and once by storing a value there. The use of the macro and the function is just to show you both options; the extra store is because Clang crashes if I don't do it. I'm not sure if it's a bug in Clang's release candidate, my patchy installation, or my own code. That's life on the cutting edge.

```c
/* Compile with:
export CFLAGS="-g -Wall -O3 --std=c11 -pthread -latomic"
make c_primes
*/
#include <stdio.h>
#include <stdatomic.h>
#include <stdlib.h> //malloc
#include <stdbool.h>
#include <pthread.h>

typedef struct {
    long int *plist;
    long int length;
    long int max;
} prime_s;

int add_a_prime(_Atomic (prime_s) *pin, long int new_prime){
    prime_s p = atomic_load(pin);
    p.length++;
    p.plist = realloc(p.plist, sizeof(long int) * p.length);
    if (!p.plist) return 1;
    p.plist[p.length-1] = new_prime;
    if (new_prime > p.max) p.max = new_prime;
    atomic_store(pin, p);
    return 0;
}
```

```
typedef struct{
    long int i;
    _Atomic (prime_s) *prime_list;
    pthread_mutex_t *mutex;
} test_s;

void* test_a_number(void *vin){
    test_s *in = vin;
    long int i = in->i;
    prime_s pview;
    do {
        pview = atomic_load(in->prime_list);
    } while (pview.max*2 < i);

    bool is_prime = true;
    for (int j=0; j < pview.length; j++)
        if (!(i % pview.plist[j])){
            is_prime = false;
            break;
        }

    if (is_prime){
        pthread_mutex_lock(in->mutex);
        int retval = add_a_prime(in->prime_list, i);
        if (retval) {printf("Too many primes.\n"); exit(0);}
        pthread_mutex_unlock(in->mutex);
    }
    return NULL;
}

int main(){
    prime_s inits = {.plist=NULL, .length=0, .max=0};
    _Atomic (prime_s) prime_list = ATOMIC_VAR_INIT(inits);

    pthread_mutex_t m;
    pthread_mutex_init(&m, NULL);

    int thread_ct = 3;
    test_s ts[thread_ct];
    pthread_t threads[thread_ct];

    add_a_prime(&prime_list, 2);
    long int max = 1e6;
    for (long int i=3; i< max; i+=thread_ct){
        for (int t=0; t < thread_ct && t+i < max; t++){
            ts[t] = (test_s) {.i = i+t, .prime_list=&prime_list, .mutex=&m};
            pthread_create(threads+t, NULL, test_a_number, ts+t);
        }
        for (int t=0; t< thread_ct && t+i <max; t++)
        pthread_join(threads[t], NULL);
    }

    prime_s pview = atomic_load(&prime_list);
    for (int j=0; j < pview.length; j++)
        printf("%li\n", pview.plist[j]);
}
```

Conclusion There is, and always will be, more to threading. It is a topic being actively researched, and there are semester-long courses about how to do it better. For example, the discussion here didn't touch the problem of synchronization across caches, although lazy synchronization can also improve speed. But most of the cases are as easy as adding a single line of code before a `for` loop, and if you are careful about how you write your code and minimize the resources shared across threads, you can make most of your code into one of those easy cases.

References

Clay Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, 2009.

Darryl Gove. *Multicore Application Programming: for Windows, Linux, and Oracle Solaris (Developer's Library)*. Addison-Wesley Professional, 2010.

Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison-Wesley, 2003.

Vladimir Nabokov. *Pale Fire*. G P Putnams's Sons, 1962.