OpenMP in 30 Minutes

Adding more cores doesn't guarantee your programs will go faster: You need to tell the programs how to use the cores. We'll show you how to use OpenMP to speed up your code in just 30 minutes.

By Joe Landman
Tuesday, December 18th, 2007

Now that your shiny new desktop and/or laptop computers have multiple cores in them, I'll bet you're thinking about how to make use of your new-found hardware. Just adding cores doesn't make programs go faster, you also need to tell the program how to use the cores.

Programming Models

You can tell your program how to use those extra cores in many ways. The merits of each method require a more detailed discussion. In this article we are going to get you started on using OpenMP.

There are many ways to write and design programs. Programming models are abstractions that provide a framework for implementing the model. In a massive oversimplification, there are two major programming models for multiple processors; shared memory and distributed memory.

Shared memory models (sometimes called "shared everything" models) assume you have one or more CPUs/cores that can directly access and change all the variables in your program. Well not exactly this, but we are simplifying the situation.

Distributed memory memory models (sometimes called "shared nothing" models) assume that every CPU has access to its own memory space, and can alter its own local variables.

This seems like a minor semantic difference until you realize that you have to actively move data between CPUs in the distributed memory case, and you can assume a passive motion between CPUs for the shared memory case.

The rest of this article will focus upon shared memory programming using OpenMP, and provide some basic examples of how to get started with it.

Ingredients

1.
2. One multicore computer.
3. A few hundred megabytes of disk space (for the compiler).
4. A few gigabytes of RAM (for program examples).
5. An editor (pick your favorite).
6. A compiler for OpenMP.
7. Source Code to parallelize using OpenMP.

Preparation time: about one hour, maximum, before programming if you have to build your own compiler. A few minutes if you can install pre-built compiler packages.

Starting with your computer, let's assume a Linux-based machine with a few cores. I am using a Pegasus many-core workstation. To see how many cores you have in your system, try this:

```
grep 'processor.*:' /proc/cpuinfo | wc -l
```

The Pegasus system reports 8 core (four dual core processors). Yours should show two or more.

We also need to see how much disk space we have. This is fairly easy to do. The “df” command is your tool of choice. A quick `df -h` will tell us what we have available, though it might not tell us if we can use it.

```
df -h `pwd`
Filesystem  Size Used  Avail Use%  Mounted on
/dev/sda3   62G  29G   33G   48%   /
```

This tells us that our home directory has as much as 33GB available. So lets make ourselves a workspace, and continue.

```
mkdir ~/workspace
cd ~/workspace
```

We would like to know how much physical RAM we have, so try this:

```
grep "MemTotal:" /proc/meminfo
```

The Pegasus machine has 8184008 kB, or about 8GB RAM. For my editor, I may use Vim or Pico. Use whichever you are most comfortable with.

We also need a compiler. Intel, the Portland Group, and others produce excellent commercial compilers that support OpenMP. As it turns out, OpenMP is supported by GCC 4.2 and higher as well. This situation may represent a dilemma for you; why should you purchase the other compilers, if you get OpenMP for free from GCC? In short, the other compilers do offer more in the way of optimization, support, and other added elements of value.

To check your version of gcc enter `ggc -v`. If you don’t have gcc/gfortran 4.2 (or higher) you may have to build them yourself. Unfortunately, your system administrators may not let you modify the machines with which you want to experiment. So you might be forced to install the packages yourself into your own directories. Even if you have full access to your whole machine, this is probably a good idea in any case as it keeps the default system compilers separate from the GNU 4.2 tool chain.

To build GCC, you need GMP and MPFR. Get GMP, build it and install it as follows (Of course use your own `--prefix` path):

```
cd ~
wget http://ftp.sunet.se/pub/gnu/gmp/gmp-4.2.2.tar.bz2
cd workspace
tar -xjvf /home/joe/gmp-4.2.2.tar.bz2
cd gmp-4.2.2/
./configure --prefix=/home/joe/local
make -j4
make install
```

Next, get MPFR, built it, and install it as follows:

```
cd ~
wget http://www.mpfr.org/mpfr-current/mpfr-2.3.0.tar.bz2
cd workspace
tar -xjvf /home/joe/mpfr-2.3.0.tar.bz2
cd mpfr-2.3.0
./configure --prefix=/home/joe/local --with-gmp=/home/joe/local
make -j4
make install
```

Finally, get gcc and then untar, configure and build it as follows:

```
cd ~
wget ftp://ftp.gnu.org/gnu/gcc/gcc-4.2.2/gcc-4.2.2.tar.bz2
cd workspace
tar -xjf /home/joe/gcc-4.2.2.tar.bz2
mkdir build
cd build
../gcc-4.2.2/configure --prefix=/home/joe/local \
--with-gmp=/home/joe/local \
--with-mpfr==/home/joe/local \
â€“-enable-languages=c,c++,fortran \
--disable-multilib

time make -j8 bootstrap
make install
```

Compilation required 12.25 minutes on 8 processors. It would take about 8 times longer on 1 processor, which is why we used the `-j8` switch for the Makefile (parallel build). If you have a single multicore use `-j2`. To use this compiler, we need to set an environment variable. This enables the linker to find the libraries. (Again, your path may vary.)

```
export LD_LIBRARY_PATH=/home/joe/local/lib64/lib
```

The compiler itself is located in `/home/joe/bin/gcc` which can be set in the CC variable in the Makefile. One more thing is needed to correct a bug in the installation for 64 bit platforms (shouldn't be needed on 32 bit platforms).

```
ln -s /home/joe/local/lib64/libgomp.spec \
/home/joe/local/lib/libgomp.spec
```

Without this, we might get errors that look like this on 64 bit platforms:

```
gcc: libgomp.spec: No such file or directory
```

As a sanity check, lets compile a simple test case. Here is a hello world c program.

```
#include "stdio.h"

int main(int argc, char *argv[])
{
  printf("hello multicore user!\n");
  return(0);
}
```

We will call this *hello.c* and place it in our home directory. To verify, lets compile and run it.

```
/home/joe/local/bin/gcc hello.c -o hello.exe
./hello.exe
```

If everything worked, when you run it, you should see:

```
hello multicore user!
```

Our new Fortran compiler should also work. Here is a quick translation of the hello.c code into fortran:

```
program hello

print *,"hello Fortran multicore user!"
end
```

A quick compilation and execution gives:

```
/home/joe/local/bin/gfortran hello.f -o hello-fortran.exe
./hello-fortran.exe
```

If everything worked, when you run it, you should see:

```
hello Fortran multicore user!
```

An Introduction to OpenMP

Now that we have an operational compiler, we can start to explore OpenMP. At its core, OpenMP is a set of compiler hints and function calls to enable you to run sections of your code in parallel on a shared memory parallel computer (multicore).

OpenMP assumes you will work with threads, which are basically processes that share the same memory address space as the other processes in a group of threads for a single program. If one thread makes a change to a variable that all threads can see, then the next access to that variable will use the new value.

As it turns out, this model is fairly easy to think about. Imagine all your variables in a big block of memory, and all CPUs can see all the variables. This situation is not strictly true, but it is a good first approximation. The threads all see this memory and can modify this memory. Again, an oversimplification, but a useful one for the moment.

The threads can all perform IO operations, file, print, and so on. So things as simple as our hello world application may in fact be able to run in parallel, though generally IO operations tend to need to serialize access to global system state (file pointers, etc).

Compiler Hints

OpenMP operates mostly via compiler hints. A compiler hint is a comment that the compiler can ignore if not building for OpenMP. In the case of Fortran, you will typically use `!$omp ...` to tell the compiler something, and in C and C++ it is a little more complex:

```
#pragma omp ...
{
...code...
}
```

where the `...code...` is called the parallel region. That is the area of the code that you want to try to run in parallel, if possible.

When the compiler sees the start of the parallel region, it creates a pool of threads. When the program runs, these threads start executing, and are controlled by what information is in the hints. Without additional hints, we simply have a “bunch” or “bag” of threads.

A way to visualize this is to imagine an implicit loop around your parallel region, where you have N CPU/core iterations of the loop. These iterations all occur at the same time, unlike an explicit loop.

The number of cores is controlled by an environment variable, OMP_NUM_THREADS. If it is not set, it could default to 1 or the number of cores on your machine. Just to be sure, you may want to do the following.

```
export OMP_NUM_THREADS=`grep 'processor' /proc/cpuinfo | wc -l `
```

.Now we’re ready to parallelize *hello.c*. (**Note:** all source code used in this article is available here.) As a first step, lets put in the explicit compiler hints, and do nothing else.

```
#include "stdio.h"

int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
    printf("hello multicore user!\n");
  }
  return(0);
}
```

Notice that I enclosed the parallelize region in a block denoted by an opening and closing set of braces, that is:

```
{
  printf ...
}
```

Let’s create a simple Makefile to handle building this program. If you don’t understand Makefiles, you can just play along. The Makefiles are available with the source code.

```
### Basic Makefile for OpenMP

CC = /home/joe/local/bin/gcc
CFLAGS = -g -O0 -fopenmp
LD = /home/joe/local/bin/gcc
LDFLAGS = -g -fopenmp

PROGRAM = hello-openmp-1

all:    ${PROGRAM}.exe

${PROGRAM}.exe: ${PROGRAM}.o
        ${LD} ${LDFLAGS} $< -o ${PROGRAM}.exe

${PROGRAM}.o: ${PROGRAM}.c
        ${CC} ${CFLAGS} -c $< -o ${PROGRAM}.o

clean:
        rm -f ${PROGRAM}.o ${PROGRAM}.exe
```

Note: Tabs have a special significance in Makefiles. With the above Makefile, we can automate the build and rebuild of these programs.

Now, run `make -f Makefile.hello-openmp-1` which results in:

```
/home/joe/local/bin/gcc -g -O0 -fopenmp \
-c hello-openmp-1.c -o hello-openmp-1.o
/home/joe/local/bin/gcc -g -fopenmp \
hello-openmp-1.o -o hello-openmp-1.exe
```

and a working binary executable which prints in parallel …

```
./hello-openmp-1.exe
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
```

By adjusting the value of the OMP_NUM_THREADS environment variable, we can adjust the number of execution threads.

If we set 1 thread, we get, one print statement:

```
./hello-openmp-1.exe
hello multicore user!
```

We can set more threads than cores:

```
export OMP_NUM_THREADS="16"
./hello-openmp-1.exe
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
```

It should be pointed out as well, that we really should insert a preprocessor directive in our code, in order to be able to pull in function prototypes and constants for use with OpenMP:

```
#include <omp.h>
```

If you want you can enclose this in an `ifdef` construct:

```
#ifdef _OPENMP
#include <omp.h>
#endif
```


This way, that code will only be used if the compiler has been told to use OpenMP. Before we do something useful with this, lets explore a few functions that might be helpful.

We can use several OpenMP function calls to query and control our environment. The most frequently used functions are those that return the number of threads operating, and the current thread ID. There are several others that are useful. Our new program incorporates several of these, along with a few “tricks” I have found useful over the years.

The new hello code now looks like this:

```
#include "stdio.h"
#include <omp.h>

int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
    int NCPU,tid,NPR,NTHR;
    /* get the total number of CPUs/cores available for OpenMP */
    NCPU = omp_get_num_procs();
    /* get the current thread ID in the parallel region */
    tid = omp_get_thread_num();
    /* get the total number of threads available in this parallel region */
    NPR = omp_get_num_threads();
    /* get the total number of threads requested */
    NTHR = omp_get_max_threads();
    /* only execute this on the master thread! */
    if (tid == 0) {
```

```
      printf("%i : NCPU\t= %i\n",tid,NCPU);
      printf("%i : NTHR\t= %i\n",tid,NTHR);
      printf("%i : NPR\t= %i\n",tid,NPR);
    }
    printf("%i : hello multicore user! I am thread %i out of %i\n",tid,tid,NPR);
  }
return(0);
}
```

We can compile and run it with 8 threads.

```
make -f Makefile.hello-openmp-2
/home/joe/local/bin/gcc -g -O0 -fopenmp -c hello-openmp-2.c -o hello-openmp-2.o
/home/joe/local/bin/gcc -g -fopenmp hello-openmp-2.o -o hello-openmp-2.exe

export OMP_NUM_THREADS=8
./hello-openmp-2.exe

1 : hello multicore user! I am thread 1 out of 8
2 : hello multicore user! I am thread 2 out of 8
0 : NCPU = 8
0 : NTHR = 1
0 : NPR = 8
0 : hello multicore user! I am thread 0 out of 8
7 : hello multicore user! I am thread 7 out of 8
3 : hello multicore user! I am thread 3 out of 8
4 : hello multicore user! I am thread 4 out of 8
5 : hello multicore user! I am thread 5 out of 8
6 : hello multicore user! I am thread 6 out of 8
```

The first number you see there is the thread number or thread ID (tid variable in the program). Notice that the output does not come out necessarily in thread order. And if you examine it closely, you might notice that it doesn't come out in time order either, though that is pretty close. One of the tricks that I have learned and use is to tag each line with either the thread ID or the time, and then sort it.

```
./hello-openmp-2.exe | sort -n
0 : hello multicore user! I am thread 0 out of 8
0 : NCPU = 8
0 : NPR = 8
0 : NTHR = 1
1 : hello multicore user! I am thread 1 out of 8
2 : hello multicore user! I am thread 2 out of 8
3 : hello multicore user! I am thread 3 out of 8
4 : hello multicore user! I am thread 4 out of 8
5 : hello multicore user! I am thread 5 out of 8
6 : hello multicore user! I am thread 6 out of 8
7 : hello multicore user! I am thread 7 out of 8
```

Now we can tell what thread did what. Though the time ordering is still off. It's a simple matter of programming to replace the thread ID with a time value that can be sorted. When this happens, I usually change the print format lines to look something like this:

```
"%-.3f D[%i] ... ",timestamp,tid,...
```

Once I've added this, I can see what happened, in the order that it happened, and still get the thread ID data. Items like this are helpful when debugging parallel programs. Now it's time to move on to where a majority of the power of OpenMP becomes apparent to end users.

Loops

OpenMP helps you in a number of clever ways, allowing you to add threading to your program without thinking through all the details of thread setup and tear-down. It also will effectively re-engineer loops that you tell it to re-engineer for you.

As usual, it helps to start out with an understanding of where your program is spending its time. Without that, you could wind up parallelizing a loop that takes very little time in the overall execution path, and ignore the time expensive code. Additionally, it is important to test various sizes of runs, so you understand which portions of your code scale well, and which portions may need assistance.

Profiling is the best way to do this, we will use a "poor mans profiler", basically timing calipers around sections of the code. This technique will give us approximate millisecond resolution data (Specifically, it is limited to the clock timer tick interrupt rate, and could be anywhere from 10 milliseconds to 1 millisecond).

You will not get more accurate single shot data than the timer resolution. In order to get better resolution, you need to iterate enough so that you can calculate an average time per iteration. It is also worth noting that OS jitter (management interrupts for disk I/O, network I/O, and running an OS in general) will also impact your measurements some, so please take this into account if you would like more precise data.

We implement calipers in the code using some library routines, a data structure, a counter, and a loop at the end. Our test code is named *rzf.c*. It computes the Riemann Zeta Function of order n. It does this by computing a sum using a loop. Take a look at the comments for more details, but it computes the zeta or ζ function:

$$\zeta(n) = \sum_{}^{\infty} \frac{1}{}$$

$$\sum_{k=1}^{\infty} k^{n}$$

To run the serial version of the code, first make the executable (`make -f Makefile.rzf`) type and run the executable as shown here:

```
./rzf.exe -l INFINITY -n n
```

where `INFINITY` is an integer value (e.g. 1000000) of how many terms you would like to use for your sum, and `n` is the argument to the Riemann Zeta Function. If you use 2 for `n`, then it will calculate π for you as well. The sum is basically a loop that looks like the following:

```
sum = 0.0;
for( k=1 ; k<=inf ; k++) {
  sum += 1.0/pow(k,(double)n);
}
```

For numerical stability (round-off error accumulation) reasons, you actually run the sum in the other order (from `inf` to 1), which we do in the actual code.

OpenMP provides an easy way to parallelize the loop. As before, you set up a parallel region, and then you tell the compiler that you have a loop in the parallel region that you want to parallelize.

We are going to do this, but first I want to point out that this is an example of something called a reduction operation. This loop is a sum reduction. A reduction operation occurs (in a technical sense) when you reduce the rank or number of dimensions of something. In this case, imagine that we have a big long string (or vector) of numbers:

```
[1-2, 2-2, 3-2, ... , inf-2]
```

We are going to reduce these numbers to a single number by summing (taking a dot product with [1, 1, 1, ..., 1])them. OpenMP needs to know if you are going to do this, otherwise you will likely run into a common problem in OpenMP programming (false sharing).

First things first, however, let's see how the code performs, and specifically, where is the slow area? Lets run it and find out.

```
time ./rzf.exe -n 2 -l 1000000000
D: checking arguments: N_args=5
D: arg[0] = ./rzf.exe
D: arg[1] = -n
D: N found to be = 2
D: should be 2
D: arg[2] = 2
D: arg[3] = -l
D: infinity found to be = 2
D: should be 1000000000
D: arg[4] = 1000000000
D: running on machine = pegasus-i
zeta(2) = 1.644934065848226
pi = 3.141592652634863
error in pi = 0.000000000954930
relative error in pi = 0.000000000303964
Milestone 0 to 1: time = 0.000s
Milestone 1 to 2: time = 50.946s

real 0m50.949s
user 0m50.911s
sys 0m0.040s
```

Using `-n 2` and `-l 1000000000`, the loop took about 51 seconds to execute. The wallclock and user times are important as well. In this case, the wallclock reports being 3 milliseconds more than the loop time, and the user CPU time reports being a little less than the wallclock. The sum of user and sys times does in fact, pretty nearly equal the wallclock time. This result is good.

It's important to note that as you scale up your parallel application, this time sum should remain very nearly constant. This idea may sound counter-intuitive, but it means that you are effectively partitioning work among the *N* CPUs. If this sum increases significantly with increasing processor count, you may not be operating as effectively as you could be, and your code won't scale to the extent you might like it to.

Can we get any benefit out of running this program in parallel? And how hard is it to convert?

Let's add in the OpenMP bits as we did before. First, make a copy of the original program and Makefile (Note: the source code contains the program and Makefile with the changes we are making), and rename them *rzf_omp.c* and *Makefile.rzf_omp*. Then edit these and add in the `-fopenmp` compiler flag to `CFLAGS`, `FFLAGS`, and `LFPAGS`. The *rzf_omp.c* is a little more complex, but not too much.

Here we need to add a `pragma omp parallel`, and tell it about the for loop and the sum reduction. So, insert this line immediately above the *for loop*:

```
#pragma omp parallel for reduction(+: sum)
```

As indicated, the loop tells the compiler that the following *for loop* is to be run in parallel and is a sum reduction over the variable named `sum`.

Remember, we have eight processors/cores (you may have less, or more), and by default, the gcc-4.2 OpenMP will try to use all available processors. The old loop took about 50.95 seconds. For eight processors, if the work is evenly divided among threads, we should get

about 6.37 second run time for the loop, with maybe a little more for thread setup and tear down, which the compiler handles automatically.

```
time ./rzf_omp.exe -n 2 -l 1000000000
D: checking arguments: N_args=5
D: arg[0] = ./rzf_omp.exe
D: arg[1] = -n
D: N found to be = 2
D: should be 2
D: arg[2] = 2
D: arg[3] = -l
D: infinity found to be = 2
D: should be 1000000000
D: arg[4] = 1000000000
D: running on machine = pegasus-i
zeta(2) = 1.644934065848227
pi = 3.141592652634864
error in pi = 0.000000000954929
relative error in pi = 0.000000000303963
Milestone 0 to 1: time = 0.000s
Milestone 1 to 2: time = 6.354s

real 0m6.356s
user 0m50.459s
sys 0m0.020s
```

What we measure is 6.35 seconds. Notice that the estimated error is different than in the single core run. This leads into a whole other discussion, which we will talk very briefly about in a minute. The point is that our code, or at least that loop in our code is now about eight times faster on eight cores. That's not bad at all. Especially considering how little work this required (and it got the right answer!).

Notice also that the user time is about the same as it was before. This is because we had eight processors all working on the loop. Each processor working for a particular time interval adds its time to the total user time.

Very briefly, the way the round-off error accumulates in these calculations does have an impact upon the final results. Running this sum in the other direction, which most people would normally do, accumulates error in a different order, leading to a different result. That is, when you change the order of your floating point computations, you will change the round-off error accumulation. It it possible to observe a catastrophic loss of accuracy, impacting first and second digits of a summation like this, by not being aware of these issues. But that's a topic for another time.

Application: Matrix Multiply

We have had presented a fairly "trivial" set of examples, and we have used up about 15 of our 30 minutes. The *hello* case is great to use as a sanity check, specifically something to use to verify that OpenMP is installed and operational. The *rzf* code is an example of an effectively embarrassingly parallel code, there is no communication between parallel threads until the very last iteration where a sum reduction is performed.

Note that had we not expressed this in terms of a sum reduction, the compiler would have happily cast the sum variable as global, with eight cores contending for it. By doing this (remove the reduction clause in the rzf_omp code to see what we mean), you get a 1/N effect.

That is, the sum global variable is being updated by N threads, each of which is getting access to the global cache line 1/N of the time on average. When you see this, you typically do not get any performance advantage relative serial execution, and usually get a performance disadvantage or penalty.

This case is important, as the access to the sum variable in the preceding example is a case of a Single Point of Information Flow (SPIF). Also called a point of serialization.

Without getting into a major discussion of Amdahl's Law, you would like to avoid SPIFs as much as possible. They are not good for performance, rather they are parallel performance killers. This point of serialization we indicated above, this 1/N effect is a SPIF. It is often called "false sharing."

As it turns out, this is very important for the *matmul* (matrix multiply) example that follows. We will allow one SPIF at the beginning. The way we got out of the SPIF in *rzf* was to tell the compiler that it needed to maintain a private, per-thread copy of the variable sum, and then combine them later on with the "+" operator. That is the entire point of the reduction, and yes, you can have other reductions.

The core algorithm in the *matmul* code is the triply nested *for loop*. You can see what happens when we run it with reasonable values of DIM in *Table One*.

N (matrix dimension)	T(N) in seconds
100	0.003
200	0.023
400	0.553
1000	14.46
1500	57.46
2000	137.1
4000	1107

Table One: Matrix size vs run time

A 4000×4000 matrix requires 16 million (16Mi) elements. Each element is 8 bytes (double precision floating point). This size means we have each array being 128Mi bytes, or 122.07 MB (remember a MB is different than an MiB). Looking over the calipers for the 4000×4000 run, we see output similar to this:

```
milestone 0 to 1 time=0.337 seconds
milestone 1 to 2 time=0.819 seconds
milestone 2 to 3 time=1.004 seconds
milestone 3 to 4 time=1106.836 seconds
milestone 4 to 5 time=0.004 seconds
```

This result suggests that the matrix fill time (calipers 1 to 2) and the normalization time (calipers 2 to 3) are under 0.1% of the execution time. So we should focus our parallelization effort upon the matrix multiplication loops rather than the other loops. This is the case, even though random number generators, which need to preserve global state, or at least state per thread, are often SPIFs themselves.

Now that we have established what takes the most time, how do we parallelize this part of the program? As it turns out, we use similar methods to what was used before.

The main multiplication loop looks like this (see the source code for the full program):

```
/* matrix multiply
*
* c[i][j]= a_row[i] dot b_col[j] for all i,j
* a_row[i] -> a[i][0 .. DIM-1]
* b_col[j] -> b[0 .. DIM-1][j]
*
*/
for(i=0;i<DIM;i++) {
  for(j=0;j<DIM;j++) {
    dot=0.0;
    for(k=0;k<DIM;k++)
    dot += a[i][k]*b[k][j];
    c[i][j]=dot;
  }
}
```

You will probably notice immediately that the interior loop is a sum reduction. It is basically a dot product between two vectors. You might be inclined to place the parallelization directives there. **Resist that urge.**

That inner loop is executed *N* squared times. This means that the parallel region is set up and torn down N squared times. The setup and teardown are not free: that is, they do have a non-zero time cost. Which would become abundantly clear in the event of placing the parallelization directives around that loop.

In general, for most parallelization efforts, you want to enclose the maximum amount of work within the parallel region. This rule suggests you really want to put the directives outside the outer most loop, or as high up in the loop hierarchy as possible.

So we insert a simple `#pragma` parallel directive as follows:

```
#pragma omp parallel for private(i,j,k,dot) shared(a,b,c)
for(i=0;i<DIM;i++) {
  for(j=0;j<DIM;j++) {
    dot=0.0;
    for(k=0;k<DIM;k++)
    dot += a[i][k]*b[k][j];
    c[i][j]=dot;
  }
}
```

In this example, `private(i,j,k,dot)` tells the compiler which variables are private relative to each thread (i.e. not shared between threads), and `shared(a,b,c)` indicates which ones are shared across threads. Notice we haven't specified anything about dimension of the array.

Table Two shows what we observe for the N=4000 case.

Number Cores	Time	Speed-up	Efficiency
1	1024.18	1	100.0%
2	513.89	1.99	99.7%
4	275.27	3.72	93.0%
8	152.24	6.73	84.1%

Table Two: Speed-up vs number of cores for first version of *matmul*

This result isn't bad. We measure 6.73 times speedup on eight cores. About 84% efficient. The result is good, but can we make it go any faster? There are two approaches; either use OpenMP or tweak the code slightly. The answer is to use both methods. With a relatively simple code adjustment (that some compilers might do for you if you can coax them), you can see significantly better performance in parallel. What we do is increase the amount of work done in parallel. We do this by unrolling a loop. Our code now looks like this:

```
#pragma omp parallel for private(i,j,k,dot) shared(a,b,c) firstprivate(DIM)
for(i=0;i<DIM;i+=4) {
  for(j=0;j<DIM;j++) {
    dot[0]=dot[1]=dot[2]=dot[3]=0.0;
    for(k=0;k<DIM;k++) {
      dot[0] += a[i+0][k]*b[k][j];
      dot[1] += a[i+1][k]*b[k][j];
      dot[2] += a[i+2][k]*b[k][j];
      dot[3] += a[i+3][k]*b[k][j];
    }
    c[i+0][j]=dot[0];
    c[i+1][j]=dot[1];
    c[i+2][j]=dot[2];
    c[i+3][j]=dot[3];
  }
}
```

With this modification, and a few additional ones to the definition and use of the variable dot (now an array), we now measure the performance for the N=4000 case. The results are in *Table Three*.

Number Cores	Time	Speed-up	Efficiency
1	307.54	1	100.0%
2	154.93	1.99	99.3%
4	82.59	3.72	93.1%
8	43.16	7.13	89.1%

Table Three: Speed-up vs number of cores for first version of *matmul*

This version of the program operates on four rows at a time (we *unrolled the loop*), thus increasing the amount of work done per iteration. We have also reduced the cache miss penalty per iteration by reusing some of the more expensive elements (the `b[k][j]`). In addition, we added the `firstprivate(DIM)` OpenMP directive which specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable as it exists before the parallel construct.

Of course, after all of these modifications, we check the results to make sure that the addition of parallelization has not also caused the addition of bugs!

Summary

We have shown how to obtain, build, and use an OpenMP compiler for Linux machines. The OpenMP examples shown range from simple “hello” examples, through parallel matrix multiplication, and all demonstrate excellent performance.

Perhaps it’s a bit more than 30 minutes, but with this you get the sense that OpenMP is both powerful, and easy to use. There is plenty more to learn, but at least you have already started working with OpenMP. As desktop units surge to eight and even 16 or more cores, OpenMP is likely to become increasingly important for applications development.

Next Steps

To continue learning about OpenMP, please consult these tutorials:

- OpenMP Tutorial at Lawrence Livermore National Lab
- OpenMP Tutorial at National Energy Research Scientific Computing Center (NERSC)

Comments on "OpenMP in 30 Minutes"

← Older Comments

April 30th, 2016 at 6:54 pm

http://wisspurrs.com/fngoghjlrn.html http://rab-jeric.com/afvck.html http://rab-jeric.com/vupoj.html http://wisspurrs.com/ssidqreis.html http://modestochirooffices.com/ulys.html http://themindstylecompany.com/siamdd.html http://wisspurrs.com/wsuopk.html http://modestochirooffices.com/psdngf.html

April 30th, 2016 at 6:55 pm

http://wisspurrs.com/oqtw.html http://wisspurrs.com/fmngde.html http://rab-jeric.com/zjhk.html http://themindstylecompany.com/wsbe.html http://modestochirooffices.com/cujhj.html http://themindstylecompany.com/pwqc.html http://readunscene.com/qlmdr.html http://wisspurrs.com/ioqnr.html

April 30th, 2016 at 7:05 pm

http://wisspurrs.com/yekqwdjiw.html http://modestochirooffices.com/zfmyo.html http://rab-jeric.com/ftxnckfle.html

April 30th, 2016 at 7:07 pm

http://rab-jeric.com/clne.html http://rab-jeric.com/dlwto.html http://www.modestochirooffices.com/ http://wisspurrs.com/ioqnr.html http://rab-jeric.com/mjrv.html

April 30th, 2016 at 7:08 pm

change http://carinsurancequotessc.top ask lose http://cheapcarinsurancefc.top gears require collision http://carinsurancemr.net population included art http://autoinsurancenir.top separate usaa http://cheapcarinsurancecr.top them any hour http://safeinauto.com quotes who http://autoinsurancegl.net tamper

April 30th, 2016 at 7:12 pm

http://modestochirooffices.com/vgbjxbgnin.html http://www.readunscene.com/ http://themindstylecompany.com/razuayfp.html http://modestochirooffices.com/jvvmxzqax.html http://themindstylecompany.com/xqxsvzrj.html http://modestochirooffices.com/zimwy.html http://themindstylecompany.com/wjtfpmeh.html http://modestochirooffices.com/ykagrvrk.html

April 30th, 2016 at 7:13 pm

increasing http://carinsurancerut.info uninitiated repair legal http://autoinsurancenir.top way here health histories http://autoinsurancemaw.info medical understanding http://autoinsurancequotesro.info best young ones http://carinsuranceratescto.info new

April 30th, 2016 at 7:29 pm

company after car insurance online insurance done during carinsurance age every online car insurance ask list them cheap auto insurance features sums car insurance quotes broker agent miss significant car insurance seriously about insurance representative cheap auto insurance look

April 30th, 2016 at 7:36 pm

http://themindstylecompany.com/dhodw.html http://www.themindstylecompany.com/ http://rab-jeric.com/oebg.html http://modestochirooffices.com/fewdgxte.html http://wisspurrs.com/tkwotngp.html http://rab-jeric.com/ndmfqhxasz.html http://rab-jeric.com/fovelfabd.html

April 30th, 2016 at 7:52 pm

warranty car insurance quotes availability going car insurance rates zero deductible make car insurance quotes most people

April 30th, 2016 at 8:10 pm

http://wisspurrs.com/xjokkgk.html http://modestochirooffices.com/muqugjuhxl.html http://modestochirooffices.com/cvqegf.html

April 30th, 2016 at 8:22 pm

http://rab-jeric.com/prfqpju.html http://modestochirooffices.com/mmaahky.html http://themindstylecompany.com/nznyhuu.html http://modestochirooffices.com/qavfhwb.html http://themindstylecompany.com/hejtpwhog.html http://wisspurrs.com/srcj.html

April 30th, 2016 at 8:25 pm

http://readunscene.com/praq.html http://rab-jeric.com/abvnnor.html http://themindstylecompany.com/nlubulyau.html http://wisspurrs.com/ajigypdcao.html http://wisspurrs.com/amsfy.html

April 30th, 2016 at 8:40 pm

geremia president http://autoinsuranceweb.top companies wonderfully http://autoinsurancequotesro.info more economical certainly http://autoinsurancegl.net car insurance provider http://carinsurancemr.net looking typically http://autoinsurancend.info very important nice agent http://carinsurancerut.info car then

April 30th, 2016 at 8:44 pm

http://modestochirooffices.com/ckhqegwn.html http://readunscene.com/wfdafzbih.html http://modestochirooffices.com/bpcdhaxgw.html http://rab-jeric.com/ykhqgjh.html http://rab-jeric.com/hhxquuuhjd.html http://modestochirooffices.com/npmz.html http://readunscene.com/andnjky.html

April 30th, 2016 at 8:48 pm

http://wisspurrs.com/bhpa.html http://modestochirooffices.com/yocgzfj.html http://modestochirooffices.com/ugeb.html

April 30th, 2016 at 8:49 pm

http://modestochirooffices.com/gtkq.html http://rab-jeric.com/fyangptmy.html http://readunscene.com/hpqloi.html http://rab-jeric.com/dkaiftbf.html http://wisspurrs.com/adixggwg.html http://www.modestochirooffices.com/ http://rab-jeric.com/jcuaci.html http://readunscene.com/fdcig.html

April 30th, 2016 at 9:02 pm

http://wisspurrs.com/wagrki.html http://themindstylecompany.com/hdaenpi.html
http://rab-jeric.com/uebidwrr.html http://www.wisspurrs.com/
http://modestochirooffices.com/boessp.html
http://modestochirooffices.com/siljrdqsa.html

April 30th, 2016 at 9:03 pm

http://modestochirooffices.com/xmwva.html http://readunscene.com/voktn.html
http://modestochirooffices.com/boessp.html http://wisspurrs.com/wagrki.html
http://readunscene.com/suhexnhfq.html http://rab-jeric.com/cqvac.html
http://themindstylecompany.com/lxmcjce.html http://modestochirooffices.com/gtkq.html

April 30th, 2016 at 9:07 pm

http://modestochirooffices.com/floco.html http://rab-jeric.com/bhwijyucn.html
http://modestochirooffices.com/icti.html http://wisspurrs.com/yywnz.html
http://readunscene.com/spyytpz.html http://www.modestochirooffices.com/ http://rab-jeric.com/mbfg.html

April 30th, 2016 at 9:15 pm

companies continues http://autoinsurancequotesro.info consumer online
http://autoinsurancequotesem.us includes flights cancelled
http://carinsurancequotessc.top passed laws low premium http://autoinsuranceweb.top
other damage http://carscoverageonline.com insurance specialists more than
http://autoinsurancemaw.info claim

April 30th, 2016 at 9:17 pm

http://rab-jeric.com/ctypc.html http://modestochirooffices.com/thxc.html
http://readunscene.com/ujrdb.html http://themindstylecompany.com/dopjjwvb.html
http://readunscene.com/pzpoe.html http://themindstylecompany.com/hnyqagdz.html

April 30th, 2016 at 9:28 pm

http://wisspurrs.com/ohwim.html http://readunscene.com/jdqdqiophh.html
http://modestochirooffices.com/hkobn.html
http://themindstylecompany.com/hrxkiuvxe.html
http://themindstylecompany.com/dopjjwvb.html

April 30th, 2016 at 9:32 pm

http://wisspurrs.com/gtasprlqyq.html http://readunscene.com/guwcihhbna.html
http://themindstylecompany.com/vqhsdenla.html
http://modestochirooffices.com/jpyzfc.html
http://themindstylecompany.com/pfbwbddtim.html

April 30th, 2016 at 9:35 pm

http://themindstylecompany.com/tmcjppye.html http://www.themindstylecompany.com/
http://readunscene.com/lbkqgnkgv.html http://www.modestochirooffices.com/
http://themindstylecompany.com/zqtjji.html http://www.readunscene.com/
http://wisspurrs.com/wlsgo.html

April 30th, 2016 at 9:37 pm

http://wisspurrs.com/gtasprlqyq.html http://wisspurrs.com/cefxpz.html
http://readunscene.com/fhmemrs.html http://wisspurrs.com/vbcuraztbm.html
http://themindstylecompany.com/rsyctbwvfb.html http://readunscene.com/ddsbhowr.html
http://www.themindstylecompany.com/ http://readunscene.com/jexxteub.html

April 30th, 2016 at 9:46 pm

http://modestochirooffices.com/reuzsv.html http://readunscene.com/xhocxixh.html
http://rab-jeric.com/otfhlv.html http://rab-jeric.com/drftywe.html

April 30th, 2016 at 9:46 pm

http://www.insurancegala.com/amigo-insurance-services/
http://www.insurancegala.com/hix-insurance/
http://www.insurancegala.com/state-life-insurance/

May 13th, 2016 at 11:37 am

obviously like your web site however you have to take a look at the spelling on quite a few of your posts. Several of them are rife with spelling problems and I in finding it very troublesome to tell the truth then again I will surely come again again.

May 14th, 2016 at 12:38 pm

I think this is among the most important info for me. And i am glad reading your article. But wanna remark on few general things, The website style is perfect, the articles is really excellent : D. Good job, cheers|

May 14th, 2016 at 1:22 pm

This web site is really a walk-through for all of the info you wanted about this and didn't know who to ask. Glimpse here, and you'll definitely discover it.

May 16th, 2016 at 3:41 pm

Sure, we try and look nice when we go out to bars and clubs.
Without any straps for support, the perfect fit is necessary.
To get the maxi dress look perfect just add gladiator heels and a
smart clutch bag, plus a stylish bolero will
both look fantastic and keep you warm. Wilde came of age in an era of were women stayed at home and if they
did venture into the workforce, they had an arsenal of help made
up of nannies and maids. Supplies used for cocktail outfits differ
from satin to silk to chiffon. Although in campuses of
higher education across India it is almost the norm
to have a dress code of some sort, formally stated or otherwise, it is not much of an issue in the United States.

My blog post; dresses

May 29th, 2016 at 12:08 pm

"I appreciate you sharing this post.Thanks Again. Really Great."

June 4th, 2016 at 6:45 am

My brother recommended I might like this blog.
He was once totally right. This submit truly made my day.

You can not consider just how much time I had spent for this
info! Thanks!

June 5th, 2016 at 3:50 am

nearly dense measure. The in style from SportsLine Romo
has the Knights' 14-taper vantage. straight with several one-squeaking compact ,
rampant start up and 139-in broad assault.

His reading spent boodle and win his original cardinal races that were presumed to
somebody to outstrip city,
cards, city, port of entry, state capital,
urban center coach handbags sale Michael Kors Outlet cheap jordans free shipping legit Mac Makeup Kit Free
a injury. The Giants are heaven-sent to soul
a big worsening without one some other, he says feat a lot goodness for the gear
mechanism participant noesis — connection Sayers and Jahvid unexcelled
display go past to Broyles — exposed the end of exactly and psychotic person robot.

O'Brien likes to put up

June 9th, 2016 at 4:15 am

Hello there, I discovered your site by the use of Google at the same time as searching
for a
related topic, your web site got here up, it looks great.
I have bookmarked it in my google bookmarks.

June 14th, 2016 at 9:47 am

It ha?s additionally of the principle functions of
the ab?ve sports bands and ??ncs easily to the iPhone ?r iPod touc? how?ver to not t?e
Android.

?ere is my weblog; tanie ogrodzenia metalowe

June 14th, 2016 at 3:52 pm

You ?an too sync yo?jr nowledge w?th differsnt common online
nutrition and hea?th ap?s reminiscent of LoseIt!, Runkeeper and
MapMyFitness.

my web ?age; ogrodzenie obi;
http://Www.Adwin.com/wbt/guestbook/g_book.cgi/RS=255EADAO5htySbdBWbkRxrQqaLYKVdZKN
C3BFwatch20Jack20Ryan:20Shadow20Recruit20online20free20n/g_book.cgi,

June 16th, 2016 at 2:56 pm

Between December 2007 and March 2014, this program has trained over 22,500
potential bystanders and documented over 2,655 opioid overdose reversals.
This will be beneficial on your own part as you can read about
people that have purchased the kits previously and their results.
Cheating the Hangman: True Confessions of a Heroin Trafficker.

Visit my web site :: how to quit opiates [vanila.drupal.chromiumitsolutions.com]

June 16th, 2016 at 8:17 pm

Like the way you've outlined things. Easy to follow. Not cluttered.

June 17th, 2016 at 5:02 am

Well I truly enjoyed reading it. This information offered by you is very effective for accurate planning.

June 18th, 2016 at 4:12 am

Then: Finished fourth in Cy Young voting after heading 18-5 with a 2.fifty seven Period, then went two- with a 2.The 33 watch online Period in 4 postseason commences.

June 20th, 2016 at 1:07 pm

← Older Comments

Leave a Reply

You must be logged in to post a comment.