First steps with OpenMP: Parallel programming for everyone

Remote Data Analysis and Visualization Center, University of Tennessee
http://rdav.nics.tennessee.edu/
Yashema C. Mack (ymack@utk.edu) and Amy F. Szczepański (aszczepa@utk.edu)

Introduction

Introduction and Welcome: We will go around the room and let everyone introduce themselves.

Your presenters today:

Yashema Mack RDAV Scientific Support, National Institute for Computational Sciences

Amy Szczepański Coordinator of Education, Outreach, and Training, RDAV

You can stay in touch with RDAV at:

email: Our email addresses are at the top of this page. Not sure who to ask? The XSEDE operations center at **help@xsede.org** will route your question to the member of the RDAV (or NICS) staff best able to answer it. This really is the best way to get in touch with RDAV.

web: Check us out at **http://rdav.nics.tennessee.edu**. We will have electronic versions of all slides and handouts posted on our web page under the **training** link.

Facebook: Find the **Remote Data Analysis and Visualization Center** on Facebook.

Twitter: Follow us at **@NICS_Nautilus**.

Agenda: Monday, July 18, 1:00 – 5:00.

- Introduction and Overview
- Programming with OpenMP, Part 1
- Break
- Programming with OpenMP, Part 2
- Running your code
- Conclusion

Overview of HPC and HPC architectures: RDAV operates Nautius, an SGI Altix UV 1000 with 1024 cores and 4 TB shared memory. More information in the brief slideshow. Nautilus is an XD resource for data analysis and visualization with allocations managed through XSEDE. We also welcome computation jobs, especially during off hours.

What is OpenMP: See http://www.openmp.org. For information about OpenMP compilers, see http://openmp.org/wp/openmp-compilers/. Later today we will show some example code that will help you determine which version of OpenMP your compiler supports.

Practical guide to programming with OpenMP

Example 1: A gentle introduction to OpenMP: `Hello World` and more.

We start with a very simple `Hello World` program. The only thing "new" is to include the `omp.h` library.

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

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

We compile this with `gcc -fopenmp hello0.c -o hello0`

Other compilers have other options. For example, we could use the Intel compiler with `icc` and the flag `-openmp`. The version of gcc that comes with most recent Macs supports OpenMP 2.5. On Nautilus, the version of gcc installed supports OpenMP 2.5. However, the Intel compiler supports OpenMP 3.0. With our experiments on Nautilus, we've found differences between gcc and icc that we'll talk about later.

The next step is to add some OpenMP **directives**. The first one that we'll use is **`#pragma omp parallel`** which creates a team of threads, each and every one of which will execute the next block of code. What do you think that this program will do?

```
int main (int argc, char *argv[])
{

    printf("Hello world.\n");

    #pragma omp parallel
    {
        printf("Hello from a thread!\n");
    }

    printf("I am sequential now.\n");
    return 0;
}
```

What happens if we compile this as above? What if we leave out the `-fopenmp` flag when compiling?

When we run the program, we may have questions such as:

- How many threads are created?
- Will it be the same on your computer as it is on my computer?
- Can we learn how many threads are present at runtime?
- How much control do we have over the number of threads that are created?

Later today we will learn about OpenMP library functions and environment variables that will help us learn about and control our threads. One example of these functions is `omp_get_thread_num()`. This function returns the number of the current thread.For example, we can change our `Hello World` program by replacing the `printf()` statement in the parallel region with

```
printf("Hello from thread %d.\n", omp_get_thread_num());
```

Now that we have used an OpenMP function instead of just a compiler directive (`#pragma`), we need to be careful to compile with the `-fopenmp` option; otherwise the compiler won't know what to do with `omp_get_thread_num()`. If we were to try compiling without the `-fopenmp` option, we'd get an error like the following:

```
Undefined symbols:
  "_omp_get_thread_num", referenced from:
      _main in ccFGzGuD.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

Later we will learn about conditional compilation and how to let the compiler know that if we aren't using OpenMP that it should just let `omp_get_thread_num()` be 0.

When we run this new version of the program several times, we will likely see that the order in which the threads do the work in non-deterministic. We'll see some examples later of what might happen if we forget that threaded programs run non-deterministically.

As we run these examples, sometimes it is helpful to pipe the output through sort, such as:

```
./hello2 | sort
```

This way we can check to make sure that we get everything that we expect, without worrying about the order.

When we have tasks that should be done by only one thread, we can use the directive `#pragma omp single`.

```
#pragma omp parallel
{
    #pragma omp single
    {
        a = 17;
        printf("Thread %d did the important work of setting a to %d.\n",
        omp_get_thread_num(), a);
    }
}
```

We have no way of predicting which thread will carry out this task. If we use `#pragma omp master`, then this work will be done by the master thread (thread 0). Ideally we want to have memory access done by a thread running on a CPU that is physically near the memory being accessed. This is one reason to favor having memory accesses done by a well-chosen thread. However, with the `#pragma omp master` directive, we need to explicitly say `#pragma omp barrier` at the end to ensure that the values of changed variables are available to all threads.

Here is another example that shows that we can have different behaviors in different threads:

```
int main (int argc, char *argv[]) {

    omp_set_num_threads(8);

    #pragma omp parallel
    {
        printf("Hello from thread %d.\n", omp_get_thread_num());

        if(omp_get_thread_num() == 2)
```

```
        {
            printf("Thread 2 is really excited!!!\n");
        }

        if(omp_get_thread_num() != 1)
        {
            printf("Thread %d sleeping.\n", omp_get_thread_num());
            sleep(20);
        }
        else
        {
            printf("Thread 1 is sick of this program.\n");
            exit(1);
        }
    }

    return 0;
}
```

If we don't want to have every thread do exactly the same thing and if we don't want to micromanage each thread individually, how do we get the threads to work for us in parallel? One very common way to do this is by parallelizing a `for` loop.

Summary of Example 1:

- To get started we need to `#include <omp.h>` and use proper compiler flags.
- The directive `#pragma omp parallel` will start a team of threads.
- To specify that work within the parallel region should be done by only one thread, use `#pragma omp single`.
- The function `omp_get_thread_num()` returns which thread we're in.
- We can use the function `omp_set_num_threads()` to specify how many threads to use.
- We haven't yet talked about how to specify the number of threads via environment variables.
- We've only run the code on a laptop, not on an HPC system.
- We haven't yet done much with variables because OpenMP relies on shared memory, and we haven't yet talked about how to get the threads to share variables appropriately.

Example 2a: Parallelizing a simple `for()` loop.

Suppose we had a `for()` loop, as follows:

```
for(i=0; i<10; i++)
{
    printf("Greetings from i = %d.\n", i);
    sleep(1);
}
```

How could we make this parallel with OpenMP? A first idea (which is wrong) would be to do what we did before and add in the directive `#pragma omp parallel`. We can try this out.

```
/* Wrong way to do this */
#pragma omp parallel
{
    for(i=0; i<10; i++)
    {
        printf("Greetings from i = %d.\n", i);
        sleep(1);
    }
}
```

Since we didn't tell the compiler that this is a `for()` loop, all manner of things go wrong. Every thread tries to execute the loop, and they are all changing the value of **i**. (If you increase the length of the `sleep()` and run it enough times, you can sometimes see runs where **i** takes on the value 11. If we were working with an array indexed by `i`, we could write past the end of the array and get a seg fault.)

The right way to do this is to add the directive **`#pragma omp for`**, as follows:

```
/* Right way to do this */
#pragma omp parallel
{
    #pragma omp for
    for(i=0; i<10; i++)
    {
        printf("Greetings from i = %d.\n", i);
        sleep(1);
    }
}
```

In this case, we see that our loop does *not* run in order. However, each value of **i** is only seen once, and each thread has its own private value of **i**. Later we will learn how to tell threads which variables are shared and which are not.

As a shortcut we can combine `#pragma omp parallel` and `#pragma omp for` with `#pragma omp parallel for`. This example was very trivial on purpose: Not all loops are suitable for parallelizing.

Example 2b: Putting a matrix into upper triangular form. This example helps us to understand which loops are parallelizable and which are not.

Parallelizable

- Number of iterations known upon entry and does not change
- Each iteration independent of all others
- No data dependence

Not Parallelizable

- Conditional loops (many `while` loops)
- Iterator loops (e.g., iterating over a `std::list<...>` in C++)
- Iterations dependent upon each other
- Data dependence

Suppose that we have an $N \times N$ matrix A, and we want to get A into upper triangular form. Furthermore, assume that all of the diagonal entries of A are non-zero. We can use the following code:

```
for (i = 0; i < N-1; i++)
{
    for (j = i+1; j < N; j++)
    {
        ratio = A[i][j]/A[i][i];

        for (k = i; k < N; k++)
        {
            A[j][k] -= (ratio*A[i][k]);
        }
    }
}
```

Understanding this code:

Outermost loop: `i`

- `N-1` iterations
- Iterations depend upon each other
- Values computed at step `i-1` used in step `i`

Inner loop: `j`

- `N-i` iterations (constant for a given `i`)
- Iterations can be performed in any order

Innermost loop: `k`

- `N-i` iterations (constant for a given `i`)
- Iterations can be performed in any order

Are these loops independent? Which can be parallelized with OpenMP? How would we use what we learned before to rewrite this code with OpenMP?

```
for (i = 0; i < N-1; i++)
{
    #pragma omp parallel for
    for (j = i+1; j < N; j++)
    {
        ratio = A[i][j]/A[i][i];

        for (k = i; k < N; k++)
        {
            A[j][k] -= (ratio*A[i][k]);
        }
    }
}
```

If we want to have nested loops, we need to have a new parallel region with `#pragma omp parallel`, not just `#pragma omp for`. One other thing to notice is that different iterations will have different amounts of work; near the beginning of the loops, there are more operations to do, and near the end there are fewer calculations. Later we will see some options for making sure that our threads stay busy instead of waiting for their colleagues to complete long calculations.

Summary of Example 2:

- We can parallelize a `for()` loop with `#pragma omp parallel for`.
- We can only parallelize loops with a known number of iterations and that are all independent.

Example 3a: Race conditions: Errors can arise if two threads change the same variable.

One very common error in threaded programming is what is called a **race condition**. This happens when one thread modifies some shared data without another thread's knowledge. (One reason why the examples so far have been so simple is to avoid having to discuss race conditions until now.) Consider the following program that creates an array `x[N]` and then sets `x[j] = j` for each of them.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (int argc, char *argv[])
{
    int i;
    int j;
    int N;
    int *x;

    N = 512;

    x = (int *)malloc(N * sizeof(int));

    #pragma omp parallel for
    for(i = 0; i < N; i++)
        {
            j = i;
            x[j] = j;
        }

    for(i = 0; i < N; i++)
    {
        if(x[i] != i)
            printf("Error at %d: x[%d] is %d.\n", i, i, x[i]);
    }

    free(x);
    return 0;
}
```

As we run this program with different values of `N`, we find that as `N` gets larger, we start finding errors in our array. Even though our program assigns `x[j] = j`, when we check at the end, there are times when `x[i] != i`. What happened and how can we fix it?

The solution here is fairly simple, we change our directive to

`#pragma omp parallel for` **`private(j)`**

By stating that `j` is a `private` variable, we are giving each thread its own copy of `j`. This will keep the value of `j` from changing in the midst of assigning a value to an array element. We can tell the compiler whether each variable is `private` or `shared`. If we say `default(none)`, then we must

specify whether each variable is private or shared.

It's important to know that variables that are `private` are always undefined on both entry to and exit from the parallel region. That is, not only does each thread have its own copy of the variable, but this copy does not exist outside of the parallel region. We can address this issue by using the `firstprivate` clause or the `lastprivate` clause instead of the `private` clause. With `firstprivate` clause, the variable brings its existing value into the parallel region; with `lastprivate`, its value will persist beyond the end of the parallel region.

Another way to avoid race conditions is with `#pragma omp critical (name)`. When a thread reaches a critical structured block, it will make sure that no other thread is executing a critical block with the same name before it enters the block.

Suppose that we had two vectors `a[N]` and `b[N]` and were trying to calculate their dot product, such as:

```
for(i=0; i<N; i++)
{
    c += a[i] * b[i];
}
```

we could rewrite this with a critical region to ensure that no two threads are accessing the variable **c** at the same time:

```
#pragma omp parallel for private(x)
for(i=0; i<N; i++)
{
    x = a[i] * b[i];

    #pragma omp critical (add_parts)
    {
        c += a[i] * b[i];
    }
}
```

One important thing to keep in mind that is even if your functions are carefully written to respect shared and private variables, you can't be certain that this is true of every function in every library that you use. In other words, you want to make every effort to ensure that the libraries that you are using are **thread safe**.

Example 3b: Reduction variables

There is an easier way to write the dot-product code. We would tell the compiler that **c** is a reduction variable, under +, with:

```
#pragma omp parallel for reduction(+:c)
    for(i=0; i<N; i++)
    {
        c += a[i] * b[i];
    }
```

This insures that two threads do not try to add their part of the result to **c** simultaneously. Using the reduction variable is a great way to have all the threads work together on this sort of calculation while avoiding a race condition.

Example 3c: Variable scope.

Consider the following snippet of code:

```
void caller(int *a, int n)
{
    int i,j,m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++)
    {
        int k=m;
        for (j=1; j<=5; j++)
        {
            callee(&a[i], &k, j);
        }
    }

void callee(int *x, int *y, int z)
{
    int ii;
    static int cnt;
    cnt++;
    for (ii=1; ii<z; ii++)
    {
        *x = *y + z;
    }
}
```

Which variables are private? Which variables are shared? [1]

variables: `a` `n` `i` `j` `m` `k` `x` `*x` `y` `*y` `z` `ii` `cnt`

Summary of Example 3:

- We need to tell the compiler which variables can be shared among the threads and which should be private to each thread. If we don't do that, then we may get unpredictable results.
- Critical regions will be executed by each thread in a team—but only one thread will execute the code at any given time.
- We can use the following clauses:

  `private` Each thread has its own copy of the variable. The value of the variable is not defined outside of the parallel region.

  `shared` All threads share this variable. We must be careful to avoid race conditions.

  `firstprivate` This private variable takes its value with it into the parallel region.

  `lastprivate` This private variable takes its value with it out of the parallel region.

  `default` By setting `default(none)`, we must specify whether each variable in the parallel region is public or private; the compiler will make no assumptions.

  `reduction` Lets us tell the compiler that this is a reduction variable. Lets all threads contribute to it while avoiding race conditions.

[1] **Answers:** shared: a, n, j, m, *x, cnt; private: i, k, x, y, *y, z, ii.

Example 4a: Other ways of controlling the threads: sections.

Suppose we had a program that called two functions. One function takes four seconds to run, and the other function takes one second to run:

```
void myfunction1()
{
    printf("This is the one second function.\n");
    sleep(1);
    return;
}

void myfunction4()
{
    printf("This is the four second function.\n");
    sleep(4);
    return;
}

int main (int argc, char *argv[])
{

    myfunction4();
    myfunction1();
    myfunction1();
    myfunction1();

    return 0;
}
```

We can control this with `sections`.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        myfunction4();

        #pragma omp section
        myfunction1();

        #pragma omp section
        myfunction1();

        #pragma omp section
        myfunction1();
    }
}
```

These could run in any order.

Example 4b: Barriers and Synchronization

Here are some more directives and clauses that can be useful in managing the flow of a program.

`#pragma omp barrier` This will synchronize all the threads. The threads will wait here until all the threads have reached this point. There is an implied barrier at the end of each parallel region. While there is an implied barrier at the end of `#pragma omp single`, there is *not* one at the end of `#pragma omp master`. Also, be careful to ensure that any barrier will be reached by *every* member of a team of threads to avoid deadlock conditions.

`nowait` This clause removes the implied barrier at the end of a parallel region, as illustrated in the following example.

`#pragma omp flush` All threads will ensure that their cached values of shared valuables match with the values in memory. This directive takes an optional argument of a list of variables to update. If this list is not provided, all of the thread's shared variables will be updated.

```
#pragma omp parallel
{
    #pragma omp for nowait
        for(i=0; i<N; i++)
        {
            a[i] = b[i] + c[i];
        }

    #pragma omp for nowait
        for(i=0; i<N; i++)
        {
            d[i] = e[i] + f[i];

        }
}
```

Example 4c: An example that runs in parallel only if there is enough work.

```
int main (int argc, char *argv[])
{
    int i;

    for(i=0; i<8; i+=2)
    {
        #pragma omp parallel if(i > 3)
        {
            printf("i = %d.\n", i);
            printf("Hello from thread \%d.\n", omp_get_thread_num());
        }
        printf("\n");
    }
    return 0;
}
```

Summary of Example 4:

- We can define tasks to be run in parallel with `#pragma omp section`.
- There are directives and clauses that we can use to tell the threads whether to wait for the whole team to catch up or to make sure that all threads agree on the values of shared variables.

- We can use the `if()` clause to determine whether our region should run in serial or parallel.

Example 5a: Some helpful functions to use at runtime and some helpful environment variables.

OpenMP functions:

`omp_get_thread_num()`	ID Number of current thread. Note that if there is a parallel region within a parallel region that the threads will renumber from 0.
`omp_get_num_threads()`	Returns the number of threads in the current team.
`omp_set_num_threads()`	Sets the number of threads on subsequent teams that don't use the `num_threads` clause.
`omp_get_max_threads()`	Number of threads that could be used in a new team without overriding any defaults
`omp_get_thread_limit()`	Number of threads available to the program (determined by architecture and OS, not necessarily by common sense).
`omp_get_num_procs()`	Number of processors available. Note: On the SGI Altix UV, `dplace` may tell your program that there is only 1 processor available.

OpenMP environment variables:

`OMP_NUM_THREADS`	Sets the number of threads to use in parallel regions. Always set this environment variable in every script that runs OpenMP programs on Nautilus.
`OMP_THREAD_LIMIT`	Sets the *maximum* number of threads that can be in the program.
`OMP_SCHEDULE`	Can be set to `static`, `dynamic`, `guided`, or `auto`. Determines scheduling and load balancing.
`OMP_DYNAMIC`	Determines whether the number of threads to use for parallel regions is set dynamically.

Other environment variables:

`KMP_AFFINITY`	When you compile with the Intel compiler, it automatically includes instructions about how to place threads and processes on CPUs. If you want to override these settings with a tool such as SGI's `dplace` or `omplace`, then you should set the environment variable `KMP_INFINITY=diasable`.

Example 5b: Scheduling

We can use the `schedule()` clause to specify how to assign iterations to threads. This clause takes two arguments: a schedule type (required) and a chunk size (optional).

`static` Breaks the task up into chunks of the specified size (or, if no value is set, as evenly as possible) and distributes them to the threads sequentially. This is good if you want to make sure that each thread has particular iterations (for example, if memory locality is an issue) and not so good if the tasks take widely varying amounts of time to complete.

`dynamic` As each thread finishes its work, it will ask for more work. The amount of work in each chunk is set via the chunk size; if no chunk size is set, then it will get one iteration each time.

guided This is similar to dynamic, but large chunks are handed out at the beginning and smaller chunks at the end.

runtime Defers to the values set in the `OMP_SCHEDULE` environment variable.

Example 5c: Conditional compilation

Remember back in the beginning that our `Hello World` program that used `omp_get_thread_num()` wouldn't compile without the `-fopenmp` compiler flag. One nice thing about OpenMP is that we can use the exact same source code for both the serial and parallel version. While we could force our program to run in parallel by forcing it to use only one thread, an even better solution is to use conditional compilation. In this case, if we don't compile with `-fopenmp`, then the program will compile as a regular serial program; we define the values of the OpenMP functions as appropriate.

```
#include <stdio.h>
#include <stdlib.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

int main (int argc, char *argv[]) {

    printf("Hello world.\n");

    #pragma omp parallel
    {
        printf("Hello from thread %d.\n", omp_get_thread_num());
    }

    printf("I am sequential now.\n");
    return 0;
}
```

We can try compiling it with:

```
gcc -o hello4-serial hello4.c
gcc -fopenmp -o hello4-parallel hello4.c
```

When we run them, we see that they run much as we would expect.

Example 5d: Which version of OpenMP are we using?

OpenMP 2.5 was released in May 2005, so if your compiler supports OpenMP 2.5, then the value of `_OPENMP` will be 200505. OpenMP 3.0 was released in May 2008, so if your compiler supports OpenMP 3.0, then the value of `_OPENMP` will be 200805.

```
int main (int argc, char *argv[])
{
#ifdef _OPENMP
     printf("Compiled with OpenMP, version dated: %d\n", _OPENMP);
#else
     printf("Not compiled with OpenMP.\n");
#endif
```

```
    return 0;
}
```

When I compile this code with gcc 4.2.1 (the standard version that came with my laptop), it tells me that it supports OpenMP 2.5. I've also installed gcc 4.6.0, which supports OpenMP 3.0.

Summary of Example 5:

- We can use environment variables and functions to determine how our program runs.
- It is essential to set environment variables properly for good performance on Nautilus.
- We have some control over how the tasks are scheduled to the threads.
- We can use the same source for both serial and parallel compilation by checking to see if the `_OPENMP` macro is defined.
- The value of `_OPENMP` tells us which version of OpenMP is supported by the compiler.

Running your code

Please see the other handout for full details on how to run your OpenMP code on Nautilus.

Moving from the desktop to HPC When we submit jobs on Nautilus, we submit them to the queue via the batch system with a script that might look somewhat like this:

```
#PBS -N myOpenMPJob
#PBS -q computation
#PBS -j oe
#PBS -l ncpus=16
#PBS -l walltime=01:00:00

export OMP_NUM_THREADS=16

./myprogram
```

If we saved this in a file called **mysubmission.pbs**, we would submit it to the queue with the command `qsub mysubmission.pbs`

We can watch its progress in the queue with `showq` and `qstat`. We can also keep on eye on how **myprogram** is running with standard tools like `top`, `ps`, and `pidstat`.

Monitoring your jobs: As we run our programs, we are concerned about how fast it runs and how it consumes resources on the system. One simple way to find out how long the program takes to run is `/usr/bin/time`. We can get a simple measure of how long our program takes with `time /.myprogram`. There are also more sophisticated ways to find out how long it spends in a given function.

One way to find out which processors your program's threads are running on is with `pidstat`. While running on Nautilus, if you give the command `pidstat -t -p ALL | grep myprogram > mypidstatfile`, the second to last column will tell you which physical processor each thread is running on. For example, a program with 16 threads, each running on a different CPU, would give an output like:

```
10:19:29 AM    652697        -     0.00    0.00    0.00   616  myprogram
10:19:29 AM        -     652697    0.00    0.00    0.00   616  |__myprogram
10:19:29 AM        -     652699    0.00    0.00    0.00   617  |__myprogram
```

```
10:19:29 AM        -     652700    0.00    0.00    0.00   618  |__myprogram
10:19:29 AM        -     652701    0.00    0.00    0.00   619  |__myprogram
10:19:29 AM        -     652702    0.00    0.00    0.00   620  |__myprogram
10:19:29 AM        -     652703    0.00    0.00    0.00   621  |__myprogram
10:19:29 AM        -     652704    0.00    0.00    0.00   622  |__myprogram
10:19:29 AM        -     652705    0.00    0.00    0.00   623  |__myprogram
10:19:29 AM        -     652706    0.00    0.00    0.00   624  |__myprogram
10:19:29 AM        -     652707    0.00    0.00    0.00   625  |__myprogram
10:19:29 AM        -     652708    0.00    0.00    0.00   626  |__myprogram
10:19:29 AM        -     652709    0.00    0.00    0.00   627  |__myprogram
10:19:29 AM        -     652710    0.00    0.00    0.00   628  |__myprogram
10:19:29 AM        -     652711    0.00    0.00    0.00   629  |__myprogram
10:19:29 AM        -     652712    0.00    0.00    0.00   630  |__myprogram
10:19:29 AM        -     652713    0.00    0.00    0.00   631  |__myprogram
```

This program is running on physical processors 616–631.

What do we need to do on Nautilus in order to ensure that our program's threads are spread out over the processors (like above)? Sometimes the automatic scheduling will have all of the threads fighting to share a single processor while the other processors sit around idle, so we would want to override the default. Also, sometimes a process will migrate from one processor to another; if we want to force a process to stay in a certain place—for example, because of memory locality—we want to be able to specify that. On Nautilus there are two tools that we can use to place our programs on the processors: `dplace` and `omplace`. While `dplace` is an all-purpose tool for placing programs, `omplace` is especially for MPI processes and OpenMP threads.

If we wanted our OpenMP program to run with 16 threads on logical CPUs 0–15, instead of running our program with `./myprogram`, we would use the following syntax:

```
omplace -nt 16 -c0-15 ./myprogram
```

To combine some of these, we could use the following submission script:

```
#PBS -N myOpenMPJob2
#PBS -q computation
#PBS -j oe
#PBS -l ncpus=16
#PBS -l walltime=01:00:00

export OMP_NUM_THREADS=16

omplace -nt 16 -c0-15 ./myprogram &
sleep 10
pidstat -t -p ALL | grep myprogram > mypidoutfile
```

On Nautilus we've found that `omplace` will work for programs compiled with gcc or with icc; `dplace` will only work with programs compiled with `icc` if you set the environment variable `KMP_AFFINITY=disabled`. Nautilus also has the `dlook` function that will create a file describing how your program is using memory. You can learn more about these tools by reading their `man` pages on Nautilus.

Resources

Book: *Using OpenMP: Portable Shared Memory Parallel Programming*, by Barbara Chapman, Gabriele Jost, and Ruud van der Pas

Cheat Sheet: `http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf`

SGI Tools: See the `man` pages for `dplace`, `omplace`, and `dlook`.

Acknowledgments

Thanks to the organizers for hosting this conference and for selecting our tutorial. This tutorial was inspired by Rebecca Hartman-Baker's OpenMP talk for the NCCS/NICS Crash Course in Supercomputing. We also thank our colleagues Scott Simmerman and Pragnesh Patel for their assistance. Most of all, thank *you* for coming to this tutorial!

If you would like to use these course materials at your organization, please get in touch.

Exercises

You can try these exercises on any laptop or desktop computer with a compiler that supports OpenMP and then try your code on Nautilus. If you don't have access to Nautilus through a Campus Champion or XSEDE staff account, you can request a start-up or educational allocation through the XSEDE portal.

1. Consider the code from **example 2b** for converting a matrix to upper triangular form.
   (a) This code does not take the step of getting it into row echelon form (ones on the main diagonal). Can you modify the code so that the matrix is in row echelon form?
   (b) What happens to the performance of this code under different settings of the `schedule` clause? Does the best choice depend on the size of the matrix? The number of threads?
   (c) Both of the two inner loops can be parallelized in the code for converting a matrix to upper triangular form. What happens if you parallelize the innermost loop (the `k` loop) instead of the `j` loop? How does the performance change?
   (d) Can you create nested parallel regions?
2. Let N be a natural number. Write an OpenMP program that calculates the number of primes p less than N. Hint: p is prime if `p % i != 0` for $1 < i < p$.
3. Use OpenMP to write the following operations on an $N \times N$ square matrix A using the naive element-by-element operations that you use for doing these calculations by hand:
   (a) Let x be a vector of length N. Calculate Ax and xA. Time your program for various values of N and for various numbers of threads. For which sizes of matrices is it worth parallelizing the code with OpenMP and for which sizes does the overhead of creating the threads cancel out any advantages?
   (b) Calculate A^tA.
   (c) Calculate the determinant of A.
   (d) Calculate A^{-1}.

   Can you set up your program so that you get efficient memory access and efficient scheduling? What happens if you change the arguments to the `schedule` clause for various sized matrices?

4. Consider the following code:

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

int main (int argc, char *argv[])
{

    int i;
    int a[512];

    #pragma omp parallel for shared(a) schedule(static,1)
    for(i=0; i<512; i++)
    {
        a[i] = i;
    }

    return 0;
}
```

   (a) Does this run faster with OpenMP or without it? What happens if you change the value of `N`?

   (b) Can we get this code to run faster by changing `int a[512];` to `int a[512][8];` and `a[i] = i;` to `a[i][0] = i;`? What happens as we change the sizes of the array and the number of threads?