1. OpenMP uses the fork-join model, which is essentially based on the same system fork command that was used in the previous projects. Just like the fork command, we can tell OpenMP to spawn a child process that will execute the exact same code that the parent process will execute. In OpenMP, the forking is handled automatically, in a more transparent and convenient manner for the programmer. In addition, with a single pragma, multiple child processes (actually threads) can be created. A thread is a runtime entity that is able to independently execute a stream of instructions. In OpenMP, a Parallel Region is a block of code executed by all threads simultaneously. Each thread is not thought of as being a parent or child, but rather the group is referred to as a “team”. The region of the program which should be run in parallel by the team is simply bounded as follows:

#pragma omp parallel

{

<portion of code which team executes in parallel>

}

If the number of desired threads is not explicitly specified by the programmer, then OpenMP will set the number of threads by default to match the number of logical hardware cores in the system**. Run the following program sequentially, that is, compile it without the -fopenmp option, then run it in parallel with the -fopenmp option specified during compilation. Explain what happens in each case.** Recall that the Intel MTL is a 32-core machine, where each physical core is Hyper-Threaded. [5 pts]

#include <stdio.h>

int main()

{

#pragma omp parallel

{

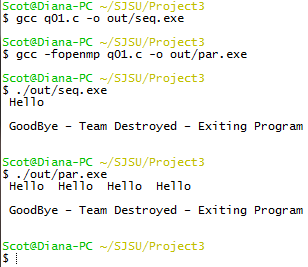
printf(“ Hello ”);

}

printf(“\n\n GoodBye – Team Destroyed – Exiting Program \n\n”);

}

In the sequential compilation, a single master thread is used since the openmp compiler was not used. The #pragma omp parallel statement is interpreted as a comment and is ignored. The program is executed utilizing a sequential pipeline with no parallelization. Setting the –fopenmp flag signals the gcc compiler to utilize the openmp compiler. During compilation, the #pragma omp parallel statement is read and the default number of threads will be set to match the number of cores in the system (4 in this case). Each thread executes the code within the parallelized block and then completed. The master thread is the only one which continued through to the end of the program.



1. Now, specify the number of threads in the “team” by using *omp\_set\_num\_threads(nthreads)* Although some compilers may not require it, note that you may also need to add the following include clause as well *#include <omp.h>*

The program from (1) should now appear as follows for nthreads = 2:

#include <stdio.h>

#include <omp.h>

int main()

{

omp\_set\_num\_threads(2);

#pragma omp parallel

{

printf(“ Hello ”);

}

printf(“\n\n GoodBye – Team Destroyed – Exiting Program \n\n”);

}

Run the program above with the nthread parameter set to 2, 4 and 8. Verify that the output is as expected for the different number of threads created in the team. **Is it possible to specify an odd number of threads? Is it possible to specify a number of threads greater than the number of logical cores (or physical cores if no HyperThreading is present) available in the hardware?** [5 pts]

In my experimentation, I have determined yes it is possible to request an odd number of threads as well as a greater number of threads than the number of physical cores. Further research shows however than creating more threads than there are cores will lead to an increased amount of overhead. At this point, a Queue must be established and processes will be assigned to one of the cores. While in my case, 4 threads will continue to work in parallel, any more will also once again introduce concurrency.

|  |
| --- |
| 2 Threads    4Threads    8Threads    3Threads |

1. The omp function call *omp\_get\_thread\_num()* retrieves the ID of the thread. Note that the master thread always has thread ID 0. That is, the numbering of the threads in the team starts from zero; not one. The other threads are given IDs of consecutively higher integers (1, 2, 3, etc.) up to nthreads-1.

By using *omp\_get\_thread\_num()* we can have each thread print out it’s ID in addition to “ Hello “. Change the single printf line with “Hello” in it from (2) above to get the following:

#include <stdio.h>

#include <omp.h>

int main()

{

omp\_set\_num\_threads(2);

#pragma omp parallel

{

printf(“\n Hello from thread = %d ”, omp\_get\_thread\_num() );

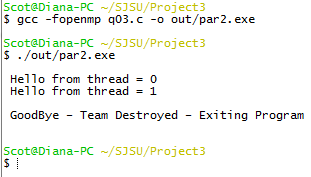
}

printf(“\n\n GoodBye – Team Destroyed – Exiting Program \n\n”);

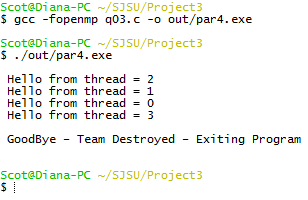
}

**Run the program for various values of nthreads. Perform several runs for each value of nthread and comment on the order in which the threads are run.** Note that you can also specify an nthreads value of 1 to default back to the sequential, single thread case. [5 pts]

From the output, we can see that order of execution when spawning multiple threads is varied through each execution of the application.



2 Threads



4 Threads



8 Threads

1. More often than not, a programmer is not just interested in having several threads all executing an identical section of code; (s)he would want to distribute the work among the threads so that it can be completed in less time and in parallel. In the previous labs that used fork, the programmer had to manually separate the work for parent vs. child by performing a test on the return value of the fork system call. For example, one branch of the IF represented parent executed code; the other branch represented the child executed code. OpenMP can do this partitioning of work among the available threads automatically.

A work-sharing construct of OpenMP divides the execution of an enclosed code region (typically a loop) among the members of the team; in other words: OpenMP splits the work between the threads. The OpenMP construct that does this for loop constructs is the keyword “for” appended onto the previously seen “parallel” pragma: *#pragma omp parallel for*

Inserting this statement before a ‘C’ for loop will distribute the iterations of the ‘for’ loop among the threads. Enter this program to trace the 16 iterations of the loop and observe how the iterations are distributed among the threads as a function of the number of threads. **Run the program for nthreads=2, 4, 8, 16 and 32. Explain what happens. What happens if the number of iterations in the for loop is not evenly divisible by the number of threads?** For example, explain what happens in the program below if nthreads were set to 5 or 7, instead of to 2 as currently shown. [10 pts]

#include <stdio.h>

#include <omp.h>

int main()

{

int i;

omp\_set\_num\_threads(2);

#pragma omp parallel for

for (i=0; i<16; i++)

{

printf(“Hello from thread number: %d Iteration: %d \n”,

omp\_get\_thread\_num(), i);

}

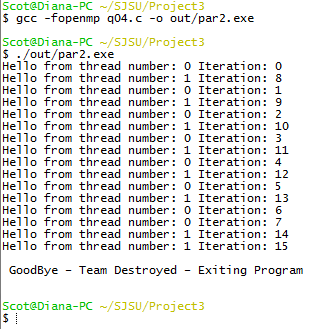
printf(“\n GoodBye – Team Destroyed – Exiting Program \n\n”);

}

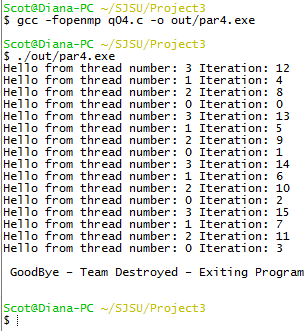
Notice that the iterations are split into contiguous chunks (rather than round robin fashion), and each thread gets one chunk of iterations. By default, OpenMP splits the iterations of a loop into chunks of equal (or roughly equal) size, assigns each chunk to a thread, and lets each thread loop through its subset of the iterations. So, for example, given 4 threads and 12 iterations, each thread gets three iterations:

T0: Iterations 0,1,2 T1: Iterations 3,4,5 T2: Iterations 6,7,8 T3: Iterations 9,10,11

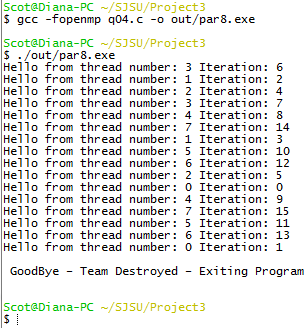
In all instances, OpenMP attemps to balance the number of threads executed between the numbers of total processes in relation to the number of iterations. In the case in which an odd number of threads exist, some threads will have more statements of execution than others. If the number of threads exceeds the total number of iterations, only the required number of threads will actually be deployed (i.e., if we ask for 32 but only need 16, then we only get 16).



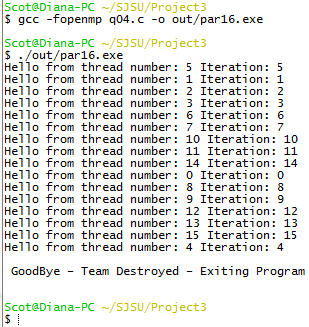
2 Threads



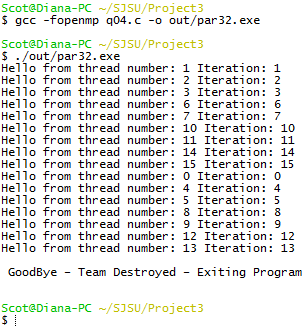
4 Threads



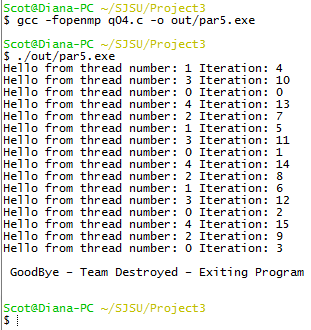
8 Threads



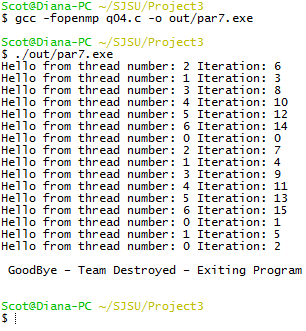
16 Threads



32 Threads



5 Threads



7 Threads

1. Although OpenMP makes most variables shared by default and visible to all threads, this is not always what is needed to provide proper and correct behavior of the parallel version of the code. The code segment below produces the sum of integers from 0 up to (COUNT-1). When run in sequential mode, it produces the correct answer. However, when parallelized incorrectly as shown below, non-deterministic errors can occur. The errors are non-deterministic because they do not always happen – they only occur under certain conditions – the erroneous answers are different from run to run – and there is no apparent error or warning message, so it appears that everything is fine.

Run the code sequentially first for the three values of COUNT to produce the correct answer. Then run it for the following values of NTHREADS and COUNT. Suggested values for NTHREADS include 2, 4, 8, 32, and 64. Suggested values for COUNT include 10, 100, and 1000. **Create a table similar to the one below, making at least ten runs at each of the 15 data points. For each of the 15 data points, estimate the probability of an incorrect answer**. If you do exactly ten runs at each data point, then this probability will be easy to report. (You can optionally do a larger number runs at each data point to get a better statistical average) **Why are some answers incorrect? What is causing the non-deterministic errors? What is the probability of obtaining an incorrect answer as a function of NTHREADS and COUNT? Is there a trend in the probability and what intuitive explanation might there be for it?** Optionally, repeat the experiment on a single/dual/quad core machine, if you have access to one. [25 pts]

Percent of Incorrect Answers Generated by Flawed Parallel OpenMP Program

#include <stdio.h>

#include <omp.h>

int main()

{

int i;

int sum=0;

omp\_set\_num\_threads(NTHREADS);

#pragma omp parallel for

for (i=0; i<COUNT; i++)

{

sum = sum + i;

printf(“Thread number: %d Iteration: %d Local Sum: %d \n“,

omp\_get\_thread\_num(), i, sum);

}

printf(“\n All Threads Done – Final Global Sum: %d \n\n”, sum);

}

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Count** | **NTHREADS=1** | **NTHREADS=2** | **NTHREADS=4** | **NTHREADS=8** | **NTHREADS=32** | **NTHREADS=64** |
| **10** | 45 | 100% | 100% | 100% | 100% | 100% |
| **100** | 4950 | 100% | 100% | 100% | 100% | **90%** |
| **1000** | 499500 | 100% | 100% | 100% | **90%** | **90%** |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **t** | **i** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **9** | **9** | **10** |
| **2** | **10** | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 |
| **4** | **10** | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 |
| **8** | **10** | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 |
| **32** | **10** | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 |
| **64** | **10** | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 |
| **2** | **100** | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 |
| **4** | **100** | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 |
| **8** | **100** | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 |
| **32** | **100** | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 |
| **64** | **100** | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | **4890** | 4950 | 4950 |
| **2** | **1000** | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 |
| **4** | **1000** | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 |
| **8** | **1000** | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 |
| **32** | **1000** | 499500 | **492094** | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 |
| **64** | **1000** | 499500 | 499500 | **498860** | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 |

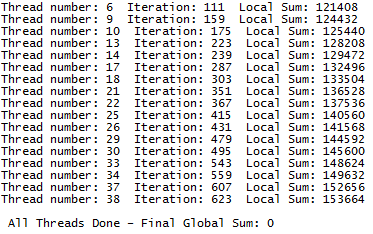
When dealing with a higher thread count, as iterations grow in size we begin to observe errors being introduced into the computations. The likely reason for an incorrect answer can be attributed to race conditions caused by using shared variables. Race conditions are non-deterministic as we can observe here. They do not always occur and are unpredictable. Given our data set, there is a 2% probability that we will receive an incorrect answer. If the current trend continues for larger data sets, we can assume an increasing trend of incorrect results in our data set. I suspect a graph with this formation to emerge.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

1. OpenMP knows to make certain variables private instead of shared, that is, visible only to individual threads. For example, the loop iteration variable, i, in the above code of (5) is automatically made private by OpenMP. OpenMP also allows a programmer to explicitly declare some variables as being private. What happens if the variable sum is made private with *#pragma omp parallel for private(sum)*

**Make this small change to (5) and document and explain what happens. Does this solve the non-deterministic incorrect answers produced from (5) above? Explain why or why not.** [10 pts]

Making sum private only partially solves the problem. We have solved the non-deterministic problem associated with the race condition but now our threads no longer communicate with one another. The result is that the original sum variable is no incremented with the updated value.



t=64 - i=1000

1. The reduction(+:sum) clause appended to the end of the #pragma omp parallel directive will solve the problem associated with (5) above. Run the program below with a few different values of NTHREADS and COUNT. Explain what reduction(+:sum) does. [10 pts]

#include <stdio.h>

#include <omp.h>

int main()

{

int i;

int sum=0;

omp\_set\_num\_threads(NTHREADS);

#pragma omp parallel for reduction(+:sum)

for (i=0; i<COUNT; i++)

{

sum = sum + i;

printf(“Thread number: %d Iteration: %d Local Sum: %d \n“,

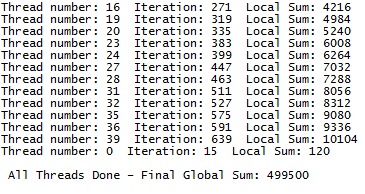
omp\_get\_thread\_num(), i, sum);

}

printf(“\n All Threads Done – Final Global Sum: %d \n\n”, sum);

}

At the beginning of the parallel block, reduction creates a private variable for the incoming thread (essentially a local copy). At the end of the parallel block, the private variable is merged into the ‘shared’ variable, allowing for accumulation.



t=64 - i=1000

1. The program listed below computes the value of PI using iteration. Run the program sequentially first, taking a time measurement.

#include <stdio.h>

#include <omp.h>

#include <time.h>

long long num\_steps = 1000000000;

double step;

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

{

double x, pi, sum=0.0;

int i;

step = 1./(double)num\_steps;

for (i=0; i<num\_steps; i++)

{

x = (i + .5)\*step;

sum = sum + 4.0/(1.+ x\*x);

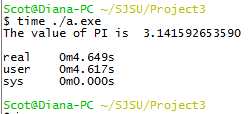
}

pi = sum\*step;

printf("The value of PI is %15.12f \n",pi);

}

You can time the sequential program by typing *time ./a.out* See page 138 of the Chapman book on Using OpenMP for more information on using the time system function. Note that the program does not need any user input and takes about 20 seconds of wall clock time on MTL to complete. **Record the sequential time.** [5 pts]



1. Parallelize the PI program above, by including the following two OpenMP parallelization clauses immediately before the ‘for loop’.

*omp\_set\_num\_threads(128);*

*#pragma omp parallel for private(x) reduction(+:sum)*

In this particular case, adding just two more lines to the sequential program will convert it to a parallel one. Also note that *omp\_set\_num\_threads(NTHREADS)* is not really necessary. OpenMP will simply set the number of threads to match the number of logical cores in the system by default. So only one additional line consisting of an OpenMP *#pragma omp parallel….* was really required to convert from sequential to parallel. We include the other one as well because we are interested in explicitly setting NTHREADS to different values as part of our experimentation. **Time the parallel program below using various values of NTHREADS.** **Record and report your findings of Time vs. NTHREADS. Include test cases involving NTHREADS > 32, the number of physical cores, and NHREADS > 64, the number of logical cores in MTL. Explain any observations.** Optionally, repeat the experiment on single/dual/quad core machine(s), if you have access to these alternate hardware platforms. [25 pts]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Threads | Real Time | User Time | Sys Time | Physical Cores | Logical Cores |
| 64 | 0m2.465s | 0m9.578s | 0m0.000s | 4 | 0 |
| 128 | 0m2.480s | 0m9.609s | 0m0.015s | 4 | 0 |
| 256 | 0m2.465s | 0m9.625s | 0m0.030s | 4 | 0 |
| 512 | 0m2.496s | 0m9.609s | 0m0.046s | 4 | 0 |
| 1024 | 0m2.543s | 0m9.594s | 0m0.061s | 4 | 0 |

As the number of threads increases, the system time increases by approximately 0m0.015s. The number of threads has greatly outnumbered the number of cores and we are observing the formation of a queue of processes waiting for each core to become available. Naturally, I suspect that by increasing the number of available cores we would greatly diminish this effect.