Programming Assignment 2

Using OpenMP to parallelize a serial program and observe the increase in the performance.

Introduction

The purpose of this programming assignment is to gain experience in writing OpenMP programs. I start with a working serial program (quake.c) that models an earthquake and add OpenMP directives to create a parallel program.

Directives and Clauses

A typical example of a directive : **#pragma omp parallel for collapse(2) private(a) firstprivate(b)**

Following directives were used in the program:

1) Parallel – Used to create different threads to run on the enclosed portion of the program individually on each of the threads. This directive can be followed by various clauses mentioned below to avoid any possibility of race conditions. This Directive was used practically before every loop that I wanted to parallelize.
2) For - Causes the work done in a for loop inside a parallel region to be divided among threads. Care should be taken that the various iterations of the for loop don't depend on each other. In case it does we might need to add various reductions to it. There are many for loops in the original quake.c program so only the for loops with significant number of iterations were selected to parallelize.

The following clauses were used with the above mentioned directives:

1) Private – This is used to create an uninitialized copy of the specified variable specific or local to each of the threads. So in case the various threads write to a common variable we can't have that variable as shared because this would lead to race condition due to all threads working in parallel. An important observation was that the private could create uninitialized copies of both variable and various statically allocated data structures.
2) First private – This helps in creating an exact copy of the specified variable local to each of the reads. I used mainly in instances when I was required to read the various values multiple times in a loop. Like when mentioning the count of number of times the loop should iterate.
3) Collapse- This clause is used to collapse the two nested loops into one and then share the workload over various threads in the system.
4) Shared – Used to specify the various variables shared across the threads.

Performance Results

Time taken by the sequential version of the quake.c : 62.304s

Timing for the parallized version of the quake.c for various number of threads:

Number of Threads	Execution Time(s)
1	63.8
2	34.19
4	20.783383
8	10.950003
16	6.32

Speed up of 16 threads program vs. Sequential program = 62.304/6.32 = 9.8582

The runtime of the program reduced as we increased the number of the threads available to the program from 63.8 seconds to 6.32 seconds at 1 and 16 threads respectively.

Also, we can see the runtime of the sequential version of the program is a bit lower than the runtime of the parallel program for 1 thread.

Expectations and Observations

Ideally as we double the number of threads the runtime should get halved as the same work would be divided amongst double the number of threads. But practically we can't reach this theoretical potential because of the following reasons:

1) There is an overhead of actually sharing the workload amongst the various threads like copying the data to the individual core and accessing the shared variables. Since the time required to access a local variable is less than time taken to access a shared variable.
2) There can be dependency amongst the various parts of the program and hence it can be practically impossible to execute two parts in parallel thereby inducing the seriality in the program.

I specifically tried to address the first issue mentioned above by parallelizing only major loops so that the communication overhead is less than the execution benefit.

Our result was approximately a speed up of 10x(approx.) considering the limitations mentioned above the results were considerably close to what was expected. But 16x still remains to be the theoretical max.

Also, as remarked before the runtime of the sequential version of the program is a bit lower than the runtime of the parallel program for 1 thread this was expected and can be explained from the fact that using a parallel program to run on a single thread adds the unnecessary overhead of various parallel constructs which are a waste when being used on a single thread.