OpenMP: Earthquake Simulation

Chirag Majithia

***Abstract*—The aim of the project is to get familiarized to OpenMP API used for multi-platform shared-memory parallel programming in C/C++/Fortran. Here we parallelize benchmarked quake simulation code by Loukas Kallivokas and David O'Hallaron.**

I. Earthquake Simulation

Given is the code used as a benchmark for simulating earthquake by Loukas Kallivokas and David O'Hallaron and later by Wes Jones, SGI for Standard Performance Evaluation Corporation. The task is to use OpenMP API to parallelize the code and evaluate it's performance with the original code. In order to understand which parts of the sequential code need speed-up by parallelizing, we use GNU profiling tool (GPROF). Then we use various OpenMP directives to debug and parallelize the code which are discussed in the section III and the evaluation of the performance is discussed in section II.

II. Profiling the sequential program

Profiling is an important aspect of software programming. Through profiling one can determine the parts in program code that are time consuming and need to be re-written. This helps make your program execution faster which is always desired.

We run GPROF on the sequential code provided to us and get following result (refer table I):

TABLE I
GPROF OUTPUT: FLAT PROFILE:

percentage time	cumulative seconds	self seconds	calls	name
64.48	42.80	42.80	3855	smvp
31.74	63.86	21.07		main
1.14	64.62	0.76	348904485	phi2
1.05	65.31	0.70	348904485	phi0
0.78	65.83	0.52	348904485	phi1
0.35	66.06	0.23	151173	elemt_mat

As indicated by the profiler, I, it is clear that the functions smvp and main took majority of computation time. Thus, on analyzing the section of code, various for loops in these two function were found to be easily parallelize.

III. OpenMP Directives

Following OpenMP directives were used:

A. *parallel*

When a thread reaches a *parallel* directive, it creates a team of threads (Fork) and becomes the master of the team. The master is a member of that team and has thread number 0 within that team. Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code. There is an implied barrier at the end of a parallel region (Join). Only the master thread continues execution past this point. If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

B. *for*

The *for* directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

C. *parallel for*

OpenMP provides *parallel for* directive as a convenience substitute to an individual *parallel* directive being immediately followed by *for* directive.

IV. OpenMP Clause

A. *Data Scope Attribute Clauses*

1) Shared: The *shared* clause declares variables in its list to be shared among all threads in the team. A shared variable exists in only one memory location and all threads can read or write to that address.

2) Private: The variables declared *private* become a new object of the same type for each thread in the team (uninitialized). All references to the original object are replaced with references to the new object. The scope of these new variable is limited to the parallel region are lost as soon as the team joins back to the master thread.

3) Default: The *default* clause allows the user to specify a default scope for all variables defined before the parallel region and extends it to the parallel region. That is, if the default scope of a variable is *shared*, all the pre-defined variable upto the start of parallel region, remain shared in the parallel region. (There is no support of *default private* and *default first private* in C/C++). The clause used during the project is default ***None***. This directive requires that the programmer explicitly scope all variables.

A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations on a single variable at the same time. This race conditions are caused by shared variables that really should have been declared private. By looking at the variables inside the parallel regions and making sure that the variables are declared private one by one, helps resolve the bug. It was used to debug the parallel construct at line 287 in the sequential code, which created erroneous results.

By default, all the variables are *shared*.

4) First Private: The *firstprivate* clause combines the behavior of the *private* clause with automatic initialization of the variables in its list, with the value of their original objects prior to entry into the parallel or work-sharing construct.

firstprivate was used at various places in code - especially places where there was a pre-initialized variable modified in the parallel construct with some computation using its previous value. However, there were instances in the code, where using *firstprivate* instead of *shared* speed up the computation time, even though the variable in use was just used to assign values to other variables.

B. Section Clauses

1) NoWait: A parallel construct starts a parallel block. This forks a team of thread with in the constructs and all of them join at the end of the construct. However, a parallel construct may have various sections/for loops. Thus at beginning of each loop or section, the master thread delegates portions of the loop for different threads in the current team. All the threads wait at the end of loop to begin the next section (implicit barrier). However, if the subsequent for loops within the construct are independent of the variables used in the previous, the programmer can direct the threads to not wait for other threads at the end of the loop and move forward with the next execution. This override of implicit barrier is done using **NoWait** clause. It was used in mem_init and initialization phase of the main loop.

2) Collapse: collapse specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space. Since all the arrays in the sequential code are either 2D or 3D, there were couple of nested loops which could be collapsed during their initialization.

V. Evaluation

The implementation of the final parallelized code is then evaluated by setting the number of omp_threads to 1, 2, 4, 8, and 16. As expected, the computation time of the program drops as a function of inverse of number of processes (TableII). However, it appears that the computation time saturates after 16 processes as the parallelizing overhead and gain due to parallelized code will eventually balance out (Fig. V).

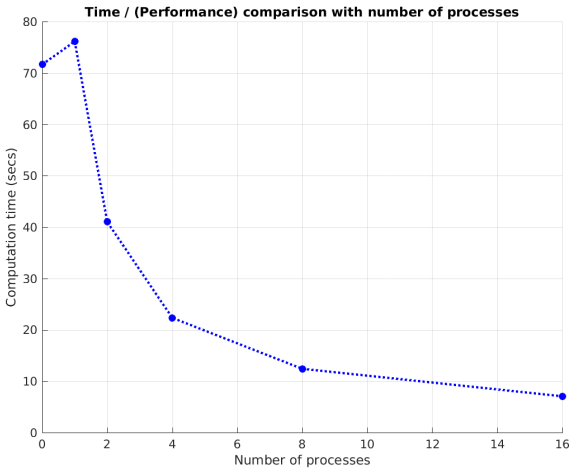

Fig. 1. "Performance vs No. of Processes"

TABLE II
No. of Processes vs Time (in secs)

Processes	Time (in secs)
Sequential (0)	71.63
1	76.13
2	41
4	22.33
8	12.41
16	7.06

Also, it is observed that the sequential code takes less time (around 5s) compared to the parallel code with just 1 thread. This is justified, as the parallel code has overhead of parallelizing the code, yet has computing power of 1 thread, forcing sequential execution.

VI. Conclusion

Various OpenMP directives and clauses were used during the parallelization of the code. An insight to debug the parallelized openMP code was developed using default(none) clause. Comparing to MPI, openMP implementation is easy to use as the API hides major intricacies of breaking sequential code into parallel. The code in the "simulation" block inside the main, can be further parallelized to achieve better performance. The project code/ report and generated outputs of the simulations are available at QuakeOpenMP repository.

References

[1] Blaise Barney, Lawrence Livermore National Laboratory *webpage*.
[2] Tim Mattson, *Introduction to OpenMP - (Intel)*
[3] The Computational and Systems Biology (CSB) at MIT *JOpenMP* Advanced Issues*