Assignment2 Report (CSE436, Summer 2016)
Kazumi Malhan, 06/08/2016

1. Backgrounds and Motivation

Programmers are always looking into new ways to improve the performance of algorithms. As hardware reaching its limit to improve serial code execution time, parallel programming is increasingly used. However, executing code in parallel causes overhead due to idling, data communication, and more. In this assignment 2, sum and matrix multiplication algorithms are implemented in both serial and parallel. The performance (execution time), speedup, efficiency, overhead, and cost are compared for both algorithms to understand the benefit and side effect of parallel execution, and determine the number of threads and scheduling policy to achieve best performance.

2. Function Implementation Description

sum.c parallel function

Before entering into parallel region, total amount of number are divided by number of tasks to determine left over. Left over identified which thread needs to perform extra work to sum all numbers. Inside parallel region, each thread obtains its thread id (tid), and determines the start and end of loop by dividing total number with number of threads (tasks) and multiplies by tid. In the first for loop, all numbers are added to temporary local variable to increase memory access. Next, only threads whose id is less than left over add the left over numbers. Finally, resulting partial result is copied to shared array to carry result out of parallel region. At the end, all elements of array are added to the result.

sum.c parallel for function

This function is mostly same as serial implementation of code except adding parallel for directives to enable penalization. After declaring local variables, parallel region is created with number of tasks. Just before the for loop, for directive is added with "schedule (runtime)". This enables various scheduling policy implementation of changing OMP_SCHEDULE without re-compilation. Partially calculated results by each threads are added together by "reduction (+:result)". Also, "nowait" is added to for directive to remove barrier as there is one at end of parallel region.

mm.c parallel row function

First, the function sets number of threads to launch and enters the parallel region. Inside, row size of matrix A (N) is divided by number of tasks and multiplied by tid to determine the start and end of first loop for each thread. This enables each thread to work on different portion of row during the matrix multiplication to perform row-wise decomposition. Other than replacing start and end of first for loop, rest of code is same as serial function.

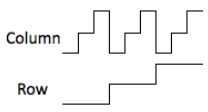
mm.c parallel col function

First, the function sets number of threads to launch and enters the parallel region. Compare to previous function, this function performs column-wise decomposition. Therefore, column size of matrix B (M) is divided by number of tasks and multiplied by tid to determine the start and end of second for loop. As a result, each thread get different portion of columns to work on. Other than start and end of second loop, rest of code is same as serial function.

mm.c parallel rowcol function

First, the function divides number of tasks by 2 to decide how row and column are decomposed. There is special case when number of task is 1, as it will cause divide by zero error. Inside parallel region, start and end of loops are determined using code segment below. For example, during istart calculation, tid needs to be divided by number of tasks in column since same loop range for row should be repeated for all variation of column. Also, modulo operation is appended at the end of each equation to ensure that result is within size of matrix. For end calculation, when result is equal to zero, it really means size of matrix (N or M). Therefore, if statement is added to take care of this operation. For column (j), modulo is used when dividing tid by task_r so each thread gets different variation of row-col decomposition. The example is shown in right. Next, matrix multiplication is performed with three for loops whose range of first and second loops are defined as istart to iend, jstart to jend, respectively.

```
istart = ((tid/task_c) * (N/task_r)%N);
iend = ((tid/task_c + 1) * (N/task_r)%N);
if (iend == 0) {iend = N;}
jstart = ((tid%task_r) * (M/task_c)%M);
jend = ((tid%task_r + 1) * (M/task_c)%M);
if (jend == 0) {jend = M;}
```



mm.c parallel for row function

The body of this function is essentially same as serial implementation. The omp_set_num_threads(num_tasks) defines the number of threads launched when entering parallel region. After the local variable declaration, parallel region is created. As this function performs row-wise decomposition, for directive is placed just before first for loop in

order to share the work. Scheduling is performed with static policy. It is notable that "no wait" is placed in for directive as there is also barrier at end of parallel region.

mm.c parallel for col function

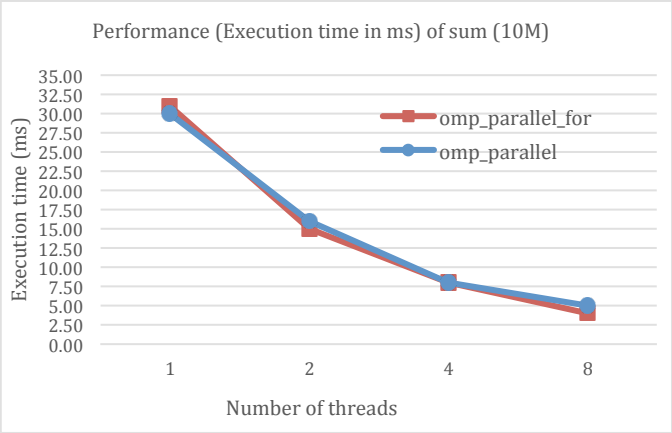
This function is similar to previous function except the method of decomposition. As this function performs column-wise decomposition, for directive is placed just before second for loop to work share columns of second input matrix.

mm.c parallel for rowcol function

This function is also similar to previous function except it needs to decompose the operation in both row-wise and column-wise. Since it is against the OpenMP rules to place nested for directives inside single parallel region, collapse (2) directive is used with for directive to collapse first two loops into single loop, and perform work sharing on combined loop.

3. Performance Report

All of performance data collection is done on **lennon.secs.oakland.edu** Linux server via VPN connection.



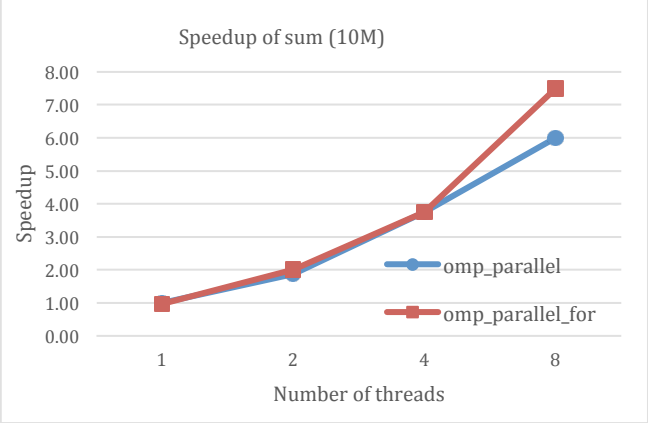


Figure 1 (above left): Performance of sum

Figure 2 (Left): Performance of mm

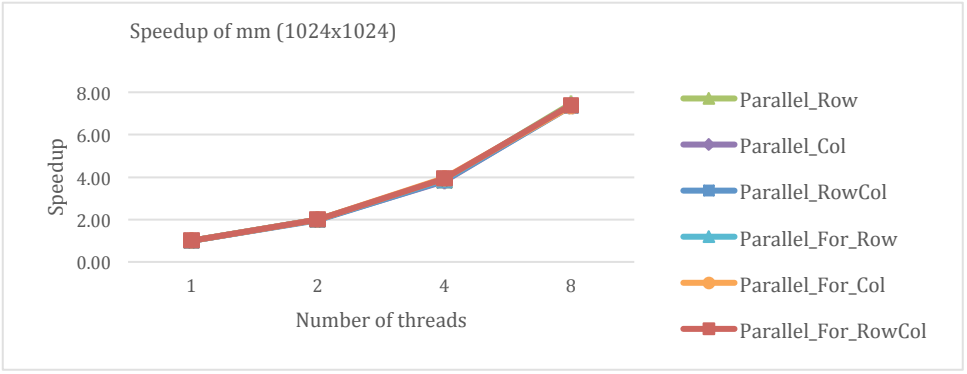
Figure 3 (above right): Speedup of sum

Figure 4 (below): Speedup if mm

Figure 5 (next page): Performance of sum with different scheduling policy

Performance of sum.c

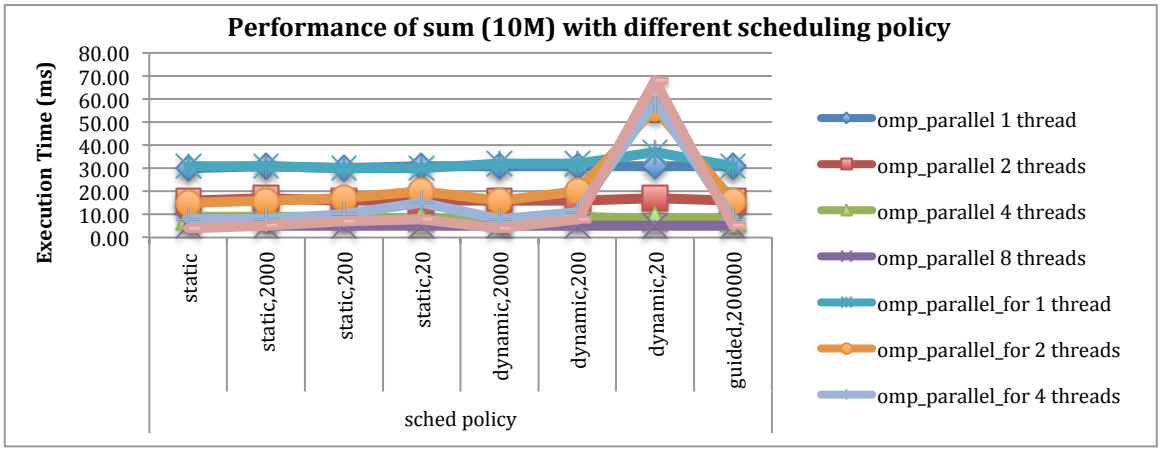
When running all version of code with only one thread, parallel for function has worst performance as there is overhead introduced with parallel and work sharing directives, which doesn't really means for single thread execution. As number of thread increases, the performance of both parallel and parallel for function increase almost in linear fashion. For sum



algorithm, the efficiency of parallel program stayed around 0.92 during p=2 to p=8. However, it significantly dropped at p=16. This demonstrates that the parallel function almost has strong scaling until number of threads reach 8. Between two parallel functions, parallel for had better performance. This is because for directive contains reduction operation, which is more efficient than using array to collect partial sum and add up using another loop. This difference is visible on speedup chart at p=8. On average, parallel code introduces about 2ms of overhead compare to serial code. For sum algorithm, running parallel for function with 8 threads demonstrates the best performance among all test cases.

Performance of mm.c

When running all functions with single thread, parallel functions are about 1.3 times slower than serial implementation due to the overhead. As number of thread increases, the performance of parallel program continuously increased. At 8 threads, the execution time reduces to 13% what it takes to run with one thread. For matrix multiplication algorithm, difference in decomposition method did not make difference in performance. This is because the there is no memory location that multiple threads are trying to write to same location, and each function has same amount of resources to perform the computation. Another finding is that as number of thread increases, the efficiency of program decreases. At p=2, efficiency is at 0.99, but it drops to 0.93 at p=8 with almost 800ms of overhead compare to serial run. From speedup point of view, the functions demonstrated close to strong scaling till p=16. At p=32, the speedup did not significantly change from p=16. Therefore, the matrix multiplication algorithm with any parallel decomposition at p=16 performs the best (around 630ms).



Performance of sum.c with different scheduling policy

In order to effectively run various types of scheduling policy, runtime option is used inside the code. To select which policy to use, environment variable "OMP_SCHEDULE" is defined. It is notable that simply stating as "static" changes to "static, 1" by GCC compiler, so it must be typed as "static, 0" for default static option.

Within each policy, larger chunk size has better performance than smaller chunk size. It is significant for dynamic scheduling. This is because smaller chunk size requires each thread to frequently check the next availability of tasks to work on, as each task is small. Between different policies, static is performing best in this experiment. It is because the task dynamic scheduling introducing additional overhead since it needs runtime processing to determine which thread gets next tasks. Between dynamic and guided, there was not significant difference in performance. For simple calculation, static scheduling seems to be the better method to choose.

4. Conclusion

Parallelizing the algorithm and executing with multiple threads has definite improvement in performance, however, it decreases efficiency of code execution. It is found that running parallel function with one thread is generally slower than purely serial function due to overhead. The sum algorithm had best performance at p=8 with parallel for function (13% of serial exaction time). For matrix multiplication, the speedup trend continued till p=16, and there was not much difference among various parallel functions. Also, efficiency drops to around 93%. When comparing scheduling policy, larger chunk size has better performance as each thread is working more time than checking for next task availability. The static policy had better performance as it uses less resource during runtime. Overall, the assignment successfully analyzed the performance improvement, overhead, and efficiency of penalization of algorithm with OpenMP.

References

How GCC compiler handles OMP_SCHEDULE environment variable for schedule (runtime)
(https://gcc.gnu.org/onlinedocs/gcc-4.3.6/libgomp/OMP_005fSCHEDULE.html#OMP_005fSCHEDULE)