

Vector Addition

Problem

This is a basic vector addition code. Modify the code to sum the two vectors in parallel. Experiment with OpenMp pragmas such as schedule and any other pragmas that you think may improve the performance of the code. Comment on the results for different variations.

Note

- Implemented different versions of the program to experiment with various OpenMp pragmas and computed the average run time by running each program 10 times.
- The serial program is **addition.c** and the speedup of each parallel version is computed using (Serial Runtime of addition.c)/(Runtime of parallel version)
- All the times are in seconds.
- Best performing results are marked green.
- The optimized code (**optimized_addition.c**) is cooked up after analysis of all the following experiments.

Analysis of the serial program

Implementation

addition.c

Serial Runtime

0.0382444 seconds

Experiment 1

Parallelizing the program by putting a '#pragma omp for' on the main loop for vector addition.

Implementation

addition_parallel_for.c

Experimental Data

Thread count	1	2	4	6	12
Average Run Time	0.0428711	0.0244711	0.0134756	0.0098727	0.0266772
Speedup	.8920	1.562	2.838	3.873	1.433

Observations

- The performance improves with more number of threads.
Analysis: This is due to the parallelism involved in the work-share model of OpenMp.

- The serial execution time is better (i.e. less) than the run time of parallel version for 1 thread.

Analysis: This is due to the fact that there is a small overhead in thread creation, maintenance and synchronization and join even though there is only the master thread (thread 0) which is doing the operation.

Experiment 2

Parallelize using OpenMp pragma 'nowait'.

Motivation

As the work assigned to each of the threads is independent, they can finish their work independently and leave the implicit barrier even without waiting for the other threads.

Implementation

addition_parallel_for_nowait.c

Experimental Data

Thread count	1	2	4	6	12
Average Run Time	0.0429496	0.0238323	0.01318	0.0097545	0.0278128
Speedup	.890	1.604	2.901	3.920	1.375

Observation

- The performance improves as compared to experiment 1.
- The serial execution time is better (i.e. less) than the run time of parallel version for 1 thread.

Analysis: This is due to the fact that there is a small overhead in thread creation, maintenance and synchronization and join even though there is only the master thread (thread 0) which is doing the operation.

Experiment 3

Parallelize with OpenMp pragma schedules: clauses static, dynamic and guided.

Static scheduling with different chunk size

Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads. We also experimented with chunk size == 1.

Implementation

addition_parallel_for_schedule_static_exp1.c // Using schedule static clause without any chunk size

addition_parallel_for_schedule_static_exp2.c // Using schedule static clause with chunk size = 1

Experimental Data

Table 1: Using static schedule without chunk size

Thread count	1	2	4	6	12
Average Run Time	0.0438986	0.0248705	0.0138678	0.0105537	0.0270016
Speedup	0.8711	1.5377	2.7577	3.6237	1.41637

Table 2: Using static schedule with chunk size = 1

Thread count	1	2	4	6	12
Average Run Time	0.042433	0.0257133	0.0150692	0.0109512	0.0275479
Speedup	0.9012	1.4873	2.5379	3.4922	1.3882

Dynamic scheduling with different chunk sizes

We have experimented with different chunk sizes like 10000, 1000, 100, 1.

Implementation

addition_parallel_for_schedule_dynamic_chunk_10000.c

addition_parallel_for_schedule_dynamic_chunk_1000.c

addition_parallel_for_schedule_dynamic_chunk_100.c

addition_parallel_for_schedule_dynamic_chunk_1.c

Experimental Data

Table 1: Using dynamic schedule with chunk size = 10000

Thread count	1	2	4	6	12
Average Run Time	0.0430198	0.0218191	0.012829	0.0098943	0.0269101
Speedup	0.888	1.752	2.981	3.865	1.4211

Table 2: Using dynamic schedule with chunk size = 1000

Thread count	1	2	4	6	12
Average Run Time	0.0403506	0.0232349	0.0141845	0.0110717	0.0240052
Speedup	0.9478	1.6459	2.6962	3.4542	1.593

Table 3: Using dynamic schedule with chunk size = 100

Thread count	1	2	4	6	12
Average Run Time	0.0436366	0.0361779	0.031442	0.0363067	0.0557251
Speedup	0.8764	1.057	1.216	1.053	0.686

Table 4: Using dynamic schedule with chunk size = 1

Thread count	1	2	4	6	12
Average Run Time	0.1094314	0.3732927	0.5639068	0.3906575	0.3192758
Speedup	0.3494	0.102	0.0678	0.0978	0.1197

Observation

- With dynamic scheduling, as the chunk size goes down, performance degrades.

Analysis: With dynamic scheduling, when the chunk size is 1, each thread has to ask the OpenMP runtime for each iteration of the distributed loop. This incurs overhead which reduces when the chunk size increases as each thread has to ask the OpenMP runtime for lesser number of times.

Guided scheduling with chunk size = 1

Implementation

addition_parallel_for_schedule_guided.c

Experimental Data

Table 1: Using guided schedule with chunk size = 1

Thread count	1	2	4	6	12
Average Run Time	0.0411292	0.0216451	0.0123379	0.009378	0.0222145
Speedup	0.9298	1.766	3.0997	4.0780	1.7215

Observation

- Guided scheduling has a better performance than others (dynamic/guided).

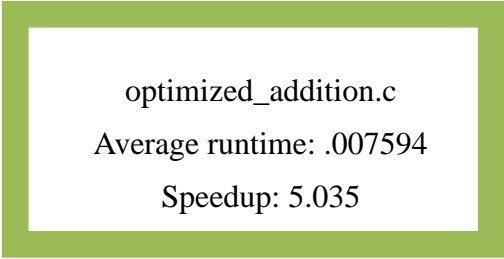
Analysis: As we know, dynamic scheduling generates significant overhead if the chunks are too small in terms of execution time. This is why it is often desirable to use a moderately large chunk size on tight loops, which in turn leads to more load imbalance. In this particular case where this might be a problem and hence the guided schedule helped a lot. Again, threads request new chunks dynamically, but the chunk size is always proportional to the remaining number of iterations divided by the number of threads. The smallest chunk size is specified in the schedule clause (default is 1). Despite the dynamic assignment of chunks, scheduling overhead is kept under control.

Moreover the reason that guided scheduling performs better than static might be attributed to load imbalance present among different threads and as a result static scheduling does not come up as a good choice.

Cooking up the Optimized Code

The leanings from the above experiments help us to cook up the optimized code with the following features:

1. ‘#pragma omp parallel for’ put to the outermost loop
2. Set OMP_NUM_THREADS to 6
3. Used schedule (guided)
4. Used ‘nowait’ clause



optimized_addition.c
Average runtime: .007594
Speedup: 5.035

Matrix Multiplication

Problem

This code multiplies two matrices using triple nested for loops. Develop at least two parallelizations using only 1 parallel for loop. Also experiment with different nested parallelizations. Create one parallelization using the collapse directive and a second using pragmas on multiple levels of the nested loop.

Note

- Implemented different versions of the program to experiment with various OpenMp pragmas and computed the average run time by running each program 10 times.
- The serial program is **multiply.c** and the speedup of each parallel version is computed using (Serial Runtime of multiply.c)/(Runtime of parallel version)
- All the times are in seconds.
- Best performing results are marked green.
- The optimized code (**optimized_multiply.c**) is cooked up after analysis of all the following experiments.

Analysis of the serial program

Implementation

multiply.c

Serial Runtime

15.092023 seconds

Experiment 1

Before we begin with some parallel optimization of the serial code, we can have one serial optimization on the code, which is the following.

Loop Reordering We have experimented with 6 loop orderings (ijk, ikj, jik, jki, kij, kji) and got the following results:

Loop Order	Average-Runtime	Speedup
ijk	15.534865	0.9714
ikj	7.748856	1.947
jik	9.908139	1.523

jki	29.200268	0.516
kij	7.944300	1.899
kji	22.363676	0.674

Implementation

```
multiply_loop_interchange_ijk.c
multiply_loop_interchange_jik.c
multiply_loop_interchange_kij.c
multiply_loop_interchange_ikj.c
multiply_loop_interchange_jki.c
multiply_loop_interchange_kji.c
```

Observation

- The loop order ikj and kij outperforms other orderings.

Analysis: We know that, the matrices A, B and C are stored in row major order. For the order ijk, A is accessed sequentially in row major order (and hence utilizing the spatial locality) but the access of B is n-strided and hence very slow. More generally, we have the following accesses in matrix multiplication,

$$A[i][j] += B[i][k] * C[k][j]$$

Now if we have i or k as the inner loop, then the consecutive accesses will be strided because of poor utilization of spatial locality. But if we avoid having i or k in the inner loop, in other words if we keep j as the inner loop, then spatial locality will benefit accesses of matrices A and C. Hence the favorable choices of orders are ikj and kij. In ikj, k is the second fastest growing index which leads to strided read access of C (during the middle iteration) and in kij, i is the second fastest growing index which lead to strided write access of A. As we know that write access are more costly (in terms of time) than read access, so the performance of ikj is better.

Experiment 2 (parallelization using OpenMp pragma for)

Introduce 'pragma omp for' parallelization on the outermost for loop and got the following results.

Thread Count	Average-Runtime	Speedup
1	16.020388	0.942
2	7.987371	1.889
4	4.212864	3.5823
6	2.964770	5.090
12	2.305540	6.545

Implementation

multiply_parallel_outer_for.c

Observation

- The performance improves with more number of threads.

Analysis: This is due to the parallelism involved in the work share model of OpenMp.

- The serial execution time is better (i.e. less) than the run time with thread count == 1.

Analysis: This is due to the fact that there is a small overhead in thread creation, maintenance and synchronization and join even though there is only the master thread (thread 0) which is doing the operation.

Experiment 3 (parallelization using OpenMp pragma collapse)

Introduce 'pragma omp collapse' parallelization (along with 'pragma for' in the outermost loop) and using that to collapse different loops.

Average Runtime					
Thread Count	1	2	4	6	12
collapsing 1 loop	21.521892	9.858506	4.667486	2.971117	2.347690
collapsing 2 loops	16.054157	7.968558	4.143699	2.882800	2.140681
collapsing 3 loops	20.067321	10.759646	5.026925	3.459996	2.407979

Speedup					
Thread Count	1	2	4	6	12
collapsing 1 loop	0.701	1.5308	3.2334	5.07	6.428
collapsing 2 loops	0.940	1.89	3.642	5.235	7.050
collapsing 3 loops	0.752	1.402	3.002	4.3618	6.2675

Implementation

multiply_forloop_collapse_1.c
multiply_forloop_collapse_2.c
multiply_forloop_collapse_3.c

Observation

- Collapsing improves the performance of 'pragma omp for' code. The speedup improves from 6.545 (experiment 2 with 12 threads) to 7.050 (experiment 3 with collapsing 2 loops and 12

threads)

Analysis: With collapse(2) , the outer two loop levels are collapsed into one with a loop length of $n*n$, and the resulting long loop will be executed by all threads in parallel. This ameliorates any load balancing problem and hence improves the performance because now more iteration space (in this case $n*n$) is available to get parallelized.

- Collapsing 3 loops is giving bad performance than collapsing 2 loops.

Analysis: Without collapse(3) we have n iteration chunks of the outer loop which will be divided among the threads and each thread runs $n*n$ iterations (of the inner loops). When we collapse 3 loops, the number of iteration chunks grows to $n*n*n$, i.e. there are many more chunks to be divided among the threads.

Experiment 4 (parallelization by putting 'omp pragma for' on different nested loop levels)

4.1. Introduced 'pragma omp parallel for' parallelization in the middle loop.

Average Runtime					
Thread Count	1	2	4	6	12
parallelizing middle loop	16.663160	8.275227	5.795834	3.679095	31.229117

Speedup					
Thread Count	1	2	4	6	12
parallelizing middle loop	0.905	1.823	2.603	4.102	0.483

Implementation

multiply_parallel_middle_for.c

Observation

- The performance is poorer than the parallelizing outermost for.

Analysis: The effect of putting '#pragma omp parallel for' in the middle loop is that for each iteration of i (in this case 1000 times), a team of threads will be created and jobs will be distributed to them. This creation of team of threads and the associated bookkeeping degrades the performance.

- The performance at thread count 12 is worse than other thread counts.

Analysis: As the number of threads increases the overhead (i.e. creation, bookkeeping and synchronization) also got tremendously large.

From these two observations, one obvious improvement is the next experiment.

4.2. Introduced 'pragma omp for' parallelization in the middle loop, but kept the thread creation overhead once only at the outer loop.

This is done by putting '#pragma omp parallel' at outer loop and '#pragma omp for' in the middle loop.

Average Runtime					
Thread Count	1	2	4	6	12
parallelizing middle loop with 1 time thread creation	16.288582	7.886915	5.512594	3.567394	16.313984

Speedup					
Thread Count	1	2	4	6	12
parallelizing middle loop with 1 time thread creation	0.9265	1.9135	2.7377	4.2305	0.925

Implementation

multiply_parallel_middle_for_ver2.c

Observation

- The performance is improved from Experiment 4.1

Analysis: In this case the thread team will be formed at the very outer loop only once and each of team members will be iterating over the i loop sequentially. Now for each value of i, the iteration space of j will be distributed among the team. This approach reduces the thread creation overhead by n times and hence its performance is better than Experiment 4.1.

But still the performance is poor as compared to parallelizing outermost for (experiment 2), as the outer loop (i) is traversed sequentially by the threads as opposed to parallel traversal in Experiment 2.

4.3. Introduced 'pragma omp parallel for' parallelization in the inner loop.

In order to parallelize inner loops k, we have to be extra cautious, as multiple threads will try to write the same location leading to a race condition. To avoid that we have to use OpenMp pragmas 'reduction'.

Average Runtime					
Thread Count	1	2	4	6	12
parallelizing inner loop	18.55	11.43	8.924	9.20320	16.120

Speedup					
Thread Count	1	2	4	6	12
parallelizing inner loop	0.813	1.320	1.6911	1.6398	0.936

Implementation

multiply_parallel_inner_for.c

Observation

- The performance is poor as compared to parallelizing any of the out loop.

Analysis: The effect of putting '#pragma omp parallel for' in the inner loop is that for each iteration of $i*j$ (in this case $1000*1000$ times), a team of threads will be created and jobs will be distributed to them. This creation of team of threads and the associated bookkeeping degrades the performance to large extent.

Cooking up the Optimized Code

The leanings from the above experiments help us to cook up the optimized code with the following features:

- Loop ordered to ikj
- '#pragma omp parallel for' put to the outermost loop
- Collapsing 2 outer consecutive loops and use the 'schedule (static)' clause to distribute the iteration space of the bigger loop formed among the threads.
- Set OMP_NUM_THREADS to 12

optimized_multiply.c

Runtime: 1.125914

Speedup: 13.4042

Matrix Transpose

Problem

This code transposes a matrix. Similar to the matrix multiplication code, develop at least two parallelizations using only 1 parallel for loop. Also experiment with different nested parallelizations. Create one parallelization the collapse directive and a second using pragmas on multiple levels of the nested loop.

Note

- Implemented different versions of the program to experiment with various OpenMp pragmas and computed the average run time by running each program 10 times.
- The serial program is **transpose.c** and the speedup of each parallel version is computed using (Serial Runtime of transpose.c)/(Runtime of parallel version)
- All the times are in seconds.
- Best performing results are marked green.
- The optimized code (**optimized_transpose.c**) is cooked up after analysis of all the following experiments.

Analysis of the serial program

Implementation

transpose.c

Serial Runtime

0.289681 seconds

Experiment 1(parallelization using 'omp pragma for')

Introduce 'pragma omp for' parallelisation on the outermost for loop and got the following results.

Thread Count	Average-Runtime	Speedup
1	0.349129	0.829
2	0.246907	1.1732
4	0.174720	1.657
6	0.109028	2.65
12	0.128158	2.260

Implementation

transpose_parallel_outer_for.c

Observation

- The performance improves with more number of threads.

Analysis: This is due to the parallelism involved in the work share model of OpenMp.

- The serial execution time is better (i.e. less) than the run time with thread count == 1.

Analysis: This is due to the fact that there is a small overhead in thread creation, maintenance and synchronization and join even though there is only the master thread (thread 0) which is doing the operation.

- Though the runtime for thread count == 12 is better than the serial version, but it is more than thread count == 6

Analysis: With increase in thread count the overhead also goes up, as the OpenMp runtime has to do some maintenance/bookkeeping/synchronization of those threads. That is the reason that though the performance is better than serial version, but it is not as better as with thread count == 6.

Experiment 2 (parallelization using OpenMp pragma collapse)

Introduce 'pragma omp collapse' parallelization (along with 'pragma for' in the outermost loop) and using that to collapse different loops.

2.1. Collapsing one outer loop

This is done just for experimentation purpose.

Thread Count	1	2	4	6	12
Average Runtime	0.316280	0.248827	0.145274	0.124198	0.133936

Thread Count	1	2	4	6	12
Speedup	0.915	1.164	1.994	2.332	2.162

Implementation

transpose_parallel_collapse_1.c

Observation

- Collapsing improves the performance over serial code. The average runtime improves from 0.289681 sec (serial code) to 0.124198 secs (6 threads) or 0.133936 sec (12 threads)

Analysis: Actually this improvement over serial code is mainly due to the 'omp pragma for'. In fact collapsing 1 loop has no significance over not collapsing any loop and more importantly this

collapsing makes the performance poor as compared to the code with 'omp pragma for' in experiment 1 (as explained in next observation.)

- Collapsing 1 loops is giving bad performance than Experiment 1.

Analysis: This is due to the fact that collapsing itself has its OpenMp runtime overheads. So it seems better not to collapse one loop (which is in fact meaningless). On the other hand, the code at experiment 1 has no such overhead.

2.2. Collapsing two loops.

The COLLAPSE clause (introduced with OpenMP 3.0) is used for perfect loop nests, i.e., with no code between the different do (and enddo) statements and loop counts not depending on each other. Here the inner j loop depends on outer i loop, so collapsing 2 loops are not possible.

Experiment 3 (parallelization by putting 'omp pragma for' on different nested loop levels)

3.1. Introduced 'pragma omp parallel for' parallelization in the inner loop.

Thread Count	1	2	4	6	12
Average Runtime	0.386463	0.173359	0.103529	0.091139	55.452560

Thread Count	1	2	4	6	12
Speedup	0.749	1.670	2.798	3.1784	0.0052

Implementation

transpose_parallel_inner_for.c

Observation

- The performance is better than the parallelizing outermost for.

Analysis: When we put '#pragma omp parallel for' on the outer loop, then the iteration space of I will be distributed among the threads and each of the threads has to do the following task.

```
for(j=i+1;j<n;j++) {  
    temp=A[i][j];  
    A[i][j] = A[j][i];  
    A[j][i] = temp;  
}
```

Now as we can see that the tasks are not balanced. In fact, as i increase, the task assigned to each thread get smaller and smaller. The result is most of the threads will finish early and be idle. This load imbalance is alleviated as we parallelize the inner loop. With that, all the threads will be working on the iteration space of j (for each i) and the threads will balance the load effectively.

- The performance at thread count 12 is worse than other thread counts.

Analysis: As the number of threads increases the overhead (i.e. creation, bookkeeping and synchronization) also got tremendously large.

Scope of improvement

The effect of putting '#pragma omp parallel for' in the inner loop is: for each iteration of i , creation of team of threads, associated bookkeeping and synchronization will happen which is an overhead and now we will target to reduce that.

The next experiment 3.2 will address that.

3.2. Introduced 'pragma omp for' parallelization in the inner loop, but kept the thread creation overhead once only at the outer loop.

This is done by putting '#pragma omp parallel' at outer loop and '#pragma omp for' in the inner loop.

Thread Count	1	2	4	6	12
Average Runtime	0.388247	0.176759	0.088556	0.067340	50.588359

Thread Count	1	2	4	6	12
Speedup	0.7461	1.63	3.27	4.301	0.0057

Implementation

transpose_parallel_inner_for_ver2.c

Observation

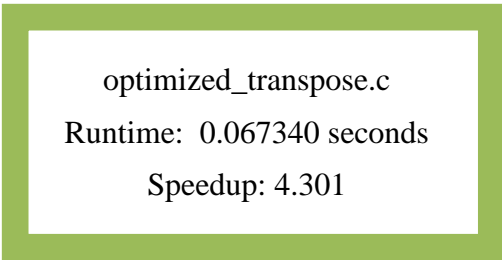
- The performance is improved from Experiment 3.1

Analysis: In this case the thread team will be formed at the very outer loop only once and each of team members will be iterating over the i loop sequentially. Now for each value of i , the iteration space of j will be distributed among the team. This approach reduces the thread creation overhead and hence its performance is better than Experiment 3.1.

Cooking up the Optimized Code

The leanings from the above experiments help us to cook up the optimized code with the following features:

1. ‘#pragma omp for’ put to the inner loop and ‘#pragma omp parallel’ to the outer loop.
2. Set OMP_NUM_THREADS to 6.



optimized_transpose.c
Runtime: 0.067340 seconds
Speedup: 4.301

Tiled Matrix Multiplication

Problem

This code takes the original matrix multiply and uses an optimized block implementation. Find at least one optimization you can make to the serial code to improve the serial performance. Similar to the matrix multiplication code, develop at least two parallelizations using only 1 parallel for loop. Also experiment with different nested parallelizations. Create one parallelization using the collapse directive and a second using pragmas on multiple levels of the loop. Experiment with different block sizes and discuss the impact of block size on performance.

Note

- Implemented different versions of the program to experiment with various OpenMp pragmas and computed the average run time by running each program 10 times.
- The serial program is **tiled.c** and the speedup of each parallel version is computed using (Serial Runtime of tiled.c)/(Runtime of parallel version)
- All the times are in seconds.
- Best performing results are marked green.
- The optimized code (**optimized_tiled.c**) is cooked up after analysis of all the following experiments.

Analysis of the serial program

Implementation

tiled.c

Serial Runtime

12.349107 seconds

Experiment 1 (Serial Optimization)

The serial optimization that we implemented is loop re-ordering. **We have shown in "Program 2. Matrix Multiplication", that the particular loop order of 'ikj' gives best performance among all other permutations of loop orders.**

Implementation

tiled_serial_optimization.c

Average runtime: 10.537829

Speedup : 1.1718

Experiment 2 (parallelization by putting 'omp pragma for' on different nested loop levels)

Table 1: Average Runtime

Thread Count	1	2	4	6	12
parallelizing loop ii	12.560247	7.717031	5.027308	3.429746	2.106260
parallelizing loop jj	13.625174	7.872090	5.210185	4.048191	3.006135
parallelizing loop kk	19.865358	51.563600	165.703819	281.942046	466.393704
parallelizing loop i	14.339194	8.622304	10.125982	12.698576	15.458028
parallelizing loop j	14.157095	9.189533	17.290484	59.393478	49.689194
parallelizing loop k	19.283338	84.690121	290.503487	402.489264	3835.162051

Table 2: Speedup

Thread Count	1	2	4	6	12
parallelizing loop ii	0.9831	1.600	2.456	3.600	5.863
parallelizing loop jj	0.906	1.568	2.370	3.050	4.107
parallelizing loop kk	0.621	0.239	0.074	0.043	0.0264
parallelizing loop i	0.8612	1.432	1.219	0.972	0.7988
parallelizing loop j	0.8722	1.343	0.714	0.207	0.248
parallelizing loop k	0.640	0.145	0.042	0.0306	0.003

Implementations

tiled_parallel_outer_for_ii.c
tiled_parallel_outer_for_jj.c
tiled_parallel_outer_for_kk.c
tiled_parallel_outer_for_i.c
tiled_parallel_outer_for_j.c
tiled_parallel_outer_for_k.c

Observation

- The performance improves with more number of threads.

Analysis: This is due to the parallelism involved in the work share model of OpenMp.

- The serial execution time is better (i.e. less) than the run time with thread count = 1.

Analysis: This is due to the fact that there is a small overhead in thread creation, maintenance and synchronization and join even though there is only the master thread (thread 0) which is doing the operation.

- The performance of parallelizing inner loops like jj, i and j is poorer than the parallelizing outermost for.

Testing Methods

While testing these cases, we have deliberately put '#pragma omp parallel' at the outermost loop level. This is because of the learning that we have got from all the previous experiments that putting '#pragma omp parallel' at an inner loop level will lead to enhanced thread creation overhead which will degrade the performance. In order to parallelize inner loops like jj, i and j, we have put '#pragma omp for' at those loop levels.

Analysis: While we are parallelizing inner loops jj, i and j, all the threads need to iterate sequentially over the outer loops and that lead to poor performance than parallelizing outer loop.

- The performance of parallelizing inner loops like kk and k is worst.

Testing Methods

While testing these cases, we have deliberately put '#pragma omp parallel' at the outermost loop level. This is because of the learning that we have got from all the previous experiments that putting '#pragma omp parallel' at an inner loop level will lead to enhanced thread creation overhead which will degrade the performance. In order to parallelize inner loops k and kk, we have to be extra cautious, as multiple threads will try to write the same location leading to a race condition. To avoid that we have to use omp pragmas like 'reduction' or 'critical'. As we know that omp reduction does not work on arrays (as there is a race condition to write C[i][j]) we use critical here.

Analysis: Usage of 'pragma critical' introduces the overhead of mutual exclusive access and synchronization which become humongous when the number of threads increases. This is because of the fact that if a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.

Experiment 3 (parallelization using omp pragma collapse)

Introduce 'pragma omp for collapse' parallelization (along with 'pragma for' in the outermost loop) and using that to collapse different loops.

Table 3: Average Runtime

Thread Count	1	2	4	6	12
collapsing 1 loop	13.018146	6.271444	4.483590	3.960699	2.539019
collapsing 2 loop	10.810555	6.569333	4.098443	3.204124	1.927341
collapsing 3 loop	11.956086	7.398638	4.058250	2.817681	1.839456

Table 4: Speedup

Thread Count	1	2	4	6	12
collapsing 1 loop	0.9486	1.96910	2.754	3.117	4.863
collapsing 2 loop	1.142	1.879	3.013	3.854	6.407
collapsing 3 loop	1.032	1.669	3.042	4.382	6.713

Implementation

tiled_collapse_1.c
 tiled_collapse_2.c
 tiled_collapse_3.c

NOTE: We cannot collapse more than 3 loops here. The reason is COLLAPSE clause (introduced with OpenMP 3.0) is used for perfect loop nests, i.e., with no code between the different do (and enddo) statements and loop counts not depending on each other. Here the inner loops (i, j, k) depend on outer loops ii, jj, kk resp., so collapsing more than 3 loops is not possible.

Observation

- Collapsing improves the performance over serial code. The average runtime improves 12.349107 sec (serial code) to 2.817681 secs (collapsing 3 loop with 6 threads) or 1.839456 (collapsing 3 loop with 12 threads).

Analysis: As we know, collapsing help in alleviating load imbalance. Collapsing 3 loops help to achieve more parallelization as now more iteration space $n*n*n$ is open for parallelization.

Experiment 4 (Experimenting with different block size on the serial code)

To achieve this we modified the serial code and add the customization code to use different block sizes (1, 2, 4, ..., 512) and got the following runtimes.

bsize	Runtime	Speedup
1	36.618085	
2	17.537852	
4	12.040961	
8	8.730942	
16	8.383561	
32	8.255950	
64	9.361085	
128	11.911677	
256	12.017908	
512	14.490553	

Implementation

tiled_block_size_exp.c

Observation

- While increasing *bsize* from 1 to 32, the performance improves.

Analysis: In the non-blocked version, we have a n-strided access on B[k,j] and hence the spatial locality is not utilized much. The main insight towards doing blocked implementation is that the block will stay in the cache and the subsequent accesses can benefit from that. As the bsize increases, the bigger blocks can reside in the cache and all the subsequent accesses in that block will be accessed very fast.

- From *bsize* 64 to 512, the performance degrades.

Analysis: If we enhance the *bsize* after a certain limit (in my case it is 32, the block size become too big to fit in the cache and as a result the cache has to evict some lines and fetch the blocks from memory to the cache, which degrades the performance.

Cooking up the Optimized Code

The leanings from the above experiments help us to cook up the optimized code with the following features:

1. Used loop ordering ikj.
2. ‘#pragma omp parallel for’ put to the outer loop.
3. Set OMP_NEUM_THREADS = 12
4. bsize = 32
5. use collapse(3) along with schedule(static)

optimized_tiled.c

Runtime: 1.5727seconds

Speedup: 7.851

Tiled Transpose

Problem

Develop a tiled version of the transpose.c example (called tiled_transpose.c) similar to the tiled.c matrix multiply example. Make any changes anywhere in the code necessary to make the loop that computes the transpose run as fast as possible. Discuss multiple different improved serial optimizations and improved parallel implementations. Experiment with different block sizes and discuss the impact of block size on performance.

Note

- Implemented different versions of the program to experiment with various OpenMp pragmas and computed the average run time by running each program 10 times.
- The serial program is **tiled_transpose.c** and the speedup of each parallel version is computed using
(Serial Runtime of tiled_transpose.c)/(Runtime of parallel version)
- All the times are in seconds.
- Best performing results are marked green.
- The optimized code (**optimized_tiled_transpose.c**) is cooked up after analysis of all the following experiments.

Analysis of the serial program

Implementation

tiled_transpose.c

Serial Runtime

0.342363 seconds with block size = 32

But the question is how do we get that particular block size of 32??

For that we performed the following experiment of running tiled transpose on different block sizes.

Experiment 1 (Serial Optimization, refining the block size)

We executed the tiled transpose on different block sizes ranging from 1, 2, 4, ..., 2^n , ..., 2^{12} and got the following observation.

Implementation

tiled_transpose_exp_with_block_size.c

bsize	Runtime
1	0.687030
2	0.477087
4	0.373344
8	0.362363
16	0.353636
32	0.342363
64	0.364753
128	0.355683
256	0.376835
512	0.351535
1024	0.406048
2048	0.432060
4096	0.439796

Observation

- While increasing bsize from 1 to 32, the performance improves.

Analysis: In the non-blocked version, we have a n-strided access on $A[j,i]$ (as j is the fastest growing loop) and hence the spatial locality is not utilized much. The main insight towards doing blocked implementation is that the block will stay in the cache and the subsequent accesses can benefit from that. As the bsize increases, the bigger blocks can reside in the cache and all the subsequent accesses in that block will be accessed very fast.

- From bsize 64 onwards, the performance degrades.

Analysis: If we enhance the bsize after a certain limit (in my case it 32, the block size become too big to fit in the cache and as a result the cache has to evict some lines and fetch the blocks from memory to the cache, which degrades the performance.

Experiment 2 (Loop unrolling and jamming)

Our next serial optimization is to unroll the inner loop.

Motivation: For the transpose, $B[i,j] = A[j,i]$, we have a n-strided access on $A[j,i]$ (as j is the fastest growing loop). So for better utilization of spatial locality we can unroll the loop and for each iteration we will try to access $A[j, i+1]$, $A[j, i+2]$, ..., $A[j, i+m]$ which are already in the cache.

Implementation

tiled_transpose_with_loop_unroll.c

Runtime

Average runtime: 0.228624 with block size == 32 and loop unroll count = 6.

Speedup: 1.497

But the question is how do we get that particular loop unroll count == 6.

For this we performed the experiment on different 'loop unroll count' and picked the best possible value.

Implementation

tiled_transpose_with_loop_unroll_loop_count_exp.c

We have taken different loop unroll count (LUC) from 1 to 100.

LUC	Runtime	LUC	Runtime	LUC	Runtime	LUC	Runtime
1	0.426090	2	0.362500	3	0.244791	4	0.230575
5	0.232028	6	0.228624	7	0.234316	8	0.229792
9	0.234277	10	0.235413	11	0.256010	12	0.250743
13	0.248435	14	0.249249	15	0.250521	16	0.254629
17	0.274835	18	0.271376	19	0.268225	20	0.272535
21	0.271691	22	0.266879	23	0.268558	24	0.268467
25	0.265445	26	0.270225	27	0.267090	28	0.269847
29	0.268615	30	0.273251	31	0.275096	32	0.277170
33	0.322991	34	0.322887	35	0.322993	36	0.322898
37	0.323131	38	0.322975	39	0.322840	40	0.322976
41	0.322869	42	0.322976	43	0.323134	44	0.322797
45	0.323011	46	0.322804	47	0.323007	48	0.322847
49	0.323040	50	0.323089	51	0.322818	52	0.322951
53	0.322922	54	0.323014	55	0.322937	56	0.322904
57	0.322966	58	0.322801	59	0.323175	60	0.322755
61	0.322959	62	0.323063	63	0.322800	64	0.322992
65	0.322854	66	0.322981	67	0.323016	68	0.322924
69	0.322965	70	0.322782	71	0.323024	72	0.322955
73	0.322978	74	0.323022	75	0.322914	76	0.322980
77	0.322786	78	0.323104	79	0.322947	80	0.322849
81	0.323082	82	0.322924	83	0.322965	84	0.323066

85	0.322821	86	0.323040	87	0.322923	88	0.323001
89	0.322819	90	0.323232	91	0.322988	92	0.322817
93	0.323099	94	0.322852	95	0.323002	96	0.323016
97	0.322865	98	0.323011	99	0.322801		

Experiment 3 (parallelization by putting 'omp pragma for' on different nested loop levels)

NOTE: This optimization is done on top of optimization of loop unrolling and jamming.

Table 1: Average Runtime

Thread Count	1	2	4	6	12
parallelizing loop ii	0.256892	0.133151	0.075387	0.054464	0.073417
parallelizing loop jj	0.255687	0.131239	0.075512	0.056326	0.275527
parallelizing loop i	0.267522	0.182008	0.251193	0.185980	0.227348
parallelizing loop j	0.267784	0.325106	0.332658	0.774928	0.616990

Table 2: Speedup

Thread Count	1	2	4	6	12
parallelizing loop ii	1.332	2.571	4.541	6.28	4.66
parallelizing loop jj	1.338	2.608	4.533	6.078	1.242
parallelizing loop i	1.279	1.881	1.362	1.840	1.505
parallelizing loop j	1.278	1.053	1.0291	0.441	0.554

Implementations

tiled_transpose_with_loop_unroll_parallel_for_ii.c

tiled_transpose_with_loop_unroll_parallel_for_jj.c

tiled_transpose_with_loop_unroll_parallel_for_i.c

tiled_transpose_with_loop_unroll_parallel_for_j.c

Observation

- The performance improves with more number of threads.

Analysis: This is due to the parallelism involved in the work share model of OpenMp.

- The performance at thread count 12 is worse than other thread counts.

Analysis: As the number of threads increases, the overhead (i.e. creation, bookkeeping and synchronization) also got tremendously large.

- The serial execution time (after loop unrolling and jamming, which is 0.228624 seconds) is better (i.e. less) than the run time with thread count == 1.

Analysis: This is due to the fact that there is a small overhead in thread creation, maintenance, synchronization and join even though there is only the master thread (thread 0) which is doing the operation.

- The performance of parallelizing inner loops like jj, i and j is poorer than the parallelizing outermost for ii.

Testing Methods:

While testing these cases, we have deliberately put '#pragma omp parallel' at the outermost loop level. This is because of the learning that we have got from all the previous experiments that putting '#pragma omp parallel' at an inner loop level will lead to enhanced thread creation overhead which will degrade the performance. In order to parallelize inner loops like jj, i and j, we have put '#pragma omp for' at those loop levels.

Analysis: While we are parallelizing inner loops jj, i and j, all the threads need to iterate sequentially over the outer loops and that lead to poor performance than parallelizing outer loop.

Experiment 4 (parallelization using omp pragma collapse)

Introduce 'pragma omp collapse' parallelization (along with 'pragma for' in the outermost loop) and using that to collapse different loops.

Table 3: Average Runtime

Thread Count	1	2	4	6	12
collapsing 1 loop	0.252463	0.203558	0.122658	0.085948	0.050225
collapsing 2 loop	0.252122	0.212255	0.121803	0.084283	0.046376

Table 3: Speedup

Thread Count	1	2	4	6	12
collapsing 1 loop	1.3560	1.681	2.791	3.983	6.816
collapsing 2 loop	1.357	1.612	2.810	4.062	7.382

Implementation

tiled_transpose_with_loop_unroll_parallel_for_ii_collapse_1.c
tiled_transpose_with_loop_unroll_parallel_for_ii_collapse_2.c

NOTE: We cannot collapse more than 2 loops here. The reason is COLLAPSE clause (introduced with OpenMP 3.0) is used for perfect loop nests, i.e., with no code between the different do (and enddo) statements and loop counts not depending on each other. Here the inner loops (i, j) depend on outer loops ii, jj, so collapsing more than 2 loops is not possible.

Observation

- Collapsing improves the performance over Experiment 3 on parallelizing outer for loop from 0.054464 secs (for 6 threads) to 0.046376 secs (collapsing 2 loop with 12 threads)

Analysis: As we know, collapsing help in alleviating load imbalance. Collapsing 2 loops help to achieve more parallelization as now more iteration space $n*n$ is opened for parallelizations.

Cooking up the Optimized Code

The leanings from the above experiments help us to cook up the optimized code with the following features:

1. Applied loop unrolling and jamming with loop unroll count = 6.
2. ‘#pragma omp parallel for’ put to the outermost ii loop.
3. Set OMP_NUM_THREADS = 12
4. bsize is set to 32
5. used collapse(2) along with schedule(static)

optimized_tiled_transpose.c

Runtime: 0.046376 seconds

Speedup: 7.382