CS 6210: Scientific and Data Computing I Fall 2024

Assignment 1

Milena Belianovich
October 30, 2024

For parallel code development and testing for this assignment, use any multicore platform with OpenMP, e.g., CADE Lab or your laptop. After code development, performance should be reported on a 20-core node on the Lonepeak CHPC cluster (use batch scripts). Note that you should not perform direct execution of programs on CHPC login nodes (account will get suspended if some threshold of total interactive CPU time is exceeded.

Submission: 1) A PDF report providing explanations as specified for each question, 2) Submission of the fixed buggy.c file and only your final par.c files for questions 2e and 2f (optional, extra credit question).

1 Problems

1. (25 points) Compile and execute the program buggy.c. It has sequential reference code and a parallel code for incrementing the values of elements in an array. The parallel code produces incorrect results. What is the reason? Fix the bug in the parallel program.

   Solution.

   The given code file buggy.c contains two sections: a sequential reference implementation and a parallel implementation designed to increment the values of an array. The task involves:

   - Compiling and executing the code to observe any discrepancies between the sequential and parallel implementations.
   - Identifying and fixing the bug causing incorrect results in the parallel version.

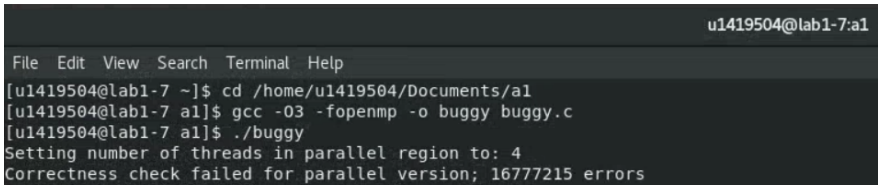

Figure 1: Results shown for buggy.c before fixing

Initial Observations:

Upon executing the original version of buggy.c, the correctness check indicated a significant number of errors in the parallel implementation. The output showed "Correctness check failed for parallel version; 16734511 errors" (as seen in Figure 1).

This discrepancy pointed towards a fundamental flaw in the way the parallel sections were implemented.

Reason for Incorrect Results:

The root cause of the incorrect results stemmed from the structure of the parallel regions. In the original code, two separate parallel regions were defined as seen in Figure 2.

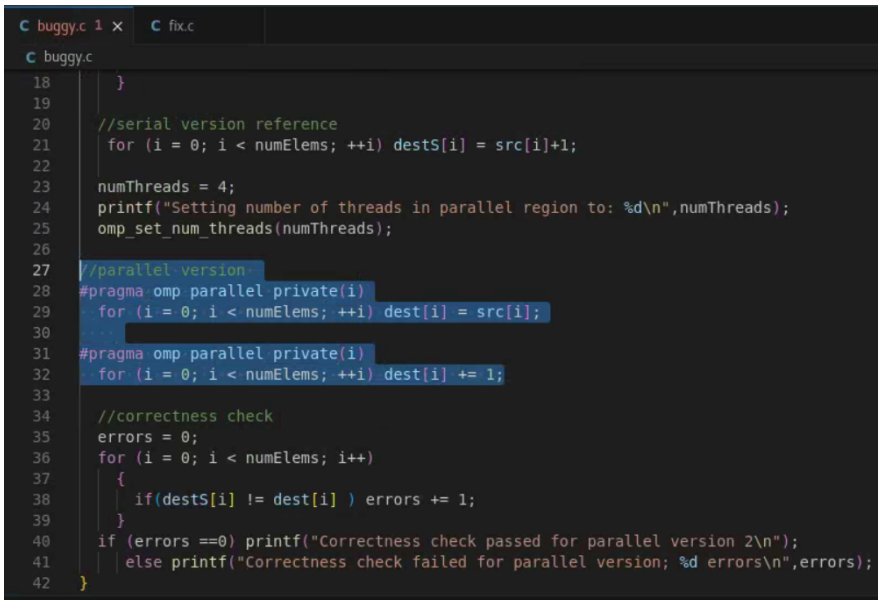

Figure 2: buggy.c inconsistency

In the first parallel region, each thread independently copies elements from src to dest. In the second parallel region, threads attempt to increment elements in dest. However, due to the lack of synchronization and work-sharing, each thread redundantly processes the entire array, leading to an inconsistent and incorrect final result.

Solution Implementation:

To resolve this issue, the parallel regions were combined into a single parallel loop. This approach effectively distributes the workload among threads, ensuring each element is correctly incremented. The corrected version is as seen in Figure 3. This solution leverages the "pragma omp parallel for" directive to distribute the loop iterations among available threads. This resolves the conflict by combining the copy and increment operations into one loop, avoiding redundant parallel execution.

Testing and Results.

The corrected code was compiled and executed on a multicore system using the CADE Lab environment, which provided OpenMP support. The results were as follows:

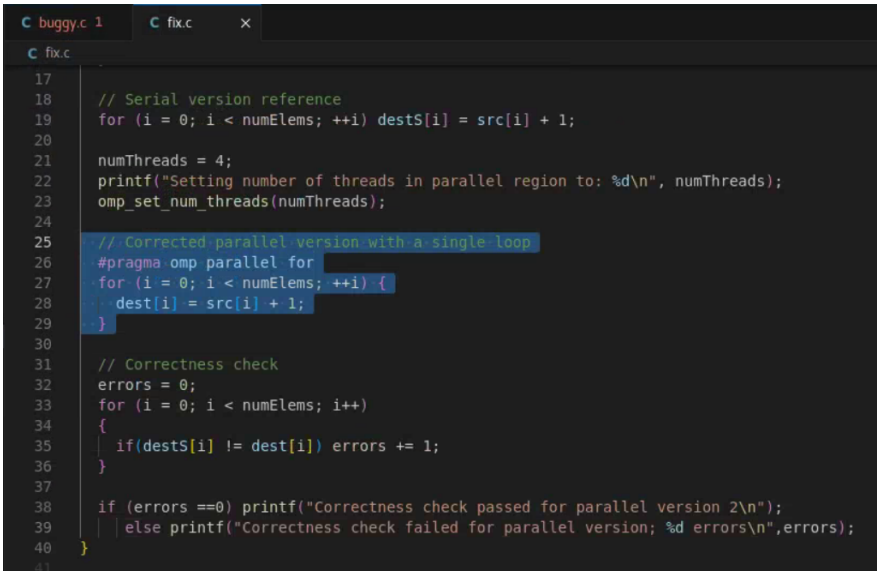

Figure 3: buggy.c corrected

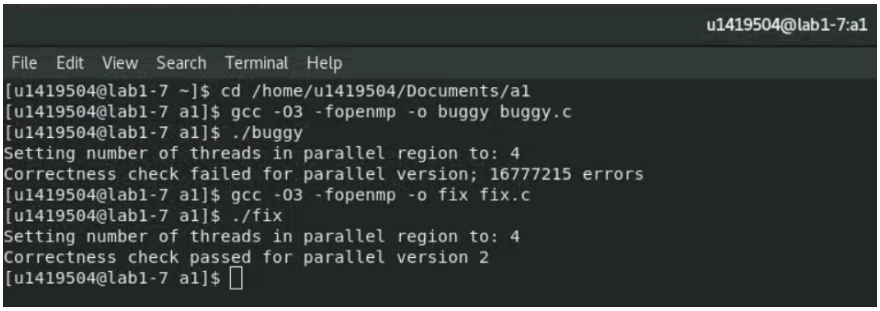

Figure 4: buggy.c corrected

- Original Code: The execution of the original code yielded $16,777,215$ errors in the correctness check.
- Fixed Code: After applying the fix, the correctness check passed with zero errors.

The comparison of the results "before" and "after" can be seen in Figure 4.

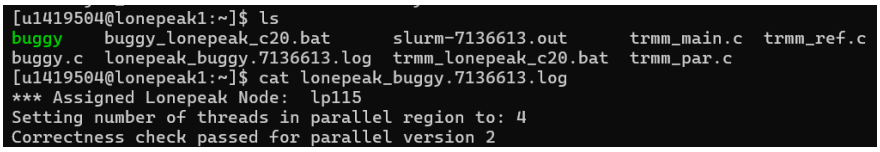

Figure 5: Lonepeak log results

Testing on Lonepeak CHPC Cluster: As per the assignment guidelines, further performance testing was conducted on the 20-core node of the Lonepeak CHPC cluster. A batch script was used to submit jobs for execution on the cluster. The job was

executed successfully on a dedicated node (assigned node: lp115), utilizing 20 cores as specified.

The output logs confirmed that the corrected parallel code passed the correctness check with zero errors, demonstrating consistent behavior across different multicore environments. The testing result from Lonepeak is shown in Figure 5, validating the scalability and correctness of the parallel implementation on a high-performance system (CADE).

2. The following code performs matrix multiplication of two upper-triangular matrices (stored as standard dense matrices with the lower triangular parts storing zero values) to produce the values for the result upper-triangular matrix.

```
for (i=0; i<600; i++)
    for (j=i; j<600; j++) // trmm
        for (k=i; k<=j; k++)
            C[i][j] += A[i][k]*B[k][j];
```

For each of the exercises below, use the files provided to compile the code. The file trmmref.c contains reference sequential code. The driver code in trmmmain.c invokes the sequential code in trmmref.c and performs multiple executions of the parallel version in trmmpar.c, using different numbers of threads, based on the number of cores on the multicore system. Performance results are reported for different thread counts, showing the best and worst performance (different executions of the same code result in run-to-run variation due to interference from other processes on the system that interleave execution with the tested code). A correctness check is also performed on the results by comparing the output of the parallel code version with the sequential code version.

(a) (15 points) Create and execute three parallel versions, one each by placing a work-sharing "pragma omp for" above each of the three loops. Provide a clear explanation for the differences you observe regarding the code's execution in the 3 cases.

Solution. The provided code was modified to create three versions:

- Outer Loop Parallelization (i loop) as seen in Figure 6.
- Middle Loop Parallelization (j loop) as seen in Figure 7.
- Inner Loop Parallelization (k loop) as seen in Figure 8.

C trmm_par_outer.c > ...

```c
void trmm_par(int N, float *__restrict__ a, float *__restrict__ b,
              float *__restrict__ c)
{
    int i, j, k;

    // Parallelize the outer loop directly with omp for
    #pragma omp parallel for private(j, k)
    for (i = 0; i < N; i++)
        for (j = i; j < N; j++)
            for (k = i; k <= j; k++)
                c[i * N + j] += a[i * N + k] * b[k * N + j];
}
```

Figure 6: Outer Loop Parallelization

Testing was conducted on both CADE and Lonepeak clusters to evaluate scalability and correctness. Initial tests were conducted on CADE Lab with up to 16 threads (CADE results can be seen in Figure 9), followed by final performance tests on the Lonepeak CHPC cluster using up to 40 threads.

Results and Analysis.

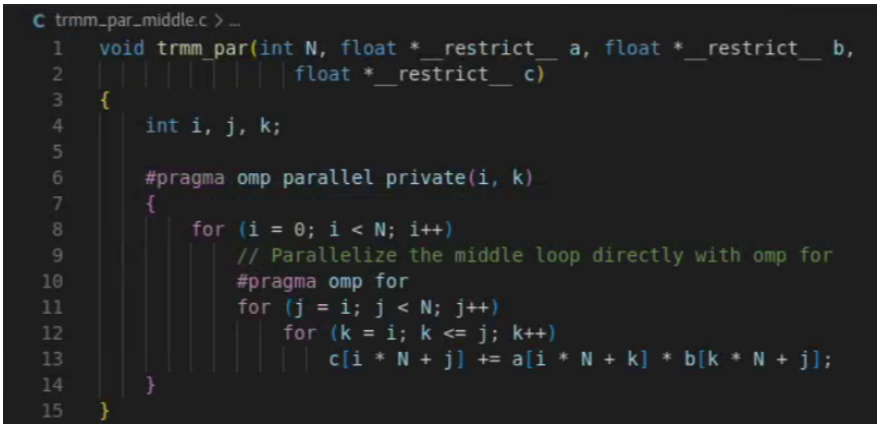

Figure 7: Middle Loop Parallelization

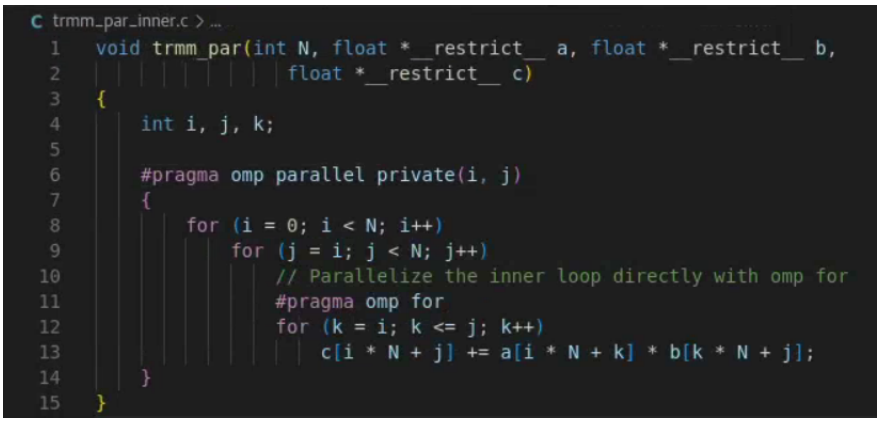

Figure 8: Inner Loop Parallelization

i. Outer Loop Parallelization
   - CADE Performance: Achieved a maximum GFLOPS of 2.31 and a speedup of 3.85 using 15 threads.
   - Lonepeak Performance: Best GFLOPS reached 16.54 with a speedup of 9.52 on 39 threads, indicating moderate scalability.
   - Explanation: This version provided reasonably large independent tasks, but the limited number of independent iterations in the outer loop eventually restricted scalability.

ii. Middle Loop Parallelization
   - CADE Performance: Reached a maximum GFLOPS of 2.79 with a speedup of 4.61 on 15 threads, outperforming the outer loop version.
   - Lonepeak Performance: Achieved the highest GFLOPS of 16.52 with a speedup of 9.54 using 39 threads.
   - Explanation: The middle loop had more independent iterations to distribute among threads, resulting in better load balancing and scalability.

iii. Inner Loop Parallelization

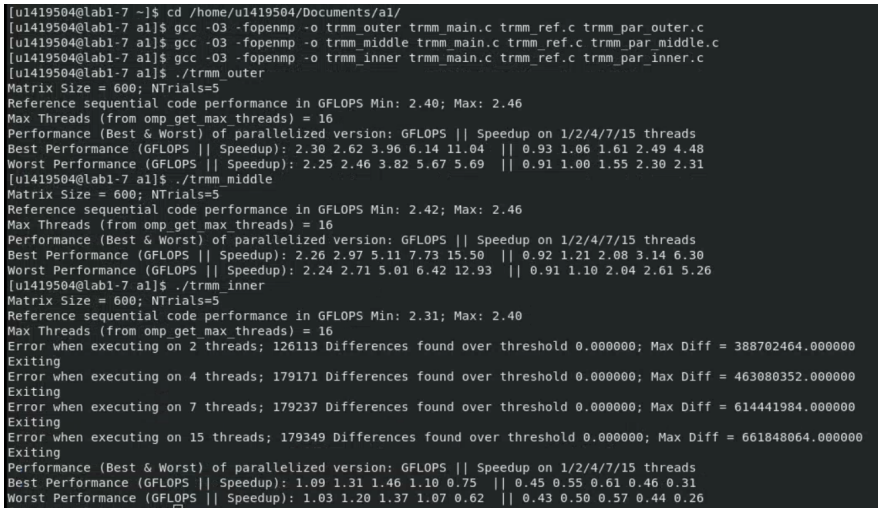

Figure 9: Results of 2a on CADE

- CADE Performance: Encountered significant errors and low GFLOPS values, indicating data inconsistencies due to parallelizing the innermost loop (k).
- Lonepeak Performance: Persisting errors with minimal performance gains showed large differences due to data race conditions.
- Explanation: Parallelizing the inner loop caused frequent updates to shared data elements, leading to race conditions and incorrect results.

Conclusion.

The middle loop parallelization strategy provided the best performance and scalability, while the outer loop strategy achieved moderate gains. Parallelizing the innermost loop resulted in incorrect results due to inherent data dependencies. These findings demonstrate the importance of choosing the appropriate loop to parallelize based on the nature of the task and the distribution of independent iterations.

(b) (15 points) For the parallel version corresponding to the "pragma omp for" on the outermost "i" loop, compare the maximum achieved GFLOPs performance for the following three variants, using **static** scheduling:

i) default chunk size;

ii) chunk size = 10;

iii) chunk size = 1.

What could explain the performance differences?

Solution. The provided code was modified to create three versions:

- Default Chunk Size (OpenMP's default behavior) as seen in Figure 10.
- Chunk Size = 10 as seen in Figure 11.
- Chunk Size = 1 as seen in Figure 12.

```c
C trmm_par_outer_chunk_def.c > trmm_par(int, float *__restrict__, float *__restrict__, float *__restrict__)
1  void trmm_par(int N, float *__restrict__ a, float *__restrict__ b,
2                float *__restrict__ c)
3  {
4      int i, j, k;
5
6      // Parallelize the outer loop directly with omp for
7      #pragma omp parallel for private(j, k) schedule(static)
8      for (i = 0; i < N; i++)
9          for (j = i; j < N; j++)
10             for (k = i; k <= j; k++)
11                 c[i * N + j] += a[i * N + k] * b[k * N + j];
12 }
```

Figure 10: Default Chunk Size (OpenMP's default behavior)

```c
C trmm_par_outer_chunk_10.c > trmm_par(int, float *__restrict__, float *__restrict__, float *__restrict__)
1  void trmm_par(int N, float *__restrict__ a, float *__restrict__ b,
2                float *__restrict__ c)
3  {
4      int i, j, k;
5
6      // Parallelize the outer loop directly with omp for
7      #pragma omp parallel for private(j, k) schedule(static, 10)
8      for (i = 0; i < N; i++)
9          for (j = i; j < N; j++)
10             for (k = i; k <= j; k++)
11                 c[i * N + j] += a[i * N + k] * b[k * N + j];
12 }
```

Figure 11: Chunk Size = 10

Testing was conducted on both CADE and Lonepeak clusters to evaluate scalability and correctness (CADE results can be seen in Figure 13).

Results and Analysis.

i. Performance on CADE:
   - Default Chunk Size: Achieved a peak performance of 10.89 GFLOPS with a maximum speedup of 4.43 on 15 threads.
   - Chunk Size = 10: Achieved a peak performance of 20.41 GFLOPS with a maximum speedup of 8.33 on 15 threads.
   - Chunk Size = 1: Achieved a peak performance of 27.19 GFLOPS with a maximum speedup of 11.04 on 15 threads.

ii. Performance on Lonepeak:
   - Default Chunk Size: Best GFLOPS of 16.56 and speedup of 9.46 on 39 threads.
   - Chunk Size = 10: Best GFLOPS of 20.69 and speedup of 11.91 on 39 threads.
   - Chunk Size = 1: Best GFLOPS of 34.75 and speedup of 19.92 on 39 threads.

Conclusion.

The results demonstrate that smaller chunk sizes provide better performance due to improved load balancing. However, the optimal chunk size depends on the

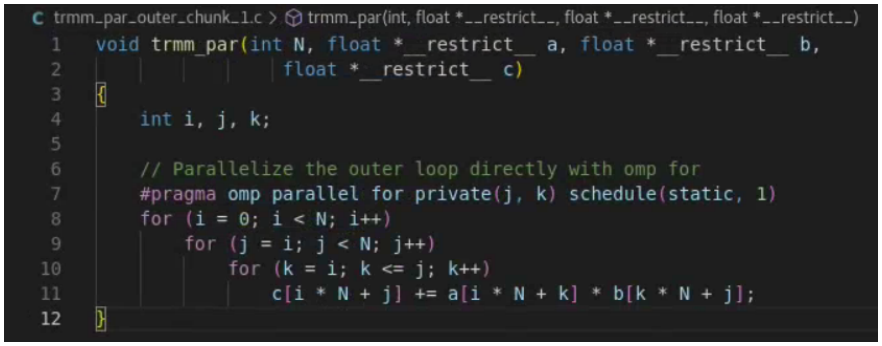

Figure 12: Chunk Size = 1

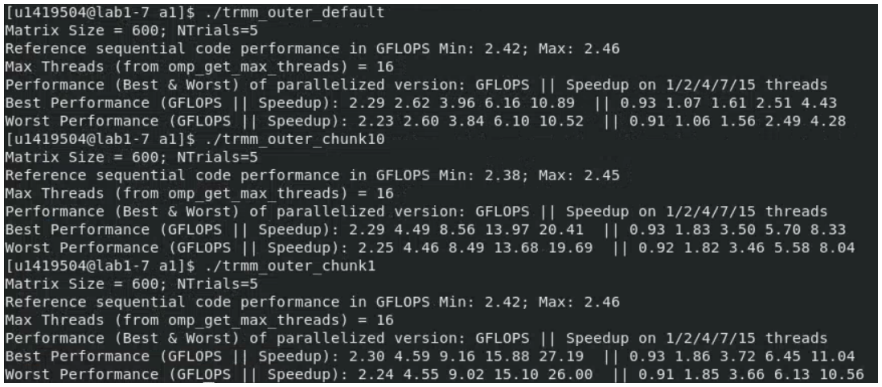

Figure 13: Results of 2b on CADE

trade-off between scheduling overhead and workload distribution. While chunk size = 1 yielded the best performance, chunk size = 10 offers a good compromise between overhead and scalability.

(c) (10 points) For the parallel version corresponding to the "pragma omp for" on the outermost "i" loop, measure the maximum achieved GFLOPs performance for the following two variants, using **dynamic** scheduling:

i) chunk size = 10;

ii) chunk size = 1.

Compare and contrast performance trends with those in part (b), attempting to explain the reason for the observed trend.

Solution. The provided code was modified to create dynamic scheduling on parallel performance using two different chunk sizes:

- Chunk Size = 10 as seen in Figure 14.
- Chunk Size = 1 as seen in Figure 15.

Testing was conducted on both CADE and Lonepeak clusters to evaluate performance (CADE results can be seen in Figure 16).

Results and Analysis.

i. Performance on CADE:

```c
// trmm_par_outer_chunk_10_dyn.c
void trmm_par(int N, float *__restrict__ a, float *__restrict__ b,
              float *__restrict__ c)
{
    int i, j, k;

    // Parallelize the outer loop with dynamic scheduling and chunk size 10
    #pragma omp parallel for private(j, k) schedule(dynamic, 10)
    for (i = 0; i < N; i++)
        for (j = i; j < N; j++)
            for (k = i; k <= j; k++)
                c[i * N + j] += a[i * N + k] * b[k * N + j];
}
```

Figure 14: Chunk Size = 10 (dynamic scheduling)

```c
// trmm_par_outer_chunk_1_dyn.c > trmm_par(int, float *__restrict__, float *__restrict__, float *__restrict__)
void trmm_par(int N, float *__restrict__ a, float *__restrict__ b,
              float *__restrict__ c)
{
    int i, j, k;

    // Parallelize the outer loop with dynamic scheduling and chunk size 1
    #pragma omp parallel for private(j, k) schedule(dynamic, 1)
    for (i = 0; i < N; i++)
        for (j = i; j < N; j++)
            for (k = i; k <= j; k++)
                c[i * N + j] += a[i * N + k] * b[k * N + j];
}
```

Figure 15: Chunk Size = 1 (dynamic scheduling)

- Chunk Size = 10: Achieved a peak performance of 28.27 GFLOPS with a maximum speedup of 11.50 on 15 threads.
- Chunk Size = 1: Achieved a peak performance of 29.98 GFLOPS with a maximum speedup of 12.14 on 15 threads.

ii. Performance on Lonepeak:

- Chunk Size = 10: Best GFLOPS of 29.79 with a speedup of 17.15 on 39 threads.
- Chunk Size = 1: Best GFLOPS of 41.44 with a speedup of 23.81 on 39 threads.

Comparison with Static Scheduling in Part 2b

- CADE: Dynamic scheduling with both chunk sizes showed better performance than static scheduling in Part 2b. The improvements are attributed to dynamic scheduling's ability to balance workload more effectively, reducing idle times.
- Lonepeak: Dynamic scheduling demonstrated significant gains in GFLOPS compared to static scheduling, particularly with chunk size 1, which achieved 41.44 GFLOPS. This improvement highlights dynamic scheduling's advantage in adapting to variations in workload distribution.

Conclusion.

Dynamic scheduling outperforms static scheduling in this scenario by effectively adapting to variations in workload across threads. The finer chunk size of 1 yielded

```
[u1419504@lab1-7 a1]$ gcc -O3 -fopenmp -o trmm_10_dyn trmm_main.c trmm_ref.c trmm_par_outer_chunk_10_dyn.c
[u1419504@lab1-7 a1]$ gcc -O3 -fopenmp -o trmm_1_dyn trmm_main.c trmm_ref.c trmm_par_outer_chunk_1_dyn.c
[u1419504@lab1-7 a1]$ ./trmm_10_dyn
Matrix Size = 600; NTrials=5
Reference sequential code performance in GFLOPS Min: 2.37; Max: 2.46
Max Threads (from omp_get_max_threads) = 16
Performance (Best & Worst) of parallelized version: GFLOPS || Speedup on 1/2/4/7/15 threads
Best Performance (GFLOPS || Speedup): 2.30 4.60 9.21 16.12 28.27  || 0.93 1.87 3.75 6.55 11.50
Worst Performance (GFLOPS || Speedup): 2.26 4.57 9.09 15.58 26.19  || 0.92 1.86 3.70 6.34 10.65
[u1419504@lab1-7 a1]$ ./trmm_1_dyn
Matrix Size = 600; NTrials=5
Reference sequential code performance in GFLOPS Min: 2.39; Max: 2.47
Max Threads (from omp_get_max_threads) = 16
Performance (Best & Worst) of parallelized version: GFLOPS || Speedup on 1/2/4/7/15 threads
Best Performance (GFLOPS || Speedup): 2.30 4.60 9.22 16.11 29.98  || 0.93 1.86 3.73 6.52 12.14
Worst Performance (GFLOPS || Speedup): 2.27 4.57 9.09 15.52 28.35  || 0.92 1.85 3.68 6.29 11.48
```

Figure 16: Results of 2c on CADE

the highest GFLOPS and speedup, due to superior load balancing. However, chunk size 10 remains a viable option, offering a good balance between scheduling overhead and performance scalability.

(d) (15 points) Repeat part (c) for a parallel version with "pragma omp for" on the middle "j" loop. Compare and contrast the trends in part (c) with that observed here. What could explain the different trends with chunk size?

Solution. The provided code was modified to create dynamic scheduling on parallel version on the middle "j" loop using two different chunk sizes:

- Chunk Size = 10 as seen in Figure 17.
- Chunk Size = 1 as seen in Figure 18.

```c
C trmm_par_outer_chunk_10_dyn_j.c > ...
1   void trmm_par(int N, float *__restrict__ a, float *__restrict__ b,
2                 float *__restrict__ c)
3   {
4       int i, j, k;
5
6       #pragma omp parallel private(i, k)
7       {
8           for (i = 0; i < N; i++)
9               // Parallelize the middle loop with dynamic scheduling and chunk size 10
10              #pragma omp for schedule(dynamic, 10)
11              for (j = i; j < N; j++)
12                  for (k = i; k <= j; k++)
13                      c[i * N + j] += a[i * N + k] * b[k * N + j];
14      }
15  }
```

Figure 17: Chunk Size = 10 (dynamic scheduling on j loop)

Testing was conducted on both CADE and Lonepeak clusters to evaluate performance (CADE results can be seen in Figure 19).

Results and Analysis.

i. Performance on CADE:
   - Chunk Size = 10: Achieved a peak performance of 17.12 GFLOPS with a maximum speedup of 6.93 on 15 threads.
   - Chunk Size = 1: Achieved a peak performance of 9.84 GFLOPS with a maximum speedup of 3.99 on 15 threads.

ii. Performance on Lonepeak:

```c
C trmm_par_outer_chunk_1_dyn_j.c > trmm_par(int, float *__restrict__, float *__restrict__, float *__restrict__)
1  void trmm_par(int N, float *__restrict__ a, float *__restrict__ b,
2                float *__restrict__ c)
3  {
4      int i, j, k;
5
6      #pragma omp parallel private(i, k)
7      {
8          for (i = 0; i < N; i++)
9              // Parallelize the middle loop with dynamic scheduling and chunk size 1
10             #pragma omp for schedule(dynamic, 1)
11             for (j = i; j < N; j++)
12                 for (k = i; k <= j; k++)
13                     c[i * N + j] += a[i * N + k] * b[k * N + j];
14     }
15 }
```

Figure 18: Chunk Size = 1 (dynamic scheduling on j loop)

```
[u1419504@lab1-7 a1]$ gcc -O3 -fopenmp -o trmm_j_10 trmm_main.c trmm_ref.c trmm_par_outer_chunk_10_dyn_j.c
[u1419504@lab1-7 a1]$ gcc -O3 -fopenmp -o trmm_j_1 trmm_main.c trmm_ref.c trmm_par_outer_chunk_1_dyn_j.c
[u1419504@lab1-7 a1]$ ./trmm_j_10
Matrix Size = 600; NTrials=5
Reference sequential code performance in GFLOPS Min: 2.43; Max: 2.47
Max Threads (from omp_get_max_threads) = 16
Performance (Best & Worst) of parallelized version: GFLOPS || Speedup on 1/2/4/7/15 threads
Best Performance (GFLOPS || Speedup): 2.26 4.11 7.28 11.39 17.12  || 0.92 1.66 2.95 4.61 6.93
Worst Performance (GFLOPS || Speedup): 2.22 4.05 7.17 11.14 16.34  || 0.90 1.64 2.90 4.51 6.62
[u1419504@lab1-7 a1]$ ./trmm_j_1
Matrix Size = 600; NTrials=5
Reference sequential code performance in GFLOPS Min: 2.39; Max: 2.47
Max Threads (from omp_get_max_threads) = 16
Performance (Best & Worst) of parallelized version: GFLOPS || Speedup on 1/2/4/7/15 threads
Best Performance (GFLOPS || Speedup): 2.09 3.42 5.69 7.28 9.84  || 0.85 1.39 2.31 2.95 3.99
Worst Performance (GFLOPS || Speedup): 2.06 3.37 5.55 7.12 9.38  || 0.84 1.37 2.25 2.89 3.80
```

Figure 19: Results of 2d on CADE

- Chunk Size = 10: Best GFLOPS of 9.70 with a speedup of 5.57 on 39 threads.
- Chunk Size = 1: Best GFLOPS of 3.98 with a speedup of 2.29 on 39 threads.

Comparison with Dynamic Scheduling on the Outer Loop (Part 2c)

i. Performance Drop: The peak GFLOPS observed with middle loop parallelization is significantly lower compared to outer loop parallelization. This trend is consistent across both chunk sizes (10 and 1) on both CADE and Lonepeak.

ii. Effect of Chunk Size:
   - For Chunk Size = 10, dynamic scheduling of the middle loop shows moderate performance but struggles to scale efficiently beyond a limited number of threads. This is due to increased dependencies and less parallel work available in the inner loops.
   - For Chunk Size = 1, there is a significant drop in GFLOPS and speedup, highlighting inefficiencies caused by increased scheduling overhead relative to the available parallel work in the inner loops.

Conclusion.

The results demonstrate that parallelizing the middle loop results in lower GFLOPS and speedup compared to parallelizing the outer loop. This discrepancy arises due to the reduced parallel work and increased scheduling overhead in the middle loop, particularly when using smaller chunk sizes. Dynamic scheduling with

chunk size 10 performed better than chunk size 1, but neither outperformed the results achieved by parallelizing the outer loop.

(e) (20 points) Perform loop permutation guided by stride analysis to enhance data locality. Attempt to maximize parallel performance by suitable choice of scheduling directives for the transformed code. Target performance: 100 GFLOPs on the target system (20-core node on Lonepeak cluster).

Solution.

Stride Analysis for Each Loop.

i. Innermost Loop (k-loop):
   - Access to a[i * N + k]: As k changes, the access pattern in a is contiguous along the row, so the stride is 1.
   - Access to b[k * N + j]: As k changes, the stride jumps between different rows of b with a stride of N elements (row-major order).
   - Access to c[i * N + j]: The element in c remains the same as k iterates. Hence, no additional memory movement occurs for c within this loop.

ii. Middle Loop (j-loop):
   - Access to a[i * N + k]: The value of i remains the same, so only the value of k changes within this scope. If k also remains constant, the accesses are not impacted by j, so the stride is small.
   - Access to b[k * N + j]: As j changes, it accesses contiguous elements in the row of b, resulting in a stride of 1 within this scope.
   - Access to c[i * N + j]: As j changes, this accesses contiguous elements in c within a row, leading to a stride of 1.

iii. Outermost Loop (i-loop):
   - Access to a[i * N + k]: As i changes, the access jumps between different rows of a, resulting in a stride of N elements.
   - Access to b[k * N + j]: The k and j remain constant, so the accesses in b are not affected within this loop.
   - Access to c[i * N + j]: As i changes, the accesses to c jump between different rows, resulting in a stride of N.

Summary of Strides:

- Innermost Loop (k-loop): Contiguous access in a, larger stride in b.
- Middle Loop (j-loop): Contiguous access in b and c.
- Outermost Loop (i-loop): Larger strides in a and c.

Evaluating Each Loop Permutation.

We have three loops: i, j, and k. Let's consider each possible permutation:

i. Loop Order: (i, j, k):
   - Pros: Contiguous accesses in c for j, minimizing strides.
   - Cons: Non-contiguous accesses in a and b, causing more cache misses.

ii. Loop Order: (i, k, j):

- Pros: Contiguous access in a when iterating over k.
- Cons: Non-contiguous accesses in b, causing larger strides in b.

iii. Loop Order: (j, i, k):
- Pros: Contiguous access in c and b for j.
- Cons: Non-contiguous access in a when iterating over i.

iv. Loop Order: (j, k, i):
- Pros: Contiguous access in a and c for k.
- Cons: Possible non-contiguous access patterns in b when iterating over k.

v. Loop Order: (k, i, j):
- Pros: Contiguous access in a for k.
- Cons: Non-contiguous access in b and c for i and j.

vi. Loop Order: (k, j, i):
- Pros: Contiguous access in b for j.
- Cons: Non-contiguous access in a and c for i.

Based on our stride analysis, the most promising loop orders appear to be:

- (j, k, i): This order achieves contiguous access in a and b for iterations over k, with minimal strides for c. It maintains relatively good cache locality for all matrices and provides ample parallelism in j.
- (k, j, i): This order allows for contiguous access in b, but at the cost of introducing non-contiguous accesses in c.

Given the stride analysis, the optimal choice appears to be (j, k, i), since this order maximizes contiguous memory access patterns in matrices a and b and offers sufficient opportunities for parallelization over j.

After trying and failing many different versions of scheduling, we concluded that the best one is guided (which is opposite to the previous results that stated that dynamic scheduling was the best) as well the fact that chunk size should not be specified. Trying chunk sizing like 5, 10, 20, 30, and 40 with guided scheduling returned worse results than automatic. However, neither of the methods of scheduling and their chunk sizing returned the most of ≈ 33 GFLOPs.

Therefore, the target performance was not achieved, but the files that have been tried with will be attached.

(f) (25 points) Extra Credit: Use loop unrolling on the version produced in part (e) to achieve performance of 150 GFLOPs on target Lonepeak 20-core node. Not all loops produce a benefit from unrolling. Why?

Solution.

Selective Unrolling: Unrolling every loop does not necessarily lead to performance gains. In matrix operations like trmm, the innermost k loop often benefits the most from unrolling due to its high iteration frequency and proximity to the core calculations. Unrolling this loop effectively reduces overhead from repeated loop control instructions (like branching and counter updates) and enhances data

reuse in cache, both of which are crucial for performance on compute-intensive operations.

Innermost Loop Benefits: By unrolling the k loop, the computation inside each unrolled segment is contiguous, which better aligns with the CPU's data fetching and cache prefetching mechanisms. For example, fetching blocks of b[k * N + j] sequentially can reduce cache misses and improve data locality, allowing the CPU to execute these operations with less latency.

Guided Scheduling: For optimal performance on a multi-core system like Lonepeak's 20-core node, guided scheduling on the j loop helps distribute work across threads dynamically. This approach manages load balancing effectively, allowing threads to handle variable workloads based on their completion times, which is especially useful for loops with varying iteration counts.

Why Other Loops Don't Benefit: Unrolling outer loops like j or i can lead to increased complexity in workload balancing across threads, especially with guided scheduling. These loops generally have fewer iterations than the innermost loop, so unrolling them can cause cache thrashing or increased memory traffic without proportional gains in performance. Additionally, the larger chunks of work from unrolling outer loops may lead to load imbalance, as some threads could finish their chunks sooner, leaving others to catch up.

Through guided scheduling and selective unrolling of the k loop, this approach maximizes GFLOPs by balancing workload distribution and optimizing innermost loop efficiency on the targeted 20-core architecture.