HW3: Parallelizing Strassen's Matrix-Multiplication Algorithm via OpenMP

Tyler Scotti

Spring 2026

1 Parallelization Approach

The serial Strassen's algorithm was parallelized using OpenMP tasks. At each recursive level, the matrix products M_1 through M_7 are spawned in as separate tasks. This allows for idle threads to pick them up. A `taskwait` directive synchronizes before the results are combined into the output blocks C_{11}, C_{12}, C_{21}, C_{22}.

The parallel section is created once in the main function with `#pragma omp parallel`, and a `#pragma omp single` directive ensures only one thread enters the top-level call. Tasks are generated recursively, with an `if` clause that stops creating new tasks when the subproblem size is close to the leaf size, avoiding excessive task overhead.

A copy constructor and copy assignment operator were needed, thus I added to the `Matrix` class to ensure correct behavior when OpenMP tasks copy matrix objects across threads.

2 Experimental Setup

All experiments were ran on a single node on Grace. The code was compiled with:

```
module load intel
icpx -O2 -qopenmp -o strassen_omp.exe strassen_omp.cpp
```

Jobs were submitted via a batch file such that my timings were collected in dedicated mode. Speedup is defined as $S_p = T_1/T_p$ where T_1 is the single-thread Strassen time and T_p is the p-thread time. Efficiency is $E_p = S_p/p$.

3 Results

3.1 Effect of Leaf Matrix Size

For each matrix size, the smallest leaf size (2^q with the smallest q) consistently gave the fastest times. This is because smaller leaves create more tasks at the bottom of the recursion tree, exposing more parallelism to OpenMP. The tradeoff is more overhead from task creation, but for matrices of size 1024 and above, the parallelism benefit outweighs the overhead.

Table 1: Strassen's execution time (seconds) for all configurations.

k	n	Leaf	T_1	T_2	T_4	T_8	T_{16}	T_{48}
10	1024	16	0.717	0.453	0.274	0.181	0.136	0.142
10	1024	32	0.659	0.392	0.237	0.148	0.129	0.169
10	1024	64	0.709	0.426	0.253	0.163	0.124	0.161
11	2048	16	5.091	3.047	1.732	1.010	0.660	0.543
11	2048	32	4.341	2.551	1.466	0.842	0.561	0.466
11	2048	64	5.037	2.841	1.618	0.893	0.580	0.504
12	4096	32	31.24	17.46	9.468	5.548	3.181	2.028
12	4096	64	35.74	19.89	10.60	6.063	3.360	2.145
12	4096	128	40.64	21.18	11.64	6.385	3.960	2.280
13	8192	32	225.1	118.2	62.49	34.63	19.74	11.18
13	8192	64	257.4	131.7	69.87	38.51	22.49	11.69
13	8192	128	283.7	147.2	78.68	41.86	23.89	12.20
14	16384	64	1805.8	—	—	—	—	80.89
14	16384	128	2005.1	—	—	—	—	87.36

3.2 Speedup and Efficiency (Best Leaf Size per k)

Using the best leaf size for each matrix size:

Table 2: Speedup and efficiency using 48 threads with the best leaf size.

k	n	Best Leaf	T_1 (s)	T_{48} (s)	Speedup (S_{48})
10	1024	32	0.659	0.169	3.91×
11	2048	32	4.341	0.466	9.31×
12	4096	32	31.24	2.028	15.41×
13	8192	32	225.1	11.18	20.14×
14	16384	64	1805.8	80.89	22.33×

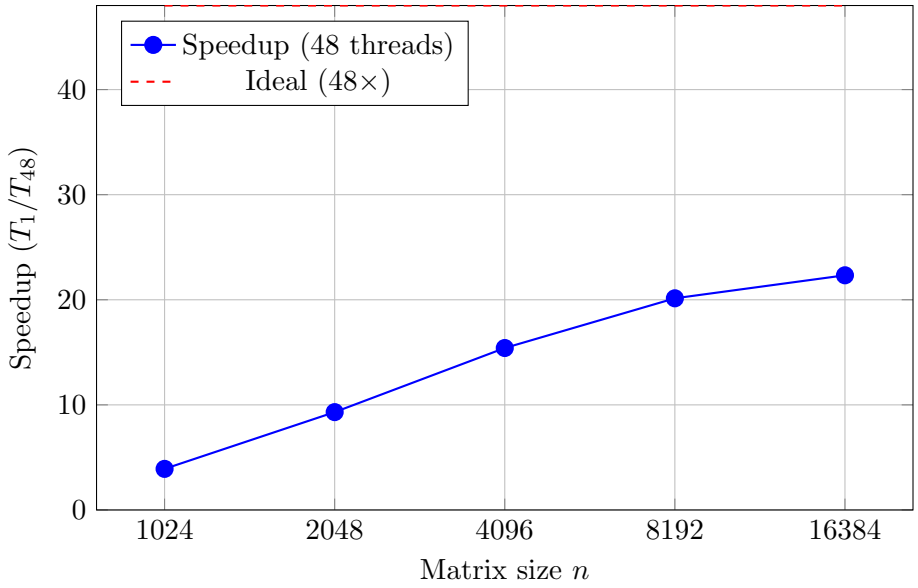

Figure 1: Speedup vs. matrix size using 48 threads.

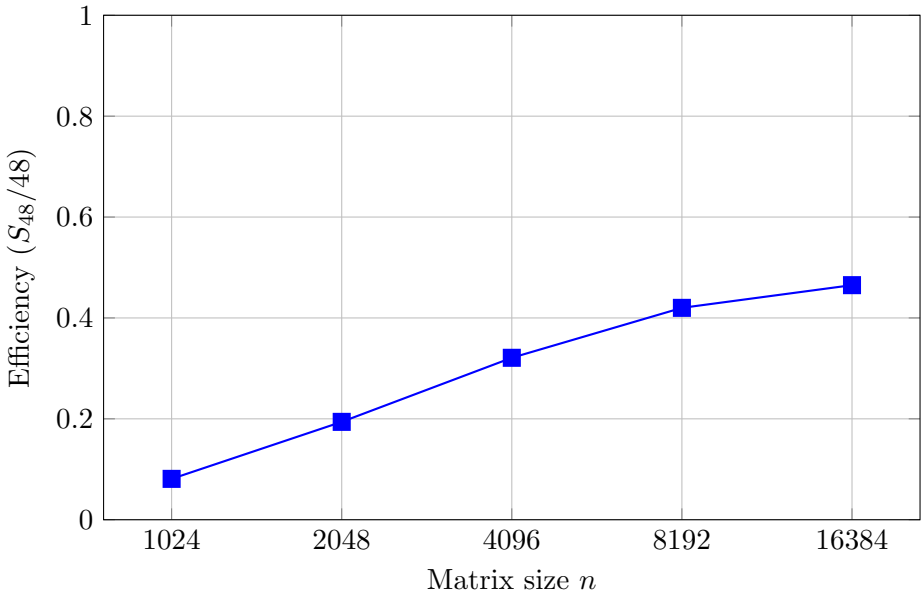

Figure 2: Efficiency vs. matrix size using 48 threads.

3.3 Speedup vs. Number of Threads ($k = 13$, Leaf $= 32$)

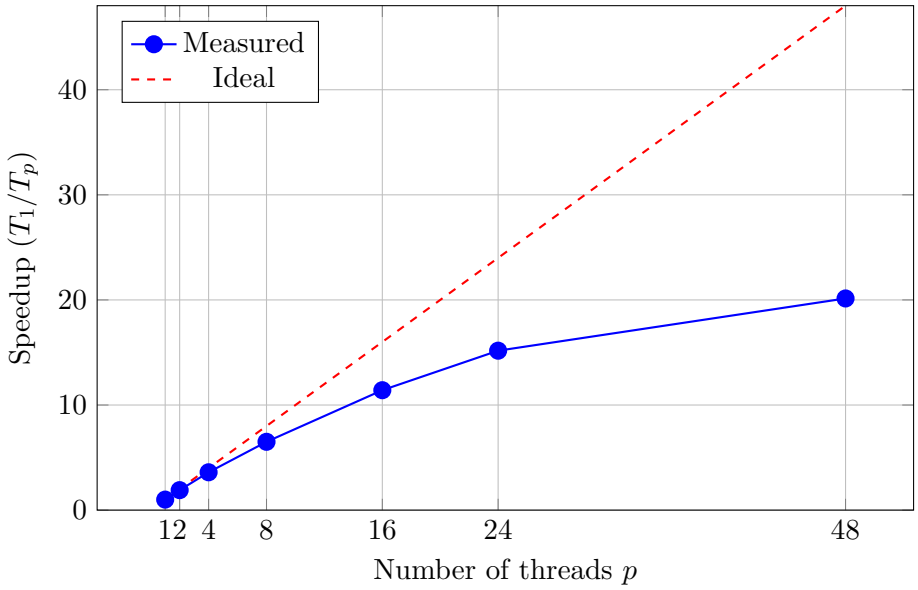

Figure 3: Speedup vs. number of threads for $k = 13$, leaf size $= 32$.

4 Insights

- **Speedup improves with problem size.** At $k = 10$ ($n = 1024$), the problem is too small for 48 threads — the task creation overhead dominates and speedup is only 3.9×. By $k = 14$ ($n = 16384$), speedup reaches 22.3× because there is enough work to keep threads busy.
- **Smaller leaf sizes give better speedup.** Leaf size 32 was consistently the best for $k = 10$–13. Smaller leaves mean more levels of recursion before hitting the base case, which creates more independent tasks for OpenMP to distribute. However, extremely small leaves (e.g., $2^4 = 16$) add overhead from extra allocations and copies.
- **Efficiency is below 50% even at the largest size.** The maximum efficiency observed was 46.5% at $k = 14$. This is partly because Strassen's recursion creates 7 tasks per level (not a power of 2), so work cannot be evenly divided among 48 threads. Memory allocation overhead and cache effects also contribute.
- **Diminishing returns beyond 24 threads.** For $k = 10$ and $k = 11$, performance actually degrades going from 24 to 48 threads due to overhead. For larger problems, the jump from 24 to 48 threads still helps but with reduced marginal benefit.

5 How to Compile and Execute on Grace

```
# Log in to Grace
ssh <NetID>@grace.hprc.tamu.edu

# Load the Intel compiler
module load intel
```

```
# Compile
icpx -O2 -qopenmp -o strassen_omp.exe strassen_omp.cpp

# Run the batch file
sbatch strassen_omp.grace_job
```