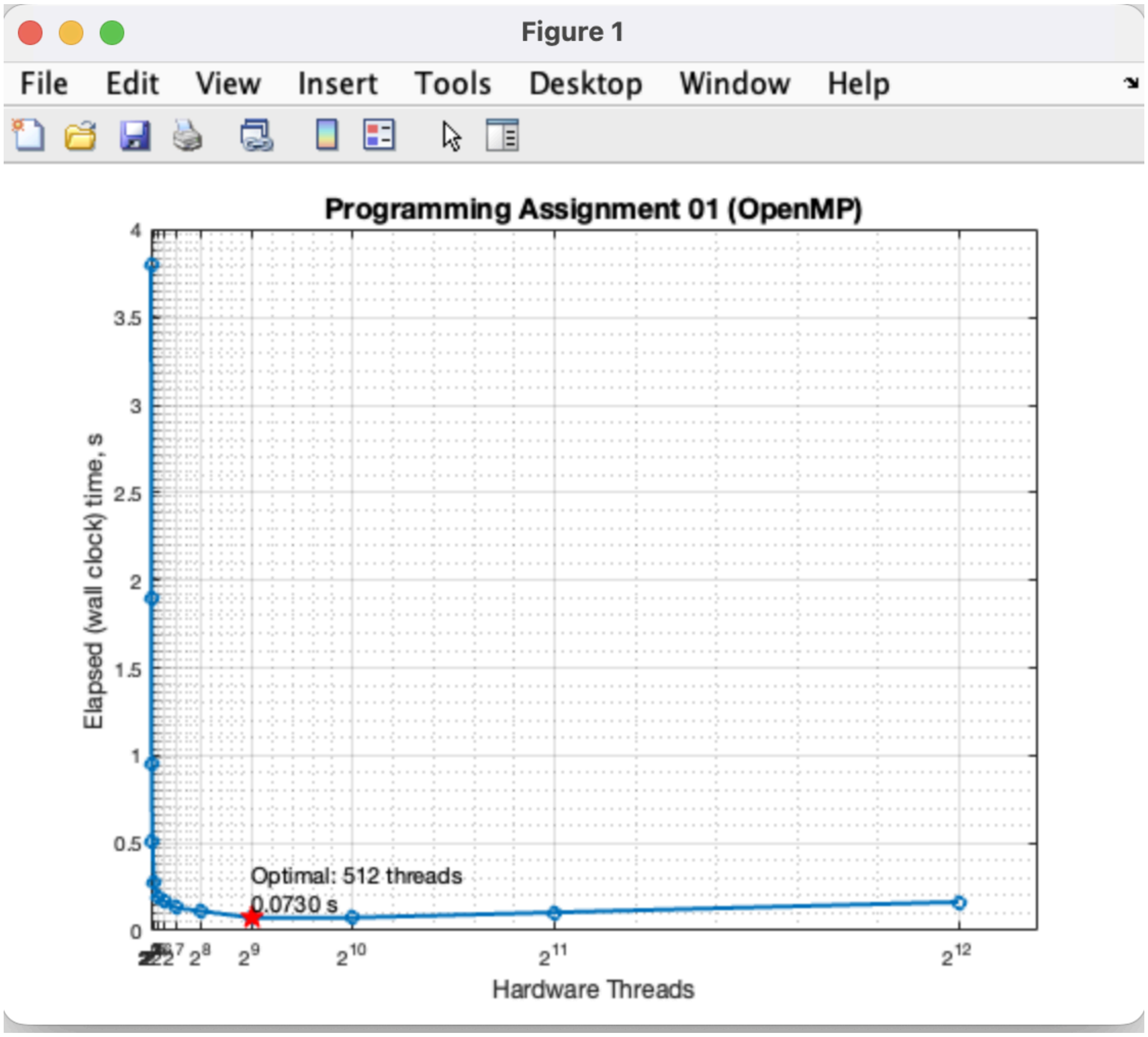
Brandon Nguyen, 827813045
COMPE596 - Spring 2026
1/30/2026

COMPE596 P01 - OpenMP Double Precision General Matrix Multiply

1) Runtime Plot



2) Code Listing

// Serial Program

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>
#include <math.h>

#define ORDER 1000
#define AVAL 3.0
#define BVAL 5.0
#define TOL 1.1102230246251568E-16
#define UNROLL_FACTOR 4

int main(int argc, char *argv[])
{
    int Ndim, Pdim, Mdim;
    int i, j, k;
    double *A, *B, *C;
    double tmp, err, errsq, cval;
    double run_time;
    double dN, mflops;
    FILE *fp;
    clock_t t_start, t_end;

    Ndim = ORDER;
    Pdim = ORDER;
    Mdim = ORDER;

    A = (double *)malloc(Ndim * Pdim * sizeof(double));
    B = (double *)malloc(Pdim * Mdim * sizeof(double));
    C = (double *)malloc(Ndim * Mdim * sizeof(double));

    if (A == NULL || B == NULL || C == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    for(i = 0; i < Ndim; i++)
        for(j = 0; j < Pdim; j++)
            A[i + j * Ndim] = AVAL;

    for(i = 0; i < Pdim; i++)
        for(j = 0; j < Mdim; j++)
            B[i + j * Pdim] = BVAL;
```

```
for(i = 0; i < Ndim; i++)
    for(j = 0; j < Mdim; j++)
        C[i + j * Ndim] = 0.0;

printf("Serial computation:\n");
t_start = clock();

for(i = 0; i < Ndim; i++) {
    for(j = 0; j < Mdim; j++) {
        tmp = 0.0;

        for(k = 0; k < Pdim - UNROLL_FACTOR + 1; k += UNROLL_FACTOR) {
            tmp += A[i + k * Ndim] * B[k + j * Pdim];
            tmp += A[i + (k+1) * Ndim] * B[(k+1) + j * Pdim];
            tmp += A[i + (k+2) * Ndim] * B[(k+2) + j * Pdim];
            tmp += A[i + (k+3) * Ndim] * B[(k+3) + j * Pdim];
        }

        for(; k < Pdim; k++) {
            tmp += A[i + k * Ndim] * B[k + j * Pdim];
        }
        C[i + j * Ndim] = tmp;
    }
}

t_end = clock();
run_time = (double)(t_end - t_start) / CLOCKS_PER_SEC;
dN = (double)ORDER;
mflops = 2.0 * dN * dN * dN / (run_time * 1.0e6);

printf("  Time: %f seconds\n", run_time);
printf("  Performance: %f mflops\n", mflops);

cval = Pdim * AVAL * BVAL;
errsq = 0.0;
for(i = 0; i < Ndim; i++) {
    for(j = 0; j < Mdim; j++) {
        err = C[i + j * Ndim] - cval;
        errsq += err * err;
        }
    }

    if (errsq > TOL) {
        printf("  Error in dgemm: %e\n", errsq);
    } else {
        printf("  dgemm passed\n");
    }

    printf("MATLAB_DATA_SERIAL: %f %f\n", run_time, mflops);

    fp = fopen("data.txt", "w");
    if (fp == NULL) {
        fprintf(stderr, "Error opening data.txt for writing\n");
    } else {
        fprintf(fp, "1 %f\n", run_time);
        fclose(fp);
    }

    free(A);
    free(B);
    free(C);

    return 0;
}
```

// Parallel Program

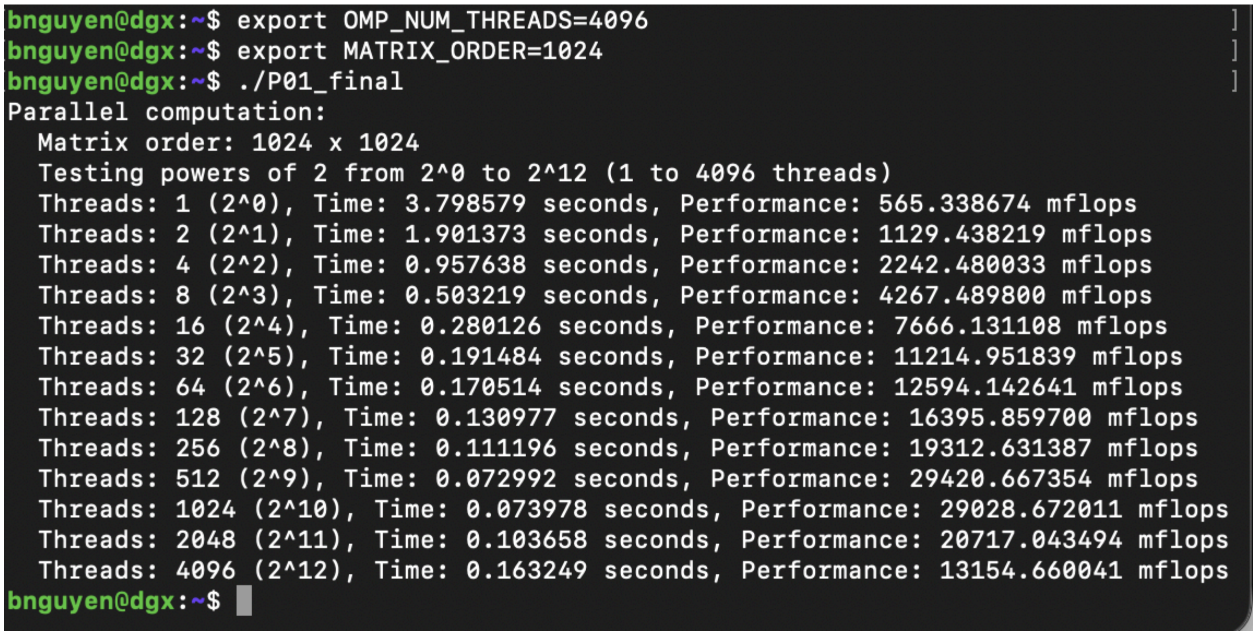
```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define AVAL 3.0
#define BVAL 5.0
#define TOL 1.1102230246251568E-16
#define UNROLL_FACTOR 4
#define DEFAULT_ORDER 1000

int main(int argc, char *argv[])
```

```
{
    int Ndim, Pdim, Mdim;
    int i, j, k;
    double *A, *B, *C;
    double tmp, err, errsq, cval;
    double start_time, run_time;
    double dN, mflops;
    int num_threads;
    int order;
    FILE *fp;
    char *env_threads;
    char *env_order;

    env_order = getenv("MATRIX_ORDER");
    if (env_order == NULL) {
        env_order = getenv("ORDER");
    }
    if (env_order == NULL) {
        order = DEFAULT_ORDER;
        fprintf(stderr, "Warning: MATRIX_ORDER or ORDER not set, using %d\n",
DEFAULT_ORDER);
    } else {
        order = atoi(env_order);
        if (order < 1) {
            order = DEFAULT_ORDER;
            fprintf(stderr, "Warning: Invalid matrix order, using %d\n", DEFAULT_ORDER);
        }
    }

    Ndim = order;
    Pdim = order;
    Mdim = order;

    env_threads = getenv("OMP_NUM_THREADS");
    if (env_threads == NULL) {
        env_threads = getenv("NUM_THREADS");
    }
    if (env_threads == NULL) {
        num_threads = 1;
        fprintf(stderr, "Warning: OMP_NUM_THREADS or NUM_THREADS not set, using
1 thread\n");
    } else {
        num_threads = atoi(env_threads);
        if (num_threads < 1) {
            num_threads = 1;
            fprintf(stderr, "Warning: Invalid thread count, using 1 thread\n");
        }
    }


    int max_power = 0;
    int temp = num_threads;
    while (temp > 1) {
        temp >>= 1;
        max_power++;
    }

    A = (double *)malloc(Ndim * Pdim * sizeof(double));
    B = (double *)malloc(Pdim * Mdim * sizeof(double));
    C = (double *)malloc(Ndim * Mdim * sizeof(double));

    if (A == NULL || B == NULL || C == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    for(i = 0; i < Ndim; i++)
        for(j = 0; j < Pdim; j++)
            A[i + j * Ndim] = AVAL;

    for(i = 0; i < Pdim; i++)
        for(j = 0; j < Mdim; j++)
            B[i + j * Pdim] = BVAL;

    printf("Parallel computation:\n");
    printf("  Matrix order: %d x %d\n", order, order);
    printf("  Testing powers of 2 from 2^0 to 2^%d (1 to %d threads)\n", max_power,
num_threads);
```

```
fp = fopen("data.txt", "w");
if (fp == NULL) {
    fprintf(stderr, "Error opening data.txt for writing\n");
}

int thread_power;
int current_threads;
for(thread_power = 0; thread_power <= max_power; thread_power++) {
    current_threads = 1 << thread_power;

    for(i = 0; i < Ndim; i++)
        for(j = 0; j < Mdim; j++)
            C[i + j * Ndim] = 0.0;

    omp_set_num_threads(current_threads);

    start_time = omp_get_wtime();


    #pragma omp parallel for private(tmp, i, j, k)
    for(i = 0; i < Ndim; i++) {
        for(j = 0; j < Mdim; j++) {
            tmp = 0.0;

            for(k = 0; k < Pdim - UNROLL_FACTOR + 1; k += UNROLL_FACTOR) {
                tmp += A[i + k * Ndim] * B[k + j * Pdim];
                tmp += A[i + (k+1) * Ndim] * B[(k+1) + j * Pdim];
                tmp += A[i + (k+2) * Ndim] * B[(k+2) + j * Pdim];
                tmp += A[i + (k+3) * Ndim] * B[(k+3) + j * Pdim];
            }

            for(; k < Pdim; k++) {
                tmp += A[i + k * Ndim] * B[k + j * Pdim];
            }
            C[i + j * Ndim] = tmp;
        }
    }

    run_time = omp_get_wtime() - start_time;
    dN = (double)order;
```

```
        mflops = 2.0 * dN * dN * dN / (run_time * 1.0e6);

        printf("  Threads: %d (2^%d), Time: %f seconds, Performance: %f mflops\n",
               current_threads, thread_power, run_time, mflops);

        cval = Pdim * AVAL * BVAL;
        errsq = 0.0;
        for(i = 0; i < Ndim; i++) {
            for(j = 0; j < Mdim; j++) {
                err = C[i + j * Ndim] - cval;
                errsq += err * err;
            }
        }

        if (errsq > TOL) {
            printf("    Error in dgemm: %e\n", errsq);
        }


        if (fp != NULL) {
            fprintf(fp, "%d %f\n", current_threads, run_time);
        }
    }

    if (fp != NULL) {
        fclose(fp);
    }

    free(A);
    free(B);
    free(C);

    return 0;
}
```

3) Program Output



```
bnguyen@dgx:~$ export OMP_NUM_THREADS=4096
bnguyen@dgx:~$ export MATRIX_ORDER=1024
bnguyen@dgx:~$ ./P01_final
Parallel computation:
  Matrix order: 1024 x 1024
  Testing powers of 2 from 2^0 to 2^12 (1 to 4096 threads)
  Threads: 1 (2^0), Time: 3.798579 seconds, Performance: 565.338674 mflops
  Threads: 2 (2^1), Time: 1.901373 seconds, Performance: 1129.438219 mflops
  Threads: 4 (2^2), Time: 0.957638 seconds, Performance: 2242.480033 mflops
  Threads: 8 (2^3), Time: 0.503219 seconds, Performance: 4267.489800 mflops
  Threads: 16 (2^4), Time: 0.280126 seconds, Performance: 7666.131108 mflops
  Threads: 32 (2^5), Time: 0.191484 seconds, Performance: 11214.951839 mflops
  Threads: 64 (2^6), Time: 0.170514 seconds, Performance: 12594.142641 mflops
  Threads: 128 (2^7), Time: 0.130977 seconds, Performance: 16395.859700 mflops
  Threads: 256 (2^8), Time: 0.111196 seconds, Performance: 19312.631387 mflops
  Threads: 512 (2^9), Time: 0.072992 seconds, Performance: 29420.667354 mflops
  Threads: 1024 (2^10), Time: 0.073978 seconds, Performance: 29028.672011 mflops
  Threads: 2048 (2^11), Time: 0.103658 seconds, Performance: 20717.043494 mflops
  Threads: 4096 (2^12), Time: 0.163249 seconds, Performance: 13154.660041 mflops
bnguyen@dgx:~$
```

4) MATLAB Plot Script

```
close all
clc
data = load('data.txt');
threads = data(:,1);
times = data(:,2);
figure;
plot(threads, times, 'o-', 'LineWidth', 2);
set(gca, 'XTick', threads);
set(gca, 'XTickLabel', ...
   arrayfun(@(x) sprintf('2^{%d}', log2(x)), threads, 'UniformOutput', false));
xlabel('Hardware Threads', 'FontSize', 12);
ylabel('Elapsed (wall clock) time, s', 'FontSize', 12);
title('Programming Assignment 01 (OpenMP)', 'FontSize', 14);
set(gcf,'Color','white');
grid on;
ax = gca;
ax.YMinorGrid = 'on';
ax.XMinorGrid = 'on';

[min_time, idx] = min(times);
opt_threads = threads(idx);
hold on
```

```
plot(opt_threads, min_time, 'rp', 'MarkerSize', 12, 'MarkerFaceColor', 'r');
text(opt_threads, min_time, ...
   sprintf('Optimal: %d threads\n%.4f s', opt_threads, min_time), ...
   'FontSize', 11, 'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'left');
hold off
figure(1);
```

5) Discussion

In P01, we are tasked with writing a program using the DGX server to observe how the wall clock runtime changes depending on the hardware thread count by the power of 2 via a DGEMM (Double Precision General Matrix Multiplication) program. There is to be a serial and parallel implementation of the program. Lastly, we are then required to plot our terminal output results in MATLAB via a data text file in order to precisely locate and display the optimal thread count. After writing the DGEMM program, I tested a range of thread counts ranging from 2^0 to 2^12 to give a good idea of runtime differences. According to my findings in the graphs above and my device's CPU specs, the optimal thread count appears to be 512 threads for my program with a time of 0.0730s before the system or program hits a saturation point and the processing time goes back up as the thread count continues to rise. All in all, this first programming assignment was a good visual and technical introduction to accelerated computing, and how to use the DGX server via a matrix multiplication program.