assignment-10-hpc-c

November 6, 2024

[1]:
```
!apt update
!apt install -y gcc g++
```

```
Get:1 https://cloud.r-project.org/bin/linux/ubuntu jammy-cran40/ InRelease
[3,626 B]
Get:2 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64
InRelease [1,581 B]
Hit:3 http://archive.ubuntu.com/ubuntu jammy InRelease
Get:4 http://security.ubuntu.com/ubuntu jammy-security InRelease [129 kB]
Get:5 https://r2u.stat.illinois.edu/ubuntu jammy InRelease [6,555 B]
Get:6 http://archive.ubuntu.com/ubuntu jammy-updates InRelease [128 kB]
Hit:7 https://ppa.launchpadcontent.net/deadsnakes/ppa/ubuntu jammy InRelease
Get:8 https://cloud.r-project.org/bin/linux/ubuntu jammy-cran40/ Packages [59.5
kB]
Hit:9 https://ppa.launchpadcontent.net/graphics-drivers/ppa/ubuntu jammy
InRelease
Get:10 http://archive.ubuntu.com/ubuntu jammy-backports InRelease [127 kB]
Hit:11 https://ppa.launchpadcontent.net/ubuntugis/ppa/ubuntu jammy InRelease
Get:12
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64
Packages [1,107 kB]
Get:13 https://r2u.stat.illinois.edu/ubuntu jammy/main all Packages [8,446 kB]
Get:14 https://r2u.stat.illinois.edu/ubuntu jammy/main amd64 Packages [2,610 kB]
Get:15 http://security.ubuntu.com/ubuntu jammy-security/universe amd64 Packages
[1,163 kB]
Get:16 http://security.ubuntu.com/ubuntu jammy-security/main amd64 Packages
[2,391 kB]
Get:17 http://archive.ubuntu.com/ubuntu jammy-updates/universe amd64 Packages
[1,452 kB]
Get:18 http://archive.ubuntu.com/ubuntu jammy-updates/restricted amd64 Packages
[3,319 kB]
Get:19 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 Packages [2,672
kB]
Fetched 23.6 MB in 6s (3,634 kB/s)
Reading package lists… Done
Building dependency tree… Done
Reading state information… Done
59 packages can be upgraded. Run 'apt list --upgradable' to see them.
```

```
W: Skipping acquire of configured file 'main/source/Sources' as
repository 'https://r2u.stat.illinois.edu/ubuntu jammy InRelease' does not seem
to provide it (sources.list entry misspelt?)
Reading package lists… Done
Building dependency tree… Done
Reading state information… Done
g++ is already the newest version (4:11.2.0-1ubuntu1).
g++ set to manually installed.
gcc is already the newest version (4:11.2.0-1ubuntu1).
gcc set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 59 not upgraded.
```

Parallel Computing Overview Parallel computing refers to the process of breaking down a computational problem into smaller, independent tasks that can be executed simultaneously across multiple processors or cores. In the context of OpenMP, this involves multithreading where different sections of code are executed concurrently using threads.

Benefits of Parallel Computing:

Improved Performance: Tasks are executed faster as multiple computations happen simultaneously. Efficiency: Workload is distributed across different cores or processors. Scalability: Large problems can be split and processed efficiently across multiple resources. OpenMP Overview: OpenMP (Open Multi-Processing) is an API that supports multi-platform shared-memory parallelism in C, C++, and Fortran. It is widely used for parallelizing loops and sections of code to utilize multi-core processors.

Directives: OpenMP uses #pragma directives to specify which parts of the code should be parallelized. Parallel Regions: Sections of code that are executed in parallel. Work-sharing: Dividing work among different threads.

[2]:
```
%%writefile gauss_elimination.c
#include <stdio.h>
#include <omp.h>
#include <time.h>

#define N 4

void gauss_elimination(float A[N][N], float B[N]) {
    int i, j, k;
    float factor;

    #pragma omp parallel for private(i, j, k, factor) shared(A, B)
    for (k = 0; k < N-1; k++) {
        for (i = k+1; i < N; i++) {
            factor = A[i][k] / A[k][k];
            for (j = k; j < N; j++) {
                A[i][j] = A[i][j] - factor * A[k][j];
            }
            B[i] = B[i] - factor * B[k];
```

```
        }
    }
}

int main() {
    float A[N][N] = {{2, -1, 1, 3},
                     {4, 5, -2, 2},
                     {1, 2, 3, -1},
                     {5, 4, 3, 2}};
    float B[N] = {8, 4, 10, 2};

    // Measure time
    clock_t start = clock();

    gauss_elimination(A, B);

    clock_t end = clock();
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Resultant matrix after Gaussian Elimination:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%f ", A[i][j]);
        }
        printf("\n");
    }

    printf("\nResultant vector B:\n");
    for (int i = 0; i < N; i++) {
        printf("%f ", B[i]);
    }
    printf("\n");

    printf("\nExecution time: %f seconds\n", time_spent);

    return 0;
}
```

```
Writing gauss_elimination.c
```

[3]:
```
!gcc -fopenmp gauss_elimination.c -o gauss_elimination
!time ./gauss_elimination
```

```
Resultant matrix after Gaussian Elimination:
2.000000 -1.000000 1.000000 3.000000
0.000000 7.000000 -4.000000 -4.000000
0.000000 0.000000 3.928571 -1.071429
0.000000 0.000000 0.000000 -0.636364
```

```
Resultant vector B:
8.000000 -12.000000 10.285714 -17.890911

Execution time: 0.000077 seconds

real    0m0.004s
user    0m0.001s
sys     0m0.000s
```

Theoretical Discussion Approach and Design of Parallel Algorithms: For the Gaussian Elimination algorithm, parallelism is achieved by parallelizing the elimination step (where rows are modified). OpenMP allows easy parallelization by using the #pragma omp parallel for directive.

Threading and Shared Memory: In OpenMP, the default model is shared memory, meaning all threads share the same global address space. This simplifies communication between threads but requires proper synchronization to avoid race conditions.

Load Balancing: In parallel algorithms, it is crucial to distribute the workload evenly across processors or threads. Poor load balancing can lead to some threads finishing earlier and remaining idle while others are still executing.

2. LU Decomposition LU Decomposition is a matrix factorization technique where a matrix is decomposed into a product of two matrices, L (Lower Triangular) and U (Upper Triangular).

i) Dolittle Method (LU Decomposition) In the Dolittle method, the lower triangular matrix L has ones on its diagonal, while the upper triangular matrix U is computed as part of the decomposition.

Code for LU Decomposition (Dolittle Method)

[5]:
```
%%writefile dolittle_lu.c
#include <stdio.h>
#include <omp.h>
#include <time.h>

#define N 4

void dolittle_lu(float A[N][N], float L[N][N], float U[N][N]) {
    int i, j, k;

    #pragma omp parallel for private(i, j, k)
    for (i = 0; i < N; i++) {
        // Upper Triangular Matrix U
        for (k = i; k < N; k++) {
            float sum = 0;
            for (j = 0; j < i; j++) {
                sum += L[i][j] * U[j][k];
            }
            U[i][k] = A[i][k] - sum;
```

```
        }

        // Lower Triangular Matrix L
        for (k = i; k < N; k++) {
            if (i == k) {
                L[i][i] = 1; // Diagonal as 1
            } else {
                float sum = 0;
                for (j = 0; j < i; j++) {
                    sum += L[k][j] * U[j][i];
                }
                L[k][i] = (A[k][i] - sum) / U[i][i];
            }
        }
    }
}

int main() {
    float A[N][N] = {{2, -1, 1, 3},
                     {4, 5, -2, 2},
                     {1, 2, 3, -1},
                     {5, 4, 3, 2}};
    float L[N][N], U[N][N];

    // Measure time
    clock_t start = clock();

    dolittle_lu(A, L, U);

    clock_t end = clock();
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Matrix L (Lower Triangular):\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%f ", L[i][j]);
        }
        printf("\n");
    }

    printf("\nMatrix U (Upper Triangular):\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%f ", U[i][j]);
        }
        printf("\n");
    }
```

```
    printf("\nExecution time: %f seconds\n", time_spent);

    return 0;
}
```

```
Overwriting dolittle_lu.c
```

[6]:
```
!gcc -fopenmp dolittle_lu.c -o dolittle_lu
!time ./dolittle_lu
```

```
Matrix L (Lower Triangular):
1.000000 0.000000 0.000000 0.000000
2.000000 1.000000 0.000000 0.000000
0.500000 0.357143 1.000000 0.000000
2.500000 0.928571 1.072727 1.000000

Matrix U (Upper Triangular):
2.000000 -1.000000 1.000000 3.000000
0.000000 7.000000 -4.000000 -4.000000
0.000000 0.000000 3.928571 -1.071429
0.000000 0.000000 0.000000 -0.636364

Execution time: 0.000299 seconds

real    0m0.002s
user    0m0.001s
sys     0m0.001s
```

ii) Crout's Method (LU Decomposition) In Crout's method, the lower triangular matrix L has non-unit elements on its diagonal, and the upper triangular matrix U has ones on its diagonal.

Code for LU Decomposition (Crout's Method)

[7]:
```
%%writefile crout_lu.c
#include <stdio.h>
#include <omp.h>
#include <time.h>

#define N 4

void crout_lu(float A[N][N], float L[N][N], float U[N][N]) {
    int i, j, k;

    #pragma omp parallel for private(i, j, k)
    for (i = 0; i < N; i++) {
        // Upper Triangular Matrix U (Diagonal 1s)
        U[i][i] = 1;
```

```
        // Lower Triangular Matrix L
        for (k = i; k < N; k++) {
            float sum = 0;
            for (j = 0; j < i; j++) {
                sum += L[k][j] * U[j][i];
            }
            L[k][i] = A[k][i] - sum;
        }

        // Upper Triangular Matrix U
        for (k = i+1; k < N; k++) {
            float sum = 0;
            for (j = 0; j < i; j++) {
                sum += L[i][j] * U[j][k];
            }
            U[i][k] = (A[i][k] - sum) / L[i][i];
        }
    }
}

int main() {
    float A[N][N] = {{2, -1, 1, 3},
                     {4, 5, -2, 2},
                     {1, 2, 3, -1},
                     {5, 4, 3, 2}};
    float L[N][N], U[N][N];

    // Measure time
    clock_t start = clock();

    crout_lu(A, L, U);

    clock_t end = clock();
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Matrix L (Lower Triangular):\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%f ", L[i][j]);
        }
        printf("\n");
    }

    printf("\nMatrix U (Upper Triangular):\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
```

```
            printf("%f ", U[i][j]);
        }
        printf("\n");
    }

    printf("\nExecution time: %f seconds\n", time_spent);

    return 0;
}
```

```
Writing crout_lu.c
```

[8]:
```
!gcc -fopenmp crout_lu.c -o crout_lu
!time ./crout_lu
```

```
Matrix L (Lower Triangular):
2.000000 0.000000 0.000000 0.000000
4.000000 7.000000 0.000000 0.000000
1.000000 2.500000 3.000000 0.000000
5.000000 6.500000 0.500000 -5.083333

Matrix U (Upper Triangular):
1.000000 -0.500000 0.500000 1.500000
0.000000 1.000000 -0.571429 -0.571429
0.000000 0.000000 1.000000 -0.833333
0.000000 0.000000 0.000000 1.000000

Execution time: 0.001915 seconds

real    0m0.004s
user    0m0.002s
sys     0m0.002s
```

3. Gauss-Seidel Method The Gauss-Seidel method is an iterative technique for solving a system of linear equations. It improves upon the initial guess and converges to the solution of the system.

Code for Gauss-Seidel Method

[9]:
```
%%writefile gauss_seidel.c
#include <stdio.h>
#include <omp.h>
#include <math.h>
#include <time.h>

#define N 3
#define MAX_ITER 1000
#define TOLERANCE 0.0001
```

```c
void gauss_seidel(float A[N][N], float B[N], float X[N]) {
    int i, j, k;
    float new_X[N], sum;

    for (k = 0; k < MAX_ITER; k++) {
        int converged = 1;

        #pragma omp parallel for private(i, j, sum) shared(A, B, X, new_X,␣
 ↪converged)
        for (i = 0; i < N; i++) {
            sum = 0;
            for (j = 0; j < N; j++) {
                if (i != j) {
                    sum += A[i][j] * X[j];
                }
            }
            new_X[i] = (B[i] - sum) / A[i][i];

            if (fabs(new_X[i] - X[i]) > TOLERANCE) {
                converged = 0;
            }
        }

        for (i = 0; i < N; i++) {
            X[i] = new_X[i];
        }

        if (converged) break;
    }
}

int main() {
    float A[N][N] = {{4, 1, 2},
                     {3, 5, 1},
                     {1, 1, 3}};
    float B[N] = {4, 7, 3};
    float X[N] = {0, 0, 0};  // Initial guess

    // Measure time
    clock_t start = clock();

    gauss_seidel(A, B, X);

    clock_t end = clock();
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Solution Vector X:\n");
```

```
    for (int i = 0; i < N; i++) {
        printf("%f ", X[i]);
    }
    printf("\n");

    printf("\nExecution time: %f seconds\n", time_spent);

    return 0;
}
```

```
Writing gauss_seidel.c
```

[10]:
```
!gcc -fopenmp gauss_seidel.c -o gauss_seidel
!time ./gauss_seidel
```

```
Solution Vector X:
0.499965 0.999962 0.499967

Execution time: 0.000338 seconds

real    0m0.003s
user    0m0.001s
sys     0m0.001s
```

Parallelization Metrics After measuring the execution time, you can compute parallelization metrics such as speedup and efficiency using the following formulas:

Speedup (S) = Time for Serial Execution / Time for Parallel Execution

Efficiency (E) = Speedup / Number of Threads

By modifying the OpenMP settings (such as setting the number of threads using omp_set_num_threads()), you can experiment with different configurations and compute these metrics.

[ ]: