Assignment 7: MPI and OpenMP Parallelization

PHYS 5v48.001 – HPC S25, Aaron Smith

ABSTRACT

This assignment explores the step-by-step process of turning a serial C++ program for an N-body gravitational system into parallel versions using: (i) OpenMP for shared-memory on a single node, (ii) MPI for distributed memory, and (iii) hybrid MPI+OpenMP, plus (iv) a shared-memory MPI approach. You will benchmark each approach for an increasing number of cores on a single HPC node (or more if you wish), measuring speedup and runtime per particle. Your goal is to understand the code modifications needed for concurrency, how HPC scheduling works, and how to measure parallel performance.

Overview: The N-body problem is a classic example of parallel computing. It involves calculating the gravitational interactions between N particles in D-dimensional space (usually $D = 3$). Each particle has an initial position ($\boldsymbol{x}$), velocity ($\boldsymbol{v}$), and mass (m). Newton's law of gravity gives the force between two particles i and j as and the acceleration on a particle i as the superposition of all pairwise forces (easy to parallelize):

$$\frac{\mathrm{d}p}{\mathrm{d}t} = G\frac{m_1 m_2}{r^2} \quad \text{(2-body Force)} \qquad \Rightarrow \qquad \boldsymbol{a}_i = G\sum_{j\neq i}^{N} \frac{m_j\,(\boldsymbol{x}_j - \boldsymbol{x}_i)}{\|\boldsymbol{x}_j - \boldsymbol{x}_i\|^3} \quad \text{(N-body Acceleration)}\,. \tag{1}$$

The system is evolved forward in time using an explicit Euler method (causal dependencies make this difficult to parallelize), updating the position and velocity for each particle at each step according to:

$$\boldsymbol{v}_i^{n+1} = \boldsymbol{a}_i^n \Delta t + \boldsymbol{v}_i^n \quad \text{(Kick Velocities)} \qquad \text{and} \qquad \boldsymbol{x}_i^{n+1} = \boldsymbol{a}_i^n \Delta t^2 + \boldsymbol{v}_i^n \Delta t + \boldsymbol{x}_i^n \quad \text{(Drift Positions)}\,. \tag{2}$$

Track the total kinetic energy for a sense of correctness. A softening parameter $\epsilon > 0$ is included to avoid divergences at very small interparticle distances. A minimal serial code is provided, but you will need to parallelize it for this assignment:

1. **Serial:** Measure the runtime for $N = \{128, 256, \dots\}$ or whichever values are feasible, then produce a benchmark chart of runtime vs. N. Save that code as your reference `nbody.cc`.
2. **OpenMP:** Copy `nbody.cc` to `nbody_omp.cc` and modify to add OpenMP pragmas for parallel loops.
3. **MPI:** Copy `nbody.cc` to `nbody_mpi.cc` and add basic MPI decomposition to handle partial sums, distributing N among ranks, etc.
4. **MPI + OpenMP:** Copy `nbody_mpi.cc` to `nbody_mpi_omp.cc` and combine the MPI decomposition with OpenMP shared memory parallelization on each rank.
5. **Shared-Memory MPI (Optional):** Copy `nbody_mpi.cc` to `nbody_mpi_shared.cc` for a shared-memory MPI approach that avoids communication costs.

In tutorial fashion we walk through the main steps to embed timing calls and incorporate OpenMP and MPI. The final HPC run is to measure performance with N as large as possible within reasonable time constraints.

1. SERIAL VERSION (NBODY.CC)

```cpp
#include <iostream>  // Standard I/O
#include <fstream>  // File I/O
#include <random>  // Random number generators
#include <vector>  // Vector (dynamic array)
#include <tuple>  // Tuple (multiple return values)
#include <chrono>  // Time utilities

// Global constants
static int N = 128;                    // Number of masses
static const int D = 3;                // Dimensionality
static int ND = N * D;                 // Size of the state vectors
static const double G = 0.5;           // Gravitational constant
static const double dt = 1e-3;         // Time step size
static const int T = 300;              // Number of time steps
static const double t_max = static_cast<double>(T) * dt; // Maximum time
static const double x_min = 0.;        // Minimum position
static const double x_max = 1.;        // Maximum position
static const double v_min = 0.;        // Minimum velocity
static const double v_max = 0.;        // Maximum velocity
static const double m_0 = 1.;          // Mass value
static const double epsilon = 0.01;   // Softening parameter
static const double epsilon2 = epsilon * epsilon; // Softening parameter^2
// Note that epsilon must be greater than zero!

using Vec = std::vector<double>; // Vector type
using Vecs = std::vector<Vec>;   // Vector of vectors type

// Random number generator
static std::mt19937 gen; // Mersenne twister engine
static std::uniform_real_distribution<> ran(0., 1.); // Uniform distribution

// Print a vector to a file
template <typename T>
void save(const std::vector<T>& vec, const std::string& filename,
          const std::string& header = "") {
  std::ofstream file(filename); // Open the file
  if (file.is_open()) { // Check for successful opening
    if (!header.empty())
      file << "# " << header << std::endl; // Write the header
    for (const auto& elem : vec)
      file << elem << " "; // Write each element
    file << std::endl; // Write a newline
    file.close(); // Close the file
  } else {
    std::cerr << "Unable to open file " << filename << std::endl;
  }
}

// Generate random initial conditions for N masses
std::tuple<Vec, Vec> initial_conditions() {
  Vec x(ND), v(ND);                  // Allocate memory
  const double dx = x_max - x_min;  // Position range
  const double dv = v_max - v_min;  // Velocity range
  for (int i = 0; i < ND; ++i) {
    x[i] = ran(gen) * dx + x_min;  // Random initial positions
    v[i] = ran(gen) * dv + v_min;  // Random initial velocities
  }
  return {x, v}; // Positions and velocities
}

// Compute the acceleration of all masses
// a_i = G * sum_{ji} m_j * (x_j - x_i) / |x_j - x_i|^3
Vec acceleration(const Vec& x, const Vec& m) {
  Vec a(ND);              // Accelerations
  for (int i = 0; i < N; ++i) {
    const int iD = i * D; // Flatten the index
    double dx[D];         // Difference in position
    for (int j = 0; j < N; ++j) {
      const int jD = j * D; // Flatten the index
      double dx2 = epsilon2; // Distance^2 (softened)
      for (int k = 0; k < D; ++k) {
        dx[k] = x[jD + k] - x[iD + k]; // Difference in position
        dx2 += dx[k] * dx[k]; // Distance^2
      }
      const double Gm_dx3 = G * m[j] / (dx2 * sqrt(dx2)); // G * m_j / |dx|^3
      for (int k = 0; k < D; ++k) {
        const int iDk = iD + k; // Flatten the index
        a[iDk] += Gm_dx3 * dx[k]; // Acceleration
      }
    }
  }
  return a; // Accelerations
}

// Compute the next position and velocity for all masses
std::tuple<Vec, Vec> timestep(const Vec& x0, const Vec& v0, const Vec& m) {
  Vec a0 = acceleration(x0, m); // Calculate particle accelerations
  Vec x1(ND), v1(ND);           // Allocate memory
  for (int i = 0; i < ND; ++i) {
    v1[i] = a0[i] * dt + v0[i]; // New velocity
    x1[i] = v1[i] * dt + x0[i]; // New position
  }
  return {x1, v1}; // New positions and velocities
}

// Main function
int main(int argc, char** argv) {
  // Start timing
  auto start = std::chrono::high_resolution_clock::now();

  // Set up the problem
  if (argc > 1) {
    N = std::atoi(argv[1]); // Update the number of masses
    ND = N * D;              // Update the size of the state vectors
  }

  // Prepare vectors for time points, masses, positions, velocities, and kinetic energy
  Vec t(T+1); // Time points
  for (int i = 0; i <= T; ++i)
    t[i] = double(i) * dt; // Time points
  Vec m(N, m_0); // Masses (all equal)
  Vecs x(T+1), v(T+1); // Positions and velocities
  std::tie(x[0], v[0]) = initial_conditions(); // Set up initial conditions

  // Simulate the motion of N masses in D-dimensional space
  for (int n = 0; n < T; ++n)
    std::tie(x[n+1], v[n+1]) = timestep(x[n], v[n], m); // Time step

  // Calculate the total kinetic energy of the system
  Vec KE(T+1); // Kinetic energy
  for (int n = 0; n <= T; ++n) {
    double KE_n = 0.; // Kinetic energy
    auto &v_n = v[n]; // Velocities
    for (int i = 0; i < N; ++i) {
      double v2 = 0.; // Velocity magnitude
      for (int j = 0; j < D; ++j) {
        const int k = i * D + j; // Flatten the index
        v2 += v_n[k] * v_n[k]; // Velocity magnitude
      }
      KE_n += 0.5 * m[i] * v2; // Kinetic energy
    }
    KE[n] = KE_n; // Kinetic energy
  }

  // Print the vector to the specified file
  save(KE, "KE_" + std::to_string(N) + ".txt", "Kinetic Energy");
  save(t, "time_" + std::to_string(N) + ".txt", "Time");

  // Output the results
  std::cout << "Total Kinetic Energy = [" << KE[0];
  const int T_skip = T / 50; // Skip every T_skip time steps
  for (int n = 1; n <= T; n += T_skip)
    std::cout << ", " << KE[n];
  std::cout << "]" << std::endl;

  // Stop timing
  auto end = std::chrono::high_resolution_clock::now();
  double elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() / 1000.;
  std::cout << "Runtime = " << elapsed << " s for N = " << N << std::endl;
  return 0;
}
```

Compile and run with:

```
g++ -O3 -std=c++17 nbody.cc -o nbody
time ./nbody 128
```

Exercise: Record these times for a range of N and plot (log-log axes) the runtime vs. N to see the scaling behavior. This will be your reference for estimating the runtime of the parallel versions. Hint: Write a bash script to automate running multiple times in a loop to colloct all data at once.

2. OPENMP PARALLELIZATION (NBODY_OMP.CC)

Exercise: Parallelize the code with OpenMP and measure the speedup for different numbers of threads. Run the code with `OMP_NUM_THREADS` in $\{1,2,4,8,16,\dots\}$ and plot (log-log axes) the runtime/N^2 vs. n_{threads} to see the speedup. It is recommended to use the same value of N for all runs, but you can also vary N if the runtime is too long (low core counts) or too short (high core counts). The basic steps are outlined below:

1. Add the `-fopenmp` flag to the compiler to enable OpenMP. Compilation then becomes:

    ```
    g++ -fopenmp -O3 -std=c++17 nbody_omp.cc -o nbody_omp
    export OMP_NUM_THREADS=4
    time ./nbody_omp 1024
    ```

2. Add `#include <omp.h> // OpenMP (parallelism)` near the top to include the OpenMP header.
3. Add a global variable to track the thread number and make it thread-private.

    ```
    static int thread; // Unique thread number
    #pragma omp threadprivate(thread, gen, ran)
    ```

4. Add the following code to the main function to set the thread number and seed the random number generator:

    ```
    #pragma omp parallel
    {
      thread = omp_get_thread_num(); // Unique thread number
      gen.seed(thread); // Seed the random number generator
    }
    ```

5. Insert `#pragma omp parallel for` in the loops in `initial_conditions`, `acceleration`, `timestep`, and the KE summation if that is also loop-based.

3. MPI PARALLELIZATION (NBODY_MPI.CC)

Exercise: Parallelize the code with MPI and measure the speedup for different numbers of MPI ranks. Add the MPI timings to the same plot as the OpenMP timings to compare the two approaches. The basic steps are outlined below:

1. Change the compiler to `mpicxx` to enable MPI. Compilation then becomes:

    ```
    mpicxx -O3 -std=c++17 nbody_mpi.cc -o nbody_mpi
    time mpiexec -n 4 ./nbody_mpi 1024
    ```

2. Insert `#include <mpi.h>  // MPI (parallelism)` near the top to include the MPI header.
3. Add global variables

    ```
    static int rank, n_ranks; // Process rank and number of processes
    static std::vector<int> counts, displs; // Counts and displacements for MPI_Allgatherv
    static std::vector<int> countsD, displsD; // State counts and displacements for MPI_Allgatherv
    static int N_beg, N_end, N_local; // Mass range for each process [N_beg, N_end)
    static int ND_beg, ND_end, ND_local; // State vector range for each process [ND_beg, ND_end)
    ```

4. Initialize MPI and get the rank and number of ranks and set up the work distribution among ranks, defining N_beg, N_end so that each process handles N_local $= N/$n_ranks plus remainder:

```
// Set up parallelism
void setup_parallelism() {
  MPI_Init(NULL, NULL); // Initialize MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Unique process rank
  MPI_Comm_size(MPI_COMM_WORLD, &n_ranks);

  // Get the current time and convert it to an integer
  auto now = std::chrono::high_resolution_clock::now();
  auto now_cast = std::chrono::time_point_cast<std::chrono::microseconds>(now);
  auto now_int = now_cast.time_since_epoch().count();

  // Pure MPI version
  gen.seed(now_int ^ rank); // Seed the random number generator

  // Divide the masses among the processes (needed for MPI_Allgatherv)
  counts.resize(n_ranks); // Counts for each process
  displs.resize(n_ranks); // Displacements for each process
  countsD.resize(n_ranks); // State counts for each process
  displsD.resize(n_ranks); // State displacements for each process
  const int remainder = N % n_ranks; // Remainder of the division
  for (int i = 0; i < n_ranks; ++i) {
    counts[i] = N / n_ranks; // Divide the masses among the processes
    displs[i] = i * counts[i]; // Displacements where each segment begins
    if (i < remainder) {
      counts[i] += 1; // Correct the count
      displs[i] += i; // Correct the displacement
    } else {
      displs[i] += remainder; // Correct the displacement
    }
    countsD[i] = counts[i] * D; // State counts for each process
    displsD[i] = displs[i] * D; // State displacements for each process
  }

  // Set up the local mass ranges
  N_beg = displs[rank]; // Mass range for each process [N_beg, N_end)
  N_end = N_beg + counts[rank]; // Mass range for each process [N_beg, N_end)
  ND_beg = N_beg * D; // State vector range for each process [ND_beg, ND_end)
  ND_end = N_end * D; // State vector range for each process [ND_beg, ND_end)
  N_local = N_end - N_beg; // Local number of masses
  ND_local = ND_end - ND_beg; // Local size of the state vectors
}
```

5. To keep the implementation simple, allocate the full memory on each process, but only initialize the local portion of the state vectors by changing the loop ranges to `for (int i = ND_beg; i < ND_end; ++i)`. Then perform a gather operation to communicate the data to all processes (as this is needed for the acceleration calculation). This is done in the initial_conditions function before the return statement:

```
if (n_ranks > 1) { // More than one process
  // Gather the initial positions and velocities
  MPI_Allgatherv(x.data() + ND_beg, ND_local, MPI_DOUBLE, x.data(),
                 countsD.data(), displsD.data(), MPI_DOUBLE, MPI_COMM_WORLD);
  MPI_Allgatherv(v.data() + ND_beg, ND_local, MPI_DOUBLE, v.data(),
                 countsD.data(), displsD.data(), MPI_DOUBLE, MPI_COMM_WORLD);
}
```

6. Similarly change the loop ranges in the `acceleration` and `timestep` functions, and communicate the updated positions and velocities. This is done in the `timestep` function before the return statement exactly as above but with the vectors `x1` and `v1`.

7. Modify the loop range for the kinetic energy calculation to only sum over the local masses. Then reduce the kinetic energy to the root process before saving the data to a file. This is done after the loop in the main function:

```
if (rank == 0) {
  // Reduce the kinetic energies
  MPI_Reduce(MPI_IN_PLACE, KE.data(), T+1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  // Protected save and print logic ...
} else {
  // Send the kinetic energies
  MPI_Reduce(KE.data(), NULL, T+1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

8. Finally, clean up MPI resources and finalize the MPI environment at the end of the main function with `MPI_Finalize(); // Finalize MPI`.

4. HYBRID MPI+OPENMP (NBODY_MPI_OMP.CC)

Exercise: Combine the MPI and OpenMP versions to create a hybrid version that uses MPI for domain decomposition and OpenMP for shared-memory parallelism within each domain. Run as large a problem as you can within 15 minutes, even going to multiple nodes if possible. Report your largest problem size. The basic steps are outlined below:

- Start from the MPI version and add the OpenMP pragmas as in the OpenMP version, almost all steps are directly applicable.
- One exception is generalizing the random number seeding for processes and threads. This can be done by combining the process rank and thread number to seed the random number generator:

```
// Hybrid MPI+OpenMP version
#pragma omp parallel
{
  thread = omp_get_thread_num(); // Unique thread number
  gen.seed(now_int ^ (thread * n_ranks + rank)); // Seed the random number generator
}
```

- In this case the compiler flags should include both MPI and OpenMP flags:

```
mpicxx -fopenmp -O3 -std=c++17 nbody_mpi_omp.cc -o nbody_mpi_omp
export OMP_NUM_THREADS=4
mpiexec -n 2 ./nbody_mpi_omp 1024
```

5. SHARED MEMORY MPI (NBODY_MPI_SHARED.CC)

Optional Exercise: Instead of replicating the global arrays on each rank, use **MPI Shared Memory** windows to reduce memory overhead and unify the code. In principle this is similar to the OpenMP version, but makes you think about the memory a bit more. The basic steps are outlined below:

- Start from the MPI version making the following changes:

  ```
  // Shared memory for masses, positions, velocities, and accelerations
  static double *m, *x, *v, *a, *x_next, *v_next; // Shared memory
  static MPI_Win win_m, win_x, win_v, win_a, win_x_next, win_v_next; // Shared windows
  ```

  This also no longer stores the full data from each time step, but only the current and next positions and velocities, requiring some changes throughout the code.

- Allocate shared memory and create shared windows in the `setup_parallelism` function:

  ```
  // Allocate shared memory for positions, velocities, and accelerations
  if (rank == 0) {
    MPI_Win_allocate_shared(N * sizeof(double), sizeof(double),
                            MPI_INFO_NULL, MPI_COMM_WORLD, &m, &win_m);
    MPI_Win_allocate_shared(ND * sizeof(double), sizeof(double),
                            MPI_INFO_NULL, MPI_COMM_WORLD, &x, &win_x);
    // Repeat for v, a, x_next, v_next
  } else {
    int disp_unit;
    MPI_Aint size;
    MPI_Win_allocate_shared(0, sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD, &m, &win_m);
    // Repeat for x, v, a, x_next, v_next
  }
  ```

- Before calline `MPI_Finalize` free the shared memory and windows:

  ```
  // Positions, velocities, and accelerations (reversed order)
  MPI_Win_free(&win_v_next);
  MPI_Win_free(&win_x_next);
  MPI_Win_free(&win_a);
  MPI_Win_free(&win_v);
  MPI_Win_free(&win_x);
  MPI_Win_free(&win_m);
  ```

- Modify the code to use the global shared memory arrays by changing the function calls to return void and placing explicit barriers when needed. For example, in the `initial_conditions` and `timestep` functions:

  ```
  MPI_Barrier(MPI_COMM_WORLD); // Synchronize all processes
  ```

  This synchronization avoids race conditions and ensures that all processes have completed their accesses.

- Before the barrier in the timestep function, swap the pointers to the current and next data:

  ```
  std::swap(x, x_next); // Update the positions and velocities
  std::swap(v, v_next); // By swapping the pointers to the arrays
  ```

- Finally, include a kinetic energy function that can be called in the time loop to calculate the kinetic energy for the local masses and reduce it to the root process before saving the data to a file.

  ```
  KE[n] = kinetic_energy(); // Kinetic energy
  ```

Note: A further optimization for multi-node HPC is to distribute the data across nodes without replication or unnecessary communication, and use one of the shared memory approaches within each node. This is more advanced and requires a different approach, but is worth considering for large-scale simulations.

6. REPORTING GUIDELINES

1. **Submission Format:** Provide a short PDF with:
   - **Key Code Snippets** or full files for each `nbody_*.cc` approach.
   - **Benchmark Results**: Time or speedup for each approach on HPC for different numbers of cores. Possibly also multi-node for the MPI + OpenMP hybrid approach.
   - **Plots or Tables**: Runtimes, scaling, and a short discussion.
2. **Evaluation Criteria**:
   - *Correctness and Completeness*: Each parallel method compiles, runs, and yields consistent results.
   - *Clarity of Code and Analysis*: Timers, HPC logs, scaling charts, discussion.
   - *Reflections on HPC* usage, domain decomposition, concurrency trade-offs.

This assignment provides practice in converting a serial HPC code to an OpenMP shared-memory version, an MPI distributed-memory version, and a hybrid approach. Remember to submit at least one job on an HPC cluster. Happy coding!