

Topic 2: Molecular Dynamics of Lennard-Jones System (continued)

The most time consuming part of an MD calculation is the evaluation of the accelerations of the particles at each time step. Between each pair of particles there action force and an equal and opposite reaction force. Since there are $N(N - 1)/2$ pairs of particles in the system, the time to calculate the forces scales like $\mathcal{O}(N^2)$.

Systolic Algorithm Speedup

In the *systolic algorithm* the N particles are divided evenly among the p processes running on the parallel machine. The calculation of forces is divided into p stages:

- The forces between the $n = N/p$ local particles on each processor are computed. This computation can proceed fully in parallel.
- The local particles are copied to the traveling particles on each processor, and the following steps are repeated $p - 1$ times:
 - The traveling particles are shifted to the right neighbor processor in the ring topology. This incurs a communication overhead not present in the serial program. The movement of the p sets of traveling particles can proceed in parallel.
 - The forces between the n local particles and the n traveling particles on each processor are computed. This computation can proceed fully in parallel.

Thus the main overhead incurred by the parallel algorithm relative to the serial is the communication of $\mathcal{O}(N)$ data values (the coordinates of the traveling particles) $p - 1$ times over the interconnection network. Thus the wall time of a parallel run is expected to behave like

$$\frac{N^2}{p}\tau_{\text{comp}} + \frac{(p-1)N}{p}\tau_{\text{comm}},$$

where τ_{comp} is the time required to process one pair of particles, and τ_{comm} is the time required to move a traveling particle from one process to another.

Usually, $\tau_{\text{comm}} \gg \tau_{\text{comp}}$, so for small N a serial program will run much faster than a parallel program.

However, for a given number p of processes and large enough N , the computational time will dominate and the parallel program will run almost p times faster than the serial program.

MPI Implementation of Systolic Algorithm

The program `md_systolic.cpp` contains an MPI implementation of the systolic molecular dynamics algorithm for the Lennard-Jones system. This program is a straightforward modification of the serial program `md.cpp`.

The following global variables are added:

```
int numberOfProcesses;
```

```
int particlesPerProcess;
int rankOfThisProcess;
double rTraveling[MAX_PARTICLES][3];
double sendBuffer[MAX_PARTICLES][3];
double wallClockTime, communicationTime;
```

The first two integer variables represent p and N/p . The two new arrays `rTraveling` and `sendBuffer` are used to process the traveling particles. The computation and communication times are estimated using the MPI function `MPI_Wtime()` which returns the elapsed time in seconds on the processor which calls it.

The systolic algorithm is implemented inside the function `void computeForces()`:

```
// copy positions of local particles to traveling particles
for (i = 0; i < particlesPerProcess; i++)
    for (k = 0; k < 3; k++)
        rTraveling[i][k] = r[i][k];

// implement systolic algorithm
for (process = 0; process < numberOfProcesses - 1; process++) {
    startTime = MPI_Wtime();
    // copy traveling particles to send buffer
    for (i = 0; i < particlesPerProcess; i++)
        for (k = 0; k < 3; k++)
            sendBuffer[i][k] = rTraveling[i][k];
    // interleave send/receive to prevent deadlock
```

```
if (rankOfThisProcess % 2 == 0) { // even processes send/receive
    // send traveling particles to next process
    MPI_Send(sendBuffer[0], 3 * particlesPerProcess, MPI_DOUBLE,
             nextProcess, 0, MPI_COMM_WORLD);
    // receive traveling particles from previous process
    MPI_Recv(rTraveling[0], 3 * particlesPerProcess, MPI_DOUBLE,
             previousProcess, 0, MPI_COMM_WORLD, &status);
} else { // odd processes receive/send
    MPI_Recv(rTraveling[0], 3 * particlesPerProcess, MPI_DOUBLE,
             previousProcess, 0, MPI_COMM_WORLD, &status);
    MPI_Send(sendBuffer[0], 3 * particlesPerProcess, MPI_DOUBLE,
             nextProcess, 0, MPI_COMM_WORLD);
}
communicationTime += MPI_Wtime() - startTime;
// calculate contributions to local forces from traveling particles
```

Note here the interleaving of `MPI_Send` and `MPI_Recv` calls between processes of even and odd rank. Since both of these functions *block*, i.e., do not return until the send or receive communication is complete, this interleaving is important to prevent *deadlock*. The case of $p = 2$ shows how interleaving works:

Deadlock: Processes 0 and 1 both call `MPI_Send` first. They both block until the data sent has been received (by the other process). But neither process has signalled that it is ready to receive!

Deadlock: Processes 0 and 1 both call `MPI_Recv` first. They both block until they receive the data (from the other process). But neither process has sent the data yet!

Interleave: Process 0 calls `MPI_Send`, and process 1 calls `MPI_Recv`. Data moves from process 0 to process 1. Process 0 then calls `MPI_Recv` and process 1 calls `MPI_Send`. Data moves from process 1 to process 0.

Additional modifications to the serial program involve synchronization of the values of global variables. For example, the function `void measureProperties ()` is modified to compute the kinetic energy and pressure:

```
kineticEnergy = virial = 0;
kineticEnergyThisProcess = virialThisProcess = 0;
for (p = 0; p < particlesPerProcess; p++) {
    for (i = 0; i < 3; i++) {
        kineticEnergyThisProcess += 0.5 * v[p][i] * v[p][i];
        virialThisProcess += r[p][i] * a[p][i];
    }
}
// accumulate total kinetic energy and virial
startTime = MPI_Wtime();
MPI_Allreduce(&kineticEnergyThisProcess, &kineticEnergy,
              1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(&virialThisProcess, &virial,
```

```
1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);  
communicationTime += MPI_Wtime() - startTime;
```

Scaling of the algorithm with N and p

The following are some typical results obtained on nash.ccr:

No. of Particles N	No. of processes p	Wall Clock Time	Communication Time
32	4	21.0 secs	20.1 secs
256	4	60.3 secs	16.7 secs
256	8	305.4 secs	281.4 secs
4000	4	10,401 secs	245.0 secs
4000	8	5,674 secs	231.6 secs