Implementation and Evaluation of MPI-Based Parallel MD Program

R. TROBEC,[1] M. ŠTERK,[1] M. PRAPROTNIK,[2] D. JANEŽIČ[2]

[1]*Jožef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia*
[2]*National Institute of Chemistry, Hajdrihova 19, 1000 Ljubljana, Slovenia*

Received 4 September 2000; revised 13 February 2001; accepted 20 February 2001

ABSTRACT: The message-passing interface (MPI)-based object-oriented particle–particle interactions (PPI) library is implemented and evaluated. The library can be used in the n-particle simulation algorithm designed for a ring of p interconnected processors. The parallel simulation is scalable with the number of processors, and has the time requirement proportional to n^2/p if n/p is large enough, which guarantees optimal speedup. In a certain range of problem sizes, the speedup becomes superlinear because enough cache memory is available in the system. The library is used in a simple way by any potential user, even with no deep programming knowledge. Different simulations using particles can be implemented on a wide spectrum of different computer platforms. The main purpose of this article is to test the PPI library on well-known methods, e.g., the parallel molecular dynamics (MD) simulation of the monoatomic system by the second-order leapfrog Verlet algorithm. The performances of the parallel simulation program implemented with the proposed library are competitive with a custom-designed simulation code. Also, the implementation of the split integration symplectic method, based on the analytical calculation of the harmonic part of the particle interactions, is shown, and its expected performances are predicted. Int J Quantum Chem 84: 23–31, 2001

Key words: parallel MD simulation; object-oriented library; ring network

Correspondence to: D. Janežič; e-mail: dusa@kihp5.cmm.ki.si.
Contract grant sponsor: Ministry of Science and Technology, Republic of Slovenia.
Contract grant numbers: J1-106-513, P1-0104-503, J1-0104-7346, and S41-104-001.

Introduction

Many different natural phenomena or artificially defined systems can be simulated using particle interactions, for example, the movement of planets in a solar system where particles interact by gravitation. What a particle represents and which attributes describe its behavior depend on the application. Table I shows some applications [1]. The particle can be a physical body (molecule), a natural group of bodies (galaxy), or an artificially declared group (group of electrons).

The simulation based on particles usually consists of the calculation of interactions between the particles and their application for the determination of new particle data. The calculation of interactions is the most time-consuming part, with complexity of $O(n^2)$ for an n-particle system [2]. However, parallel implementations exist, with a linear speedup, that reduce the complexity to $O(n^2/p)$ on a p-processor system [3].

There are different applications as well as many different programs for essentially the same tasks. The same program should be run with different numerical methods and/or different particle data. The sequential code of each new, but slightly different, program is usually adopted from an older version. The development of a parallel version is a more demanding task, and requires some deeper computer knowledge. If a parallel computer algorithm for the calculation of particle interactions is proved to be optimal, then the principle of parallelization is the same for all applications. All of the programming necessary for the parallel implementation can be encompassed by a multipurpose message-passing interface (MPI)-based [4] program library for particle interactions. Two of the general principles from the object-oriented programming [5], classes and templates, can be used in such an implementation.

The structure of this article is as follows. In the following section, the parallel algorithm for particle interactions is described. Some basic concepts of the proposed programming library for particle–particle interactions (PPI) are presented in the third section. Finally, parallel test programs for molecular dynamics (MD), implemented with the PPI library, are described and evaluated on a cluster with different numbers of computing nodes. The work concludes with some comments on results and directions for future work.

Parallel Algorithm for Particle Interactions

In most applications, the interaction between two particles is symmetrical, and no particle can interact with itself. The sequential algorithm for n particles for each time step is thus

```
for (i = 0; i < n; i++)
    for (j = i+1; j < n; j++)
        calculateInteraction(i, j);
```

There are $n(n-1)/2$ calculations of particle interactions.

Other possibilities exist for system simulation, such as calculating the field of interactions from all particles (e.g., the gravity field in the case of a solar system simulation) and applying it to all particles [1].

We have implemented the approach known as "particle–particle" simulation. On a parallel system with p processors numbered as $0, 1, \ldots, p-1$ and connected in a ring, the particles are divided uniformly among processors [3]. All local particles on a processor are called self-particles. The copies of self-particle attributes, being transmitted to the neighboring ring processors, are called guest particles. Several variations of parallel algorithms for

TABLE I
Some applications of particle interactions.

Application	Gas	Semiconductor	Solar system	Galaxy clusters
Particle	molecule	10^4 electrons or holes	planet	galaxy
Attributes	molecule position, relative positions of atoms, velocity	charge, position	mass, position, velocity	mass, position, velocity, radius
No. of particles	10^3–10^5	10^5–10^9	10–1000	10^4–10^5
Simulated time [s]	10^{-12}	10^{-9}	$10^6 - 10^{10}$	10^{17}

Pass	Processor 0	Processor 1	Processor 2	Processor 3
0	**0 1 2** 0 1 2	**3 4 5** 3 4 5	**6 7 8** 6 7 8	**9 10 11** 9 10 11
	0-1 0-2 1-2	3-4 3-5 4-5	6-7 6-8 7-8	9-10 9-11 10-11
1	**0 1 2** 9 10 11	**3 4 5** 0 1 2	**6 7 8** 3 4 5	**9 10 11** 6 7 8
	0-9 0-10 0-11 1-9 1-10 1-11 2-9 2-10 2-11	3-0 3-1 3-2 4-0 4-1 4-2 5-0 5-1 5-2	6-3 6-4 6-5 7-3 7-4 7-5 8-3 8-4 8-5	9-6 9-7 9-8 10-6 10-7 10-8 11-6 11-7 11-8
2	**0 1 2** 6 7 8	**3 4 5** 9 10 11	**6 7 8** 0 1 2	**9 10 11** 3 4 5
	0-6 0-7 0-8 1-7 1-8	3-9 3-10 3-11 4-10 4-11	6-1 6-2 7-2 8-2	9-4 9-5 10-5 11-5

FIGURE 1. Parallel algorithm for particle interactions on four processors. For each calculation pass, the states of all processors are shown. Upper boxes show selfparticles (bold), lower boxes show guest particles; calculated pass interactions are shown as pairs *particle*$_1$–*particle*$_2$.

particle interactions were tested. We describe here the fastest variation only.

Each processor performs, within a particular time step, $p/2$ calculation passes following these rules.

1. In pass 0, calculate the interactions between self-particles.
2. Send copies of self-particle attributes to the right neighbor, and receive guest-particle (copies of self-particle) attributes from the left neighbor. The interpretation of left and right direction is arbitrary.
3. In the next pass, calculate the interactions between all pairs: self-particle–guest particle, and accumulate the calculated interactions with the self-particle and guest-particle attributes.
4. Send guest particles to the right neighbor, and receive new guest particles from the left.
5. Repeat steps 3 and 4 until the particles have come halfway around the ring.
6. If the number of processors is even, the same pairs of particles are now on processors i and $i+p/2$. Each processor therefore calculates half of the interactions in the last pass.
7. After all of the interactions are calculated, but not yet added to all particles, it is necessary to return guest particles halfway around the ring to their original processors which add all of the remaining interactions.

The algorithm is graphically explained in Figure 1. The example shows 12 particles on 4 processors. In pass 0, the local $(n/p)(n/p-1)/2 = 3$ interactions are calculated as in a sequential algorithm. In pass 1, all $(n/p)^2 = 9$ interactions are calculated on each processor, and added to self- and guest-particle attributes. For example, processor 1 calculates the interaction 4–1, and adds it to self-particle 4 and to guest particle 1. In pass 2, the interaction 6–1, for example, is calculated on processor 2 only, and not on processor 0. Finally, the guest particles are rotated for two more steps in order to reach the initial positions. The interaction of guest particle 1, already calculated in pass 2 on processor 2, is now added to self-particle 1.

This algorithm is optimal because the number of calculated interactions remains $n(n-1)/2$, the minimum possible, and the calculation load is evenly distributed among the processors.

`PPI` Library

In the design phase, we have followed the idea of making a simple and unified tool for parallel programming in the area of particle simulation. The basic assumption was that a potential user has no deep knowledge of parallel computing. All supporting tasks, known in advance, should be ready to use. Our goal was to reach the execution speed of the custom-designed MD programs.

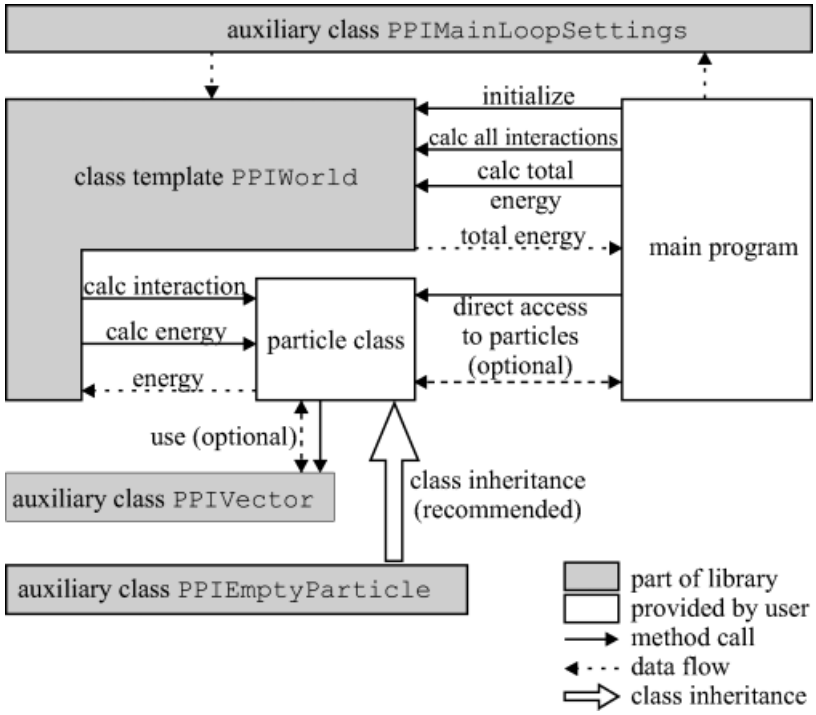

FIGURE 2. Structure of a parallel simulation program using the `PPI` library.

We have designed an object-based programming library. Previous requirements suggested the use of an object-oriented programming language. The need for fast implementation precludes the use of class inheritance and virtual functions, so that in line functions have to be used instead. The template mechanism from the programming language C++ [5] seems to be the optimal choice.

The program library for particle–particle interactions provides a standard interface to the simulation system that implements all necessary tasks for system initialization, parallel execution, and output of results. The user has to provide declarations of particles and their attributes with an algorithm for the calculation of interactions.

Different elements of a parallel simulation program written within the `PPI` library are shown in Figure 2. The main part of the library is the `PPIWorld` class template. There are also some auxiliary classes, used optionally, for easier and safer programming. The particle class and the main program have to be provided by user. The particle class is used in the template `PPIWorld` that acts as a simulator system for particles represented by the particle class, and controlled by the main program.

CLASS `PPIWorld`

A simple simulation program would use `PPIWorld` as follows.

1. Create a variable of the type `PPIWorld`, e.g., `PPIWorld<myParticleClass> world;`
2. Set initial particle data; either read from file with `world.readParticles();` or set to random values with `world.setRandomParticles();`
3. Call `world.mainLoop();` with suitable parameters—number of iterations, version of the basic algorithm, etc.
4. When necessary, calculate the total system energy with `world.getTotal Energy();` or write particle data to a file with `world.writeParticles();` etc.

`PPIWorld` includes, for example, the following methods, where methods represent procedures declared in an object-oriented class.

`readParticles();` —deletes old particle data, reads new particle data, and distributes them among the processors.

`gatherParticles();` —gathers particle data on processor 0, allocates memory space, and returns pointer to processor 0, returns `NULL` to all other processors.

`distributeParticles();` —distributes particles from processor 0 among all of the processors.

`mainLoop();`—runs the simulation for a desired number of time steps.

`getTotalEnergy();` —accumulates energy for all particles.

PARTICLE CLASS

The use of class `PPIWorld` requires us to provide a particle class, which includes some methods that are called from `PPIWorld`.

`interact();`—calculates the interaction between two particles, and writes the result into both particles' attributes.

`calcNewSpeed();`—calculates the change of particle velocity due to the interaction, and adds it to the velocity attribute.

`postStep();` —is either left empty or used for a specific calculation that should be performed on all particles at the end of a time step.

`createMPIType();` —creates a description of an MPI type that is used in communication. The user would have to study the MPI documents [4] at this step only. This method is necessary provided that a parallel system composed of different computers is used. However, on homogeneous computer clusters, leaving out this method will only result in minimal degradation of the communication performance.

PARALLELISM IN `PPI` LIBRARY

The most important feature of the class `PPIWorld` is the parallel implementation of the main loop. In the current version of the `PPI` library, the algorithm "particle–particle," described in the second section, was implemented. The `PPI` library was designed in such a way that more complex systems, such as molecules, could be simulated. The class `PPIWorld` takes care of distributing and gathering the particles. Should the user need to run a certain calculation that has no support in the class and does not need to be parallelized, the `PPIWorld` can be used for gathering particle data. After this calculation is completed, the parallel simulation proceeds in a standard way.

Test Examples

To test the `PPI` library, we applied it to the standard method for the MD simulation of a monoatomic system. Also, the theoretical estimation of the calculation complexity of a system of butadiyne molecules is performed.

LEAPFROG VERLET SIMULATION

To present some basic principles of MD simulation, a simple monoatomic system of argon particles is described, based on the leapfrog Verlet algorithm (LFV) given in [6].

The evaluation model of n argon particles in a cube of size L was chosen to give realistic merits of the parallel method efficiency. The periodic boundary conditions were imposed to conserve the number of particles. The nonbonded interactions were calculated applying the *minimum image convention* [7].

Argon atoms represent particles with these attributes: position $\mathbf{r}$, velocity $\mathbf{v}$, and force $\mathbf{F}$, while mass is the same for all particles, and thus is used as a constant in the force calculations. The force on particle i by particle j is calculated as

$$\mathbf{F}_{ij} = (\mathbf{r}_i - \mathbf{r}_j)\left(r_{ij}^{-14} - \tfrac{1}{2} r_{ij}^{-8}\right). \tag{1}$$

For each time step, new positions are calculated as

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \Delta t \mathbf{v}(t) + \frac{\Delta^2 t}{2}\mathbf{F}(t). \tag{2}$$

There is a similar equation for the calculation of new velocities. Appropriate units are chosen in order to eliminate multiplication by constants.

The left part of Figure 3 shows some of the main methods from the `PPI` library that are used in the particle class `Argon`. Forces, new positions, and new velocities are calculated in `calcPartialForce()`, `calcNewSpeed()`, and `calcNewPos()` according to the given equations. To achieve a realistic system state, it is necessary to scale velocities after a certain number of steps by a factor $s = \sqrt{E_k/E'_k}$, where E_k is the initial kinetic energy and E'_k is the new kinetic energy. After several applications of the scaling procedure, a stable system is acquired that preserves a constant energy. The scaling is implemented in the `postIteration()` method.

```
class Argon: public PPIEmptyParticle {
public:
    CTriple position, speed, force;

    void clearInteraction() {
        force.reset();

    }
    void addPartialForce(CParticle &other) {
        calculate Lennard-Jones force between
            atoms *this and other
        add it to *this and subtract from other

    }
    void calcNewSpeed() {
        speed.addTo(force);
    }
    void calcNewPos() {
        position.addTo(speed); position.addTo(force);
    }
    void postIteration(int iter, double startEnergy,
                       double energy) {
        if necessary scale velocity

    }
};

int main(int argc, char **argv) {
    PPIWorld<Argon> world(argc, argv, &cout);
        //create class from template
        //this also initializes MPI
    ifstream inFile(argv[1], "r");
    world.readParticles(inFile);
        //read initial states from file
    world.mainLoop(atoi(argv[2]), settings);
        //run argv[2] steps
}
```

```
class Butadiyne: public PPIEmptyParticle {
public:
    molecule position/speed and atom relative
        positions/speeds/forces
    normal and Cartesian coordinates

    void preIteration() {
        harmonical part of dynamics
            (vibration, rotation, translation)
        transform normal coordinates to Cartesian
    }
    void clearInteraction() {
        reset all atom forces
        calculate bond forces
        calculate other forces between atoms
            within molecule
    }
    void addPartialForce(CParticle &other) {
        calculate Lennard-Jones and Coulomb forces for
        each pair:  atom from *this -- atom from other
        add to atoms in *this
        subtract from atoms in other
    }
    void calcNewSpeed() {
        apply non-harmonic components to speed
    }

    void postIteration(int iter, double startEnergy,
                       double energy) {
        transform Cartesian coordinates to normal
        harmonical part of dynamics
            (vibration, rotation, translation)
    }
};

int main(int argc, char **argv) {
    PPIWorld<Butadiyne> world(argc, argv, &cout);

    analyse harmonical part
    ifstream inFile(argv[1], "r");
    world.readParticles(inFile);
        //read initial states from file
    world.mainLoop(atoi(argv[2]), settings);
        //run argv[2] steps
}
```

FIGURE 3. Pseudocodes of particle classes `Argon` and `Butadiyne`, and programs `main` for LFV (left) and SISM (right). Differences are shown in italic.

The particle class is used in the template `PPIWorld` that is controlled by the program `main` given after the class.

SPLIT INTEGRATION SYMPLECTIC METHOD

The new split integration symplectic method (SISM) [8] was derived in terms of the Lie algebraic language. The formula

$$\mathbf{x}|_{t_0+\Delta t} = \exp(\Delta t \hat{L}_H)\mathbf{x}|_{t_0}, \tag{3}$$

where $\hat{L}_H$ is the Poisson bracket operator and $\mathbf{x} = (\mathbf{q}, \mathbf{p})$ is a vector in phase space composed of the coordinates and momenta of all particles, provides a way for integrating the Hamiltonian system in terms of Lie operators [9].

A typical model MD Hamiltonian [7] is

$$\begin{aligned} H = &\sum_i \frac{\mathbf{p}_i^2}{2m_i} + \sum_{\text{bonds}} k_b(b-b_0)^2 + \sum_{\text{angles}} k_\phi(\phi-\phi_0)^2 \\ &+ \sum_{\text{dihed}} k_\vartheta\big(1+\cos(n\vartheta-\delta)\big) + \sum_{i>j}\frac{e_i e_j}{r_{ij}} \\ &+ \sum_{i>j} 4\varepsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] \end{aligned} \tag{4}$$

where i and j run over all atoms, m_i is the mass of the ith atom, b_0, ϕ_0, and δ are reference values for bond lengths, angles, and dihedral angles, respectively, k_b, k_ϕ, and k_ϑ are corresponding force constants, e_i denotes the charge on the ith atom, r_{ij} is the distance between atoms i and j, and ε_{ij} and σ_{ij} are the corresponding constants of the Lennard-Jones potential.

The construction of an efficient algorithm rests with the ability to separate the Hamiltonian into parts which are themselves integrable, and also efficiently computable. Suppose that the Hamiltonian H is split into two parts:

$$H = H_0 + H_r \tag{5}$$

where

$$H_0 = H^{\text{harm}}(m_i, b_0, \phi_0, \delta, k_b, k_\phi, k_\vartheta) \tag{6}$$

and H^{harm} denotes the harmonic approximation.

Then the following approximation for $\mathbf{x}|_{t_0+\Delta t}$ can be used:

$$\mathbf{x}|_{t_0+\Delta t} \approx \exp\left(\frac{\Delta t}{2}\hat{L}_{H_0}\right) \exp(\Delta t \hat{L}_{H_r}) \exp\left(\frac{\Delta t}{2}\hat{L}_{H_0}\right) \mathbf{x}|_{t_0} \tag{7}$$

which prescribes how to propagate from one point in phase space to another. First, the system is propagated a half-step evolution with H_0, then a whole step with H_r, and finally another half step with H_0. This integration scheme was used as the basis for the development of SISM, a second-order symplectic integration algorithm for MD integration. The Hamiltonian H_0 represents the dynamically leading contribution, and depends only on the constant parameters of the simulation, and is resolved analytically. H_r represents the remaining part, and is resolved numerically. This separation of the potential function allows the analytical treatment of high-frequency terms in the Hamiltonian, which permits the SISM to employ up to an order-of-magnitude larger integration step size than can be used by other methods of the same order and complexity. To evaluate the parallel efficiency of SISM, the system of butadiyne molecules was used as a validation suite [10].

The pseudocode of the particle class `Butadiyne` used in the SISM simulation with the accompanying program `main` are given on the right side of Figure 3. Italic program code has to be rewritten if the parallel SISM program (right column in Fig. 3) is devised from the existing LFV simulation program (left column in Fig. 3).

RESULTS

The parallel program for argon simulation, designed with the program library `PPI`, was devised from existing sequential Fortran codes for argon simulators. A sequential version based on `PPI` was compared with Fortran counterparts with different simulation parameters, e.g., number of particles [11], in order to prove that the simulation

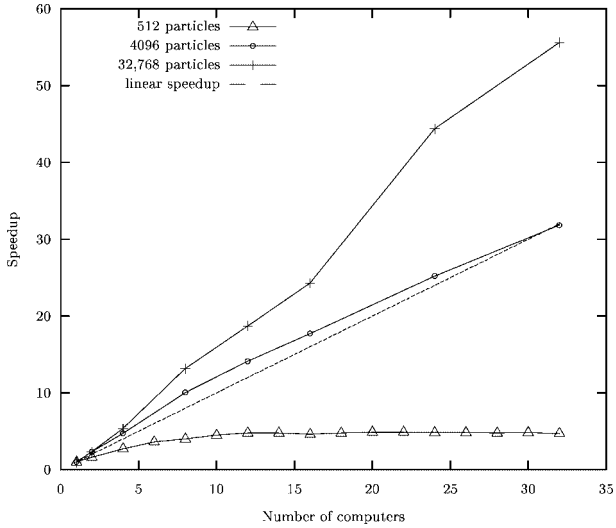

FIGURE 4. Speedup of parallel `PPI`-based MD integrations for various problem sizes.

results are the same, and that the execution time is also similar. The parallel `PPI` program was evaluated on a cluster of 32 AMD Athlon 750 [12] computing nodes, connected into a hypercube using Fast-Ethernet network cards and running Linux. The required ring topology was embedded into the hypercube. The speedup of the parallel `PPI`-based MD is given in Figure 4 for different numbers of particles as a function of the number of computers.

For 512 particles, the ratio between the calculation time and communication time is small; thus, the speedup is less than linear, particularly for a greater number of computers. The speedup remains approximately the same from 14 to 32 computers. Its maximum value 4.9 is reached with 22 computers.

For 4096 particles, we noticed superlinear speedup up to 24 computers because of the cache memory. The speedup on 32 computers is no more superlinear. Extrapolating the results, the number of computers that would finish the job in the shortest possible time would be about 160, with a speedup of about 50.

For 32,768 particles, the speedup is superlinear up to 32 computers, and also in this case, the cache memory is not large enough for all particles. The maximum superlinearity would be achieved with about 40 computers.

The results show that, at the same ratio of computing to communication speed as used in the test cluster, employing more than 128 particles per computer will give an approximately linear speedup.

More than 800 particles per computer will result in superlinear speedup. The shortest computation time for a given problem size will be achieved with about 26 particles per computer, but such a system is usually not cost effective. Parallel PPI-based MD simulation performances in all tests were almost identical to the custom-designed parallel simulation code.

EVALUATION OF RESULTS

The superlinearity can be explained by analyzing the effects of communication complexity and the use of cache memory for different numbers of particles n and computers p. In each simulation step, the total number of particles which pass through each computer is n, and does not depend on p. Furthermore, if the number of particles per computer n/p is small (because of large p), more short messages increase the communication time.

The program efficiency also depends on memory bandwidth. For large n/p, the computers must access the main memory more frequently because the cache memory is not large enough for all local and guest particles. As p increases, the amount of available cache memory increases as well, improving the cache hit rate, and consequently resulting in superlinear speedup. After the whole simulated system is stored in the cache memory, the speedup gradually falls to linear, which is seen in Figure 4 for the example of 4096 particles. However, for the 32,768 particles example, the superlinearity is significantly greater.

From our experiments follows that the speedup was sublinear for $n/p < 128$, superlinear for $200 < n/p < 15{,}000$, and linear elsewhere. These boundaries are approximate, and depend on the network hardware and cache memory architecture.

The parallel simulation results for SISM are not reported here. In accordance with theoretical comparisons of complexities [11], similar performances for LFV and SISM were expected. Even if the complexity of the SISM seems to be high compared to the LVF algorithm, it is not so since the most demanding part of the SISM, i.e., the calculation of nonbonding forces and energy, is almost the same as for the LVF. All of the extra work (coordinate transformations) is in the range of $O(n)$, and thus prevailed by nonbonding force and energy calculation. Cartesian as well as normal coordinates have to be held in memory, modestly increasing the memory requirements. One SISM step thus requires slightly greater calculation and communication time, particularly for systems smaller than 500 particles. For large systems of more than 200,000 particles, the time-step complexity is almost the same for both methods since the calculation of interactions prevails.

In comparison with the LFV, the range of superlinearity for SISM moves toward a smaller number of particles because of increased memory requirements. The range of sublinear speedups for SISM moves toward a larger number of particles because of the increased communication time.

Conclusion

In this work, a parallel library PPI for particle interactions on a ring network is described. It is shown that the proposed approach can be used on different areas of applications with particle interactions. The program code is transparent and robust. With adequate changes in the program code, an entirely different simulation experiment can be performed. Testing implementations of argon and butadiyne simulations are given, and the speedup for various problem sizes for LFV is analyzed. A greater speedup can be achieved if the number of particles per processor is sufficiently large to preserve a high ratio of calculation time/communication time, and to utilize the cache memory. The PPI-based algorithm performances are optimal with respect to calculation complexity.

Further work lies primarily in the development of the parallel library PPI for more complex and realistic systems of particles and methods (e.g., SISM) in order to give to a potential user a wider range of possible applications.

ACKNOWLEDGMENTS

This research was funded under grants J1-106-513, P1-0104-503, J1-0104-7346, and S41-104-001 from the Ministry of Science and Technology of the Republic of Slovenia.

References

1. Hockney, R. W.; Eastwood, J. W. Computer Simulation Using Particles; Institute of Physics Publishing, 1988.
2. Fox, G. C.; Johnson, M. A.; Lyzenga, G. A.; Otto, S. W.; Salmon, J. K.; Walker, D. W. Solving Problems on Concurrent Proceessors, Volume 1: General Techniques and Regular Problems; Prentice-Hall International, 1988.

3. Trobec, R.; Jerebic, I.; Janežič, D. Parallel Comput 1993, 19, 1029–1039.
4. Snir, M.; Otto, S.; Huss-Lederman, S.; Walker, D.; Dongarra, J. MPI: The Complete Reference; M.I.T. Press: Cambridge, MA, 1996.
5. Stroustrup, B. The C++ Programming Language; Addison-Wesley: Reading, MA, 1997.
6. Verlet, L. Phys Rev 1967, 159, 98–103.
7. Allen, M. P.; Tildesley, D. J. Computer Simulation of Liquids; Clarendon: Oxford, England, 1987.
8. Janežič, D.; Merzel, F. J Chem Inf Comput Sci 1995, 35, 321–326.
9. Sanz-Serna, J. M.; Calvo, M. P. Numerical Hamiltonian Problems; Chapman & Hall: London, England, 1994.
10. Janežič, D.; Praprotnik, M. Int J Quantum Chem 2001, 84, 2–12.
11. Trobec, R.; Merzel, F.; Janežič, D. J Chem Inf Comput Sci 1997, 37, 1055–1062.
12. Shimpi, A. L. AMD Athlon, http://www.anandtech.com, Aug. 1999.