exaMD User Guide

A scalable proto-app library for Molecular Dynamics using the Adaptive Midpoint method

Christopher Goodyer and Jon Gibson

Numerical Algorithms Group

Version 2.0 – October 2016

Contents

1 Overview 3

2 The Adaptive Midpoint method 3

3 Implementation 6

3.1 Domain Decomposition and Partitioning 6

3.2 The Midpoint Algorithm and Adaptivity 7

3.2.1 The Overlap Variant 8

3.3 MPI Communication 8

4 Installation and Usage 9

4.1 Building the Proto-app 9

4.2 Running the Proto-app Driver 9

4.3 Proto-app Driver Output 10

4.4 The Proto-app Library 11

4.4.1 exaMD_Init 11

4.4.2 exaMD_read_params 11

4.4.3 exaMD_domain_create 12

4.4.4 exaMD_particles_generate 12

4.4.5 exaMD_read_particles 12

4.4.6 exaMD_write_particles 12

4.4.7 exaMD_partition_domain 13

4.4.8 exaMD_setup_partitioned_grid 13

4.4.9 exaMD_partition_work 13

4.4.10 exaMD_setup_midpoint_interactions 13

4.4.11 exaMD_start_comms_posn 14

4.4.12 exaMD_forces_midpoint_all 14

4.4.13 exaMD_forces_midpoint_overlap 14

4.4.14 exaMD_wait_comms_forces 14

4.4.15 exaMD_Finalize 14

5 Performance 14

References 15

ACKNOWLEDGEMENTS

The Adaptive Midpoint method is implemented here, following the original idea and suggestion of Simone Melchionna, in order to investigate its potential use for Molecular Dynamics simulations in the presence of coarse-grained force fields and intrinsically sparse systems, such as occur in macromolecular suspensions and cellular environments. Thanks are due to Simone Melchionna and the other developers of the MUPHY simulation software for the base application around which this proto-app library has been built and for their assistance and guidance during its development.

This code has been developed as part of the European Union's Seventh framework Programme for research, technological development and demonstration under grant agreement No. 610741.

We acknowledge PRACE for awarding us access to resource MareNostrum based in Spain at Barcelona Supercomputing Center and resource Marconi based in Italy at Cineca.

1 Overview

The `exaMD` proto-app parallel library implements the particle interaction calculations of a Molecular Dynamics (MD) code using the Midpoint method. An example driver program (`exaMD.f90`) and cost function (`lennardjones_12` in `user_func.f90`) are provided to illustrate the use of the routines. The code is written in Fortran with MPI.

The Midpoint method is described in Section 2 and its implementation in `exaMD` in Section 3. Instructions on building and using the library are given in Section 4. Testing has been done up to 6,000 cores and good results achieved for cases with sufficient computational work. The performance is discussed in Section 5.

2 The Adaptive Midpoint method

Generally, in Molecular Dynamics the forces are calculated between all pairs of particles with a separation less than a given cutoff distance, R_c. Recently, new methods have been devised in order to optimize the search volume for particle interactions. These include the Tower-Plate method [1] and the Midpoint method [2], the latter having been extended in an adaptive sense in [3].

In the conventional Midpoint method, the domain is split into blocks of side $R_c/2$ so that distances and interactions need only be considered between neighbouring blocks. Subsequently, a search radius of $R_c/2$ is considered around every home block so that particles within a distance of R_c from each other can be found by searching from a point somewhere between the two. Force calculations are performed on the process that owns this midpoint block. The advantage of this method is that the search volume is $\frac{4}{3}\pi(\frac{R_c}{2})^3$ rather than $\frac{4}{3}\pi R_c{}^3$, a reduction by a factor of eight. This means that, on average, eight times fewer particle pairings need to be considered. This is balanced against sometimes needing to send more messages post-calculation, given that both particles in a calculation on one process may live on other processes.

When parallelised, with each process owning a subset of blocks, this method typically means that each process exchanges particle information with those processes responsible for the neighbouring domains. By considering the order in which interactions between particles in different blocks are calculated, it is possible to ensure that each interaction is only calculated once. Each block would only have to consider a subset of its neighbours, although if the blocks are on different processes then the result would have to be communicated to the one that doesn't perform the calculation.

The conventional Midpoint method is suboptimal when applied to sparse molecular systems, such as occur in coarse-grained Molecular Dynamics simulations, because sparsity causes an imbalance in the communication that impacts negatively on the parallel performance. The Adaptive Midpoint method specifically targets these situations by providing a dynamic and general way to determine the best communication pattern as molecules move in time [3]. The key idea is to perform an on-the-fly recalculation of the communication pattern as molecules move away from a starting position. Given the large difference in timescales between the fast local motion of atoms and the diffusive displacement of molecules as a whole, re-partitioning can be done sporadically as timestepping advances, thus taking only a negligible fraction of the total computing time (further details are found in [3]).

In our approach, the domain is divided into cubic blocks of width $R_c/2$ and then the blocks are partitioned amongst the processes. Typically, a process would have many blocks and would communicate with a number of other processes. Figure 1 shows the set of neighbouring blocks for both two and three dimensions, with the home block being number 5 in each case. Obviously, two particles in the same block would also have their midpoint in the block. However, when the particles are in different blocks (and so potentially also different processes), a calculation may be required to determine in which block the midpoint lies. In order to avoid this, a set of rules can be defined which specify which block will own a particular calculation. For the simpler 2D example, these rules are shown in Table 1. Note the following points.

- The main diagonal shows interactions between particles in the same block. These will only be evaluated on

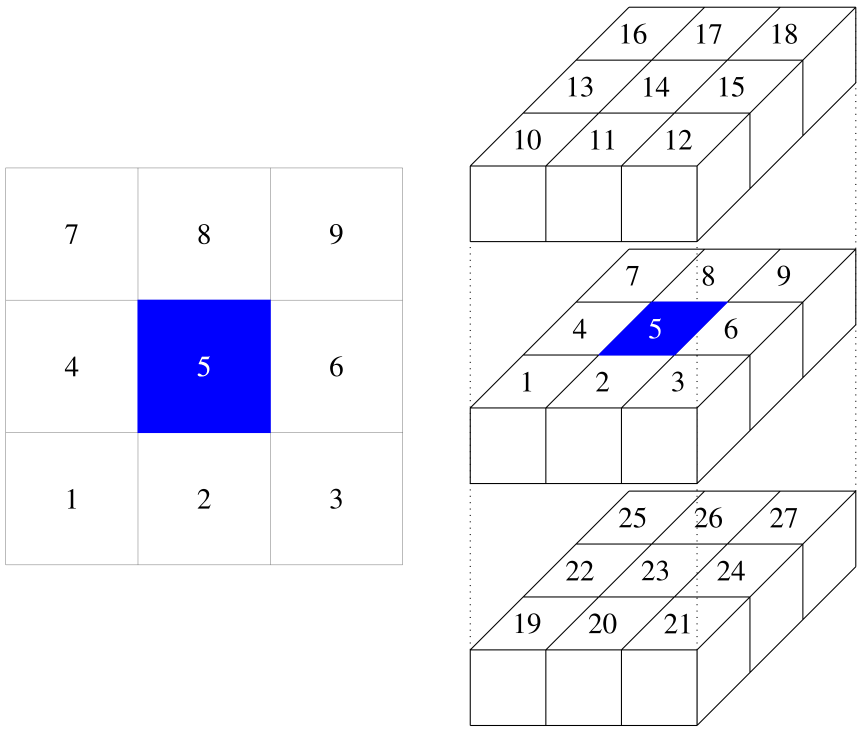

Figure 1: Schematic showing neighbouring blocks in (a) 2D and (b) 3D

Particle two	Particle one								
	1	2	3	4	5	6	7	8	9
1	✗	✗	✗	✗	✗	✗	✗	✗	✓
2		✗	✗	✗	✗	✗	✗	✓	✓
3			✗	✗	✗	✗	✓	✓	✗
4				✗	✗	✓	✗	✗	✓
5					✓	✓	✓	✓	✓
6						✗	✓	✗	✗
7							✗	✗	✗
8								✗	✗
9									✗

Table 1: Midpoint interactions in 2D. The same colour indicates equivalent operations.

P2	Particle one																									
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
2		✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
3			✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
4				✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
5					✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
6						✗	✓	✗	✗	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
7							✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
8								✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
9									✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
10										✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
11											✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✓	✓
12												✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗
13													✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
14														✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
15															✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗
16																✗	✗	✗	✗	✗	✓	✗	✓	✓	✗	✗	✗
17																	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗
18																		✗	✓	✓	✗	✓	✓	✗	✗	✗	✗
19																			✗	✗	✗	✗	✗	✗	✗	✗	✗
20																				✗	✗	✗	✗	✗	✗	✗	✗
21																					✗	✗	✗	✗	✗	✗	✗
22																						✗	✗	✗	✗	✗	✗
23																							✗	✗	✗	✗	✗
24																								✗	✗	✗	✗
25																									✗	✗	✗
26																										✗	✗
27																											✗

Table 2: Midpoint interactions in 3D

the home block, hence (5,5) is the only tick. All the colours along this diagonal are blue to indicate that they are all equivalent operations.

- The interactions below the main diagonal need not be evaluated, given the symmetry. Avoiding the repetition of calculations within a given block must also be considered.
- For particles in the home block (5), the only interactions to consider are those with the three blocks above (7, 8 and 9) and the one to the right (6). The interactions with the other four blocks will be calculated while those blocks are the home block. For example, the interaction 4↔5 is evaluated as 5↔6 when the home block is one to the left. This neighbouring relationship is replicated in the pairings 1↔2, 2↔3, 7↔8 and 8↔9. The colouring on the table is thus a red ticked cell at (5,6), with red crosses at (1,2), (2,3), (4,5), (7,8) and (8,9).
- Other equivalent pairs are similarly coloured to highlight which pairs need to be calculated on the home block.
- In total there are 13 pairs of blocks which require their interactions to be calculated on the home block, out of a possible 45.

When considering the 3D problem, also shown in Figure 1, there are obviously many more combinations of blocks to consider, although the principles are the same. The interactions that need calculating on any given home block are shown in Table 2. The table shows that only 63 pairs need to be considered out of a total of 378 potential interactions.

Algorithm 1 Pseudo-code showing how non-blocking communication allows overlap of computation and communication in a single force calculation step of the midpoint method.

```
 1: Decompose the domain into blocks (cubes of side length $R_c/2$).
 2: Generate the particle distribution (either randomly or read from file).
 3: Partition the domain between processes, i.e. map blocks to processes.
 4: Initiate sending of particle positions to processes with neighbouring blocks using non-blocking communication.
 5: for $block_{interior}$ = 1 to $n_{interior}$ do
 6:   if $block_{interior}$ = a multiple of 10% of $n_{interior}$ and all messages have newly arrived then
 7:     for $block_{boundary}$ = 1 to $n_{boundary}$ do
 8:       for permutation_index = 1 to 63 do
 9:         b1=permutation(permutation_index,1)
10:         b2=permutation(permutation_index,2)
11:         Compute interaction(b1,b2)
12:       end for
13:     end for
14:     Send forces back to neighbours using non-blocking communication.
15:   end if
16:   for permutation_index = 1 to 63 do
17:     b1=permutation(permutation_index,1)
18:     b2=permutation(permutation_index,2)
19:     Compute interaction(b1,b2)
20:   end for
21: end for
22: Complete receiving of forces.
```

3 Implementation

Algorithm 1 shows how a single force calculation step of the midpoint method has been implemented in the exaMD proto-app library. The first three steps of the algorithm are described in Section 3.1, domain decomposition and partitioning, with the remainder in the following section on the midpoint algorithm, which also includes a description of the overlap variant of the algorithm. Section 3.3 then describes the MPI communication within the code.

3.1 Domain Decomposition and Partitioning

In the first step of Algorithm 1, the global box is divided up into blocks of size $\frac{R_c}{2} \times \frac{R_c}{2} \times \frac{R_c}{2}$, with the three faces at the greatest distance from the origin having larger blocks, to match the global box boundary. If the global box is cubic with a side of length L and the block length is $d = R_c/2$, then there will be $\text{int}\left(\frac{L}{d}\right)$ blocks along each side of the box and the total number of blocks will be

$$n_b = \left[\text{int}\left(\frac{L}{d}\right)\right]^3$$

Along each side there will be

$$\text{int}\left(\frac{L}{d}\right) - 1 \quad \text{blocks of width} \quad d$$

$$\text{and } 1 \quad \text{block of width} \quad L - d\left[\text{int}\left(\frac{L}{d}\right) - 1\right]$$

5↔5	5↔6	5↔7	5↔8	5↔9	5↔10	5↔11	5↔12	5↔13	5↔14
5↔15	5↔16	5↔17	5↔18	4↔6	4↔9	4↔15	4↔18	6↔7	6↔10
6↔13	6↔16	3↔7	3↔8	3↔16	2↔8	2↔9	2↔16	2↔17	2↔18
1↔9	1↔18	7↔12	7↔15	8↔10	8↔11	8↔12	9↔10	10↔23	10↔27
11↔23	11↔25	11↔26	11↔27	12↔23	12↔25	13↔23	13↔24	13↔27	14↔23
15↔22	15↔23	15↔25	16↔21	16↔23	16↔24	17↔20	17↔21	17↔23	18↔19
18↔20	18↔22	18↔23							

Table 3: Block interactions that need calculating in 3D

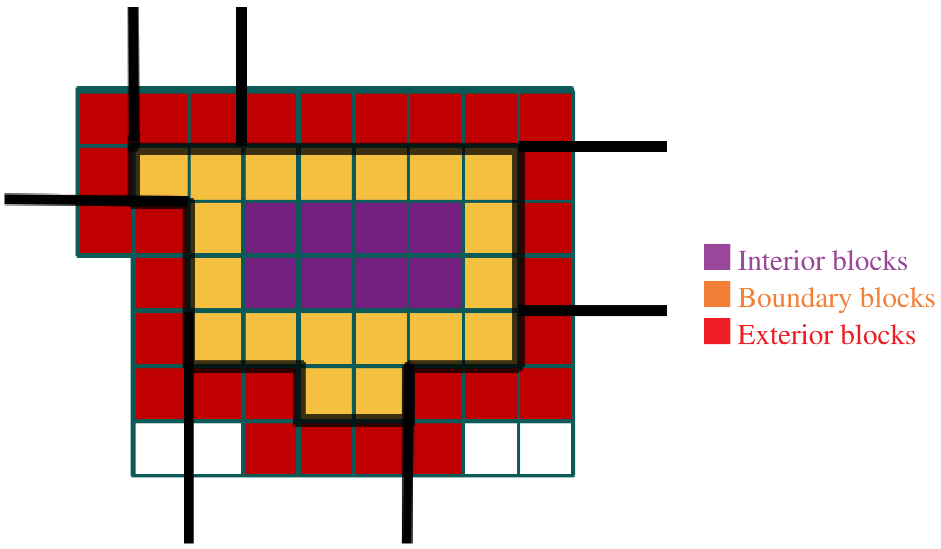

Figure 2: The different types of blocks on a process

The list of particles is then created, either by randomly generating positions within the global domain or by reading a previously-generated distribution from file. The particles are subject to the constraint that they must be a distance of at least 1.0 from all other particles, including over the periodic boundaries. This imposes an upper limit on the number of particles that can be placed in a domain of a given size. As each particle is generated, the block in which it is located is determined.

To help facilitate arbitrary process counts and high quality partitions, the package METIS [4] is used to partition the blocks amongst the processes (step 3 of Algorithm 1). METIS is software that performs graph partitioning based on multilevel recursive-bisection that is particularly efficient for sparse matrices. It creates contiguous partitions with minimal communication costs between neighbouring processes. In a full production code, ParMETIS [5] could be used instead to perform the partitioning in parallel. Alternatively, another partitioner could be used. The routine `exaMD_partition_domain` in `exaMD_partitioning.f90` could be adapted to call another partitioner or use a previously set-up partition. Regardless of which method is used, after partitioning all processes need to call `exaMD_setup_partitioned_grid` to set up the correct data structures.

3.2 The Midpoint Algorithm and Adaptivity

The code employs a number of look-up tables. Table 3 shows the "permutation list", extracted from the interactions shown in Table 2. Lists are also built up of the blocks owned by each process. These are subdivided into lists of *interior blocks* (i.e. with no neighbouring blocks on other processes), lists of *boundary blocks* (i.e. with at least one of its 26 neighbours on a different process) and lists of *exterior blocks* (i.e. blocks that need to be received from a neighbouring process). These are illustrated in Figure 2, where the solid black lines denote process boundaries. Individual blocks also have a list of the processes they need to communicate with.

Algorithm 2 Pseudo-code for the overlap variant of the midpoint method implementation.

```
4: Initiate sending of particle positions to processes with neighbouring blocks using non-blocking communication.
5: for $block_{interior}$ = 1 to $n_{interior}$ do
6:   if $block_{interior}$ = a multiple of 10% of $n_{interior}$ and some messages have newly arrived then
7:     for $block_{boundary}$ = 1 to $n_{boundary\_newly\_arrived\_messages}$ do
8:       for permutation_index = 1 to 63 do
9:         b1=permutation(permutation_index,1)
10:        b2=permutation(permutation_index,2)
11:        Compute interaction(b1,b2)
12:      end for
13:    end for
14:    Send forces back to neighbours using non-blocking communication.
15:  end if
16:  for permutation_index = 1 to 63 do
17:    b1=permutation(permutation_index,1)
18:    b2=permutation(permutation_index,2)
19:    Compute interaction(b1,b2)
20:  end for
21: end for
22: Complete receiving of forces.
```

With a view to writing efficient parallel code on large process counts, computation and communication were overlapped. This is achieved by initiating the sending of the positions (line 4 of Algorithm 1) before computing any interactions. Using the lists of block types, the code then computes the interactions of the interior blocks until all exterior blocks have been received. At this point, line 6 in Algorithm 1, all interactions for the boundary blocks can be calculated. Once these boundary blocks are computed, the relevant force information is sent back to the neighbouring processes (line 14) before completing the calculations on the interior blocks (lines 16-20).

The adaptive nature of the Midpoint method leverages the high quality partitioning created by METIS (or by other equivalent, graph-based partitioning methods), in order to perform on-the-fly recalculation of the communication pattern as molecules move away from their starting positions. The user can modify the basic scheme presented in `exaMD.f90` in order to call METIS at regular intervals.

3.2.1 The Overlap Variant

The overlap variant processes a given boundary block as soon as the messages that block requires have been received, rather than waiting for all that process's incoming messages to have been received and then processing all the boundary blocks. The intention is to increase the degree of overlap between computation and communication. This alternative approach is shown in Algorithm 2 (note that only lines 6 and 7 have changed from Algorithm 1 but showing the other lines makes it easier to understand).

3.3 MPI Communication

The code's MPI communication is made up of four distinct phases.

- After the initial distribution of the particles has been calculated, process 0 broadcasts the initial particle counts on every block to all processes. It then sends to each process all the particle positions for the blocks that process owns.

- At the start of the calculation phase, each process sends to the processes owning its exterior blocks, arrays of any particles positions that they require. They will then receive the particle positions required from the relevant processes. Note that it may be necessary to send data from the same block to multiple processes.
- At the end of the calculation phase, the forces calculated for particles in exterior blocks need communicating back to their owning processes. These forces then need adding to both the forces calculated locally and those from other processes.
- After the forces have been calculated, a global sum of their values and the square of their values is calculated to help verify the results. This is done using calls to `MPI_Reduce`.

An important design decision was to locally gather all information being sent to a particular process so that it could be sent in a single message. This packing and unpacking may make the code slightly more complicated but the latency of multiple messages could become a significant overhead otherwise.

4 Installation and Usage

4.1 Building the Proto-app

The file `exaMD/make.inc` is included by Makefiles in the `exaMD` and `exaMD/SRC` directories and should be the only file you need to edit. Set the variable `F90` in `make.inc` to point to your MPI Fortran compiler and `FLAGS` to the appropriate compiler flags. By default, the partitioner used is METIS. If this is not currently installed on your system, it can be downloaded from `http://glaros.dtc.umn.edu/gkhome/metis/metis/download`. The variable `METIS_DIR` in `make.inc` needs to be set to your top-level METIS directory.

To compile, simply run

```
% make
```

from the `exaMD` directory. This will create both an executable, `exaMD`, and a library, `LIB/libexaMD.a`.

If you want to compile the library and not the executable, then run

```
% make lib
```

To remove all object and module files, the library and executable, run

```
% make clean
```

4.2 Running the Proto-app Driver

The application should be run using the standard method for running MPI jobs on your system. For example, if this is using `mpirun`, then to launch a 16 process job you would use

```
% mpirun -np 16 ./exaMD
```

There are a number of flags that can be given to `exaMD`:

`-atoms`	the number of particles/atoms to use in the simulation (without this flag, a default value of 500,000 is used)
`-readfile`	load the particle positions from the file `exaMD_particles.dat`
`-writefile`	write the set of particle positions to the file `exaMD_particles.dat`
`-overlap`	use the overlap variant, described in Section 3.2.1

Therefore to run a 1024 process job with 16,000,000 particles using the overlap variant, you would use

```
% mpirun -np 1024 ./exaMD -atoms 16000000 -overlap
```

4.3 Proto-app Driver Output

The output to stdout for a typical run is shown below.

```
  % mpirun -np 8 ./exaMD -atoms 100000
 -----------------------------------------------------------------------
 The exaMD Proto-app Example Program
 -----------------------------------

 Number of atoms: 100000
 Simulation box size:  100.0  100.0  100.0
 Cut-off distance:  8.00
 Number of blocks in each dimension: 25 25 25
 Number of MPI processes: 8
 Floating-point precision: 8
 Overlap option: not selected
 -----------------------------------------------------------------------
 Number of particles generated:      10000
 Number of particles generated:      20000
 Number of particles generated:      30000
 Number of particles generated:      40000
 Number of particles generated:      50000
 Number of particles generated:      60000
 Number of particles generated:      70000
 Number of particles generated:      80000
 Number of particles generated:      90000
 Number of particles generated:     100000
 -----------------------------------------------------------------------
 Calling METIS

Timing Information -------------------------------------------------
 Multilevel:              0.476
     Coarsening:                  0.120
            Matching:                     0.056
            Contract:                     0.064
     Initial Partition:           0.200
     Uncoarsening:                0.156
          Refinement:                     0.088
          Projection:                     0.044
     Splitting:                   0.000
********************************************************************
 -----------------------------------------------------------------------
 Checks on force calculations
  SUM(fxx, fyy, fzz):   0.1E-13  -0.6E-13  -0.6E-12
  SUM(fxx^2, fyy^2, fzz^2):   0.8E+06   0.8E+06   0.8E+06

 Times from process 3
  Communication times (secs):  Pre    0.0015;  Post     0.0025
  Computation time (secs):    0.1631
  Total time (secs):    0.1672

  Stopping.... Done
```

The output begins with the values of the important simulation parameters. After that, a line is printed after the generation of every 10,000 particles. This is followed by the output from METIS. For a sufficiently large case, the output will pause at this point while the forces are calculated. Once completed, the global values of $SUM(fxx, fyy, fzz)$ and $SUM(fxx^2, fyy^2, fzz^2)$ are output to provide a quick check on the validity of the forces calculated. The first set of values should be close to round-off, typically less than 1×10^{-12} in double precision; the second set should be almost identical between different runs of the same case.

The final few lines give timing information. The computation time gives the time taken for the midpoint forces calculation, with the pre and post communication times being the times taken for the communication before and after this calculation, respectively. The "total time" is the combined time for this communication and computation (note that rounding differences may mean that its value differs slightly from the sum of the values output for communication and computation times, as is the case in the above example). The times are given for the process with the greatest total time.

If the code is compiled with the `-DPRINTALLFORCES` flag, it will then print out the values of $SUM(fxx^2, fyy^2, fzz^2)$ for each block. This can be useful during library development. For example, the forces from a serial run could be compared to those from a parallel one in the following way.

```
% mpirun -np 1 ./exaMD | sort -g > serial_out
% mpirun -np 8 ./exaMD | sort -g > parallel_out
% diff serial_out parallel_out
```

Note how the sort command is being used to order the output in terms of block number, which is printed at the start of each line before the force values. This is necessary because of the unordered nature of the output from parallel runs. A graphical file difference program, such as `tkdiff` or `xxdiff`, may be useful here if there are a large number of differences between the files.

4.4 The Proto-app Library

The proto-app library contains the following routines. An example of their use can be found in the `exaMD.f90` driver program. Note that all the library calls begin with an `exaMD_` prefix. Only the routines that a user is likely to call directly have been listed.

4.4.1 exaMD_Init

```
exaMD_Init(pid, noprocs, input_comm)
  Integer, Intent(out) :: pid, noprocs
  Integer, Intent(in) :: input_comm
```

Initialises MPI, if it hasn't been already. It creates a copy of the input MPI communicator, `input_comm`, for use within the library, called `COMM_EXAMD`. This is declared within the module `exaMD_mod`. The routine returns the number of processes, `noprocs`, and the MPI rank, `pid`, of each process. This is a collective call.

4.4.2 exaMD_read_params

```
exaMD_read_params(natms_tot)
  Integer, Intent(inout) :: natms_tot
```

Loads the values of `natms_tot` and `globalbox` (in module `exaMD_mod`) from a file in the current directory called `exaMD_particles.dat`. This should only be called by a single process. Note that all processes require the value of `globalbox` so this should be broadcast after calling this routine and before calling `exaMD_domain_create`, which uses its value.

4.4.3 EXAMD_DOMAIN_CREATE

```
exaMD_domain_create(block_length, nblocks, nblocks_xyz)
  Real(kind=PRC), Intent(in) :: block_length
  Integer, Intent(out) :: nblocks, nblocks_xyz(3)
```

Divides the domain into cubic blocks of side length `block_length`, which must be set to $R_c/2$ on input (step 1 of Algorithm 1). On output, `nblocks` is set to the total number of blocks and the array `numblocks_xyz` holds the numbers of blocks along the `x`, `y` and `z` dimensions.

4.4.4 EXAMD_PARTICLES_GENERATE

```
exaMD_particles_generate(particle_positions, natms_tot, ntot, &
                         first_particle, next_particle, nblocks, &
                         nblocks_xyz, block_length)
  Real(kind=PRC), Allocatable, Intent(inout) :: particle_positions(:,:)
  Integer, Allocatable, Intent(inout) :: ntot(:), first_particle(:)
  Integer, Allocatable, Intent(inout) :: next_particle(:)
  Integer, Intent(in) :: natms_tot, nblocks, nblocks_xyz(3)
  Real(kind=PRC), Intent(in) :: block_length
```

Called by a single process, this routine randomly generates particles within the domain and then determines which block the particle is in. If there is already a particle within a distance of 1.0 (the value of the parameter `PROXIMITY` in the module `exaMD_mod`), then it is discarded. The inputs `pid` and `noprocs` were returned by `exaMD_Init`. The inputs `nblocks` and `numblocks_xyz` are outputs from the routine `exaMD_domain_create`. The total number of atoms to generate is determined by the variable `natms_tot`, which should have been given a value in the user code. The output `ntot` is an array of size `nblocks` that holds the number of particles in each block. The output array `particle_positions` holds the positions of all the particles, in the order in which they were generated. The output `first_particle` is an array of size `nblocks` that gives the index of the first particle in each block within the `particle_positions` array. The output `next_particle` is an array of size `natms_tot` that gives the index of the next particle in the same block within the `particle_positions` array. These outputs are required by the routine `exaMD_partition_work`.

4.4.5 EXAMD_READ_PARTICLES

```
exaMD_read_particles(particle_positions, natms_tot, ntot, first_particle, &
                     next_particle, nblocks, nblocks_xyz, block_length)
  Real(kind=PRC), Allocatable, Intent(inout) :: particle_positions(:,:)
  Integer, Allocatable, Intent(inout) :: ntot(:), first_particle(:)
  Integer, Allocatable, Intent(inout) :: next_particle(:)
  Integer, Intent(in) :: natms_tot, nblocks, nblocks_xyz(3)
  Real(kind=PRC), Intent(in) :: block_length
```

Loads previously saved particle positions from a file in the current directory called `exaMD_particles.dat`. Note that the routine arguments and their meanings are identical to those of `exaMD_particles_generate`. This should only be called by a single process.

4.4.6 EXAMD_WRITE_PARTICLES

```
exaMD_write_particles(particle_positions, natms_tot)
```

```
  Real(kind=PRC), Allocatable, Intent(inout) :: particle_positions(:,:)
  Integer, Intent(in) :: natms_tot
```

Writes the set of particle positions to a file in the current directory called `exaMD_particles.dat`. This routine should only be called by a single process.

4.4.7 EXAMD_PARTITION_DOMAIN

```
exaMD_partition_domain(nblocks, nblocks_xyz, ntot, pid, noprocs, part)
  Integer, Intent(in) :: nblocks, nblocks_xyz(3), pid, noprocs
  Integer, Allocatable, Intent(in) :: ntot(:)
  Integer, Allocatable, Intent(inout) :: part(:,:,:)
```

A routine run on a single process to partitition the blocks amongst the processes using METIS. The number of atoms in the blocks are used as weights in the partitioning process. The only output is the 3D array `part`, which contains the partition information for each block. This is step 3 of Algorithm 1.

4.4.8 EXAMD_SETUP_PARTITIONED_GRID

```
exaMD_setup_partitioned_grid(part, nblocks, nblocks_xyz, pid)
  Integer, Intent(in) :: nblocks, nblocks_xyz(3), pid
  Integer, Allocatable, Intent(in) :: part(:,:,:)
```

Creates the lists of interior, boundary and exterior blocks, as well as a list of the processes each block has to communicate with. This information is held in the structure `grid`, declared in the module `exaMD_mod`. This routine should be called by each process. The input `part` is the output from the routine `exaMD_partition_domain`. It is therefore necessary to broadcast the values of this array to all processes before this routine is called.

4.4.9 EXAMD_PARTITION_WORK

```
exaMD_partition_work(particle_positions, ntot, first_particle, &
                     next_particle, nblocks, nblocks_xyz, pid, noprocs)
  Integer, Intent(in) :: nblocks, nblocks_xyz(3), pid, noprocs
  Integer, Allocatable, Intent(in) :: ntot(:), first_particle(:)
  Integer, Allocatable, Intent(in) :: next_particle(:)
  Real(kind=PRC), Allocatable, Intent(in) :: particle_positions(:,:)
```

This is a collective call. Particle information is sent from process 0 to the processes that own those particles.

4.4.10 EXAMD_SETUP_MIDPOINT_INTERACTIONS

```
exaMD_setup_midpoint_interactions()
```

Sets up the list specifying which of the interactions between different pairs of blocks are to be calculated on the home block, as shown in Table 3. This routine should be called by all processes since they will all need the list, which is declared within the module `exaMD_mod`.

4.4.11 EXAMD_START_COMMS_POSN

```
exaMD_start_comms_posn()
```

Initiates the communication of particle positions on boundary blocks to the neighbouring processes that require them (step 4 of Algorithm 1). Non-blocking sends and receives are posted. A non-blocking receive is also posted for receiving the forces back after their calculation.

4.4.12 EXAMD_FORCES_MIDPOINT_ALL

```
exaMD_forces_midpoint_all(user_func, cutoff, noprocs)
  Real(kind=PRC), Intent(in) :: cutoff
  Integer, Intent(in) :: noprocs
  External :: user_func
```

This routine implements lines 5 to 21 of Algorithm 1. The argument `user_func` is the name of a user-supplied function for calculating the forces between a pair of particles. The input `cutoff` is the interaction cut-off distance, R_c.

4.4.13 EXAMD_FORCES_MIDPOINT_OVERLAP

```
exaMD_forces_midpoint_overlap(user_func, cutoff, pid, noprocs)
  Real(kind=PRC), Intent(in) :: cutoff
  Integer, Intent(in) :: pid, noprocs
  External :: user_func
```

This routine implements lines 5 to 21 of Algorithm 2, the overlap variant. The argument `user_func` is the name of a user-supplied function for calculating the forces between a pair of particles. The input `cutoff` is the interaction cut-off distance, R_c.

4.4.14 EXAMD_WAIT_COMMS_FORCES

```
exaMD_wait_comms_forces(noprocs)
  Integer, Intent(in) :: noprocs
```

Waits for all the forces to have been received (line 22 of Algorithm 1).

4.4.15 EXAMD_FINALIZE

```
exaMD_Finalize(text)
  Character(len=*) :: text
```

Finalizes MPI, deallocates memory and exits the code. The input `text` will be printed to stdout by process 0.

5 PERFORMANCE

The code was intially tested on the Anselm supercomputer at IT4Innovations in the Czech Republic. Subsequently, bigger runs were carried out on MareNostrum at Barcelona Supercomputer Centre and Marconi at Cineca, both

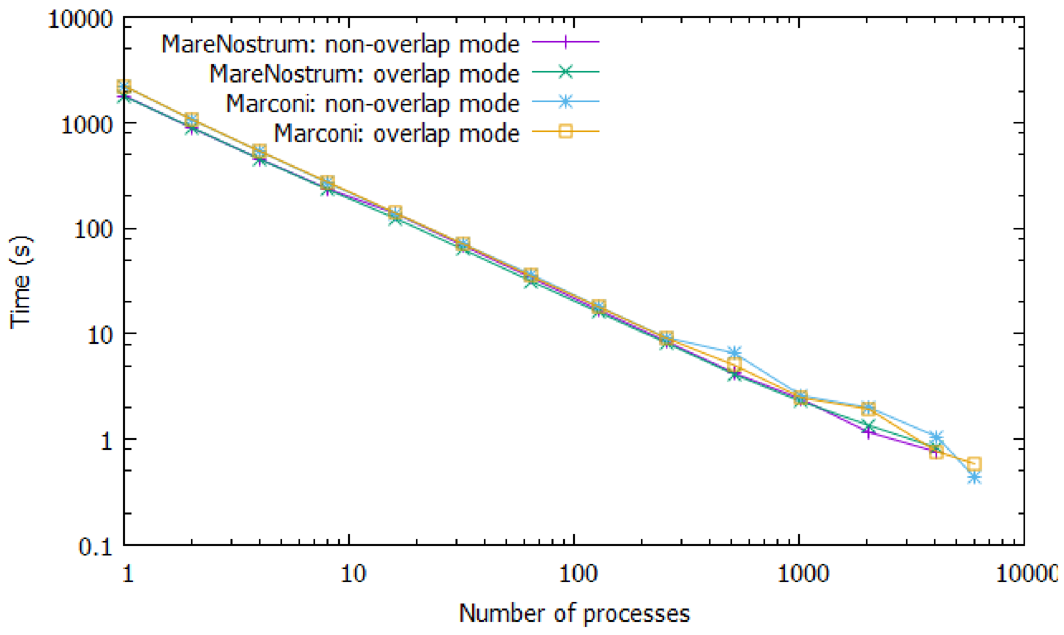

Figure 3: Time to solution for runs of 32,000,000 particles in a 400^3 domain with a cut-off radius of 8.0 on MareNostrum and Marconi.

made available through PRACE. Figure 3 shows the "total times" output by both machines (as explained in Section 4.3) for runs with and without the overlap option. The runs were performed with 32 million particles in a $400 \times 400 \times 400$ box with a cut-off distance of 8.0 for a range of core counts up to 4,096 cores (the runs on Marconi have an additional point at 6,000 cores because of its greater limit to what is permitted to run in the standard queues).

Figure 4 shows weak scaling performance on the same two machines, where both the number of particles and the size of the box are scaled with the core count. There are 15,625 particles per core and a box size of $50 \times 25 \times 25$ per core. The cut-off distance was always 8.0. One dimension of the box is doubled with each doubling of the core count, so at two cores the box size will be $50 \times 50 \times 25$, at four cores it will be $50 \times 50 \times 50$ and at eight cores it will be $100 \times 50 \times 50$. This means that for the biggest runs at 4,096 cores, the particle count was 64 million and the box size was $800 \times 400 \times 400$. Each point is the mean of five runs.

The results show that MareNostrum consistently performs the better of the two machines. Its scaling graphs are also smoother. Comparing the overlap variant with the standard method on either machine, it is not possible to draw any conclusions about one method being better. Both the strong and the weak scaling graphs show good scaling up to the largest jobs run. Having said that, a comparison with a non-midpoint method is required in order to draw any conclusions about whether and in what circumstances it would be a better approach. Hopefully, this proto-app would allow someone to make such a comparison for their particular problems.

References

[1] R.H. Larson, J.K. Salmon, R.O. Dror, M.M. Deneroff, C. Young, J.P. Grossman, Y. Shan, J.L. Klepeis, and D.E. Shaw. 'High-Throughput Pairwise Point Interactions in Anton, a Specialized Machine for Molecular Dynamics Simulation' IEEE 14th International Symposium on High Performance Computer Architecture. (2008)

[2] K.J. Bowers, E. Chow, Huageng Xu, R.O. Dror, M.P. Eastwood, B.A. Gregersen, J.L. Klepeis, I. Kolossvary, M.A. Moraes, F.D. Sacerdoti, J.K. Salmon, Yibing Shan and D.E. Shaw. 'Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters' Supercomputing 2006, Proceedings of the ACM/IEEE. (2006)

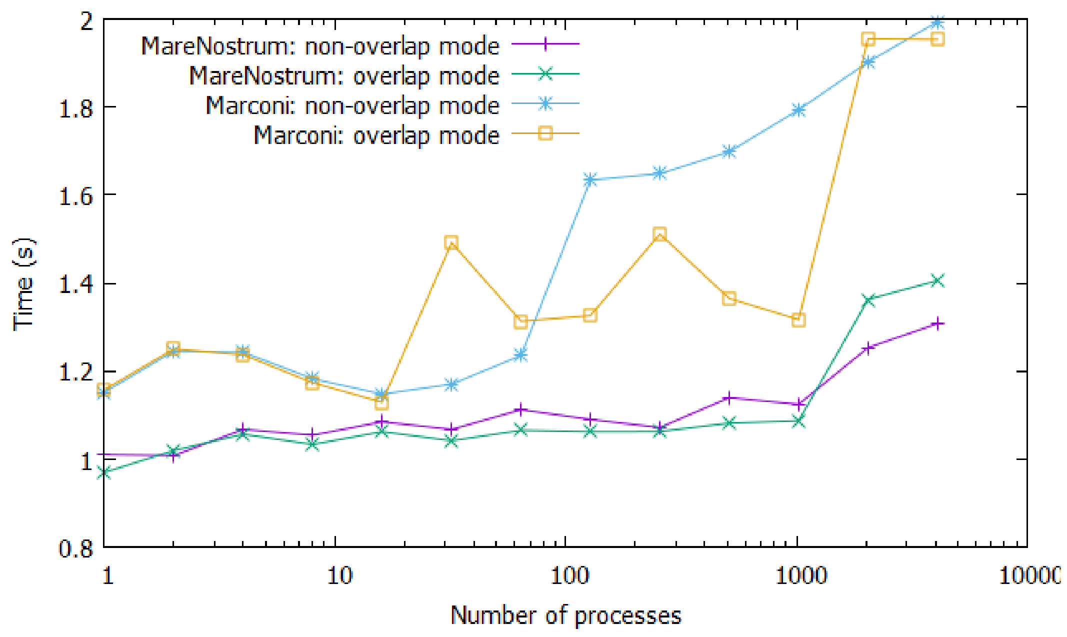

Figure 4: Time to solution for weak-scaling runs on MareNostrum and Marconi.

[3] F. Sterpone and S. Melchionna. ‘Adaptive Midpoint method for parallel Molecular Dynamics’ in preparation.

[4] A. Abou-Rjeili and G. Karypis. ‘Multilevel Algorithms for Partitioning Power-Law Graphs.’ IEEE International Parallel & Distributed Processing Symposium (IPDPS), (2006).

[5] K. Schloegel, G. Karypis, and V. Kumar. ‘Parallel static and dynamic multi-constraint graph partitioning.’ *Concurrency and Computation: Practice and Experience* **14**:3 pp 219–240. (2002)