GRAPE-6 User's Guide - Multi-Cluster version without monolithic configuration

Junichiro Makino

Department of Astronomy,
School of Sciences,
University of Tokyo,
Hongo 7-3–1, Bunkyo-Ku, Tokyo 113-0033, Japan
Phone : 81–3–5841–4276
Fax : 81–3–5841–7644
E-mail : makino@astron.s..u-tokyo.ac.jp

Version 0.0: March 29, 2000
Version 0.1: July 10, 2000
Version 0.1.1: July 12, 2000
Version 0.1.2: Sept 11, 2000
Version 0.1.3: Sept 28, 2000
Version 0.2: Feb 26, 2001
Version 0.3: May 8, 2001
Version 0.4: Sept 26, 2001
Version 0.5: Feb 3, 2002
Version 0.51: Feb 9, 2002
Version 0.6: Oct 2, 2002
Version 0.7: Mar 18, 2003
Version 0.71: Mar 22, 2003
Version 0.71a: Apr 1, 2003
Version 0.8: Dec 31, 2004

Abstract

I give the full description of the GRAPE-6 interface software package for multi-cluster system, but yet without the support for monolithic configuration.

Contents

1 TODO

- Support and documentation for simulator library
- Installation Guide and Export Kit
- What else?

2 Changes

2.1 Version 0.0

- March 29, 2000 — created
- Neighbor list not supported in software yet.

2.2 Version 0.1

- July 10, 2000
- Neighbor list support added (Not much tested yet...)

2.3 Version 0.1.1

- July 12, 2000
- `g6_npipes` documented.
- a number of small corrections.

2.4 Version 0.1.2

- Sept 11, 2000
- Sections for local information added
- a number of small corrections.

2.5 Version 0.1.3

- Sept 28, 2000
- Sections for local information updated

2.6 Version 0.2

- Feb 26, 2001
- descriptions for `g6_reset` and `g6_reset_fofpga` added
- example replaced to show the use of the firsthalf/lasthalf pair.

2.7 Version 0.3

- May 8, 2001
- "Getting started" section added.
- Reference section reorganized.
- As of May 8 2001, NOT consistent with actually installed library yet.
- `g6_set_calculate_accel_scaling_mode` added.

2.8 Version 0.4

- September 26, 2001
- Two BUG descriptions added.

2.9 Version 0.5

- Feb 3, 2002
- Athlon-specific informations added.

2.10 Version 0.51

- Feb 9, 2002
- Description of the return values of neighbour list-related functions extended.

2.11 Version 0.6

- Oct 2, 2002
- Description for library update for timesharing control in section **??**

2.12 Version 0.7

- Mar 18, 2003
- Description of `g6_set_overflow_flag_test_mode`.

2.13 Version 0.71

- Mar 22, 2003
- Description of `g6_set_neighbour_list_sort_mode` added.
- Apr 1, 2003
- A typo in function name corrected (Thanks to T. Saito)

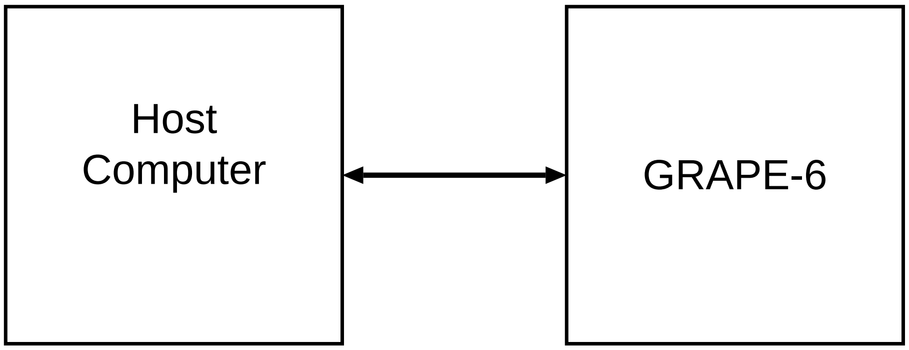

Figure 1: Basic Single-Cluster configuration of GRAPE-6

2.14 Version 0.8

- Dec 31, 2004
- Descriptions of `g6calc_firsthalf0`, `g6calc_lasthalf0`, `g6calc_lasthalf0a`, `g6_getnjmax`.

2.15 Version 0.81

- Jan 31, 2005
- Descriptions for the behaviour of `nj` added to FAQs section.

3 Overview of GRAPE-6 and its operating principles

GRAPE-6 is the successor of GRAPE-4, designed for high-accuracy integration of gravitational N-body system using the individual timestep and Hermite scheme. It works as a backend processor, connected to a host computer through PCI interface. Thus, from the viewpoint of a user, GRAPE-6 system (single cluster) looks like that in figure 1.

In the following, I first describe what is calculated how, first on this simple (single cluster, or SC) configuration. What GRAPE-6 SC calculates is the forces and their time derivative on 48 particles, from all particles loaded into its memory.

To be more precise, GRAPE-6 calculates the following

$$\mathbf{a}_i = \sum_j Gm_j \frac{\mathbf{r}_{ij}}{(r_{ij}^2+\epsilon^2)^{3/2}} \tag{1}$$

$$\dot{\mathbf{a}}_i = \sum_j Gm_j \left[\frac{\mathbf{v}_{ij}}{(r_{ij}^2+\epsilon^2)^{3/2}} - \frac{3(\mathbf{v}_{ij}\cdot\mathbf{r}_{ij})\mathbf{r}_{ij}}{(r_{ij}^2+\epsilon^2)^{5/2}}\right], \tag{2}$$

$$\phi_i = \sum_j Gm_j \frac{1}{(r_{ij}^2+\epsilon^2)^{1/2}}, \tag{3}$$

where

$$\mathbf{r}_{ij} = \mathbf{x}_{p,j} - \mathbf{x}_i, \tag{4}$$

$$\mathbf{v}_{ij} = \mathbf{v}_{p,j} - \mathbf{v}_i. \tag{5}$$

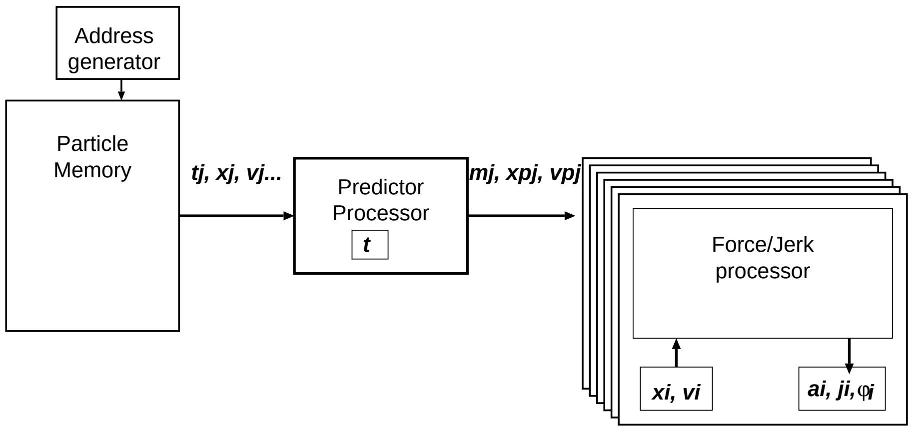

Figure 2: GRAPE-6 SC from the application viewpoint

Here, $\mathbf{x}_i$, $\mathbf{v}_i$, $\mathbf{a}_i$, $\dot{\mathbf{a}}_i$ are the position, velocity, acceleration, time derivative of acceleration of particle i, G is the gravitational constant and m_j is the mass of particle j. With GRAPE-6, G is fixed to unity. The parameter ϵ is the usual plummer softening parameter.

The position and velocity of particle j have additional suffix p to denote they are "predicted" values at time t using the following formulae:

$$\Delta t = t - t_j \tag{6}$$

$$\mathbf{x}_p = \frac{\Delta t^4}{24}\mathbf{a}_0^{(2)} + \frac{\Delta t^3}{6}\dot{\mathbf{a}}_0 + \frac{\Delta t^2}{2}\mathbf{a}_0 + \Delta t \mathbf{v}_0 + \mathbf{x}_0 \tag{7}$$

$$\mathbf{v}_p = \frac{\Delta t^3}{6}\mathbf{a}_0^{(2)} + \frac{\Delta t^2}{2}\dot{\mathbf{a}}_0 + \Delta t \mathbf{a}_0 + \mathbf{v}_0, \tag{8}$$

Here, we dropped the subscript j for clarity. Position $\mathbf{x}_p$ and velocity $\mathbf{v}_p$ are the predicted values, at time t, $\mathbf{x}_0$, $\mathbf{v}_0$, $\mathbf{a}_0$, $\dot{\mathbf{a}}_0$ $\mathbf{a}_0^{(2)}$ are true values of position etc at time t_j.

Thus, from the viewpoint of the host computer (and the application program on the host), the internal structure of GRAPE-6 SC looks like that in figure 2.

If you have multiple clusters connected to a single host, they work just completely independently. In all library functions that actually communicate with GRAPE-6 hardware, you simply specify the identity of the cluster as an argument.

4 Getting started

As described in the previous section, from the point of the view of the host computer, a GRAPE-6 hardware consists of three components: memory, pipeline processors and control logic. The memory stores the data of particles which exert the force. pipeline processors calculate the force on particles which they received from the host, and send the calculated force back to the host computer. In the following, I describe how the application programmer can actually use GRAPE-6 hardware.

4.1 Shared timestep

Though GRAPE-6 is designed for individual (block) timestep, many applications use it just with shared timestep. In addition, shared timestep is simpler. So we start with the shared timestep algorithm. Also, for simplicity we discuss only the direct summation method.

A typical sequence of using GRAPE-6 hardware with shared timestep algorithm is the following.

1. acquire the GRAPE-6 hardware.
2. set coordinate scaling etc.
3. write particle data to GRAPE-6 memory.
4. repeat the steps 5 to 8 until forces on all particles are obtained
5. send 48 particles (48 is the logical number of pipelines on GRAPE-6) to GRAPE-6 force calculation pipelines and let them start calculation
6. wait until calculation ends and force are returned
7. If GRAPE-6 returns hardware error, go back to step 1 and start over again.
8. If the application needs the list of the neighbours, let GRAPE-6 sends back the neighbor lists and receive the data.
9. go back to step 4
10. release the GRAPE-6 hardware

To acquire the GRAPE-6 hardware, one call `g6_open(clusterid)`. The argument, `clusterid` indicates which of the GRAPE-6 cluster is to be used, when multiple GRAPE-6 hardwares are connected to a single host. In most of sites, only one hardware is connected, and in this case you just supply 0 as argument. It is probably a good idea to make this argument a global variable within program which you can set through command line option or something like that.

If someone else is using the requested hardware, this routine put the process into sleep. It is automatically waken up when the GRAPE-6 hardware becomes available.

After you get GRAPE-6, the first thing you need to do is to set the scaling. GRAPE-6 stores both position and time with fixed point format. So you need to specify the scaling so that the range of time and distance used in the application program is correctly expressed within GRAPE-6. These scaling factors are used by other library routines to convert the data to and from GRAPE-6 internal format.

Time is not used with shared timestep, so we start with the position scaling.

Position scaling is set by `g6_set_xunit(xunit)`. The integer argument `xunit` gives the number of bits used to express the fractional part. The default is 51, which should be fine if the application program is written using nondimensional units.

Let me give an example. If you do the cosmological simulation which cover several Gpcs and you use parsecs as unit, you need around 33 bits to express the integral part. The bit length for position is 64. So give $64 - 33 = 31$, or something a bit smaller like 28 or so, as `xunit`.

Beware that no error check is performed for overflow or underflow. It's application programmer's responsibility to provide a proper scaling.

Now that we specified the scaling. W set particle data to GRAPE-6 memory. This is achieved by calling

```
int g6_set_j_particle_mxonly(int clusterid,
                             int address,
                             int index,
```

```
                    double mass,
                    double x[3])
```

Args `address` and `index` probably need further explanation. `Address` specifies the (logical) location within GRAPE-6 memory to which this particle is stored. If you send `n` particles to the hardware, `address` must be in the range of $[0, n]$ and should not overlap. If you send the data to same `address` twice, the former data is simply overwritten. If you do not write to some address `j`, the data in that location is of course garbage. GRAPE-6 always calculates the force from particles in consecutive addresses. So particle data in the specified range must all be valid.

`Index` is a new feature of GRAPE-6, which was not used in any of previous GRAPE hardwares. This is an integer value assigned to each particle data, which is used for the following two purposes.

- The neighbor list hardware stores this index, not the memory address in hardware.
- Each pipeline use this index to avoid self interaction.

With simple direct summation, this is not particularly useful since you almost always set index equal to address. However, if you use treecode, address are not same as the identity of the particles in your code.

The capability to avoid self-interaction is only important in the case of the individual timestep. So we do not discuss it further here.

After you store all necessary particle data to the memory, you can now let the GRAPE-6 hardware actually start calculation. From the point of view of an application, a GRAPE-6 always look like a collection of 48 pipelines. So you send one particle to each pipeline and let all of them calculate the force from particles in the memory.

To achieve this, you call

```
void g6calc_firsthalf(int clusterid,
                      int nj,
                      int ni,
                      int index[],
                      double xi[][3],
                      double vi[][3],
                      double fold[][3],
                      double j6old[][3],
                      double phiold[],
                      double eps2,
                      double h2[])
```

Here, `nj` is the number of particles stored in GRAPE-6 memory. `Ni` is the number of particles you want to send, which should not exceed 48. Other args are the actual data of particles to be sent. `Index` is here to be used to avoid self-interaction. `Xi` and `vi` are position and velocity, `eps2` and `h2` are the softening and the radius of the neighbor sphere (both squared). For convenience, `eps2` is a scalar value, though the hardware can set different values to each pipeline.

Here, one important thing is that you need to send `fold`, `j6old` and `phiold`, which are used to determine the internal scaling for force, jerk and potential for each pipeline.

GRAPE-6, unlike GRAPE-4, accumulates the result in fixed point format, which greatly reduced the complexity of the overall design of hardware. Unfortunately, this simplification does not come free. In order to avoid overflow or underflow, GRAPE-6 hardware should know beforehand roughly how large is the result. It (actually the library functions) determines the scaling accordingly for each pipeline. `Fold` and other two args are just used to determine the scaling.

Except for the first timestep at which you do not know what the force in the previous timestep is, you can just supply the force in the previous timestep, since it would not change by a large factor in one timestep.

For the first timestep, you need to give some "educated guess" for the magnitude of the force. For example, if you are working with the standard units ($M = 1$ etc), to supply order-of-unity values would be fine.

This `firsthalf` routine send the data to GRAPE-6 and let it start calculation.

The next step is to wait GRAPE-6 to finish the calculation and receive the data. This is achieved by calling `lasthalf` routine, which takes almost the same arguments as `firsthalf`. When called, it first checks if the calculation is completed, and if not, it just waits until the calculation finishes. Then it receives the data sent from GRAPE-6, performs the necessary scaling, and returns the results.

If the overflow occurs, this `lasthalf` routine tries to change the scaling and let GRAPE-6 do the calculation again. This is the reason why positions and velocities are still passed to this routine.

Note that, at least at the time of this writing, GRAPE-6 hardware is not 100% reliable. The library routine detects various kind of hardware error. If `lasfhalf` returns non-zero value, that means it detected some hardware error which is not corrected. At this point, the application program should reinitialize the hardware and start over. The routine to reinitialized GRAPE-6 is `g6_reinitialize(clusterid)`. It resets the GRAPE-6 hardware, releases it, and reattachs it. Thus, somebody else might use the GRAPE-6 hardware. Therefore, the content of the GRAPE-6 memory may have been changed after your call to `g6_reinitialize(clusterid)`, and you have to send particles to the memory again.

If `lasthalf` returns zero, you can proceed to calculate the forces on next set of 48 particles, or you can retrieve the neighbor list if you need.

You read the neighbor list in two steps. First, you call `g6_read_neighbour_list`, which just transfer the content of the hardware neighbour memory to internal storage of the library. Then, you call `g6_get_neighbour_list` to obtain the list for a specified particle. In case of the overflow of the list, the `read` function returns 1. The recovery from overflow will be discussed in more detail in section 8

4.2 Individual (block) timestep

A typical sequence of using GRAPE-6 hardware with block timestep algorithm is the following.

1. acquire the GRAPE-6 hardware.
2. set coordinate scaling etc.
3. write all particle data to GRAPE-6 memory.
4. find the particles in the current blockstep.
5. repeat the steps 8 to 13 until you get the forces on all particles in the current block
6. send 48 particles (48 is the logical number of pipelines on GRAPE-6) to GRAPE-6 force calculation pipelines and let them start calculation
7. wait until calculation ends and force are returned

8. If GRAPE-6 returns hardware error, go back to step 1 and start over again.
9. If the application needs the list of the neighbours, let GRAPE-6 sends back the neighbour lists and receive the data.
10. go back to step if there are still more particles in the current block
11. update the orbit, time, timestep etc.
12. send updated particles to GRAPE-6 memory.
13. If you have not reached the time to stop, go back to step 4.
14. release the GRAPE-6 hardware

One can see there is not really much change. The first two steps are the same. To write data to GRAPE-6 memory, now you need to specify the full predictor data. So you use `g6_set_j_particle`, which accepts all the necessary data for predictor.

The force calculation, neighbour list read, and error recovery procedures are the same as those in the case of shared timestep.

The final small difference from the shared-timestep version is that at the end of the each blockstep, you send the particles which are integrated to GRAPE-6 memory. You need not send all particles at the beginning of the each timestep. You only send the updated ones at the end.

4.3 Athlon and JP buffering

The AMD Athlon processor offers performance significantly better than what offered by Intel Pentium X, in particular with GCC compilers. However, almost all existing motherboards for AMD Athlon processors share one common problem: The bandwidth achieved with PCI PIO access is very low, even though the DEC Alpha box, which uses exactly the same chipset (AMD 750/760), offers very good performance.

In order to get reasonable performance on AMD, we need to activate DMA transfer for all three major data transfers: sending i- and j-particles and getting result. DMA is by default used to get data on x86 box, since Intel processors are also slow in PIO read. However, since PIO write is very fast on Intel processors, DMA is not used by default for sending particles.

To use DMA in sending particles, one need to do the followings:

- Get the EPROM and libraries newer than DEC 2001.
- Make sure the configuration file contains the lines:

  ```
  JPSPACE 10
  IJPDMA 1
  ```

  The number after keyword `JPSPACE` may be different from 10 after installation. If so, DO NOT CHANGE THAT VALUE.
- With the above change, DMA is used to send i-particles, but to send j-particles using DMA, you need to modify your code. Before calling `g6_set_j_particle`, call, just once,

  ```
  int g6_initialize_jp_buffer(int clusterid, int size)
  ```

The second number is the size of the buffer, which should be something reasonably large (10000 or so should be okay). With this function called, `g6_set_j_particle` just set the data to the internal buffer on the host memory. When the buffer becomes full, it is automatically flushed. However, before you calculate the force using firsthalf/lasthalf pair, you need to explicitly call the function to flush the buffer:

```
int g6_flush_jp_buffer(int  clusterid)
```

so that all j-particle data are actually sent to GRAPE-6.

5 Reference Manual for subroutines

All subroutines are written in C language. However, for simple and unified treatment of Fortran and C application programs, most of routines are written in the form which is directly callable from both Fortran and C. **HOWEVER, THIS DUAL FORM IS NOT YET AVAILABLE ON THE BABY-GRAPE6 LIBRARY.** On the BABY-6, you need to use the Fortran interface even if you use C/C++. This means that all functions have trailing underscore and all arguments are pointers.

5.1 Overview

Initialization routines

- g6_open: acquires the GRAPE-6 hardware.
- g6_close: releases the GRAPE-6 hardware.
- g6_reinitialize: reinitialize the GRAPE-6 hardware in the case of hardware error.
- g6_initialize_jp_buffer: Initialize the DMA work area for sending j-particles.

Scaling

- g6_set_tunit: Specifies the binary point for time.
- g6_set_xunit: Specifies the binary point for position.
- g6_set_overflow_flag_test_mode: Specifies the test mode for result overflow.
- g6_get_overflow_flag_test_mode: Inquires the current test mode for result overflow.

Sending data to memory

- g6_set_j_particle: sends one j-particle data to memory.
- g6_set_j_particle_mxonly: sends one j-particle data (pos
- g6_flush_jp_buffer: flush the DMA buffer for j-particles. and mass only) to memory.

Setting current time for individual timestep algorithm

- g6_set_ti; sets the current time for prediction.

Force calculation

- g6calc_firsthalf: sets scales for i particles, sends i particles, sets the numbers of i- and j-particles, and starts the calculation on GRAPE-6.
- g6calc_lasthalf: waits GRAPE-6 to finish calculation and receives the results.
- g6calc_lasthalf2: similar to the above but get the indices for the nearest neighbour as well.

Neighbour list

- g6_read_neighbour_list: stores the content of the hardware neighbor list to host memory.
- g6_get_neighbour_list: retrieves the neighbour list for one particle.
- g6_set_neighbour_list_sort_mode: set the sort flag for the neighbour list.
- g6_get_neighbour_list_sort_mode: returns the sort mode.

High-level function for shared-timestep algorithm

- calculate_accel_by_grape6_separate_trial_noopen: simple interface for shared-timestep code.
- calculate_accel_by_grape6_noopen: similar to the above but with more stricr error checks.

Low-level functions ... (you need not use these stuff... Well, then why are they listed here?)

- g6_reset: reset the GRAPE-6 hardware.
- g6_reset_fofpga: force reinitialization of the GRAPE-6 hardware.
- g6_set_nip: sets the number of i-particles.
- g6_set_njp: sets the number of j-particles.
- g6_set_i_particle_scales_from_real_value: sets scalings for force calculation.
- g6_adjust_ip_scales: adjusts the scaling in cases of overflow.
- g6_set_i_particle: sends one i particle data.
- g6_get_force: receives one force.
- g6_get_force_etc: similar to the above but returns more.
- g6_guestimate_acc_etc: Internal function. Listed for completeness only.
- g6_set_calculate_accel_scaling_mode: More internal function...

Note that functions `g6_set_nip` through `g6_get_force_etc` are called internally from `g6calc_firsthalf` and `g6calc_lasthalf`, and therefore one need not to use these routines, unless one wants direct access to the low-level functionalities of the GRAPE-6 hardware.

5.2 Initialization

5.2.1 g6_open

```
int g6_open(int clusterid)

subroutine g6_open(clusterid)
integer clusterid
```

Initializes the GRAPE-6 hardware and interface package for the specified cluster. If multiple clusters are available on the system, they are numbered from zero. The function which returns the available number of clusters is not implemented yet.... If someone else is using the requested GRAPE-6 cluster, it put the process in the queue and sleep, using the lockf(2) system call.

Note that currently this function takes several seconds to complete when called for the first time.

5.2.2 g6_close

```
int g6_close(int clusterid)
subroutine  g6_close(clusterid)
```

This function releases the specified GRAPE-6 cluster and allows other users to acquire it.

5.2.3 g6_reinitialize

```
int g6_reinitialize(int clusterid)
subroutine  g6_reinitialize(clusterid)
```

This function performs a hard reset on the GRAPE-6 hardware. The user need to reload all data to GRAPE-6 after this function is called.

5.2.4 g6_initialize_jp_buffer

```
int g6_initialize_jp_buffer(int clusterid, int size)
subroutine  g6_initialize_jp_buffer(clusterid)
```

This function initialized the DMA buffer for sending j-particles. The size argument should be something reasonably large, around 10^4 or so. Note that DMA for sending j-particles is only used when specified in the configuration file. However, the buffer itself is used if this function is called even when DMA is not used. In such cases, the content of the buffer is sent to GRAPE using PIO write.

5.3 Scaling

5.3.1 g6_set_tunit

```
void g6_set_tunit(int newtunit)

subroutine g6_set_tunit(newtunit)
integer newtunit
```

GRAPE-6 handles the time for predictor in 64 bit fixed point format. Thus, one need to specify where to put the binary point. The argument `newtunit` gives the point of the binary point counted from LSB. Thus, the value zero means the time expressed in GRAPE-6 is truncated to integral values, while, say, 50 means the resolution is 2^{-50}. The default value is 51.

What value should be used depends on the system of units that the application uses. If one uses the "standard" or so-called Heggie units, the default value should be fine, but one must be careful that the time would not overflow. With the default value, there are only 13 bits available above the binary point. Also, current implementation of the library software does not use the uppermost 2 bits. So the number of bits available above the binary point is $(62 - \texttt{tunit})$, which is, with default value, 2048. If the simulation covers the time much longer than 10^3, the default should be changed.

If you are using GRAPE for, say, galactic simulation and use real physical unit such as Myr as the time unit, you might want to use smaller value for `newtunit` than the default, since the default does not cover the Hubble time, and the resolution is too high (around 1 milliseconds).

When time unit is changed through the call to this function, the content of the GRAPE-6 memory becomes inconsistent with the time unit, and if individual timestep is used, all "j-particles" should be resent.

5.3.2 g6_set_xunit

```
void g6_set_xunit(int newxunit)

subroutine g6_set_xunit(newxunit)
integer newxunit
```

Similar to `g6_set_tunit`, but gives the binary point for space coordinate. The default is again 51, which is good if the size of the system is order unity. If your system is much larger or much smaller, you should change the default value accordingly.

5.3.3 g6_set_overflow_flag_test_mode

```
void g6_set_overflow_flag_test_mode(int force_test_mode,
    int jerk_test_mode,
    int pot_test_mode);
subroutine g6_set_overflow_flag_test_mode(force_test_mode,
          jerk_test_mode,
          pot_test_mode)
integer force_test_mode, jerk_test_mode, pot_test_mode
```

This function specifies which overflow flags are checked by the library. To let the library test the overflow for force, you set `force_test_mode` to one, and to ignore overflow, set it to zero.

The most likely use of this is to set `jerk_test_mode` to zero if you do not use it for time integration (for example, if you use leapfrog).

5.3.4 g6_get_overflow_flag_test_mode

```
void g6_get_overflow_flag_test_mode(int *force_test_mode,
    int *jerk_test_mode,
    int *pot_test_mode);
```

```
subroutine g6_get_overflow_flag_test_mode(force_test_mode,
                    jerk_test_mode,
          pot_test_mode)
integer force_test_mode, jerk_test_mode, pot_test_mode
```

This function returns the current values of the flags to test overflow. Initially all three flags are set to one.

5.4 Sending data to memory

5.4.1 g6_set_j_particle

```
int g6_set_j_particle(int  clusterid,
                      int address,
                      int index,
                      double tj, /* particle time */
                      double dtj, /* particle time */
                      double mass,
                      double a2by18[3], /* a2dot divided by 18 */
                      double a1by6[3], /* a1dot divided by 6 */
                      double aby2[3], /* a divided by 2 */
                      double v[3], /* velocity */
                      double x[3] /* position */)

integer function g6_set_j_particle(clusterid, address, index, tj, dtj, mass,
                       a2by18, a1by6, aby2, v, double x)
integer clusterid, address, index
double precision  tj, dtj, mass, a2by18(3), a1by6(3), aby2(3), v(3), x(3)
```

This function sends the so-called j-particle data to the memory unit of GRAPE-6. `x`, `v`, `aby2`, `a1by6`, `a2by18` are position, velocity, half of the acceleration, 1/6 of the first time derivative of the acceleration, and 1/18 of the second time derivative. These must all be the arrays of size three. `tj`, `dtj` and `mass` are the time, timestep and mass of the particle. `address` is the location in the GRAPE-6 memory to store the particle, starting from index 0, and `index` is the identifier for the particle itself, which may be different from `address` above.

The time and timestep must be acceptable for the blockstep algorithm, which means that the timestep must be the powers of two (negative values allowed, but smaller than the time resolution set by `g6_set_tunit` should result in an error). Also the time must be an exact integer multiple of the timestep.

Note that the GRAPE-6 predictor unit can make use of the second time derivative of the force, unlike the GRAPE-4 predictor unit which can use only up to the first time derivative. If the application program cannot provide the second time derivative, it must give the pointer of the array with size three filled with zeros.

The parameter `index` is introduced to GRAPE-6 to achieve

- Inhibition of the self-interaction base on particle identity
- Easier use of the neighbor list than GRAPE-3/4/5

The function g6calc_firsthalf also has this index as an argument. Thus, if the indices are the same, the accumulation of the result is skipped on hardware. Thus, effectively, one can achieve something like the following:

```
C calculate the force on particle i from all other particles
      do j = 1, n
          if (index(i) .ne. index(j)) then
C             do the actual force calculation
          endif
      enddo
```

The functions for the neighbor list will return the list of particles using this index, and not the location of the particles in the memory as used to be so on older GRAPE systems. This should make the application program somewhat simpler, in particular when the number of particles is larger than what fits on the memory of GRAPE-6.

5.4.2 g6_set_j_particle_mxonly

```
int g6_set_j_particle_mxonly(int clusterid,
                       int address,
                       int index,
                       double mass,
                       double x[3] /* position */)

integer function g6_set_j_particle_mxonly(clusterid, address, index, mass,x)
integer clusterid, address, index
double precision  mass, x(3)
```

This function works in the same way as g6_set_j_particle except that it sets only mass and position. This is handy if the application does not uses the predictor.

5.4.3 g6_flush_jp_buffer

```
int g6_flush_jp_buffer(int clusterid, int size)
subroutine  g6_flush_jp_buffer(clusterid)
```

This function flushes the data in the DMA buffer for j-particles. Should be called before fasthalf/lasthalf pair, if any j-particle was sent using g6_set_j_particle (or its variant) in DMA mode (g6_initialize_jp_buffer called).

5.5 Setting current time for individual timestep algorithm

5.5.1 g6_set_ti

```
void g6_set_ti(int clusterid, double ti)

subroutine  g6_set_ti_(clusterid, ti)
integer clusterid
double precision ti
```

This function sets the present time (t_i) for the predictor unit.

5.6 Force calculation

5.6.1 g6calc_firsthalf

```
void g6calc_firsthalf(int clusterid,
                      int nj,
                      int ni,
                      int index[],
                      double xi[][3],
                      double vi[][3],
                      double fold[][3],
                      double j6old[][3],
                      double phiold[],
                      double eps2,
                      double h2[])
subroutine g6calc_firsthalf(clusterid, nj, ni, index, xi, vi, fold,
                      j6old, phiold, eps2, h2)
integer clusterid, nj, ni, index(*),
double precision xi(3,*), vi(3,*), fold(3,*), j6old(3,*), phiold(*),
                 eps2, h2(*)
```

This function just calls `g6_set_ip_scales`, `g6_set_i_particle`, `g6_set_nip` and `g6_set_njp` in that order. In other words, it sets the necessary scalings for i-particles, sends them to GRAPE-6, set number of particles (both i and j) and starts the calculation. Note that `fold`, `j6old` and `phiold` are used to determine the scaling. Therefore, they should have the value corresponding to the particle.

The application program can calculate the force by calling this function and `g6calc_lasthalf` in that order.

5.6.2 g6calc_firsthalf0

```
void g6calc_firsthalf0(int clusterid,
                      int nj,
                      int ni,
                      int index[],
                      double xi[][3],
                      double vi[][3],
                      double fold[][3],
                      double j6old[][3],
                      double phiold[],
                      double eps2[],
                      double h2[],
                      int * mode)
subroutine g6calc_firsthalf0(clusterid, nj, ni, index, xi, vi, fold,
                      j6old, phiold, eps2, h2, mode)
integer clusterid, nj, ni, index(*), mode
double precision xi(3,*), vi(3,*), fold(3,*), j6old(3,*), phiold(*),
                 eps2, h2(*)
```

This function is almost same as the function `g6calc_firsthalf`, except that the last argument `mode` is added. If `mode=1`, it behaves in the same way as `g6calc_firsthalf`, but if `mode=0`, `eps2` is also an array. Thus, using this function, the user can specify different values of `eps2` for different i-particles.

5.6.3 g6calc_lasthalf

```
int g6calc_lasthalf(int clusterid,
                    int nj,
                    int ni,
                    int index[],
                    double xi[][3],
                    double vi[][3],
                    double eps2,
                    double h2[],
                    double acc[][3],
                    double jerk[][3],
                    double pot[])

integer function  g6calc_lasthalf(clusterid, ni, nj, index, xi, vi,
                     eps2, h2, acc, jerk, pot)
integer clusterid, nj, ni, index
double precision xi(3,*), vi(3,*), eps2, h2(*), acc(3,*), jerk(3,*), pot(*)
```

This function waits for the end of calculation and retrieves the calculated result. In the case of the scaling error, this function tries to adjust the scaling factors appropriately and retries the calculation. If some hardware error occurs and is detected, this function returns non-zero value. Otherwise the return value is zero.

5.6.4 g6calc_lasthalf2

```
int g6calc_lasthalf2(int clusterid,
                     int nj,
                     int ni,
                     int index[],
                     double xi[][3],
                     double vi[][3],
                     double eps2,
                     double h2[],
                     double acc[][3],
                     double jerk[][3],
                     double pot[],
                     int nnbindex[])
i
integer function  g6calc_lasthalf2(clusterid, ni, nj, index, xi, vi,
                     eps2, h2, acc, jerk, pot, nnbindex)
integer clusterid, nj, ni, index, nnbindex(*)
double precision xi(3,*), vi(3,*), eps2, h2(*), acc(3,*), jerk(3,*), pot(*)
```

Same as g6calc_lastfalf except that this function returns nnbindex, the indices for the nearest neighbours.

5.6.5 g6calc_lasthalf0

```
int g6calc_lasthalf0(int clusterid,
                     int nj,
                     int ni,
                     int index[],
                     double xi[][3],
                     double vi[][3],
                     double eps2,
                     double h2[],
                     double acc[][3],
                     double jerk[][3],
                     double pot[],
                     int * mode)
i
integer function  g6calc_lasthalf0(clusterid, ni, nj, index, xi, vi,
                     eps2, h2, acc, jerk, pot, 0)
integer clusterid, nj, ni, index, mode
double precision xi(3,*), vi(3,*), eps2, h2(*), acc(3,*), jerk(3,*), pot(*)
```

Same as g6calc_lastfalf except that this function should be called if one uses g6calc_firsthalf0

5.6.6 g6calc_lasthalf0a

```
int g6calc_lasthalf0a(int clusterid,
                     int nj,
                     int ni,
                     int index[],
                     double xi[][3],
                     double vi[][3],
                     double eps2,
                     double h2[],
                     double acc[][3],
                     double jerk[][3],
                     double pot[],
                     int nnbindex[],
                     int * mode)
i
integer function  g6calc_lasthalf0(clusterid, ni, nj, index, xi, vi,
                     eps2, h2, acc, jerk, pot, 0)
integer clusterid, nj, ni, index, nnbindex(*), mode
double precision xi(3,*), vi(3,*), eps2, h2(*), acc(3,*), jerk(3,*), pot(*)
```

Combines both of lasthalf2 and lasthalf0 functionality.

5.7 Neighbour list

5.7.1 g6_read_neighbour_list

```
int g6_read_neighbour_list(int clusterid)

integer function g6_read_neighbour_list(clusterid)
integer clusterid
```

This function transfers the content of the hardware neighbor list of GRAPE-6 to the working memory of the interface library. Return value of zero means successful completion, -1 some internal error and 1 overflow of the hardware neighbour list memory. This function must be called only once after `g6_calc` (or `g6calc_lasthalf`) is called.

Zero return value means normal end. Negative values indicate some hardware error (you need to reset the hardware and restart calculation). Positive return values mean overflow (radius too large).

Note that GRAPE-6 always calculates the force, and therefore neighbour lists, for 48 particles, even when you send only one particle with `g6calc_firsthalf`. For particles not supplied by the present call, the values last set are used. Thus, if you have neighbour list overflow with force calculation for less than 48 particles, that may be caused by the values of the neighbour sphere radius for "unused" locations.

If you want to use the neighbour list with less than 48 particles, always make sure that you do not set large values to "unused" locations. The simplest way to guarantee this is of course just to copy the last particle to all "unused" locations.

5.7.2 g6_get_neighbour_list

```
int  g6_get_neighbour_list_(int clusterid,
                            int ipipe,
                            int maxlength,
                            int * nblen,
                            int nbl[])

integer function  g6_get_neighbour_list(clusterid, ipipe, maxlength, nblen, nbl)
integer clusterid ipipe maxlength nblen, nbl(*)
```

This function extracts the actual neighbor list of particle calculated for particle `ipipe` from the working memory of the interface library prepared by `g6_read_neighbour_list`. The argument `maxlength` gives the size of the array `nbl`. Return value of zero means successful completion and 1 overflow. On successful return, `nblen` has the length of the neighbour list and `nbl` actual list (sorted by the indices).

5.7.3 g6_set_neighbour_list_sort_mode

```
void g6_set_neighbour_list_sort_mode(int mode);

subroutine g6_set_neighbour_list_sort_mode(mode)
integer mode
```

This function sets the sort option flag for the neighbour list function. When set to 1, the neighbour list returned by `g6_get_neighbour_list` is sorted by the indices of the neibours. this sorting is achieve by using `qsort(3)` function, and thus takes rather significant time. If you do not need your neighbour list sorted, you can get some speedup by calling this function with argument 0. Initial value of the flag is 1.

5.7.4 g6_get_neighbour_list_sort_mode

```
int g6_get_neighbour_list_sort_mode();
```

```
integer function g6_get_neighbour_list_sort_mode
```

This function returns the sort option flag for the neighbour list.

5.8 High-level function for shared-timestep algorithm

5.8.1 calculate_accel_by_grape6_separate_trial_noopen

```
int calculate_accel_by_grape6_separate_trial_noopen(int clusterid,
                                                    int ni,
                                                    double xi[][3],
                                                    double vi[][3],
                                                    int nj,
                                                    double xj[][3],
                                                    double vj[][3],
                                                    double m[],
                                                    double a[][3],
                                                    double jerk[][3],
                                                    double pot[],
                                                    double eps2)
```

This function, for historical compatibility reasons, does not have Fortran interface. This function is designed as a quick shortcut to use GRAPE-6 with simple shared-timestep code, which do not need predictors and time derivatives of accelerations. `Ni, xi, vi` are the number of particles, position and velocity on which the accelerations are calculated, and here `ni` can be arbitrary large number (not limited to 48). `Nj, xj, vj, m` are the number of particles, position, velocity and mass of the particles that exert accelerations. `A, jerk, pot` are the calculated results and `eps2` is the softening parameter.

This routine, as well as `calculate_accel_by_grape6_noopen`, use `g6_guestimate_acc_etc` internally. So use them with care. You can control the use of `g6_guestimate_acc_etc` by calling `g6_set_calculate_accel_scaling_mode`

5.8.2 calculate_accel_by_grape6_noopen

```
int calculate_accel_by_grape6_noopen(int clusterid,
                                     int n,
                                     double x[][3],
                                     double v[][3],
                                     double m[],
                                     double a[][3],
```

```
                                      double jerk[][3],
                                      double pot[],
                                      double eps2)
```

This is similar to `calculate_accel_by_grape6_separate_trial_noopen`, but calculates the self-interaction in direct summation and see if the force symmetry is achieved (if the total acceleration of the system is zero or not). If the test failed, it assumes that there may be some hardware error, and retry the same calculation.

5.9 Low level functions

5.9.1 g6_reset

```
int g6_reset_(int *clusterid)

subroutine g6_reset(clusterid)
integer clusterid
```

Performs the hardware reset. Usually one need not use this function, except for error recovery.

5.9.2 g6_reset_fofpga

```
int g6_reset_fofpga_(int *clusterid)

subroutine g6_reset_fofpga(clusterid)
integer clusterid
```

Force the reinitialization of all FPGA devices on board at the next call of `g6_open`. See the section of error recovery for details.

5.9.3 g6_npipes

```
int g6_npipes_()
integer function g6_npipes
```

Returns the number of pipelines available on the particular GRAPE-6 system in use. In most cases, it just returns 48, the number of physical pipelines per chip. In future, there may be some version of multi-cluster library which returns different numbers.

5.9.4 g6_getnjmax

```
int g6_getnjmax(int clusterid)
integer function g6_npipes(clusterid)
integer clusterid
```

Returns the maximum number of j-particles which can be stored in the board memory. The return value depends on the available number of chips. With some new hardware, the returned value may be wrong (too small).

5.9.5 g6_set_nip

```
void g6_set_nip_(int * clusterid, int * nip)
subroutine g6_set_nip(clusterid, nip)
integer clusterid, nip
```

Set the number of particles on which the forces (etc) are calculated. The maximum value for `nip` is 48, which is the number of virtual pipelines per chip. At least currently, the range of the arguments are not checked. So the application programmer has the responsibility to put them within the allowed range.

5.9.6 g6_set_njp

```
void g6_set_njp_(int * clusterid, int * njp)
subroutine g6_set_njp(clusterid, njp)
integer clusterid, njp
```

This function sets the number of the particles on the memory to on-chip registers of GRAPE-6. As a side effect, it let GRAPE-6 start the calculation.

5.9.7 g6_set_i_particle_scales_from_real_value

```
void g6_set_ip_scales_(int * clusterid,
                       int *address,
                       double acc[3],
                       double jerk[3],
                       double *phidouble *jfactor,
                       double *ffactor)

subroutine g6_set_ip_scales(clusterid, address, acc, jerk, phi,
                            jfactor, ffactor)
integer clusterid, address
double precision acc(3), jerk(3), phi, jfactor, ffactor
```

This function sets the binary point for the internal accumulator for force etc. GRAPE-6 performs the accumulation of the force etc is done in fixed-point format, in order to simplify the hardware and yet extend effective accuracy. Therefore, the hardware should know where to place the binary point before starting the calculation, and in order to do so the library function should know approximate size of force (etc). A good guess is the values at the previous timestep, and therefore within the time integration routine one can just pass the actual values of acceleration, jerk and potential to this routine.

Two additional parameters, `jfactor` and `ffactor`, are both used to allow more subtle control on the scaling of jerk. While acceleration and potential are accumulated in 64-bit format, jerk is accumulated in 32 bit. Thus, in order to allow reasonable accuracy, I chose the margin for jerk much smaller than that for acceleration and potential. The parameter `jfactor` is used to change the margin. The passed values of jerk is *multiplied* by `jfactor` before being used to calculate the scale. Thus, suppling `jfactor` larger than unity significantly reduce the chance of overflow, but might result in slightly less accurate value of jerk. I do not recommend the use of `jfactor` > 10.

The parameter `ffactor` is used to calculate the scale for jerk from the acceleration. To be specific, the scaling factor is calculated by

```
jmax = fabs(jerk[k])*(*jfactor) + fabs(acc[k])*(*ffactor);
```

This parameter is used to further reduce the chance of overflow. Since the jerk is the time derivative of the acceleration, acceleration divided by the timestep would give pretty good upper limit for the jerk.

For the first call, the use of this function is a bit problematic, since the application program might not have good knowledge on how larger is the force on a particle. In this case, I'd recommend to initialize the arrays for acceleration etc with fairly large values and then do the force calculation twice. In the first call, a good guess for the actual value is obtained. However, the calculated force itself should not be used, since the binary point might be inappropriate.

5.9.8 g6_adjust_ip_scales

```
void g6_adjust_ip_scales_(int * clusterid,
                          int *address,
                          int * flagp)

subroutine g6_adjust_ip_scales(clusterid, address, flagp)
integer clusterid, address, flagp
```

Even if the scaling factors is set according to the previous values, the calculated result might still overflow in some rare cases, and if you integrate N-body system long enough, every "rare case" would show up. This function adjust the scaling factors for particular (i-)particle, according to the flag returned by the function `g6_get_force`.

5.9.9 g6_set_i_particle

```
void g6_set_i_particle_(int * clusterid,
                        int *address,
                        int *index,
                        double x[3], /* position */
                        double v[3], /* velocity */
                        double * eps2,
                        double * h2)

subroutine g6_set_i_particle(clusterid, address, index, x, v, eps2, h2)
integer clusterid, address, index,
double precision x(3), v(3), eps2, h2
```

T his function sends the data of particles on which the force etc are to be calculated. The argument `address` is the location within GRAPE-6 register files, and it should be within the range of $[0, 47]$. The meaning of the index is described in the section for `g6_set_j_particle`. `Eps2` is the softening parameter squared. Setting this zero would not cause error, if `index` is correctly handled, since the self-interaction is avoided through this index. `H2` is the radius of the neighbor sphere.

5.9.10 g6_get_force

```
int g6_get_force_(int * clusterid,
                  double acc[][3],
                  double jerk[][3],
```

```
                   double phi[],
                   int flag[])

integer function  g6_get_force(clusterid, acc, jerk, phi, flag)
integer clusterid, flag(*)
double precision acc(3,*),jerk(3,*), phi(*)
```

This function returns the calculated result. If the calculation is not finished, it waits until the end.

If non-zero value is returned, it means some error and that the recalculation is required. The meaning of the non-zero error code is not specified yet. Arguments `acc, jerk, phi` are the calculated acceleration, jerk and potential, respectively. `Flag` indicates the status. The function to analyze and report flags will be supplied soon.

5.9.11 g6_get_force_etc

```
int g6_get_force_etc_(int * clusterid,
                   double acc[][3],
                   double jerk[][3],
                   double phi[],
                   int nnbindex[],
                   int flag[])

integer function  g6_get_force_etc(clusterid, acc, jerk, phi, nnbindex, flag)
integer clusterid, flag(*), nnbindex(*)
double precision acc(3,*),jerk(3,*), phi(*)
```

Essentially the same as `g6_get_force`, except that this function returns the indices of the nearest neighbours in `nnbindex` as well. Note that the returned indices are NOT the memory memory address but the `index` set by `g6_set_i_particle`.

5.9.12 g6_guestimate_acc_etc

```
void g6_guestimate_acc_etc( int n,
                         double eps2,
                         double m[],
                         double a[][NDIM],
                         double j[][NDIM],
                         double p[])
```

This function is not really for public use, but listed here for completeness. The only thing this function does is to assign some "reasonable" values for force, jerk and potential so that they would not cause floating point error when they passed to actual GRAPE-6 library. The use of this function is discouraged. As stated before, the application programmer should provide a good guess for force etc. before passing them to `g6calc_firsthalf_`.

5.9.13 g6_set_calculate_accel_scaling_mode

```
int g6_set_calculate_accel_scaling_mode(int mode);
```

As stated earlier, by default calculate_accel_by_grape6_separate** functions calls g6_guestimate_acc_etc internally. You can suppress the call by calling g6_set_calculate_accel_scaling_mode with mode=0 before calling calculate_accel_by_grape6_separate**. By calling this function with mode=1 you can restore the default mode.

6 Example

6.1 Shared timestep

The following code evaluates accelerations on NI particles from other NJ particles by direct summation.

```
int calculate_accel_by_grape6_separate_trial_noopen(int clusterid,
                                                    int ni,
                                                    double xi[][3],
                                                    double vi[][3],
                                                    int nj,
                                                    double xj[][3],
                                                    double vj[][3],
                                                    double m[],
                                                    double a[][3],
                                                    double jerk[][3],
                                                    double pot[],
                                                    double eps2)
{
#define IPLIMIT MAXPIPELINESPERCHIP
    double ajtmp[3];
    double jjtmp[3];
    double j2jtmp[3];
    double ti, tj, dtj;
    int j0, i0;
    int i,k,ii, iend;
    int flag[MAXPIPELINESPERCHIP+1];
    int index[MAXPIPELINESPERCHIP+1];
    double h2[MAXPIPELINESPERCHIP+1];
    double eps2array[MAXPIPELINESPERCHIP+1];
    int nharderror = 0;
    int ipmax = g6_npipes();
    ti = 0.0;tj =0.0; dtj = 0.0078125;
    for(k=0;k<3;k++){
        ajtmp[k] = 0.0;
        jjtmp[k] = 0.0;
        j2jtmp[k] = 0.0;
    }
START:
    g6_set_ti(clusterid, 0.0);
    for(i=0;i<nj;i++){
        g6_set_j_particle_mxonly(clusterid,i,i,m+i,xj[i]);
    }
    for(i=0;i<ipmax;i++){
        h2[i] = eps2;
        eps2array[i] = eps2;
    }
```

```
    g6_flush_jp_buffer(clusterid);
    for(i=0;i<ni;i+=ipmax){
        int error;
        iend = ipmax; if  (iend+i > ni) iend = ni-i;
        for(ii=0;ii<iend;ii++)index[ii]=i+ii;
        g6calc_firsthalf(clusterid, nj, iend,index,&(xi[i]),&(vi[i]),&(a[i]),&(jerk[i]),
                         pot+i,eps2, h2);
        if (error = g6calc_lasthalf(clusterid,nj,iend,index,&xi[i],&vi[i],eps2,h2,
                                    &a[i], &jerk[i], pot+i)){
            nharderror ++;
            fprintf(stderr,"(calculate_accell_trial_noopen) hard error %d\n", error);
            g6_reinitialize(clusterid);
            if (nharderror < 10){
                goto START;
            }else{
                fprintf(stderr,"(calculate_accell_trial_noopen) too many errors... %d returning -1\n", nharderror);
                return -1;
            }
        }
    }
    return 0;
}
```

6.2 Block timestep

The following is a C++ example to show some idea...

6.2.1 Initialization

```
    g6_open(0);
    g6_initialize_jp_buffer(0,nbody);

    for(int i=0;i<nbody;i++)if (pb[i].get_grape_index() == -1) {
        pb[i].set_timestep(0.0078125);
        pb[i].set_grape_index(i);
    }
    vector j218 = vector(0.0,0.0,0.0);
    particle * bi = pb;
    for(int i = 0; i<nbody; i++){
        vector j6 = bi->get_old_jerk()*ONE_SIXTH;
        vector a2 = bi->get_old_acc()*0.5L;
        g6_set_j_particle(grape6_id,bi->get_grape_index(),
                          bi->get_index(),
                          bi->get_time(),
                          bi->get_timestep(),
                          bi->get_grape_mass(),
                          (real*)&j218,
                          (real*)&j6,
                          (real*)&a2,
                          (real*)bi->pget_vel(),
                           (real*)bi->pget_pos());
        bi++;
    }
}
```

6.2.2 Force calculation

```
void calculate_acc_and_jerk_for_list_on_grape6(particle_system* ps,
                                               particle* pb,
                                               int nbody,
                                               int nbh,
                                               node_time* nt,
                                               int  n_next,
                                             real eps2)
{
    int error;
    do{
        error = 0;
        if (grape6_open == 0) {
            // actual initialization should be performed here...
        }
        real sys_t = nt[0].next_time;
        for (int i = 0; i < n_next; i++) {
            particle *bi = nt[i].pptr;
            bi->predict_loworder(sys_t);
        }
        g6_set_ti(grape6_id,sys_t);
        for (int i = 0; i < n_next; i+= npipes) {
            particle *bi = nt[i].pptr;
            int np = npipes;
            if (i+np > n_next) np = n_next - i;
            for (int ii = 0; ii < np; ii++){
                particle *bi = nt[i+ii].pptr;
                gindex[ii] = bi->get_index();
                xi[ii] = bi->get_pred_pos();
                veli[ii] = bi->get_pred_vel();
                acci[ii] = bi->get_acc();
                jerki[ii] = bi->get_jerk();
                poti[ii] = bi->get_pot();
            }
            if ( (np > 0) && (error == 0))
                g6calc_firsthalf(grape6_id,nbody, np, gindex,
                                   cvector xi, cvector veli, cvector acci, cvector jerki,
                                   poti, eps2, h2i);
            if ( (np > 0) && (error == 0)){
                error = g6calc_lasthalf(grape6_id, nbody, np, gindex,
                                          cvector xi, cvector veli,  eps2, h2i,
                                          cvector acci, cvector jerki, poti);
                if (error){
                    error = 1;
                }else{
                    for (int ii = 0; ii < np; ii++){
                        particle *bi = nt[i+ii].pptr;
                        bi->set_acc(acci[ii]);
                        bi->set_jerk(jerki[ii]);
                        bi->set_pot(poti[ii]);
                    }
```

```
            }
        }

    }
    if (error){
        cerr << "GRAPE hardware error" <<endl;
        grape6_reinitialize(grape6_id);
        grape6_open = 0;
    }
    }while (error);
}
```

6.3 Update memory

```
void particle_system::update_grape_data(int  n_next)
{
     vector j218 = vector(0.0,0.0,0.0);

     for (int i = 0; i < n_next; i++) {
        particle *bi = nt[i].pptr;
        vector j6 = bi->get_jerk()*ONE_SIXTH;
        vector a2 = bi->get_acc()*0.5L;
        g6_set_j_particle(grape6_id,bi->get_grape_index(),
                          bi->get_index(),
                          bi->get_time(),
                          bi->get_timestep(),
                          bi->get_grape_mass(),
                          (real*)&j218,
                          (real*)&j6,
                          (real*)&a2,
                          (real*)bi->pget_vel(),
                          (real*)bi->pget_pos());
    }
    g6_flush_jp_buffer(grape6_id);
}
```

7 Error recovery

As you can see in the previous example and in some of the routines, they might return error, and the application program has to handle these errors to continue calculation.

In particular, if non-zero value is returned by `g6calc_lasthalf`, it almost always implies an intermittent hardware error. The following code fragment shows the error recovery section of nbody1 with multi-cluster support.

```
         do ic = 1, g6ncluster
            if (g6error(ic) .eq. 1) then
               write(6,*) 'GRAPE-6 hard error on cluster', ic
               do jj = 1, g6ncluster
                  call g6_reinitialize(jj-1)
                  call g6_set_ti(jj-1,time)
               enddo
               do i = nbh+1, n
                  call g6_set_j_particle(g6jcid(i),g6jcloc(i), i,
```

```
$                 t0(i), step(i), mharp(i), j2,
$                 fdot(1,i), f(1,i), x0dot(1,i), x0(1,i))
            enddo
            goto 1500
         endif
      enddo
```

The idea is just to reset the hardware and send all data again if any error occurred. Since the error is intermittent, on the second try the error would normally vanish. If there is real hardware problem, well, contact your friend with hardware knowledge...

8 Neighbour list

As it was with previous GRAPE hardwares, GRAPE-6 also has the hardware support to construct the list of neighbours for each particle. The use of this function, however, is not "fool-proof".

The basic use is quite simple. When calling `firsthalf/lasthalf` pair, you specify the radius for the neighbour search. The GRAPE-6 hardware then constructs the list during calculation, and tt read_neighbour{`get_neighbour` pair returns the actual neighbour list.

The problem is what one should do if the neighbour list overflows. You should check for the overflow by testing the return value of the function `g6_read_neighbour_list`. If the return value is 1, the list is probably not correct.

Well, it is difficult to give a universal procedure to recover from the overflow, since there are variety of reasons why the overflow occurs.

So let me here explain how the GRAPE-6 hardware handles the neighbour list. Though from the application's viewpoint there is only 48 pipelines, a single GRAPE-6 cluster might consist of up to 256 GRAPE-6 chips. Logically, each of these 256 chips have 48 pipelines, and all of them calculate the forces on the same set of 48 particles, but from different subset of particles you send to the memory of GRAPE-6. In other words, the "GRAPE-6 memory" is actually partitioned to small memories which are local to each GRAPE-6 chip.

The storage for the neighbour list is also local to each chip. Each chip has three memory units for the neighbour list, each of which is shared by 16 pipelines. They can hold up to 256 particles. Thus, if any of the memory units in any of the processor chip is overflown, at least the lists for the 16 particles which share the memory unit in question can be incomplete.

In most cases, this is not a very severe limitation. A single board with 32 chips can store 8192 neighbours for 16 particles. So even if there is no overlap between the neighbour lists for these 16 particles, one particle can, on average, up to 512 neighbours without causing overflow. This is true only if the neighbours are distributed evenly on different chips, which, I hope, is mostly the case.

If you need, on average, more than $16n$ neighbours per particle on n-chip system, the only safe way is to reduce the number of particles you send with `firsthalf`. If you send just one particle, it can use all memories which is normally shared by 16 particles. So it can have up to $256n$ neighbours, which should be okay for most applications. If you need even more, well, it is probably faster if you construct the neighbour list on frontend using tree...

If you need, say, less than $4n$ neighbours on average, the overflow must be very rare. The simplest solution (well, at least for me, but probable also for you) is to construct the neighbour lists on frontend. If the overflow is sufficiently rare, $O(N)$ operation on host should not cause much performance penalty.

If you want to be smarter, you could have a routine which set $h2$ to be all zero but for one particle, and calculate the force and neighbour list for that particle. In this way, the probability of overflow is very small, but still not zero. So you still need a next level of backup routine which calculates the neighbour list on the frontend.

Unless you see performance problem, I'd recommend the simple solution of to construct the neighbour list on the frontend when overflow occurs.

9 TIMESHARING

The grape6 interface offers a rather primitive method for sharing the GRAPE-6 hardware by several programs running simultaneously. A program acquires GRAPE-6 by calling `g6_open`. If someone is already using GRAPE-6, `g6_open` prints some message and put the process into the sleep state. When the hardware is released, the process is waken up and `g6_open` returns. A program can release GRAPE-6 by calling `g6_close`. Thus, a program occupies GRAPE-6 hardware between the calls to `g6_open` and `g6_close`.

In order to attain the sharing of GRAPE-6 between multiple users, therefore, programmers must write their program so that it releases GRAPE-6 at an reasonable time interval, which is around 1 minutes.

9.1 Forced Timesharing

Note: Currently, this function is implemented only on X86 platforms.

In comuter centers with many users, it might be more appropriate to have more strict control on the allocation of resources to jobs submitted by different users. In order to achieve this, the administrator can set soft limit and hard limit for the CPU time for which application programs occupy GRAPE-6 hardware.

The soft limit is specified by adding the following line to the configuration file

```
TIMESOFTLIMIT xxx
```

Here, `xxx` is the soft limit in unit of seconds. For example,

```
TIMESOFTLIMIT 300
```

sets the soft limit to 300 seconds. The soft limit works in the following principle.

If the time one program occupied GRAPE-6 exceeded the soft limit, the excess time is recorded in the library when `g6_close` is called. When `g6_open` is called next time, if someone else is using the GRAPE-6 hardware, the process which calls `g6_open` is put to the sleep state for the recorded excess time. If no one else is using the GRAPE-6 hardware, the process is not put to the sleep state.

The hard limit is specified by adding the following line to the configuration file.

```
TIMEHARDLIMIT xxx
```

Here, `xxx` is the hard limit in unit of seconds. For example,

```
TIMEHARDLIMIT 1000
```

sets the hard limit to 1000 seconds.The hard limit works in the following principle.

First, the function `g6calc_lasthalf` periodically checks the CPU time, and if the hard limit is exceeded, it returns the error status. If error status is detected, application program *must* release and reinitialize GRAPE-6, since this error should be treated in the same way as the hardware error.

Second, if the program does not release the GRAPE-6 hardware even after this error is forced, `g6calc_lasthalf` will call `exit(-1)` when CPU time reachs twice the hard limit.

Defaults for these limits are some large numbers.

10 LINKING (Local Information)

On EV6 machines (In Hongo, currently neomuscat and alexandria), use:

```
-L/usr2/makino/disk3src/harplibs -lg6  -lg6sim2  -lm
```

Alexandria is a Linux box. In order to run a program on this machine, compile and link your program on neomuscat, port or margaux , with `-non_shared` flag passed to `ld`. This binary should also work on neomuscat, which is a Tru64 UNIX (aka Digital UNIX) box. You can also compile on neomuscat.

On EV5 machines (In Hongo, currently bourgogne only), use:

```
-L/usr2/makino/disk3src/harplibs -lg6_ev5   -lg6sim2  -lm
```

On Linux-x86 machines (In Hongo, currently g6hostx), use:

```
-L/usr2/makino/disk3src/harplibs -lg6lx  -lg6lxsim2 -lm
```

Currently, only g++/g77 compilers are available on x86 Linux boxen. I plan to try Intel or PGI.

These are temporary and subject to change.

Note that bourgogne *does not* have a f77 compiler. You can compile fortran programs on neomuscat. If you want to run your program on bourgogne, which is currently an EV5 box, you'd better to supply "-arch ev5 -tune ev5" options to prevent EV6 native instructions to be emitted by the compiler.

11 RUN-TIME SUPPORT

The present library reads the system configuration file during initialization. This feature makes it possible to use the same executable on machines with different number of chips or boards. The configuration file has the information of defective (or nonexistent) chips on board. In addition, the number of processor boards is supplied from the environment variable. The necessary environment variables are set by sourcing the file `/usr2/makino/disk3src/harplibs/set_envs`. Add in your `.cshrc` file the line

`source /usr2/makino/disk3src/harplibs/set_envs`

DO NOT make your local copy of this file, since the content of this file may change without notice.

12 Sample programs

So far, three programs which make use of GRAPE-6 exist.

12.1 grape6

`grape6` is a simple shared-timestep direct-summation program. Its source files live in `/usr2/makino/src/grape`. The makefile for this program is `Makefile.grape6`. The source file is not too easy to read, since it's the result of the evolution since the days of GRAPE-1...

Anyway, you can run the program by:

```
/usr2/makino/src/grape/grape6 /usr2/makino/src/grape/testparm2x
```

The output would look like:

```
(read_config_file) MAXCHIP, NCHIP = 16 16, CHIPS:
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
testing LED ...
LED test end.
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_2p.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_2p.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_2p.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/pb_jp/cbfin.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_rh.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_rh.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_rh.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_rh.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_rh.ttf
enter n, dt, tstop, dtout, dtsnapout, eps, pesample:
LONG F format read
Enter out_snap file name( ~ for no output)
enter scales for x and v:enter power index for rho = r\^\{-p\}:enter omega:CPU sec.=       7.43
T=       0.000  E =    -0.266529909
DE= 0.00000000e+00  V.R. =   0.244443
initialize_grape6, retcode = 0
argc = 2, argv[1]=/usr2/makino/src/grape/testparm2x
Readinf DOUBLE parameters....
n=8192 dt= 0.0312 tstop=   5.000 dtout=  0.250 dtsnapout=  0.031
eps= 0.25000 pesample=8
output snap = ~
x, v scale factor = 1.000000 0.200000 power = 0.000000 omega=0.800000
Etot = -0.404249 1.616996 0.786404
Enter diag:CPU sec.=       7.43
T=       0.000  E =    -0.266529909
DE= 0.00000000e+00  V.R. =   0.244443
CM : 1.87567e-17 -2.35949e-17  2.78504e-17
CMV:-1.39456e-17 -6.74916e-18 -8.18234e-19
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       7.44
CPU sec.=       8.52
T=       0.250  E =    -0.266529875
DE=-1.28383736e-07  V.R. =   0.246534
Enter diag:CPU sec.=       8.52
T=       0.250  E =    -0.266529875
DE=-1.28383736e-07  V.R. =   0.246534
CM : 2.64545e-17 -3.08998e-17  2.51061e-17
CMV:-2.38524e-18 -1.23057e-17 -1.66357e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       8.52
```

```
CPU sec.=        9.60
T=       0.500  E =    -0.266529766
DE=-5.34901433e-07  V.R. =   0.253028
Enter diag:CPU sec.=        9.60
T=       0.500  E =    -0.266529766
DE=-5.34901433e-07  V.R. =   0.253028
CM : 8.55164e-18 -4.96294e-17  3.81842e-17
CMV:-1.08556e-17  8.26704e-19 -5.11947e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=        9.61
CPU sec.=       10.69
T=       0.750  E =    -0.266529566
DE=-1.28686658e-06  V.R. =   0.264026
Enter diag:CPU sec.=       10.69
T=       0.750  E =    -0.266529566
DE=-1.28686658e-06  V.R. =   0.264026
CM : 1.21295e-17 -3.98173e-17  1.91633e-17
CMV:-1.27123e-17 -1.13164e-17  4.09964e-19
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       10.69
CPU sec.=       11.78
T=       1.000  E =    -0.266529238
DE=-2.51785768e-06  V.R. =   0.279704
Enter diag:CPU sec.=       11.78
T=       1.000  E =    -0.266529238
DE=-2.51785768e-06  V.R. =   0.279704
CM : 1.30646e-17 -3.74321e-17  2.04914e-17
CMV: 5.57009e-18 -7.43356e-18 -1.31121e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       11.78
CPU sec.=       12.86
T=       1.250  E =    -0.266528718
DE=-4.46870731e-06  V.R. =   0.300321
Enter diag:CPU sec.=       12.86
T=       1.250  E =    -0.266528718
DE=-4.46870731e-06  V.R. =   0.300321
CM : 8.13152e-20 -5.15267e-17  2.80131e-17
CMV:-7.42678e-18 -7.54198e-18  2.98494e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       12.87
CPU sec.=       13.95
T=       1.500  E =    -0.266527893
DE=-7.56231468e-06  V.R. =   0.326237
Enter diag:CPU sec.=       13.95
T=       1.500  E =    -0.266527893
DE=-7.56231468e-06  V.R. =   0.326237
CM : 6.92873e-18 -4.38560e-17  2.48791e-17
CMV:-9.89334e-18 -5.29226e-18 -2.58514e-18
```

```
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       13.95
CPU sec.=       15.04
T=       1.750  E =    -0.266526574
DE=-1.25127809e-05  V.R. =   0.357927
Enter diag:CPU sec.=       15.04
T=       1.750  E =    -0.266526574
DE=-1.25127809e-05  V.R. =   0.357927
CM : 9.88657e-18 -1.52330e-17  2.93226e-17
CMV:-4.43168e-18 -1.79097e-17 -6.33581e-19
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       15.04
CPU sec.=       16.12
T=       2.000  E =    -0.266524499
DE=-2.02997490e-05  V.R. =   0.395985
Enter diag:CPU sec.=       16.12
T=       2.000  E =    -0.266524499
DE=-2.02997490e-05  V.R. =   0.395985
CM : 2.45563e-17 -6.04172e-17  2.43209e-17
CMV: 1.04354e-18  2.92057e-18 -1.01712e-17
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       16.13
CPU sec.=       17.21
T=       2.250  E =    -0.266521847
DE=-3.02470587e-05  V.R. =   0.440989
Enter diag:CPU sec.=       17.21
T=       2.250  E =    -0.266521847
DE=-3.02470587e-05  V.R. =   0.440989
CM : 1.48875e-17 -5.50775e-17  2.72044e-17
CMV:-1.58836e-17 -3.45589e-19 -4.31648e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       17.21
CPU sec.=       18.29
T=       2.500  E =    -0.266524723
DE=-1.94587790e-05  V.R. =   0.492597
Enter diag:CPU sec.=       18.29
T=       2.500  E =    -0.266524723
DE=-1.94587790e-05  V.R. =   0.492597
CM :-6.35614e-18 -5.10117e-17  1.12926e-17
CMV: 4.55365e-18 -1.06049e-17  1.79571e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       18.30
CPU sec.=       19.38
T=       2.750  E =    -0.266569675
DE= 1.49178145e-04  V.R. =   0.544339
Enter diag:CPU sec.=       19.38
T=       2.750  E =    -0.266569675
DE= 1.49178145e-04  V.R. =   0.544339
```

```
CM :-3.93023e-18 -5.45760e-17  1.72761e-17
CMV:-1.30985e-17  5.14996e-18  3.81842e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       19.38
CPU sec.=       20.46
T=       3.000  E =    -0.266596553
DE= 2.49980823e-04  V.R. =   0.576093
Enter diag:CPU sec.=       20.46
T=       3.000  E =    -0.266596553
DE= 2.49980823e-04  V.R. =   0.576093
CM :-8.60585e-18 -7.16929e-17  2.26056e-17
CMV:-1.44199e-17 -6.89824e-18 -1.09132e-17
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       20.47
CPU sec.=       21.55
T=       3.250  E =    -0.266566256
DE= 1.36352907e-04  V.R. =   0.586571
Enter diag:CPU sec.=       21.55
T=       3.250  E =    -0.266566256
DE= 1.36352907e-04  V.R. =   0.586571
CM : 1.58971e-17 -5.20010e-17  4.50994e-17
CMV: 4.58075e-18  4.91957e-18  7.93839e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       21.55
CPU sec.=       22.63
T=       3.500  E =    -0.266572612
DE= 1.60192107e-04  V.R. =   0.591606
Enter diag:CPU sec.=       22.63
T=       3.500  E =    -0.266572612
DE= 1.60192107e-04  V.R. =   0.591606
CM : 2.15079e-17 -7.04122e-17  2.40998e-17
CMV:-1.62224e-17 -4.33681e-18  5.67512e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       22.64
CPU sec.=       23.72
T=       3.750  E =    -0.266583967
DE= 2.02781044e-04  V.R. =   0.591086
Enter diag:CPU sec.=       23.72
T=       3.750  E =    -0.266583967
DE= 2.02781044e-04  V.R. =   0.591086
CM : 5.27193e-18 -7.40493e-17  1.45080e-17
CMV:-4.45878e-18 -8.29415e-18 -5.85808e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       23.72
CPU sec.=       24.81
T=       4.000  E =    -0.266573960
DE= 1.65247645e-04  V.R. =   0.580252
Enter diag:CPU sec.=       24.81
```

```
T=       4.000  E =    -0.266573960
DE= 1.65247645e-04  V.R. =   0.580252
CM :-9.96111e-18 -7.44711e-17  1.76318e-17
CMV:-7.45389e-19 -6.91179e-18 -2.19890e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       24.81
CPU sec.=       25.90
T=       4.250  E =    -0.266555207
DE= 9.49058810e-05  V.R. =   0.560385
Enter diag:CPU sec.=       25.90
T=       4.250  E =    -0.266555207
DE= 9.49058810e-05  V.R. =   0.560385
CM : 1.63579e-17 -6.38460e-17  2.36339e-17
CMV:-2.85145e-17 -9.93400e-18  5.25499e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       25.90
CPU sec.=       26.98
T=       4.500  E =    -0.266542816
DE= 4.84248779e-05  V.R. =   0.536870
Enter diag:CPU sec.=       26.98
T=       4.500  E =    -0.266542816
DE= 4.84248779e-05  V.R. =   0.536870
CM :-7.83336e-18 -7.74120e-17  2.45445e-17
CMV:-9.85269e-18 -1.29020e-17 -4.65190e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       26.98
CPU sec.=       28.07
T=       4.750  E =    -0.266538246
DE= 3.12788341e-05  V.R. =   0.514305
Enter diag:CPU sec.=       28.07
T=       4.750  E =    -0.266538246
DE= 3.12788341e-05  V.R. =   0.514305
CM :-1.46367e-18 -6.38053e-17  1.26218e-17
CMV:-5.25838e-18 -5.43456e-18 -1.29867e-17
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       28.07
CPU sec.=       29.15
T=       5.000  E =    -0.266539550
DE= 3.61696770e-05  V.R. =   0.494341
Enter diag:CPU sec.=       29.15
T=       5.000  E =    -0.266539550
DE= 3.61696770e-05  V.R. =   0.494341
CM :-9.41731e-18 -6.57027e-17  1.44555e-17
CMV:-4.45878e-18  7.45389e-19  2.54449e-18
AM : 2.44037e-03 -8.75096e-04 -4.04957e-01
Exit  diag:CPU sec.=       29.16
Errors: jp, ip, ecc, ecc(u), cm = 0 0 0 0 0
```

12.2 nbody1

`nbody1` is a basic individual- (block-) timestep integrator. The particular version of nbody1 with GRAPE-6 support lives in source files live in `/usr2/makino/src/bhnbody1`. The makefile for this program is `Makefile.grape6`. The source file is again not too easy to read, since it's the result of the evolution since the days of GRAPE-2...

Anyway, you can run the program by:

```
/usr2/makino/src/bhnbody1/nbody1_g6 <  /usr2/makino/src/bhnbody1/samplein
```

The output would look like:

```
 G6NCLUSTER =           1




           N  NBH NRAND   ETA   DELTAT   TCRIT    QE       CUTOFF

         200   0     42   0.02     0.1     0.5    0.00001  0.00


           OPTIONS     0   2   0   2   1   0   0   0   0   1

 data n =         2048
 Body end
 X end
 V end
 exit data
 call g6open
(read_config_file) MAXCHIP, NCHIP = 16 16, CHIPS:
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
testing LED ...
LED test end.
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_2p.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_2p.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_2p.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/pb_jp/cbfin.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_rh.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_rh.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_rh.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_rh.ttf
(send_fpga_data) file to open: /usr2/makino/src/grape6board/processor_board/fo_unit_with_fifo_rh.ttf
 return  g6open
 fpoly0, eps2 =   1.000000000000000E-006 npipe =          48
  ic, nj  =           1       2048
Errors: jp, ip, ecc, ecc(u), cm = 0 0 0 0 0

 T =    0.000  Q = 0.50  STEPS =          0          0          0
```

```
  DE=  0.000E+00  0.000E+00  E = -0.25066
 CM  :   -0.130498E-07  -0.978288E-08  -0.265915E-07
 CMV :    0.510450E-08  -0.886712E-09  -0.433698E-09
 AM  :   -0.838052E-02  -0.543369E-03   0.823090E-02
 new eta =   2.200000000000000E-002
CPU min =          0.122586
 Number of pipes =          48
 intgrt,   0.125244140625000       0.125000000000000
CPU min =          0.141260
Errors: jp, ip, ecc, ecc(u), cm = 0 0 0 0 0

 T =    0.125  Q = 0.50  STEPS =      55332        774          0
  DE=  2.535E-08 -5.050E-08  E = -0.25066
 CM  :    0.196896E-08  -0.416279E-08  -0.218563E-08
 CMV :    0.439653E-08  -0.483714E-09  -0.118700E-08
 AM  :   -0.838052E-02  -0.543364E-03   0.823092E-02
 new eta =   2.420000000000000E-002
CPU min =          0.141276
 Number of pipes =          48
 intgrt,   0.250488281250000       0.250000000000000
CPU min =          0.157315
Errors: jp, ip, ecc, ecc(u), cm = 0 0 0 0 0

 T =    0.250  Q = 0.50  STEPS =     103030       1600          0
  DE=  1.270E-08 -7.582E-08  E = -0.25066
 CM  :   -0.272468E-08  -0.172687E-07   0.233900E-08
 CMV :    0.372973E-08  -0.493510E-08  -0.741453E-08
 AM  :   -0.838052E-02  -0.543359E-03   0.823092E-02
 new eta =   2.662000000000000E-002
CPU min =          0.157331
 Number of pipes =          48
 intgrt,   0.375122070312500       0.375000000000000
CPU min =          0.171825
Errors: jp, ip, ecc, ecc(u), cm = 0 0 0 0 0

 T =    0.375  Q = 0.50  STEPS =     146129       2218          0
  DE=  1.458E-08 -1.049E-07  E = -0.25066
 CM  :    0.238592E-08  -0.286509E-09  -0.960970E-08
 CMV :    0.273282E-08   0.490193E-08  -0.218992E-07
 AM  :   -0.838054E-02  -0.543361E-03   0.823092E-02
 new eta =   2.928200000000001E-002
CPU min =          0.171841
 Number of pipes =          48
 intgrt,   0.500244140625000       0.500000000000000
CPU min =          0.185635
Errors: jp, ip, ecc, ecc(u), cm = 0 0 0 0 0

 T =    0.500  Q = 0.50  STEPS =     186832       2713          0
```

```
  DE=  2.113E-08 -1.471E-07  E = -0.25066
 CM  :    0.366066E-08  -0.332199E-08  -0.149043E-07
 CMV :    0.225550E-08   0.159237E-08  -0.219382E-07
 AM  :   -0.838053E-02  -0.543371E-03   0.823092E-02
 new eta =   3.221020000000001E-002
CPU min =          0.185668
 Number of pipes =          48
 intgrt,   0.625122070312500       0.625000000000000
CPU min =          0.198616
Errors: jp, ip, ecc, ecc(u), cm = 0 0 0 0 0

 T =    0.625  Q = 0.50  STEPS =     224995       3309          0
  DE=  7.072E-09 -1.612E-07  E = -0.25066
 CM  :   -0.215099E-08  -0.329190E-08  -0.834841E-08
 CMV :   -0.187221E-08  -0.532395E-08  -0.378905E-07
 AM  :   -0.838054E-02  -0.543368E-03   0.823092E-02
 new eta =   3.543122000000001E-002

        TIME =   0.63  TCOMP =    0.20  KZ(1) = 0
```

12.3 Kira

`Kira` is a rather fancy N-body integration program specialized to star clusters, with the capability to handle stellar evolution, binary evolution, stellar collisions, galactic tidal fields and all the "realistic" additional physics. This program will however need some separate documentation...

13 Known bugs and problems

13.1 g77 fails to link...

g77 fails to link the library, complaining that there is no such functions like `g6_open__`.

This is because of a rather unusual feature of g77, which adds additional second underscore to names of functions which has one or more underscores in the middle. To avoid this problem, use the option `-fno-second-underscore` when compiling your Fortran programs.

13.2 g6calc_firsthalf and/or g6calc_lasthalf fails with SIGFPE

Libraries with date before Sept 26, 2001 contains this bug. If you are using older library, please update it to newer one.

14 LIMITATIONS

Maximum number of particles which can be stored is $262144 \times n$, where n is the number of boards in a cluster.

15 FAQs

Q1: My program produces a lot of messages like the following:

```
(g6_test_flag) scaling error 2da00
(g6calc_lasthalf) overflow for  3  2da00  24 16165 48-- change scales
(adjust_ip_scales) decrimenting scale for jerk-333
```

Is this something I should care?

A: This message means that the initial guess for force, jerk or potential (in the above case, it's jerk) was too small and it caused the overflow of the result. The lasthalf routine automatically recalculates the force after decrementing the scaling index by 5 (increasing the scale factor by 32).

In the above message, "3" and "2da00" are things I do not quite want to explain, and "24" is the index (within i-particle array) of the particle which resulted the overflow. "16165" is the index of that same particle specified in the argument of firsthalf, and "48" is the number of i-particles. The last number, "-333" is the new value of the scale factor (which by itself is not too useful).

It's hard to avoid the scaling error completely since jerk sometimes changes by very large factor even if time integration goes fine. However, if you see too many of them, in particular the repeated one for one force calculation, it is likely that your initial guess is too far from the true one. You might want to try a better way to make the initial guess.

Q2: It seems GRAPE-6 calculates the force from particles with index larger than `nj`. Why?

A: Well, "this is a feature, not a bug". The problem here is that j-particles are actually distributed to a number of chips. Thus, if we use p chips to calculate the force on one particle, each chip calculates forces from $[(N+1)/p]+1$ particles. Unless N is an exact integer multiple of p, some chips calculate the forces from particles with indices larger than N. GRAPE-6 library takes care of this by clearing out the unused location when a new value of `nj` is set in `firsthalfx`. So *if you do not overwrite them again*, the calculated result is correct. If you accidentally overwrite these locations, GRAPE-6 just calculates the forces from them. So you should not write the particle data with index larger than you want to use. (But why you want to do so?)

AUTHOR

Junichiro Makino