G5PIPE User's Guide

for GRAPE-7 software package version 2.1

K & F Computing Research Co.
E-mail: support@kfcr.jp

Abstract

In this document, we give a full description of G5PIPE, a backend logic for GRAPE-7 add-in card that calculates gravitational forces among particles.

Contents

1 G5PIPE Overview

G5PIPE calculates gravitational forces among particles using hardwired pipelines (figure 1). All other calculations, such as time integration of particle orbits, are performed on the host computer. The pipeline is basically equivalent to that of GRAPE-5[1], except that it does not calculate gravitational potential nor does neighbor particles.
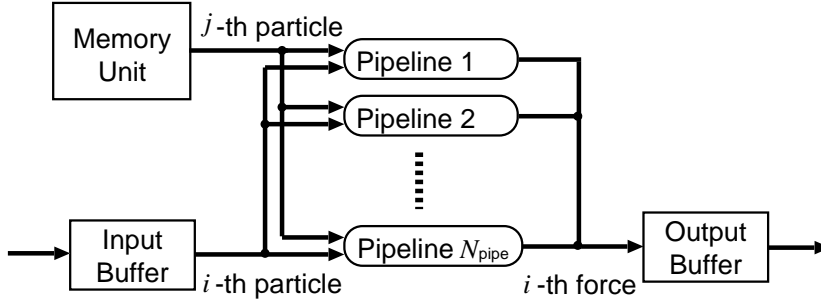

Figure 1: A schematic of G5PIPE.

In the following, we describe a force-calculation procedure using G5PIPE. Here and hereafter, we denote a particle at which force is evaluated as an i-particle, while a particle that exerts force as a j-particle.

step 1. The host computer sends a set of j-particles to the memory unit of G5PIPE. The number of j-particles sent to each G5PIPE should not exceed its memory size.

step 2. The host computer sends a set of i-particles to the input buffer of G5PIPE. The number of i-particles should not exceed the buffer size.

step 3. The host computer sends a command to indicate start of the runs.

step 4. From the input buffer, N_{pipe} particles are retrieved. Each of them is sent to one of N_{pipe} pipelines. Here, N_{pipe} is the number of pipelines integrated into one G5PIPE.

step 5. The pipelines start a run. Each pipeline calculates forces from all j-particles stored in the memory unit to its own i-particle. During the run, a j-particles is broadcasted to all N_{pipe} pipelines in every clock cycle. Each pipeline accumulates one pairwise force from a j-particle to its own i-particle into its internal register, in every clock cycle.

step 6. When a pipeline complete the run, i.e., it finishes accumulation of forces from all j-particles to its own i-particle, it sends the accumulated force to the output buffer of G5PIPE. Contents of the buffer are automatically sent back to the host (In the case of Model 300 and Model 600, outputs from all pFPGAs are accumulated before they are sent back to the host).

step 7. Step 4–6 is repeated until all i-particles in the input buffer are processed.

step 8. Now forces from all j-particles sent to the memory unit at step 1, to all i-particles sent to the input buffer at step 2, are calculated. The host computer repeats step 2–7 for different sets of i-particles, until all i-particles in the simulation are processed.

step 9. Now forces from all j-particles sent to the memory unit at step 1, to all i-particles in the simulation, are calculated. The host computer stores these forces to the main memory, and then process step 1–8 for a different set of j-particles. Obtained forces are added to the stored forces. The host computer repeats this procedure for yet another set of j-particles, until all j-particles in the simulation are processed.

Table 1 summaryises specification of G5PIPE. Figures per add-in card are shown in the table. The peak performance is, for example, that of one FPGA chip multiplied by the number of chips on the card. The number of pipelines is that of one FPGA multiplied by the number of chips. The figures shown are subject to change, and it is not recommended to hardcode them into your application program.

Table 1: G5PIPE Specifications

	Peak performance (Gflops)	Pipeline number	clock cycle(MHz)	Memory unit size (# of particles)
Model 100	101	20	133	4095
Model 300	228	60	100	12285
Model 600	456	120	100	24570
Model 800	827	128	170	32764

2 Usage of G5PIPE

2.1 Compilation and Linkage

G5PIPE can be conroled via G5PIPE library functions. In order to use the library in your own application program, you need to include `g5util.h` in your code. You also need to link G5PIPE library (`libg75.a`), HIB library (`libhib.a`), and C-language standard math library (`libm.a`). Following line shows an example of compilation command:

```
cc -o foo foo.c -L/usr/g7pkg/lib -I/usr/g7pkg/include -lg75 -lhib -lm
```

For complete description of G5PIPE library, see section 3 *Reference for G5PIPE Library Functions*. Users of legacy GRAPE-5 library (`libg5a.a`) and GRAPE-6A library (`libg65.a`) might not need full examination of that section. Most of the `g5_` functions in the legacy libraries remain unchanged in the new G5PIPE library. Those users should, however, pay attention to newly added, changed, or obsolete features listed in section 2.4.

2.2 Environment Variables

Resource Assignment (Multiple Cards User Only): If you have a system with multiple cards installed, by default, the G5PIPE standard library functions use all of them. In order to modify this behavior, you can set a list of device IDs to an environment variable `G5_CARDS`. If the list is set, G5PIPE functions use the listed cards only. For example,

```
csh> setenv G5_CARDS "0 2 3"
 sh> export G5_CARDS="0 2 3"
```

would indicate the cards with device ID 0, 2, and 3 should be used. This environment variable might be useful, when you share your system with someone else.

In the case of Model 800, four iFPGAs on a single card have four device IDs different from each other, and therse iFPGAs can be assigned to different simulations. For example, on a system with one Model 800 installed, you can set

```
csh> setenv G5_CARDS "0 2"
 sh> export G5_CARDS="0 2"
```

in order to run a simulation on two iFPGAs with device ID 0 and 2. You can run another simulation simultaneously, using iFPGAs with device ID 1 and 3.

For one more example, consider a system with two Model 600 and one Model 800. In order to see device ID of each card, type `lsgrape` [1].

```
localhost>/usr/g7pkg/scripts/lsgrape
devid grape(model)       backend-logic
   0   GRAPE-7(model600)   G5
   1   GRAPE-7(model800)   G5
   2   GRAPE-7(model800)   G5
   3   GRAPE-7(model800)   G5
   4   GRAPE-7(model800)   G5
   5   GRAPE-7(model600)   G5
```

This output shows that device IDs 0 and 5 are assigned to the two Model 600s, and 1 to 4 are assigned to the four iFPGAs of the Model 800. Therefore, you can set:

```
csh> setenv G5_CARDS "3 4 5"
 sh> export G5_CARDS="3 4 5"
```

to use two iFPGAs of the Model 800 (with device IDs 3 and 4) and one of the Model 600 (with device ID 5) for one simulation. The rest devices (two iFPGAs of the Model 800 with device IDs 1 and 2 and one Model 600 with device ID 0), can be used for another simulation.

---

[1]for usage of `lsgrape`, see *GRAPE-7 Installation Guide* section 5

Warning Message Control: You can use an environment variable `G5_WARNLEVEL` to control the warning message output. The variable can be set to 0, 1, 2, or 3. The larger number indicates the more verbose outputs. The number 1 or 2 is recommended for normal usage. The number 3 would be nice for debugging purpose. The default value is 2. The number 0 suppress all but fatal error messages. The variable should not be set to 0 when you run the functionality check script (`/usr/g7pkg/scripts/check.csh`). Otherwise it would fail.

Data Transfer Mode Setting: By default, the G5PIPE library send data from the host computer to the card using Programmed I/O Write (PIOW) transfer. Here, PIO is a transfer initiated by the host computer. If you set an environment variable `G5_SENDFUNC` to `DMAR`, the library uses Direct Memory Access Read (DMAR) transfer instead:

```
csh> setenv G5_SENDFUNC "DMAR"
 sh> export G5_SENDFUNC="DMAR"
```

DMA is a transfer initiated by the card. The performances of PIOW and DMAR depend on the host computer. You can choose the faster one to improve performance of the simulation. For most of the recent PCs, PIOW is faster, if MTRR is set by `/usr/g7pkg/hibutil/setmtrr.csh`. DMAR may be a good choice, if the MTRR cannot properly be set for some reason (see *GRAPE-7 Installation Guide* section 3.1 for `setmtrr.csh` and MTRR).

2.3   Running Sample Programs

The GRAPE-7 software package includes following sample codes:

```
/usr/g7pkg/direct/direct
/usr/g7pkg/direct/directa
/usr/g7pkg/direct/directmc
/usr/g7pkg/direct/directtest
/usr/g7pkg/directf77/direct
/usr/g7pkg/vtc/vtc
/usr/g7pkg/direct/directnb
/usr/g7pkg/direct/directnba
```

Here we assume the package is installed in `/usr/g7pkg`. In the directory `/usr/g7pkg/direct`, you can find `direct`, `directa`, `directmc`, `directtest`, `directnb` and `directnba`.

`direct` is the simplest example of direct-summation code in C. `directa` is the same as `direct` except that it shows a tiny animation. `directmc` is the same as `direct`, except that it uses "Primitive Functions" instead of "Standard Functions", so that the user can specify which card should be involved in the simulation. For details of the "Primitive" and "Standard" functions, see section 3.

`directtest` is the same as `direct`, except that the number of j-particles it can handle is not limited by the size of the memory unit (See section 4.3 for the detail).

/usr/g7pkg/directf77/direct is a counterpart of direct in Fortran.

/usr/g7pkg/vtc/vtc is an implementation of Barnes-Hut tree algorithm[3] in C. See /usr/g7pkg/vtc/00README for the usage.

directnb and directnba are for G5nbPIPE. It is an extention of G5PIPE, which has an additional function to store neighbor-particle lists. See *G5nbPIPE User's Guide* (/usr/g7pkg/doc/g5nbuser.pdf) for its usage.

In the following, we give a brief description for programs in /usr/g7pkg/direct/. The programs direct and directa take two command line arguments, namely, input file name and output file name. The input file gives the initial distribution of particles, and the output file stores the final distribution of the particles. Both are in NEMO[2] 'stoa' format, which is the output format of NEMO 'stoa' program used to convert NEMO snapshot file to ascii format. The format is as follows:

```
NOBJ                    number of particles [int]
NDIM                    number of dimensions [int] (MUST BE 3)
TIME                    time of snapshot [double]
MASS(i)                 particle masses, i = 1...NOBJ [double]
.....
X(i) Y(i) Z(i)          particle positions, i = 1...NOBJ [double]
.....
U(i) V(i) W(i)          particle velocities, i = 1...NOBJ [double]
.....
```

The file /usr/g7pkg/direct/pl2k is an example of the input file containing 2048 particles.

The program directmc takes three command line arguments. The first two arguments are the same as those of direct. The third one is the device ID of the card to be used for the run.

To see how the program works, change directory to /usr/g7pkg/direct and type ./direct pl2k outfile. You will see the following outputs (The outputs on your system may not exactly be the same, depending the hardware configuration and software environment):

```
nj: 2048
use g5_cards[0]
GRAPE-7 model100  g5_nchip:1 g5_npipes:20 g5_jmemsize:4096 g5_eps2format:floating-point
Warning: g5_get_forceMC() does not calculate potential.
The value returned is just a dummy.
ke: 0.246207
step: 10 time: 1.000000e-01
e: 2.462658e-01 de: 5.902496e-05
ke: 2.462658e-01 pe: 0.000000e+00
ke/pe: inf
.................

step: 90 time: 9.000000e-01
```

```
e: 2.554286e-01 de: 9.221770e-03
ke: 2.554286e-01 pe: 0.000000e+00
ke/pe: inf
```

The final distribution of particles are stored in `outfile`.

Let's have a look inside the source code, `direct.c`. In this file, you can find a function `calc_gravity()`:

```
void
calc_gravity(double *mj, double (*xj)[3], double (*vj)[3],
      double eps, double (*a)[3], double *p, int n)
{
    g5_set_jp(0, n, mj, xj);
    g5_set_eps2_to_all(eps*eps);
    g5_set_n(n);
    g5_calculate_force_on_x(xj, a, p, n);
}
```

This function calculates forces among particles located at `xjs`. `g5_set_jp()` stores the mass and position of the particles to the memory unit of G5PIPE (section 1 step 1). `g5_set_eps2_to_all()` set the softening parameter. `g5_set_n()` set the total number of particles. `g5_calculate_force_on_x()` repeats runs on G5PIPE as described in section 1 step 8, in order to obtain forces from all n j-particles to all n i-particles.

Just below `calc_gravity()`, you can see another force calculator, `calc_gravity2()`. Its behavior is the same as that of `calc_gravity()`, but it is explicitly showing the procedure described in section 1 step 2 to 7. Is may seem to be more complex than `calc_gravity()`, but is more flexible. You can insert another job between the step 2 and 7, for example.

In `directmc.c`, you can find yet another force calculator, `calc_gravity3()`. This is an example of "Primitive Functions" usage. By device ID (the first argument `id`), you can specify a card with which `calc_gravity3()` performs the force calculation.

In `directtest.c`, you can find `calc_gravity4()`. This extends the function `calc_gravity2()`, so that the number of j-particles it can handle is not limited by the size of the memory unit (See section 4.3 for the detail).

2.4 Difference from GRAPE-5/GRAPE-6A

Although most functions in the legacy GRAPE-5 library (`libg5a.a`) and GRAPE-6A library (`libg65.a`) remain unchanged in the new G5PIPE library, some of them are changed or removed, and several new features are added. Users of the legacy libraries should pay attention to the following changes:

- The library name changed. Use `libg75.a` and `libhib.a`, instead of `libg5.a` and `libphibdma.a`.

- The header-file name changed. It is recommended to use `g5util.h`, although traditional `gp5util.h` still exists as a symbolic link to `g5util.h`.

7

- `g5_set_xj()` and `g5_set_mj()` do not exist anymore. Use `g5_set_jp()` instead.

- G5PIPE calculates gravitational force only. It does not calculate gravitational potential. For creation of neighbor-particle lists, use G5nbPIPE, an extention of G5PIPE. See *G5nbPIPE User's Guide* (`/usr/g7pkg/doc/g5nbuser.pdf`) for its usage.

- The function shape of the force cutoff is not programmable. It is fixed to the one used with P^{3}M method. `g5_set_cutoff_table()` exists only for backward compatibility. It actually does nothing.

- The particle memory unit of G5PIPE is smaller than that of GRAPE-5 by ten fold. GRAPE-5 user might need to add 'j - loop' to their own codes, in order to handle particles exceeding the memory size. The memory size can be obtained by `g5_get_jmemsize()`. The number returned by this function should not be hardcoded into application programs, since it is subject to change. You can find a coding example with additional 'j - loop' in section 4.3.

- The number of virtual pipelines of G5PIPE (i.e. size of the input buffer show in figure 1) is larger than that of GRAPE-5. The number can be obtained by `g5_get_number_of_pipelines()`. Currently, the number is set to 256. However, this number should not be hardcoded into application programs, since it is subject to change.

- Several environment variables are added, some are deleted. See section 2.2 for a complete list of environment variables.

2.5   Use in Multi-Thread Environment

Here we list up some additional information necessary to use G5PIPE from a multi-threaded process.

- Do not use the "Standard Functions" described in section 3.1.1. Use "Primitive Functions" only.

- Do not use one card from multiple threads. i.e., do not call any library functions from multiple threads using the same device ID.

3 Reference for G5PIPE Library Functions

3.1 Synopsis

3.1.1 Standard Functions in C

The following functions provide high-level programming interface to manipulate G5PIPE.
The functions described here hide the number of the cards to the user. This approach
simplifies the user program. As long as user controls G5PIPE via these functions, the
user can handle multiple cards as a single object. The user does not need to care about
how many cards are attached to the host computer.

```
void   g5_open(void);
void   g5_close(void);
void   g5_set_range(double xmin, double xmax, double mmin);
void   g5_set_jp(int adr, int nj, double *m, double (*x)[3]);
void   g5_calculate_force_on_x(double (*x)[3], double (*a)[3], double *p, int ni);
void   g5_set_xi(int ni, double (*x)[3]);
void   g5_run(void);
void   g5_set_n(int nj);
void   g5_set_eps2(int ni, double *eps2);
void   g5_set_eps2_to_all(double eps2);
void   g5_get_force(int ni, double (*a)[3], double *pot);
int    g5_get_number_of_pipelines(void);
int    g5_get_jmemsize(void);
void   g5_set_eta(double eta);
void   g5_set_h_to_all(double h);
void   g5_set_h(int ni, double *h);
int    g5_read_neighbor_list(void);
int    g5_get_neighbor_list(int ip, int *list);
int    g5_get_nbmemsize(void);
int    g5_set_nbmemsize(int size);


The followings are obsolete. They may not be supported in the next version.

double g5_get_pcibus_freq(void);
void   g5_get_range(double *xmin, double *xmax, double *mmin);
void   g5_set_xmj(int adr, int nj, double (*xj)[3], double *mj);
void   g5_set_eps(int ni, double *eps);
void   g5_set_eps_to_all(double eps);
void   g5_set_cards(int *c);
void   g5_get_cards(int *c);
int    g5_get_number_of_cards(void);
int    g5_get_number_of_real_pipelines(void);
void   g5_set_cutoff_table(double (*ffunc)(double), double fcut, double fcor,
                           double (*pfunc)(double), double pcut, double pcor);
```

3.1.2 Primitive Functions in C

The following functions provide low-level programming interface to manipulate individual G5PIPE configured in each GRAPE-7 card. For each invocation of these functions, user need to specifies the device ID (`devid`) explicitly. The device ID of each GRAPE-7 card is obtained using `/usr/g7pkg/hibutil/lsgrape` utility. See *GRAPE-7 Installation Guide* for its usage. Meaning of the arguments other than `devid` are the same as those of corresponding standard function described in the previous section.

```
void   g5_openMC(int devid);
void   g5_closeMC(int devid);
void   g5_set_rangeMC(int devid, double xmin, double xmax, double mmin);
void   g5_set_jpMC(int devid, int adr, int nj, double *m, double (*x)[3]);
void   g5_set_xiMC(int devid, int ni, double (*x)[3]);
void   g5_runMC(int devid);
void   g5_set_nMC(int devid, int n);
void   g5_set_eps2MC(int devid, int ni, double *eps2);
void   g5_set_eps2_to_allMC(int devid, double eps2);
void   g5_get_forceMC(int devid, int ni, double (*a)[3], double *pot);
int    g5_get_number_of_pipelinesMC(int devid);
int    g5_get_jmemsizeMC(int devid);
void   g5_set_etaMC(int devid, double eta);
void   g5_set_h_to_allMC(int devid, double h);
void   g5_set_hMC(int devid, int ni, double *h);
int    g5_read_neighbor_listMC(int devid);
int    g5_get_neighbor_listMC(int devid, int ip, int *list);
int    g5_get_nbmemsizeMC(int devid);
int    g5_set_nbmemsizeMC(int devid, int size);

The followings are obsolete. They may not be supported in the next version.
double g5_get_pcibus_freqMC(int devid);
void   g5_get_rangeMC(int devid, double *xmin, double *xmax, double *mmin);
void   g5_set_xmjMC(int devid, int adr, int nj, double (*xj)[3], double *mj);
void   g5_set_epsMC(int devid, int ni, double *eps);
void   g5_set_eps_to_allMC(int devid, double eps);
int    g5_get_number_of_real_pipelinesMC(int devid);
void   g5_set_cutoff_tableMC(int devid,
                        double (*ffunc)(double), double fcut, double fcor,
                        double (*pfunc)(double), double pcut, double pcor);
```

3.1.3 Standard Functions in Fortran

The following subroutines and functions provide programming interface in Fortran. They provide the same functionality as their counterparts in C, except for dummy g5_set_cutoff_table().

```fortran
subroutine g5_open

subroutine g5_close

subroutine g5_set_range(xmin, xmax, mmin)
real*8 xmin
real*8 xmax
real*8 mmin

subroutine g5_set_jp(adr, nj, mj, xj)
integer adr
integer nj
real*8 mj(*)
real*8 xj(3, *)

subroutine g5_calculate_force_on_x(x, a, p, ni)
real*8 x(3, *)
real*8 a(3, *)
real*8 p(*)
integer ni

subroutine g5_set_xi(ni, xi)
integer ni
real*8 xi(3, *)

subroutine g5_run

subroutine g5_set_n(n)
integer n

subroutine g5_set_eps2(ni, eps2)
integer ni
real*8 eps2(*)

subroutine g5_set_eps2_to_all(eps2)
real*8 eps2

subroutine g5_get_force(ni, a, p)
integer ni
real*8 a(3, *)
```

```
real*8 p(*)

function g5_get_number_of_pipelines
integer g5_get_number_of_pipelines

function g5_get_jmemsize
integer g5_get_jmemsize

subroutine g5_set_eta(eta)
real*8 eta
```

The followings are obsolete. They may not be supported in the next version.

```
function g5_get_pcibus_freq
real*8 g5_get_pcibus_freq

subroutine g5_get_range(xmin, xmax, mmin)
real*8 xmin
real*8 xmax
real*8 mmin

subroutine g5_set_xmj(adr, nj, xj, mj)
integer adr
integer nj
real*8 xj(3, *)
real*8 mj(*)

subroutine g5_set_eps(ni, eps)
integer ni
real*8 eps(*)

subroutine g5_set_eps_to_all(eps)
real*8 eps

subroutine g5_set_cards(c)
real*8 c(*)

subroutine g5_get_cards(c)
real*8 c(*)

function g5_get_number_of_cards
integer g5_get_number_of_cards

function g5_get_number_of_real_pipelines
integer g5_get_number_of_real_pipelines
```

3.1.4 Primitive Functions in Fortran

The following subroutines and functions provide programming interface in Fortran. They provide the same functionality as their counterparts in C, except for dummy `g5_set_cutoff_tableMC()`.

```
subroutine g5_openMC(devid)
integer devid

subroutine g5_closeMC(devid)
integer devid

subroutine g5_set_rangeMC(devid, xmin, xmax, mmin)
integer devid
real*8 xmin
real*8 xmax
real*8 mmin

subroutine g5_set_jpMC(devid, adr, nj, mj, xj)
integer devid
integer adr
integer nj
real*8 mj(*)
real*8 xj(3, *)

subroutine g5_set_xiMC(devid, ni, xi)
integer devid
integer ni
real*8 xi(3, *)

subroutine g5_runMC(devid)
integer devid

subroutine g5_set_nMC(devid, n)
integer devid
integer n

subroutine g5_set_eps2MC(devid, ni, eps2)
integer devid
integer ni
real*8 eps2(*)

subroutine g5_set_eps2_to_allMC(devid, eps2)
integer devid
real*8 eps2
```

```
subroutine g5_get_forceMC(devid, ni, a, p)
integer devid
integer ni
real*8 a(3, *)
real*8 p(*)

function g5_get_number_of_pipelinesMC(devid)
integer devid
integer g5_get_number_of_pipelines

function g5_get_jmemsizeMC(devid)
integer devid
integer g5_get_jmemsize

subroutine g5_set_etaMC(devid, eta)
integer devid
real*8 eta
```

The followings are obsolete. They may not be supported in the next version.

```
function g5_get_pcibus_freqMC(devid)
integer devid
real*8 g5_get_pcibus_freq

subroutine g5_get_rangeMC(devid, xmin, xmax, mmin)
integer devid
real*8 xmin
real*8 xmax
real*8 mmin

subroutine g5_set_xmjMC(devid, adr, nj, xj, mj)
integer devid
integer adr
integer nj
real*8 xj(3, *)
real*8 mj(*)

subroutine g5_set_epsMC(devid, ni, eps)
integer devid
integer ni
real*8 eps(*)

subroutine g5_set_eps_to_allMC(devid, eps)
integer devid
real*8 eps
```

```
function g5_get_number_of_real_pipelinesMC(devid)
integer devid
integer g5_get_number_of_real_pipelines
```

3.2 Description

3.2.1 Standard Functions in C

void g5_open(void) gets access permission to G5PIPE. This function must be called before invocation of any other library functions except for `g5_set_cards()`. If `g5_open()` is already invoked by another program, it outputs a message:

    Someone is using hib[n]. Sleep...

and wait until the another program releases the access permission by calling `g5_close()`.

By default, all G5PIPEs on all GRAPE-7 cards attached to the host computer are opened by a single `g5_open()` call. You can modify this behavior by setting a list of device IDs to an environment variable `G5_CARDS`. If the variable is set, `g5_open()` opens the listed cards only. See section 2.2 for detailed usage of the environment variable `G5_CARDS`.

void g5_close(void) releases access permission to G5PIPE. When you share one GRAPE-7 system with your colleagues, you should invoke this function at regular intervals, say, each one minute. This will give a chance to another program to open G5PIPE. Once you released G5PIPE, you need to invoke `g5_open()` again to get access to G5PIPE.

void g5_set_range(double *xmin***, double** *xmax***, double** *mmin***)** specifies the scaling factor of spatial coordinate and mass.

The arguments *xmin* and *xmax* determine the upper and lower limit of the coordinate system. For all particles, each component of the position coordinate should be in the range of (*xmin/2*, *xmax/2*). The resolution (i.e., the minimum expressible length) of this coordinate system is $\Delta x = (xmax - xmin)/2^{32}$. This is due to the fact that G5PIPE pipeline uses 32-bit fixed-point format for position coordinate. For example, if you set $xmin = -64$ and $xmax = 64$, the resolution is $\Delta x = (64 - (-64))/2^{32} = 1/2^{25} \approx 3 \times 10^{-8}$. With this setting, all particles must reside in a cubic space which has side length 64 and centered at (0, 0, 0), otherwise the calculation results would be incorrect.

In the case of force calculation under periodic bondary condition, such as Particle-Mesh force calculation in P^{3}M method[4], the arguments *xmin* and *xmax* should be set so that they describe the unit cell. For example, in order to express a unit cell which has side length 128 and centered at (0, 0, 0), you should set $xmin = -64$ and $xmax = 64$. By doing so, the closest particle image is automatically chosen for the force calculation. Forces from all other replicas are omitted, if the scale length η of the cutoff function is properly set (*cf.* `g5_set_eta()`).

The argument *mmin* determines the lower limit of mass of particles. `g5_set_range()` scales the mass so that the force from a particle which has mass *mmin* does not underflow even at maximum distance $\sqrt{3}(xmax/2 - xmin/2)$. If you use a particle which has mass smaller than *mmin*, the force can be so small that a part of its lower bits are lost. The resolution of the mass is *mmin*, i.e., all masses in the range of [*mmin*, 2×*mmin*) are treated as *mmin*. The maximum expressible number of the mass is $511 \times 2^{31} \times mmin \approx 10^{12} \times mmin$. However, the mass should be chosen so that the following relation always

holds during the simulation, in order to avoid the overflow of the force.

$$\frac{m}{mmin} \leq \left(\frac{r}{rmin}\right)^2 \tag{1}$$

Here, m is the mass of the particle, r is the distance (including softening) between the particle and the position at which force is evaluated, and rmin $\approx 10^{-7} \times (xmax - xmin)$. This limitation comes from the fact that G5PIPE pipeline uses 57-bit fixed-point format for force.

For example, in the case of simulation with equal-mass particles, you can simply set the mass of the particle to $mmin$. The softening should be larger than $rmin$ so that the equation (1) holds. In the case of simulation with particles whose masses are different each other, set $mmin$ small enough to resolve the masses of different particles. And set the softening of i-th particle, ϵ_i, large enough to satisfy the relation $\epsilon_i \geq rmin\sqrt{m_i/mmin}$. Here m_i is the mass of i-th particle.

void g5_set_jp(int *adr,* **int** *nj,* **double ****mj,* **double (****xj)*[3]**)**   stores masses *mj[0]...mj[nj−1]* and positions *xj[0]...xj[nj−1]* of *j*-particles to the memory unit (*cf.* section 1 step 1). The masses and positions of *nj j*-particles are stored from the *adr*-th location to the *(adr+nj−1)*-th location of the memory unit. The number *adr + nj* must not exceed the size of the memory, which can be obtained by `g5_get_jmemsize()`.

void g5_set_xi(int *ni,* **double (****xi)*[3]**)**   stores positions *xi[0]...xi[ni−1]* of *ni i*-particles into the input buffer (*cf.* section 1 step 2). When you start calculation by `g5_run()`, forces are evaluated at these positions. The number *ni* must not exceed the value returned by `g5_get_number_of_pipelines()`.

void g5_set_n(int *n***)**   sets the number of particles *n* stored in the memory unit. The forces from the *n* particles are accumulated at the positions set by `g5_set_xi()`, when you invoke `g5_run()`. The number *n* must not exceed the size of the memory, which can be obtained by `g5_get_jmemsize()`.

void g5_run(void)   starts the force calculation. The forces from *j*-particles stored in the memory unit are evaluated and accumulated at the positions of *i*-particles set by `g5_set_xi()`.

void g5_set_eps2(int *ni,* **double ****eps2***)**   sets softening parameters to *ni i*-particles. Set square of the softening parameters to *eps2[0]...eps2[ni−1]*. The number *ni* must not exceed the value returned by `g5_get_number_of_pipelines()`. **This function call must precede `g5_set_xi()`.**

void g5_set_eps2_to_all(double *eps2***)**   sets an identical softening parameter for all *i*-particles. Set square of the softening parameter to *eps2*. **This function call must precede `g5_set_xi()`.**

void g5_get_force(int *ni*, **double (****a***)[3], double ****p***)** waits the force calculation finishes, and retrieves *ni* forces from pipeline units. The forces and potential are set to *a[0]...a[ni−1]* and *p[0]...p[ni−1]*, respectively. The number *ni* must not exceed the value returned by **g5_get_number_of_pipelines()**. Usually, the value *ni* given to this function is the same as that given to the corresponding **g5_set_xi()**.

Note that G5PIPE currently does not calculate gravitational potential, and thus **the potential values returned to** *p* **are not valid**.

void g5_calculate_force_on_x(double (**x***)[3], double (****a***)[3], double ****p***, int** *ni***)** is a wrapper for force-calculation loop (*cf.* section 1 step 2 to 7) which would often appear in user's application program.

It calculates the forces at *ni* positions of *i*-particles given by *x*, by internally invoking basic **g5_** functions. It sets the positions of *i*-particles by **g5_set_xi()**, starts the force calculation by **g5_run()**, wait until the force calculation finishes, retrieves the calculated force by **g5_get_force()**, and returns the result to *a*.

Note that G5PIPE currently does not calculate gravitational potential, and thus **the potential values returned to** *p* **are not valid**.

If the number *ni* exceeds the value returned by **g5_get_number_of_pipelines()**, this function automatically split the positions into subgroups, and calculates the forces for each subgroup. That is, repetition described in the step 8 of section 1 is automatically handled).

int g5_get_number_of_pipelines(void) returns the size of the input buffer (figure 1). This number denotes the maximum number of *i*-particles which can be processed by a single sequence of **g5_set_xi()**, **g5_run()**, and **g5_get_force()** call (*cf.* section 1 step 2 to 7).

int g5_get_jmemsize(void) returns size of the memory unit, i.e., the maximum number of *j*-particles which can be stored into the memory unit. If multiple cards are in use, the total number of particles which can be stored using all of them is returned. Table 1 summarises the size for each model.

void g5_set_eta(double eta) sets the scale length η for the cutoff function. The function shape is fixed to the one used to calculate Particle-Mesh force in P^{3}M method[4], and is not programmable. Once η is set, the force is multiplied by the cutoff function $g_{\mathrm{P3M}}(R)$ expressed as:

$$g_{\mathrm{P3M}}(R) = \begin{cases} 1 - \dfrac{1}{140}(224R^3 - 224R^5 + 70R^6 + 48R^7 - 21R^8) & \text{for} \quad 0 \leq R < 1 \\[2mm] 1 - \dfrac{1}{140}(12 - 224R^2 + 896R^3 - 840R^4 + 224R^5 + 70R^6 - 48R^7 + 7R^8) \\[2mm] & \text{for} \quad 1 \leq R < 2 \\[2mm] 0 & \text{for} \quad R \geq 2, \end{cases}$$

$$(2)$$

where R is the inter-particle distance (includes softening) scaled by η. For application of the GRAPE hardwares to the P^{3}M method, see [5].

18

double g5_get_pcibus_freq(void)   returns the operation cycle of the PCI-X bus in MHz unit. The returned value should be one of 133.0, 100.0, 66.0. This function might be useful to calculate the peak performance of the system on the fly (you can find an example in /usr/g7pkg/direct/directtest.c). **This funtion is obsolete and may not be supported in the next version.**

void g5_get_range(double *xmin,* **double** *xmax,* **double** *mmin)*   returns the scaling factor of spatial coordinate and mass. **This funtion is obsolete and may not be supported in the next version.**

void g5_set_xmj(int *adr,* **int** *nj,* **double (****xj)***[3], double ****mj)*   is the same as g5_set_jp(), except that the meaning of the 3rd and 4th argument is interchanged. This function is provided only for backward compatibility. **This funtion is obsolete and may not be supported in the next version.**

void g5_set_eps(int *ni,* **double ****eps)*   sets softening parameters to *ni i*-particles. Set the softening parameters to *eps[0]...eps[ni−1]*. The number *ni* must not exceed the value returned by g5_get_number_of_pipelines(). **This function call must precede** g5_set_xi(). **This funtion is obsolete and may not be supported in the next version.**

void g5_set_eps_to_all(double *eps)*   sets an identical softening parameter *eps* for all *i*-particles. **This function call must precede** g5_set_xi(). **This funtion is obsolete and may not be supported in the next version.**

void g5_set_cards(int *c)   specifies the cards to be used. In order to use a card which has device ID *i*, set the *i*-th element of the array *c* to unity. Set elements for non-used cards to zero. For example, in order to use cards with device ID 0, 2, and 3 on a system with 5 cards in total, set the elements of *c* as:

```
c = {1, 0, 1, 1, 0};
```

The number of elements of the array *c* should be larger or equal to the number of cards attached to the host computer. **This funtion is obsolete and may not be supported in the next version.**

void g5_get_cards(int *c)   returns to array *c* a list of cards currently in use. The meaning of the list is the same as that for g5_set_cards(). **This funtion is obsolete and may not be supported in the next version.**

int g5_get_number_of_cards(void)   returns the number of cards in use. By defaults, it returns the number of cards attached to the host computer. The number is automatically calculated when the environment variable G5_CARDS is set, or g5_set_cards() is invoked. **This funtion is obsolete and may not be supported in the next version.**

int g5_get_number_of_real_pipelines(void)   returns the number of G5PIPE pipelines in use. If multiple cards are in use, the total number of the pipelines is returned. This function might be useful to calculate the peak performance of the system on the fly (you can find an example in `/usr/g7pkg/direct/directtest.c`). Table 1 summarises the number of pipelines for each model. **This funtion is obsolete and may not be supported in the next version.**

void g5_set_cutoff_table(double (*ffunc)(double), double fcut, double fcor, double (*pfunc)(double), double pcut, double pcor)   exists only for backward compatibility. It actually does nothing. **This funtion is obsolete and may not be supported in the next version.**

3.2.2   Primitive Functions in C, Functions in Fortran

The behavior of all primitive functions is mostly the same as that of standard ones. The only difference is that the former can individually handle each card. The behavior of all functions in Fortran is the same as that of C's.

4 Examples

4.1 An example in C

The following code fragment shows how to evaluate accelerations on nj particles by direct-summation algorithm. Note that **the number of particles it can handle is limited by the number returned by** `g5_get_jmemsize()`. For an example without this limitation, see section 4.3.

```c
#include <stdio.h>
#include "g5util.h"
#define NJMAX (10000)

void main(int argc, char **argv)
{
    int i, nj, step, final_step = 100;
    double eps, size;
    static double mj[NJMAX], xj[NJMAX][3], a[NJMAX][3], p[NJMAX];

    readnbody(&nj, mj, xj, vj, argv[1]); // set initial distribution of particles.
    g5_open();
    size = 10.0;
    g5_set_range(-size/2.0, size/2.0, 1.0);

    for (step = 0; step < final_step; step++) {
        g5_set_jp(0, nj, mj, xj);     // send particles which exert forces.
        g5_set_eps2_to_all(eps*eps); // send a softening parameter.
        g5_set_n(nj);                 // set number of particles which exert forces.
        g5_calculate_force_on_x(xj, a, p, nj); // send particles which ``feel''
                                      // the force, then G5 calculate
                                      // the force on particles, and send
                                      // them back to 'a'. 'p' is not used
                                      // for now.

        integrate(xj, vj, a, dt, nj);  // update particle position.
    }
    g5_close();
    writenbody(nj, mj, xj, vj, argv[2]);
}
```

4.2 An example in Fortran

The following code fragment shows how to evaluate accelerations on nj particles by direct summation. Note that **the number of particles it can handle is limited by the number returned by** g5_get_jmemsize(). For an example without this limitation, see section 4.3.

```
#include <g5util.h>
#define REAL real*8
#define NJMAX (10000)

       program direct_summation

       REAL mj(NJMAX), xj(3, NJMAX), vj(3, NJMAX)
       REAL a(3, NJMAX), p(NJMAX)
       REAL xmax, xmin, mmin
       REAL eps, epsinv, dt
       integer nj
       integer step, final_step
       integer i, j, k

C      set initial distribution of particles.
       call readnbody(nj, mj, xj, vj)

       final_step = 100
       eps = 0.02
       dt = 0.01
       xmax = 10.0
       xmin = -10.0
       mmin = mj(1)
       call g5_open()
       call g5_set_range(xmin, xmax, mmin)

       do step=1,final_step
C          send particles which exert forces.
           call g5_set_jp(0, nj, mj, xj)

C          send a softening parameter.
           call g5_set_eps2_to_all(eps*eps)

C          set number of particles which exert forces.
           call g5_set_n(nj)

C          send particles which ''feel'' the force, then
C          G5 calculate the force on particles, and send
```

```
C          them back to 'a'.
C          'p' is not used for now.
           call g5_calculate_force_on_x(xj, a, p, nj)

C          update particle positions.
           call integrate(xj, vj, a, dt, nj)
        enddo

        call g5_close()
        call writenbody(nj, mj, xj, vj)

        end
```

4.3   Another example in C

The following code fragment shows how to evaluate accelerations on nj particles by direct-summation algorithm. The number of particles it can handle is not limited by g5_get_jmemsize(), in contrast to the example shown in section 4.1. The particles are splited into subgroups which do not exceed the size of memory unit. Forces from those subgroups are calculated individually, and then the results are summed up by the host computer.

```c
#include <stdio.h>
#include "g5util.h"
#define NJMAX (10000)

void main(int argc, char **argv)
{
    int i, j, nj, njtmp, step, final_step = 100, jmemsize;
    double eps, size;
    static double mj[NJMAX], xj[NJMAX][3], a[NJMAX][3], p[NJMAX];
    static double atmp[NJMAX][3], ptmp[NJMAX][3];

    readnbody(&nj, mj, xj, vj, argv[1]);
    g5_open();
    size = 10.0;
    g5_set_range(-size/2.0, size/2.0, 1.0);
    jmemsize = g5_get_jmemsize();
    for (step = 0; step < final_step; step++) {
        for (i = 0; i < nj; i++) {  // clear the force buffer for all nj particles.
            for (k = 0; k < 3; k++) {
                a[i][k] = 0.0;
            }
        }
        for (j = 0; j < nj; j += jmemsize) { // for each subgroups that contains
            njtmp = jmemsize;                // jmemsize or smaller number of particles,
            if (j + jmemsize > nj) {
                njtmp = nj - j;
            }
            g5_set_jp(0, njtmp, mj + j, xj + j); // set njtmp particles to the particle
                                                 // memory unit.
            g5_set_eps2_to_all(eps*eps);
            g5_set_n(njtmp);               // calculate force from njtmp particles
            g5_calculate_force_on_x(xj, atmp, ptmp, nj); // to all nj particles.
            for (i = 0; i < nj; i++) {  // accumulate force from njtmp particles.
                for (k = 0; k < 3; k++) {
                    a[i][k] += atmp[i][k];
                }
            }
```

```
            }
        integrate(xj, vj, a, dt, nj);
    }
    g5_close();
    writenbody(nj, mj, xj, vj, argv[2]);
}
```

References

[1] Kawai A., Fukushige T., Makino J., and Taiji M.,
   GRAPE-5: A Special-Purpose Computer for N-Body Simulations,
   Publ. Astron. Soc. Japan (2000), Vol. 52, p. 659,
   `http://xxx.lanl.gov/abs/astro-ph/9909116`.

[2] Teuben P. J.,
   NEMO - A Stellar Dynamics Toolbox,
   `http://bima.astro.umd.edu/nemo/`.

[3] Barnes J. E. and Hut P.,
   A Hierarchical $O(N \log N)$ Force Calculation Algorithm,
   Nature (1986), Vol. 324, p. 446.

[4] Hockney R. W. and Eastwood J. W.,
   Computer Simulation Using Particles,
   (McGraw-Hill, New York, 1981) ch5.

[5] Yoshikawa K. and Fukushige T.,
   *PPPM and TreePM Methods on GRAPE Systems for
   Cosmological N-Body Simulations*,
   Publ. Astron. Soc. Japan (2005), Vol. 57, p. 849.

Acknowledgment

Modification History

version	date	description	author(s)
2.1	11-Feb-2008	Support for Model 800.	AK
2.0	23-Apr-2007	Description for cutoff function added.	AK
1.3	11-Mar-2007	Section 2.4 added.	AK
1.2	03-Mar-2007	Acknowledgement added.	AK
		Description of `G5_SENDFUNC` added.	
		A sample code added.	
1.1	19-Feb-2007	Description of `g5_set_range` corrected.	AK
1.0	13-Feb-2007	Description for new features added.	AK
0.0	26-Jan-2007	Created	AK