Ruprecht-Karls-University Heidelberg
faculty of computer science
Special Interest Group on High Performance I/O

Software pratical for beginners

Distribution functions in PVFS2, hints in MPICH2 and performance measurement

Tobias Eberle
Dossenheimer Landstraße 90
69121 D-Heidelberg
Germany
phone: +49-6221-8936490
e-mail: tobias.eberle@gmx.de

My partner:
Frederik Grüll
e-mail: frederik.gruell@web.de

March and April 2005

Supervisors:
Dipl. Inf. Hipolito Vasquez
Prof. Dr. Thomas Ludwig

PVFS2 is an open source parallel filesystem. A file saved on PVFS2 is scattered over several I/O-servers. This scattering is done by a distribution function whose implementation is described here. Also it is presented how PVFS2 does the data file $\rightarrow$ I/O-server assignment and how you can hardcode a new assignment pattern. To set distribution parameters by MPI programs, using the MPICH2 implementation, new hints such as `distribution_name` for using a specified distribution function are described. At last the performance of several distribution functions are measured and compared.

Contents

1 Introduction

Assume that you have written an MPI program using MPICH2 that calculates something very difficult and now you want to write this results into one single file. The simplest and slowest solution is to send all data to *one I/O-server* which writes it to its harddisk. A faster solution is to use a parallel filesystem such as *PVFS2* which scatters the file across several servers and the fastest solution is to use *PVFS2 and assure that every compute node writes its data to its own harddisk.* The task was to implement these three solutions and to measure the bandwidth that can be achieved by each one for comparing their performance.

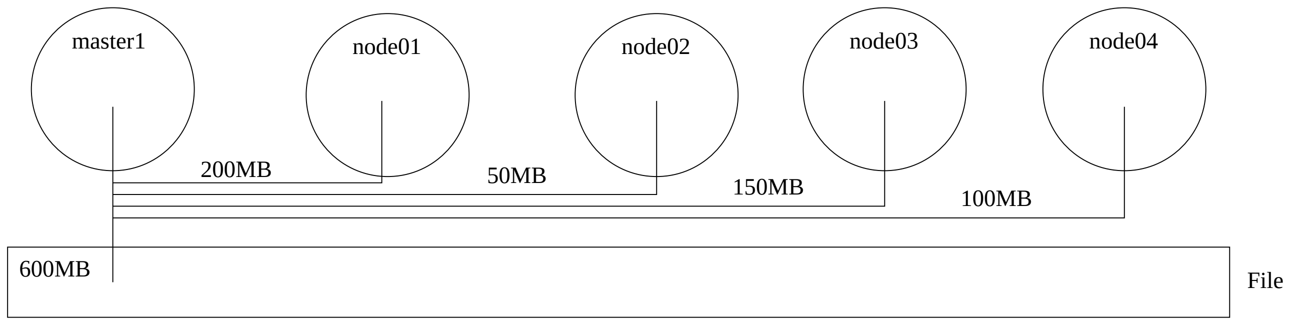

Figure 1.1: Solution 1: Every compute node sends its data to master1

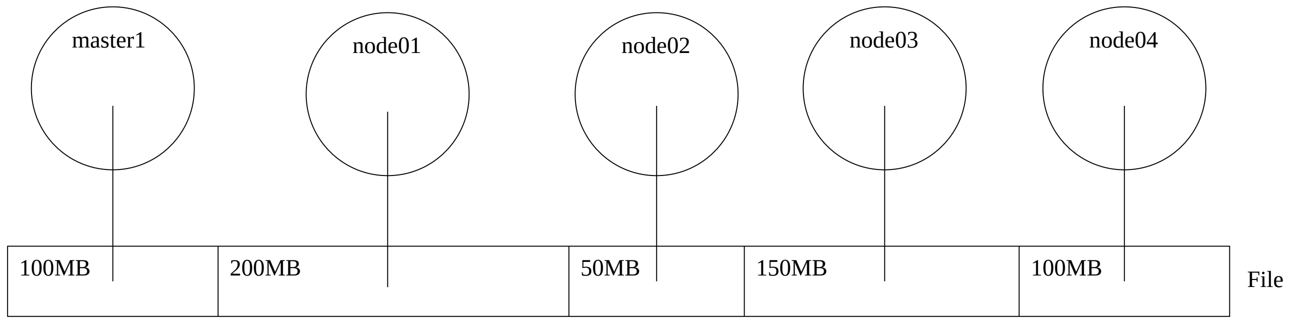

Figure 1.2: Solution 3 using varstrip distribution: One single file, but every compute node writes to its own harddisk (see chapter 2.3)

2 Distribution functions in PVFS2

2.1 Task of a distribution and data files

The task of a distribution function is to scatter a file across the existing I/O-servers in a specific way. Each I/O-server is assigned to a data file which is saved on the harddisk of the server.

2.2 Distribution functions shipped with PVFS2

There are two distribution functions that are shipped with PVFS2:

- **basic_dist**
  `basic_dist` is a very simple distribution function: All the data is saved at one I/O-server. Therefore it is not very useful, maybe for testing purpose.

- **simple_stripe**
  A file is devided into strips of the same size. The first strip is saved on I/O-server 1, the second on I/O-server 2 and so on. If all I/O-servers were used the next strip is saved on the first one. This method is also called round-robin. The default strip size is 64KB. You can change the strip size within an MPI program by using the hint `striping_unit`. The `striping_unit` hint is not implemented by MPICH2 version 1.0.1 yet. You must apply the patch described in chapter 3.3.1.

2.3 New distribution function: varstrip_dist

To implement the third solution[1] we need a more flexible distribution function than `simple_stripe`. `varstrip_dist` is like `simple_stripe`, but the strips are flexible in size and assigned to a specified data file. `varstrip_dist` is configured with a string which has the following format:

```
<config> ::= {<datafile number>:<strip size>[k|K|m|M|g|G];}+
<datafile number> ::= a value between 0 and (number of I/O servers - 1)
<strip size> ::= strip size in: k, K: Kilobyte
                                m, M: Megabyte
                                g, G: Gigabyte
                                no specification: Byte
```

As you can see data file numbers can be included more than once. If you write more bytes than the whole size specified by this configuration string, the same format is used again.

Example

For the diagram shown in the introduction the strips parameter would be:

```
strips = "0:100M;1:200M;2:52428800;3:150M;4:102400K"
```

[1]Using PVFS2 and asuring that every compute node writes to its own harddisk

2.4 How to implement a new distribution function

2.4.1 Location of distribution functions in the PVFS2 source code

The distribution function specific code is located at `/src/io/description/` in the PVFS2 source code:[2]

```
teberle@master1:~/src/pvfs2/src/io/description$ ls
dist-basic.c              module.mk               pint-distribution.h  pvfs-request.c
dist-varstrip-parser.c    module.mk.in            pint-request.c       usage-notes.txt
dist-varstrip-parser.h    pint-dist-utils.c       pint-request.h
dist-varstrip.c           pint-dist-utils.h       pvfs-distribution.c
dist-simple-stripe.c      pint-distribution.c     pvfs-distribution.h
```

The `dist-*` files are the distributions themself. We will have a closer look at them in chapter 2.4.3. `pint-dist-utils.*` contain the default implementation of some distribution function methods and the distribution registration and `pint-distribution.*` contain the internal representation of a distribution. The other files are not needed for our purpose.

2.4.2 Distribution function header file

This file should be located at `/include` and be named `pvfs2-dist-distribution_name.h`. It includes a struct that contains all parameters of the distribution function.

Example: varstrip_dist

The parameter struct is defined as

```
struct PVFS_varstrip_params_s
{
  char strips[PVFS_DIST_VARSTRIP_MAX_STRIPS_STRING_LENGTH];
};
typedef struct PVFS_varstrip_params_s PVFS_varstrip_params;
```

The format of this string is described in 2.3.

2.4.3 Description of methods a distribution function must implement

First, we need two naming conventions: The `logical offset` of a file is the offset of a byte inside of the single file we write and the `physical offset` is the offset of the same byte inside of the data file it belongs to.

Distribution functions have to implement to following methods:

```
  PVFS_offset (*logical_to_physical_offset)(void* params,
                                            uint32_t dfile_nr,
                                            uint32_t dfile_ct,
                                            PVFS_offset logical_offset);
```

`logical_to_physical_offset()` returns the `physical offset` inside of the data file, specified with `dfile_nr`, that corresponds to the `logical offset`. `dfile_ct` contains the number of data files used and `params` is a pointer to the parameter structur. The documentation[3] says that if the logical offset does not belong to the data file, the last physical offset belonging to the data file should be returned. In practise this issue never happend to me and I have not implemented this to `varstrip_dist`.

[2] You can download the source code from http://www.pvfs.org/pvfs2/
[3] /doc/design/distributions.pdf

```
PVFS_offset (*physical_to_logical_offset)(void* params,
                                          uint32_t dfile_nr,
                                          uint32_t dfile_ct,
                                          PVFS_offset physical_offset);
```

`physical_to_logical_offset()` is `logical_to_physical_offset()` vice versa. It returns the logical offset that belongs to the physical offset inside of data file number `dfile_nr`.

```
PVFS_offset (*next_mapped_offset)(void* params,
                                  uint32_t dfile_nr,
                                  uint32_t dfile_ct,
                                  PVFS_offset logical_offset);
```

`next_mapped_offset()` returns the logical offset next to `logical_offset` which belongs to the data file number `dfile_nr`. If `logical_offset` already belongs to it then `logical_offset` must be returned.

```
PVFS_size (*contiguous_length)(void* params,
                               uint32_t dfile_nr,
                               uint32_t dfile_ct,
                               PVFS_offset physical_offset);
```

Getting the physical offset of `dfile_nr` `contiguous_length()` returns the number of bytes that are continguous in the logical file and saved into this data file. For example if the distribution function is `simple_stripe`, the strip size 65536 bytes and the physical offset 30000, then $65536 - 30000 = 35536$ is returned. Attention: Is physical offset equal to 65535 (the last byte in the strip), then 1 is returned!

```
PVFS_size (*logical_file_size)(void* params,
                               uint32_t dfile_ct,
                               PVFS_size *psizes);
```

`logical_file_size()` returns the size of the logical file in bytes. `*psizes` is a pointer to an array with `dfile_ct` items that contains the file sizes of each data file.

```
int (*get_num_dfiles)(void* params,
                      uint32_t num_servers_requested,
                      uint32_t num_dfiles_requested)
```

`get_num_dfiles()` returns the number of used data files. This number must not be greater than `num_servers_requested`. `num_dfiles_requested` is the number of data files the user requested by his program.[4] There is a default implementation called `PINT_dist_default_get_num_dfiles()` which returns either the number of data files requested if greater than zero or the number of servers requested.

```
int (*set_param)(const char* dist_name,
                 void* params,
                 const char* param_name,
                 void* value);
```

Sets a parameter. `dist_name` contains the distribution name, `params` is a pointer to the structure that contains all parameters of the distribution function, `param_name` is the name of the parameter defined in the structure and `*value` is a pointer to a variable that contains the value. It has the same type as the parameter. There is a default implementation called `PINT_dist_default_set_param()`.

```
void (*encode_lebf)(char **pptr, void* params);
```

This method is used to save all parameters inside of a contiguous space in memory. `**pptr` points to the space in memory that must be used. You must save the parameter in little endian format. To convert parameters to this format PVFS2 contains many predefined macros. To use them

[4]An MPI program uses the hint striping_factor to request a number of data files.

you must call `#define __PINT_REQPROTO_ENCODE_FUNCS_C` before you include `pvfs2-types.h` and `pint-distribution.h` respectively. Have a look at `src/proto/endecode-funcs.h` to discover the converting macros.

```
  void decode_lebf(char** pptr, void* params)
```

This method does the opposite of `encode_lebf`.

```
  void registration_init(void* params)
```

`registration_init()` is called at distribution registration time. We must register our parameters here. To do so, we must call `PINT_dist_register_param()` with the following parameters: distribution name, parameter name as string, parameter structure as defined by the header file and the parameter itself. For example varstrip distribution calls:

```
  PINT_dist_register_param(PVFS_DIST_VARSTRIP_NAME, "strips",
                           PVFS_varstrip_params, strips)
```

All the implemented methods should be saved to `/src/io/description/dist-distribution_name.c`. Do not forget to include this filename into `/src/io/description/module.mk.in` and run `./configure` again, otherwise the distribution function does not get compiled by running `make`.

2.4.4 Register the distribution

In order that PVFS2 does know about a distribution function it must be registered. This is done in two steps:

1. Add the following to the file that contains the distribution function methods:

```
/* default parameters */
static PVFS_varstrip_params varstrip_params = { "\0" };

/* struct with all my implemented methods */
static PINT_dist_methods varstrip_methods = {
  logical_to_physical_offset,
  physical_to_logical_offset,
  next_mapped_offset,
  contiguous_length,
  logical_file_size,
  get_num_dfiles,
  set_param,
  encode_lebf,
  decode_lebf,
  registration_init
};
/* fill the internal pvfs2 distribution interface */
PINT_dist varstrip_dist = {
  PVFS_DIST_VARSTRIP_NAME,                /* distribution name */
  roundup8(PVFS_DIST_VARSTRIP_NAME_SIZE), /* name size */
  roundup8(sizeof(PVFS_varstrip_params)), /* param size */
  &varstrip_params,
  &varstrip_methods
};
```

2. Add “`extern PINT_dist varstrip_dist`“ to `/src/io/description/pint-dist-utils.c` add and

```
    /* Register the varstrip distribution */
    PINT_register_distribution(&varstrip_dist);
```

to `int PINT_dist_initialize(void)`.

2.5 Changing data file assignment to I/O-servers

After we have written a new distribution function the next problem to solve is to assure that each compute node writes on its own harddisk. The data file → I/O-server assignment is done inside of

```
/src/common/misc/pint-cached-config.c::PINT_cached_config_get_next_io()
```

The current default behaviour is to select one server randomly that gets data file number 0 and then go through the list of available servers. To get fixed data file assignment edit this method and set `jitter` to a fixed value. You can print the assignment by adding

```
int iTmp;
fprintf(stderr, "Data file number: %d - Server name: %s\n", i,
        PVFS_mgmt_map_addr(msg_p->fs_id, sm_p->cred_p,
                           msg_p->svr_addr, &iTmp));
```

to the end of the `for-loop` inside of

```
/src/client/sysint/sys-create.sm::create_datafiles_setup_msgpair_array()
```

You can also use my patch to add this code.[5]

[5]see chapter B.2 in the appendix

3 Hints in MPICH2

Hints are used by MPI programs to pass parameters to `MPI_File_*` methods. They are simple key-value pairs.

3.1 Existing hints

Currently available hints provided by MPICH2 version 1.0.1 that can be used with PVFS2 are:

- `striping_factor`: striping_factor can be used to request the number of data files that should be used. The striping factor is only relevant at file creation time.

3.2 Using hints

Using hints is pretty easy:

```
MPI_File mpiFileHandle;
MPI_Info mpiFileInfo;
/* create MPI Info object */
MPI_Info_create(&mpiFileInfo);
/* set striping_unit to 8MB*/
MPI_Info_set(mpiFileInfo, "striping_unit", "8388608");
/* open the file write only, create the file if not exists */
MPI_File_open(MPI_COMM_WORLD, "pvfs2:/path/file",
          MPI_MODE_WRONLY | MPI_MODE_CREATE,
          mpiFileInfo, &mpiFileHandle);
/* set file view of this process */
MPI_File_set_view(mpiFileHandle,
                  iRank * iMbytes * 1024 * 1024 * sizeof(char), MPI_BYTE,
                  MPI_BYTE, "native", MPI_INFO_NULL);
/* write some data to file */
MPI_File_write(mpiFileHandle, acBuffer, iMbytes * 1024 * 1024, MPI_BYTE,
               MPI_STATUS_IGNORE);
/* close file */
MPI_File_close(&mpiFileHandle);
/* delete MPI Info object */
MPI_Info_free(&mpiFileInfo);
```

The example above opens a file write only and writes some data to it. It uses the `simple_stripe` distribution with a strip size of about 8MB. To use hints an `MPI_Info` object must be created using `int MPI_Info_create(MPI_Info *info)`. With `int MPI_Info_set(MPI_Info info, char *key, char *value)` hints are set. After finishing the use of the info object it must be destroyed with `int MPI_Info_free (MPI_Info *info)`; the parameter is automatically set to `MPI_INFO_NULL`.
For further information about methods manipulating an info object have a look at http://www-unix.mcs.anl.gov/mpi/www/www3/

3.3 New hints

3.3.1 striping_unit

With `striping_unit` the strip size of `simple_stripe` can be set. Default is 65536 bytes.

Example

```
MPI_Info_set(mpiFileInfo, "striping_unit", "8388608");
```

3.3.2 distribution_name

`distribution_name` is used to choose the distribution function. `simple_stripe` is the default distribution function that is used if the hint is not set. If the specified distribution function does not exist program execution is aborted.

Example

```
MPI_Info_set(mpiFileInfo, "distribution_name", "simple_stripe");
```

3.4 Add-on for distribution function parameters

Because it is very circumstantial to implement a hint for every parameter of all distribution functions and to avoid recompilation of MPICH2 if a new distribution function should be used I have implemented a general solution which consists of a defined key format that is parsed by MPICH2.

Key format

```
<Key> ::= <distribution name>:<type>:<parameter name>
<distribution name> ::= simple_stripe|basic_dist|varstrip_dist|...
<type> ::= [unsigned ]int|[unsigned ]int64|char|string|double
<parameter name> ::= <parameter the name defined by the distribution>
```

Examples

1. The striping_unit can now be set another (more complicated ;-)) way. The parameter of `simple_stripe` is called `strip_size` that is from type `PVFS_size` which is a `int64`.

   ```
   MPI_Info_set(mpiFileInfo, "simple_stripe:int64:strip_size", "8388608");
   ```

2. The parameter of `varstrip_dist` is a string called `strips`:

   ```
   MPI_Info_set(mpiFileInfo, "varstrip_dist:string:strips", "0:100000;1:200000");
   ```

Notes

- To use this add-on you must set `distribution_name` even if the default distribution function is used!
- Parameters containing not used distribution function names are ignored.

3.5 How to implement a new hint

3.5.1 Overview

MPICH2 includes ROMIO[1] which implements the I/O features of MPI2[2]. The file system specific code of ROMIO is called ADIO[3]. In the source code of MPICH2[4] PVFS2 specific code is located at

```
/src/mpi/romio/adio/ad_pvfs2/
```

The important files are:

- `ad_pvfs2_hints.c`

  ```
  void ADIOI_PVFS2_SetInfo(ADIO_File fd, MPI_Info users_info, int *error_code)
  ```

  is implemented in this file. `user_info` contains all hints the user set in his MPI program. The method puts them to `fd->info`. It also checks that all processes belonging to the communicator have got the same hint value.
  Only hints that are implemented by this method can be used by the open, read and write methods.

- `ad_pvfs2_open.c`
  Opens a file. All hints that are only relevant at file creation time must be implemented there.

- `ad_pvfs2_read.c`, `ad_pvfs2_write.c`
  Reading data from a file and writing data to a file. Hints that are passed to `MPI_File_read()` and `MPI_File_write()` must be implemented there.

3.5.2 Example: Implementation of distribution_name

`/src/mpi/romio/adio/ad_pvfs2/ad_pvfs2_hints.c`

The code for the new hint must be included into the body of this if-clause:

```
/* any user-provided hints? */
if (users_info != MPI_INFO_NULL) {
  /* code for new hint */
}
```

Code for `distribution_name` added there:

```
/* hint: distribution_name */
/* allocate memory for value */
char *szDistributionName = (char *)ADIOI_Malloc((MPI_MAX_INFO_VAL+1)*sizeof(char));
/* initialize memory */
memset(szDistributionName, '\0', MPI_MAX_INFO_VAL+1);
/* get the value */
MPI_Info_get(users_info, "distribution_name",
             MPI_MAX_INFO_VAL, szDistributionName, &flag);

/* does the user set the hint? */
```

[1] http://www-unix.mcs.anl.gov/romio/
[2] http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html
[3] abstract device I/O layer
[4] http://www-unix.mcs.anl.gov/mpi/mpich2/

```
  if (flag)
  {
    /* assure that every process belonging to the communicator does have
     * the same hint. */
    char *szTmpValue = (char *) ADIOI_Malloc( (MPI_MAX_INFO_VAL+1)*sizeof(char));
    strcpy(szTmpValue, szDistributionName);
    /* rank 0 broadcasts the distribution name to the other ranks */
    MPI_Bcast(szTmpValue, strlen(szDistributionName) + 1, MPI_CHAR, 0, fd->comm);
    /* broadcasted value equal to my value? */
    if (strcmp(szTmpValue, szDistributionName) != 0)
    {
      /* no it is not! */
      /* cleanup and error handling */
      ADIOI_Free(szTmpValue);
      szTmpValue = NULL;
      ADIOI_Free(szDistributionName);
      szDistributionName = NULL;
      MPIO_ERR_CREATE_CODE_INFO_NOT_SAME(myname,
              "distribution_name",
              error_code);
      return;
    }
    /* every process has the same value */
    /* set hint to fd->info */
    MPI_Info_set(fd->info, "distribution_name", szDistributionName);
    /* cleanup */
    ADIOI_Free(szTmpValue);
    szTmpValue = NULL;
  }
  else
  {
    /* hint is not set */
    /* set default value */
    strcpy(szDistributionName, "simple_stripe");
    MPI_Info_set(fd->info, "distribution_name", szDistributionName);
  }
  /* cleanup */
  ADIOI_Free(szDistributionName);
```

After getting the value from `user_info` we have to assure that each process has got the same `distribution_name` value. Otherwise an error is raised. If everything is alright set the hint to `fd->info` and if the hint does not exist set a default value.

`/src/mpi/romio/adio/ad_pvfs2/ad_pvfs2_open.c`

Opening a file using PVFS2 system interface is done by `fake_an_open()`. I have added the parameter `MPI_Info mpiInfo` to this function that contains all hints set by `ADIOI_PVFS2_SetInfo()`. To use a distribution function other than the default one is done by calling `PVFS_sys_dist_lookup("distribution name")` and passing the return value to `PVFS_sys_create()`:

```
  /* get distribution name */
  char *szDistributionName = (char *)ADIOI_Malloc(sizeof(char)*(MPI_MAX_INFO_VAL+1));
  memset(szDistributionName, '\0', MPI_MAX_INFO_VAL+1);
  int iFlag;
```

```
MPI_Info_get(mpiInfo, "distribution_name", MPI_MAX_INFO_VAL,
             szDistributionName, &iFlag);
/* initialize myDist */
PVFS_sys_dist *myDist = NULL;
if (iFlag)
{
  /* lookup the distribution */
  myDist = PVFS_sys_dist_lookup(szDistributionName);
  if (myDist == NULL)
  {
    /* distribution does not exist! -> Error handling */
    fprintf(stderr, "Could not find distribution %s",
            szDistributionName);
    o_status->error = -1;
    return;
  }
}
/* cleanup */
ADIOI_Free(szDistributionName);
szDistributionName = NULL;
/* open file */
ret = PVFS_sys_create(resp_getparent.basename,
                      resp_getparent.parent_ref, attribs,
                      &(pvfs2_fs->credentials), myDist, &resp_create);
PVFS_sys_dist_free(myDist);
```

4 Performance measurement of distribution functions

In order to get an impression how fast PVFS2 is and to compare simple stripe and varstrip distribution, the performance must be measured. The cluster, the measurement was taken on, consists of one master and four node computers. The four nodes are I/O-servers and compute nodes, the master I/O-server, meta data server and compute node.

4.1 Results

The plots can be found in appendix A.2. The x-axis of the plots is the size of the file part each node writes and the y-axis is the bandwidth.

4.1.1 ext3

At first it is necessary to measure raw harddisk performance because the achieved bandwidth by this test is the maximum performance PVFS2 can reach in the tests below.
Figure A.2 shows the write and figure A.3 the read performance. Read performance is about 50MB/s per node and write performance about 40MB/s except of node01 whose write performance is only about 30MB/s. To check if the results are correct `bonnie++` was run one some nodes.[1] Bonnie++' output (see appendix A.1) shows the same results as we measured. Therefore we can assume that we measured the real and not cache performance. Another argument pro is the measurement done in figure A.1. The plot shows the read performance with cache prevention disabled and results in 50MB/s at file sizes greater than 500MB. Beginning at this point the file is too big to fit into the cache.

4.1.2 PVFS2: simple stripe

strip size: 64KB, 8MB, 16MB

Write performance of 64KB strip size is displayed in figure A.4. All nodes are close together. The bandwidth is about 22MB/s which is 55% of the maximum. This is not impressive but better than the bad performance Philipp Sadleder describes in his bachelor thesis [5, chapter 7, “Messungen und Testprogramme”]. Read performance is shown in figure A.5. It is about 20MB/s which is 40% of the maximum. To check if the strip size is too small and produces too many server requests, a measurement with 8MB and 16MB strip size was done (see figure A.6, A.7 and A.8, A.9). The results are disappointing. The write performance is slightly better than using 16KB strip size but not significantly. Read performance is the same. The bottleneck must be elsewhere.

strip size: fitted

The last plot in this series shows simple stripe with a strip size that corresponds to the file size each node writes. Each node writes only to its own or to another harddisk. The write figure (A.10) shows

[1] Thanks to Hipolito Vasquez who runs the tests.

a bandwidth of about 30MB/s which is 75% of the maximum. Node04 has a little lesser bandwidth. This can be explained by the fact that node04 has written to the harddisk of node01 whose harddisk is slower than the others (see 4.1.1). The reading figure (A.11) shows that master1 is reading from its own harddisk with full speed and the other nodes are reading from a harddisk of another node. The bandwidth of about 40MB/s (80%) is very good.

4.1.3 PVFS2: varstrip distribution

`varstrip_dist` distribution is configured with strip sizes that fit exactly to the size each node writes. Furthermore it is assured that every node writes to its own harddisk. The measurement shows full speed for every node on reading and 80% on writing. PVFS2 loses performance on writing.

4.1.4 mpi-io-test

I have run `mpi-io-test` with 64KB strip size. The results are shown in figure A.14 and A.15. The bandwidth output of `mpi-io-test` is devided through the number of I/O-servers so that the plots are comparable to the other. The `mpi-io-test` writing results are similar to ours but the reading plot shows cache effects. At big file sizes the cache is too small and the performance is a little bit better than in our measurement.

4.1.5 b_eff_io

`b_eff_io` cannot be run with PVFS2 because shared file pointers are necessary for valid test results but PVFS2 does not support them yet.

4.2 How to do performance measurement

4.2.1 Writing

Writing is done by opening the file with `MPI_File_Open()`, setting a view with `MPI_File_set_view()`, writing data with `MPI_File_write()` and closing the file with `MPI_File_close()`. The time measurements starts before `MPI_File_write()` and stops after finishing. `MPI_File_sync()` is not called because it synchronizes all processes which falsifies the time needed by a single process. Although it is not necessary to synchronize manually because PVFS2's default value (that was used) is to syncronize every 256KB. The following code snippet does the measurement:

```
/* allocate enaugh memory */
acBuffer = (char *)malloc(sizeof(char) * iMbytes * 1024 * 1024);
memset(acBuffer, 'A', iMbytes * 1024 * 1024);

/* file to open */
char *sFilename = "pvfs2:/pvfs2/performance.out";
MPI_File iFileHandle;
/* setting file hints, etc. */
[...]
/* opens the file */
MPI_File_open(MPI_COMM_WORLD, sFilename,
    MPI_MODE_WRONLY | MPI_MODE_CREATE,
    mpiFileInfo, &iFileHandle);

/* sets the file view */
```

```
MPI_File_set_view(iFileHandle,
    iRank * iMbytes * 1024 * 1024 * sizeof(char), MPI_BYTE,
    MPI_BYTE, "native", MPI_INFO_NULL);

/* synchronizes and starts measuring time */
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

/* writes the data */
MPI_File_write(iFileHandle, acBuffer, iMbytes * 1024 * 1024, MPI_BYTE,
               MPI_STATUS_IGNORE);

/* stops measuring writing time */
stop = MPI_Wtime();
/* calculate time duration */
dWriting = stop - start;

/* synchronize: only for carefulness */
MPI_File_sync(iFileHandle);

/* close the file */
MPI_File_close(&iFileHandle);
/* and cleanup */
free(acBuffer);
```

4.2.2 Reading

Reading is done like writing. The only differences are that `MPI_File_open()` is called with `MPI_INFO_NULL` instead of `mpiFileInfo` and `MPI_File_read()` is used instead of `MPI_File_write()`.

4.2.3 Get rid off cache effects on reading

Caches can affect reading times very heavily. They make you believe to have more performance than the system really has if they are not used. For instance reading a file from harddisk with warm caches can be happen with bandwidthes of about 500MB/s but the harddisk can read the data only with 50MB/s in reality. The problem is that you do not know all the caches that are used in the layers between your `read()` call and the disk. One cache that affects reading times supposedly at most, is the file cache of the linux operating system. Fortunatly it is easy to get rid off it. We allocate a big part of the system memory and write this data to a temporary file. Afterwards we delete the file. Here is the code:

```
/* allocate buffer. All our nodes have 1GB RAM. 700MB is enaugh to get rid
 * off the caches. To be sure you should run a test with caches enabled
 * and take a size that is greater than the file size at the point the
 * performance goes down in the graph. Figure \ref{xx} A.1 shows such
 * a test for our cluster. */
acBuffer = (char *)malloc(sizeof(char) * 700 * 1024 * 1024);
memset(acBuffer, 'a', 700 * 1024 * 1024);

/* open the file */
int fd;
fd = open("/tmp/emtycache", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
if (fd == -1)
```

```
{
    perror("");
    MPI_Finalize();
    return -1;
}
/* write the data */
if (write(fd, acBuffer, sizeof(char)* 1024 * 1024 * 700) == -1)
{
    /* do some error handling here */
    return -1;
}

/* sync() is really important! Otherwise you do not know when the operating
 * system flushes the data to disk and your performance measuring will be
 * influenced. */
fsync(fd);
close(fd);
/* cleanup */
if (unlink("/tmp/emtycache") == -1)
{
  /* do some error handling here */
  return -1;
}
free(acBuffer);
```

A Results of the measurements

A.1 Bonnie++' output

		Sequential Output						Sequential Input			
Chunk Size		Per Char		Block		Rewrite		Per Char		Block	
		K/sec	%CPU	K/sec	% CPU	K/sec	%CPU	K/sec	%CPU	K/sec	%CPU
master1	2G	24693	99	41559	26	22289	13	23462	80	51839	10
node01	2G	24353	99	33071	21	19236	10	26438	89	52278	9
node06	2G	24423	99	43704	28	22658	11	26486	88	52994	9
node07	2G	24300	99	44347	29	22694	11	26767	89	53240	9

A.2 Plots

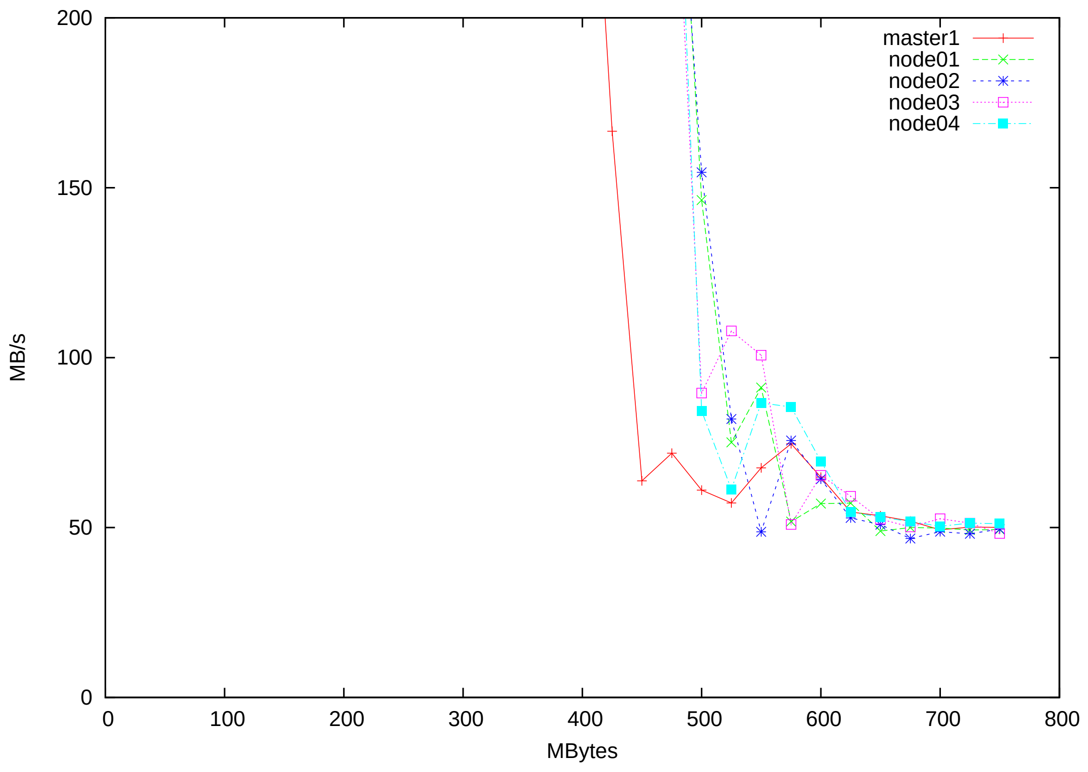

Figure A.1: *Reading* from own harddisk using UNIX `open()`, `write()` and `close()` commands with *caches enabled.*

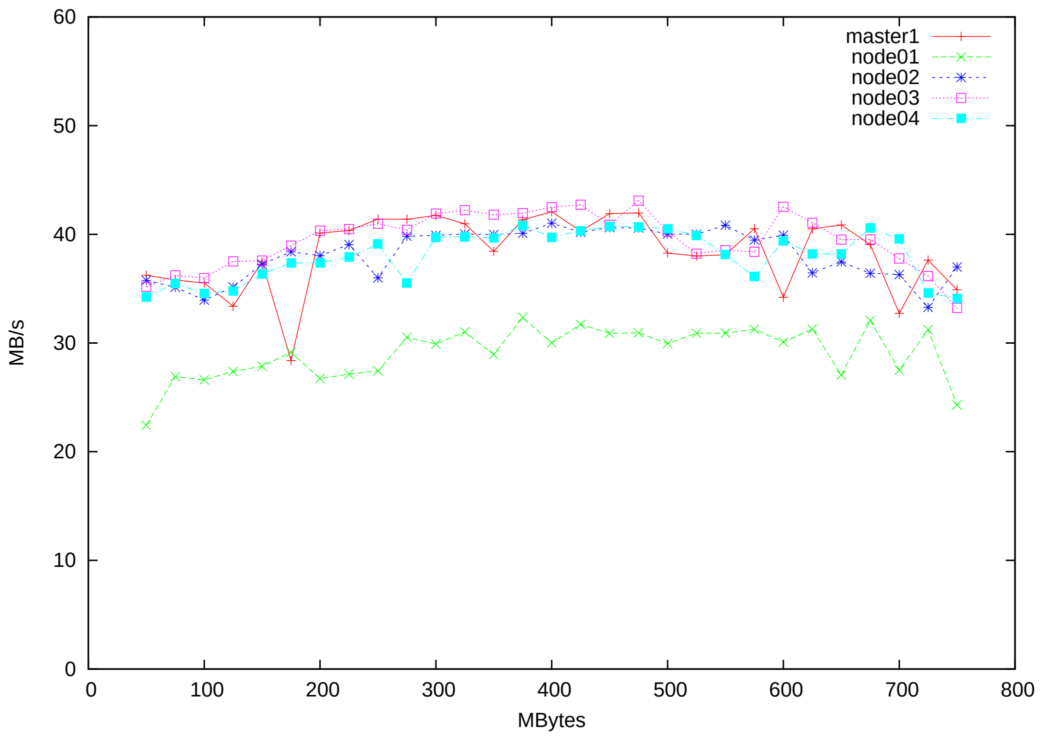

Figure A.2: *Writing* to own harddisk using UNIX `open()`, `write()` and `close()` commands

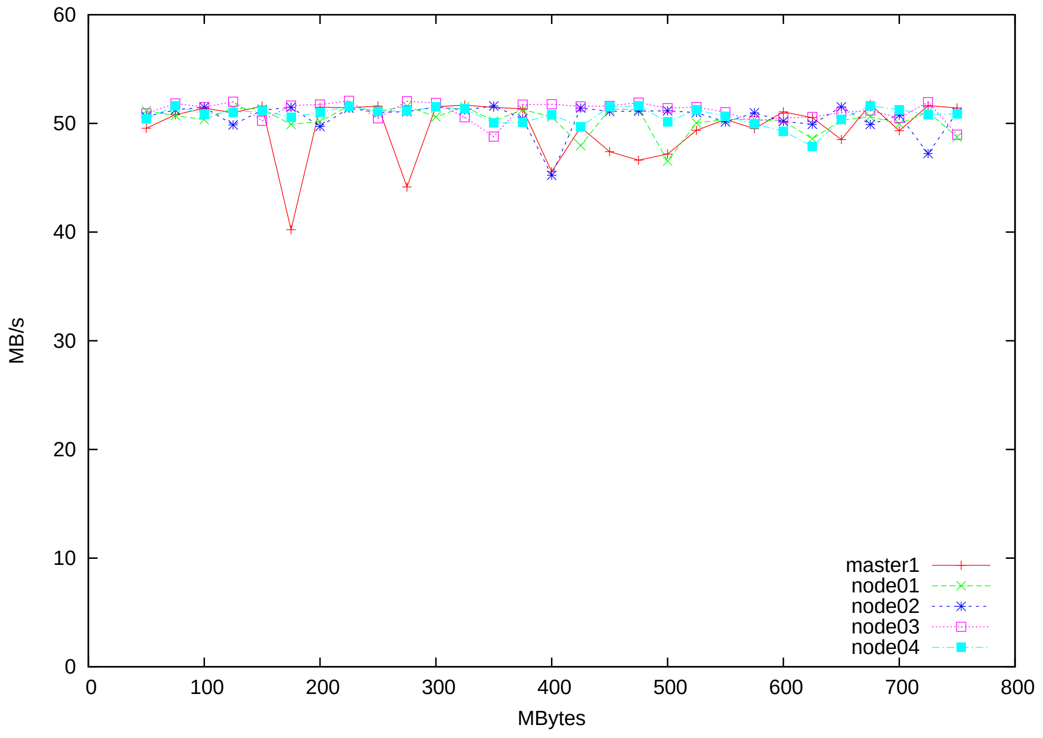

Figure A.3: *Reading* from own harddisk using UNIX `open()`, `write()` and `close()` commands

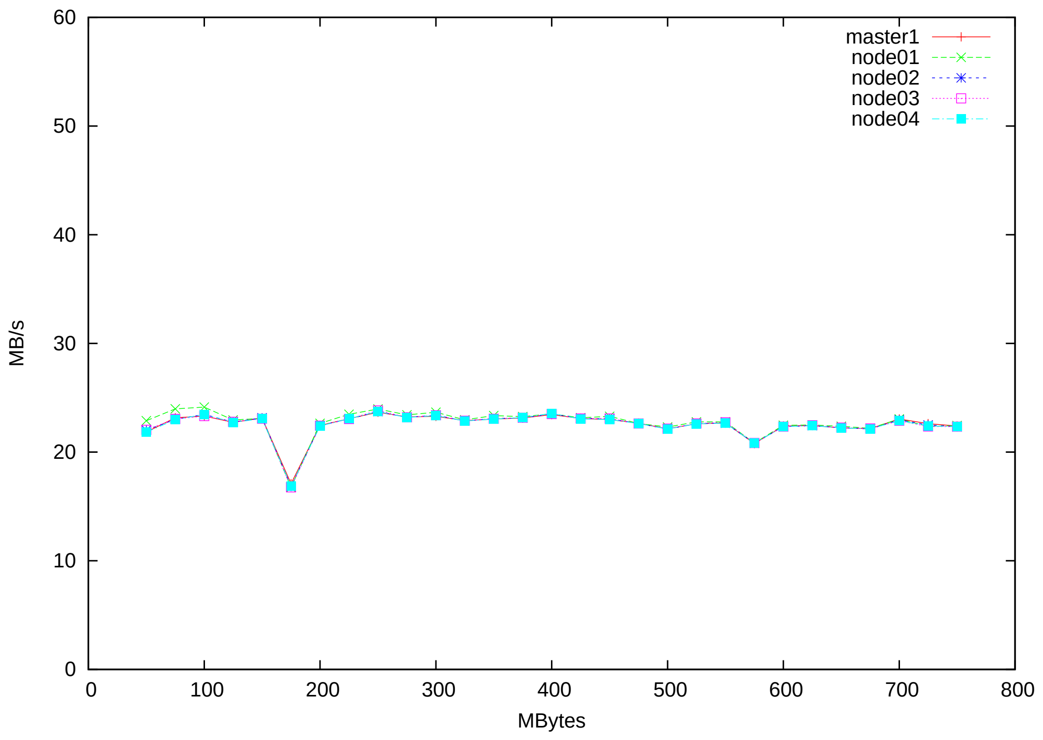

Figure A.4: *Writing* to PVFS2, simple stripe, 64KB strip size

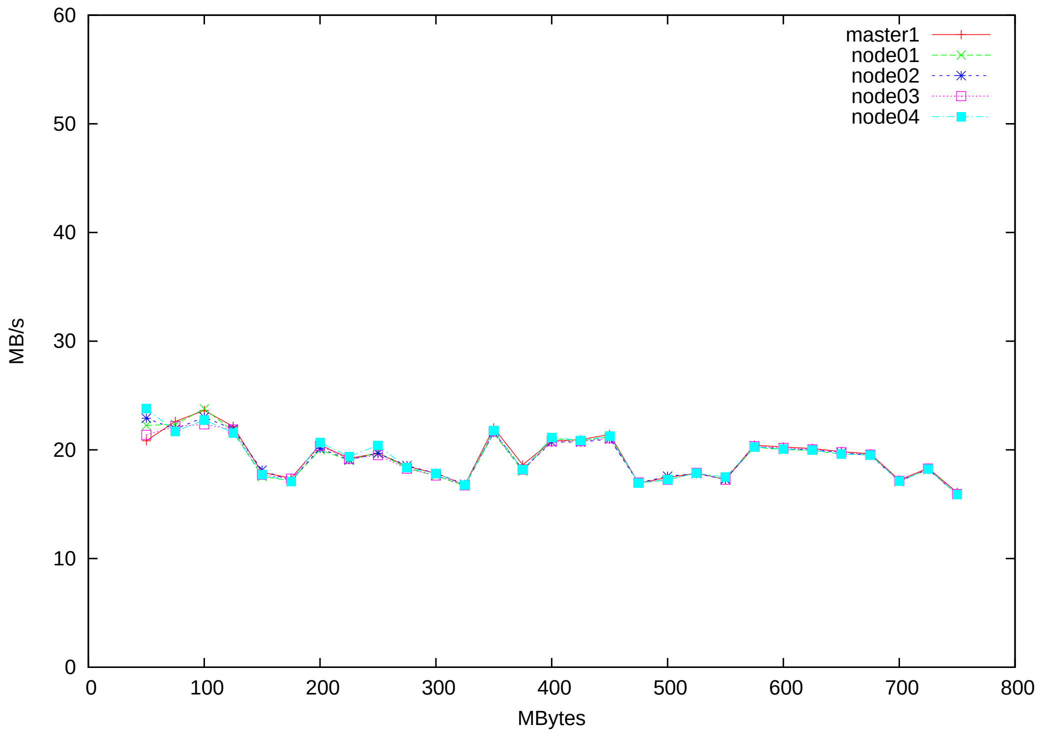

Figure A.5: *Reading* from PVFS2, simple stripe, 64KB strip size

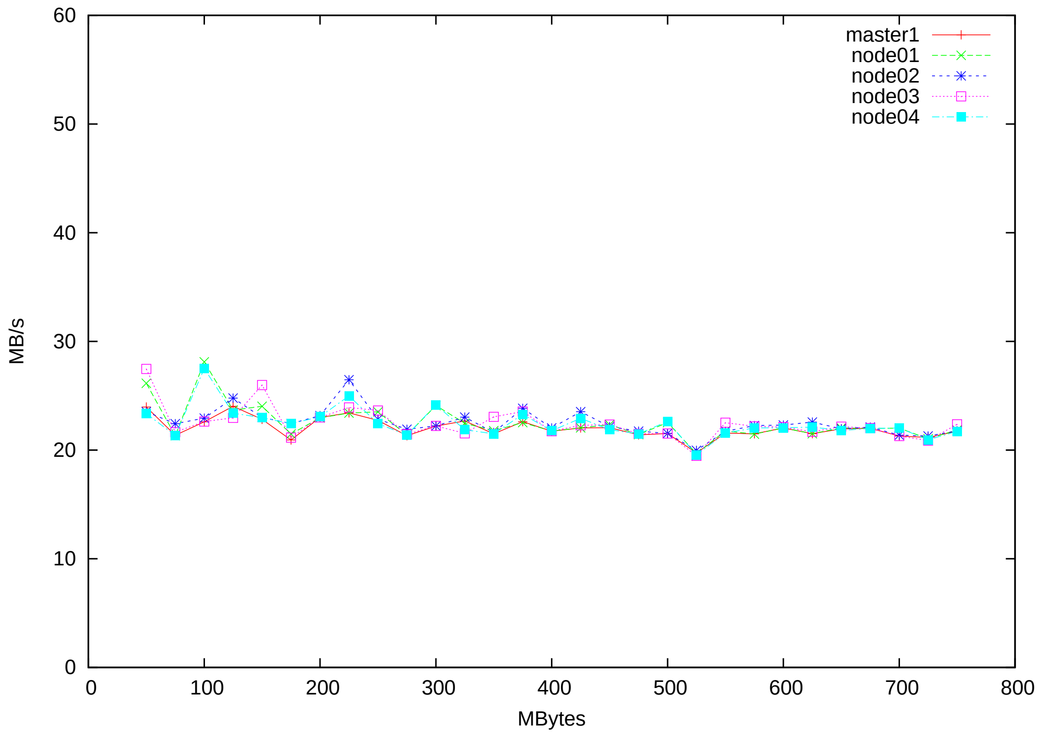

Figure A.6: *Writing* to PVFS2, simple stripe, 8MB strip size

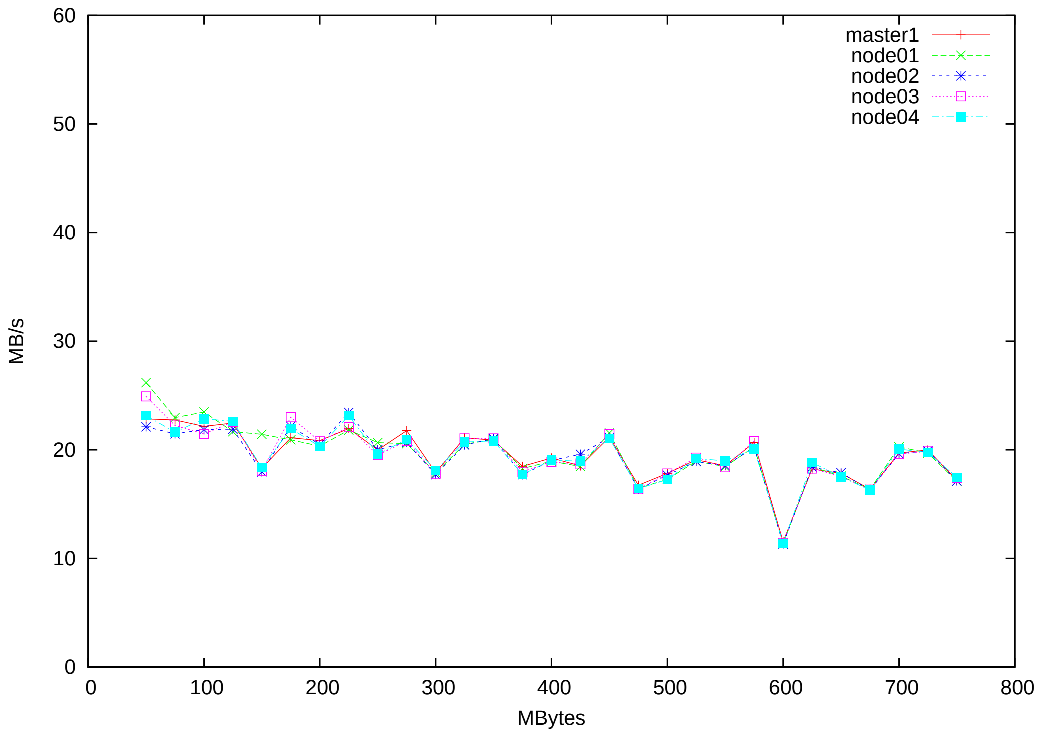

Figure A.7: *Reading* from PVFS2, simple stripe, 8MB strip size

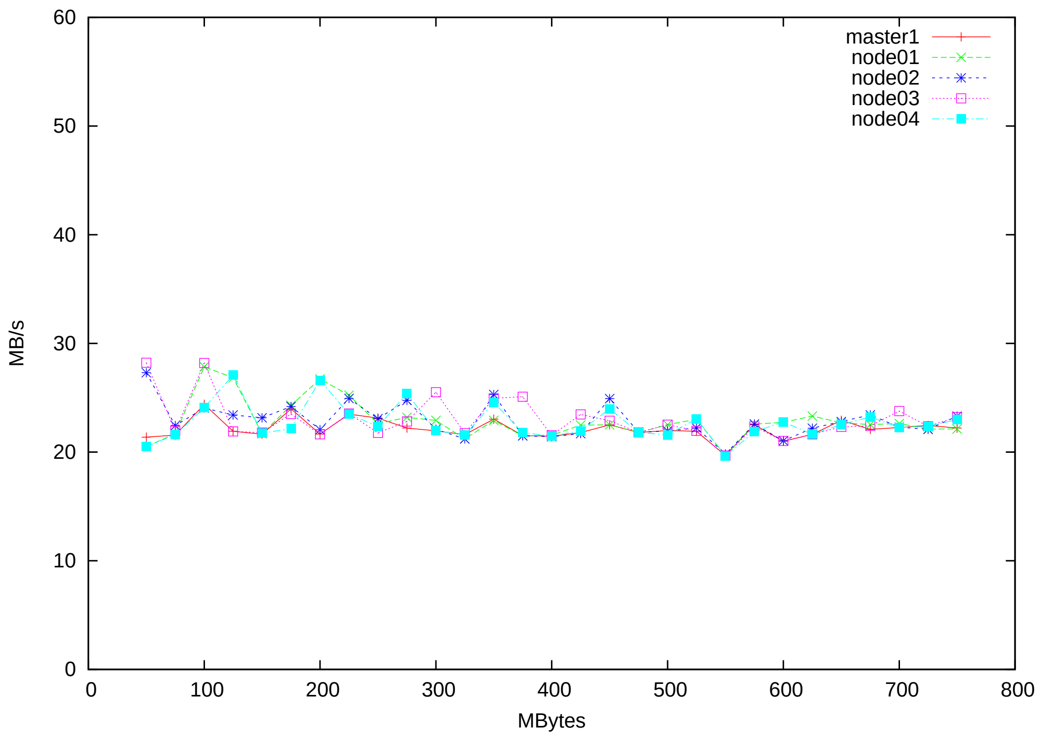

Figure A.8: *Writing* to PVFS2, simple stripe, 16MB strip size

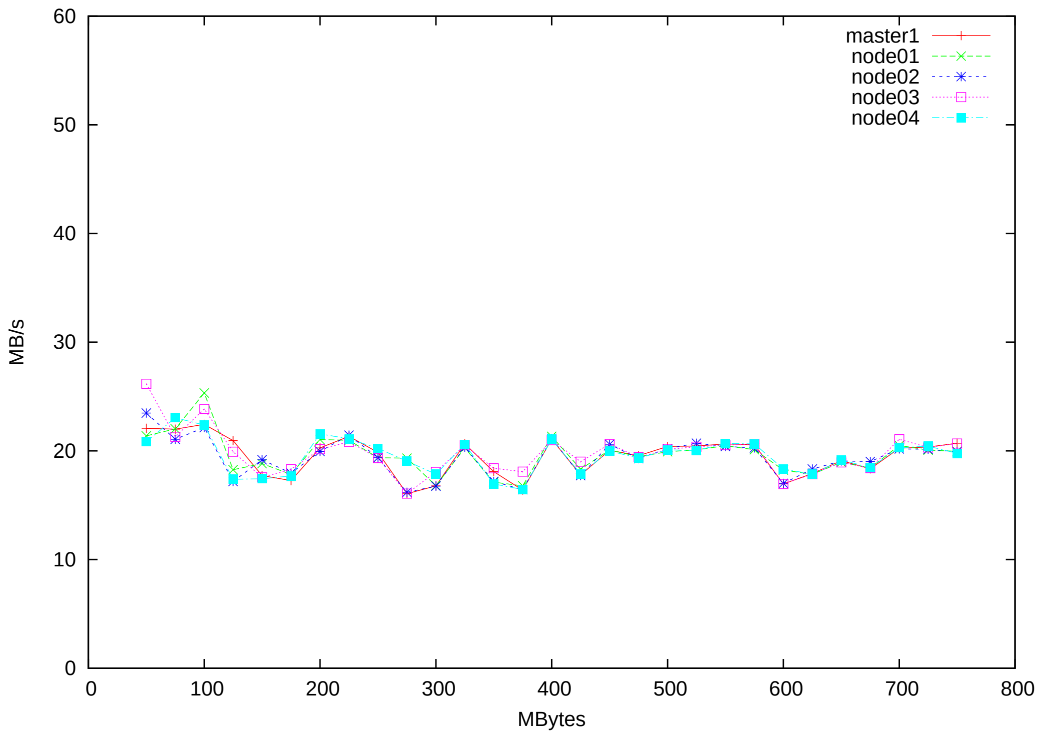

Figure A.9: *Reading* from PVFS2, simple stripe, 16MB strip size

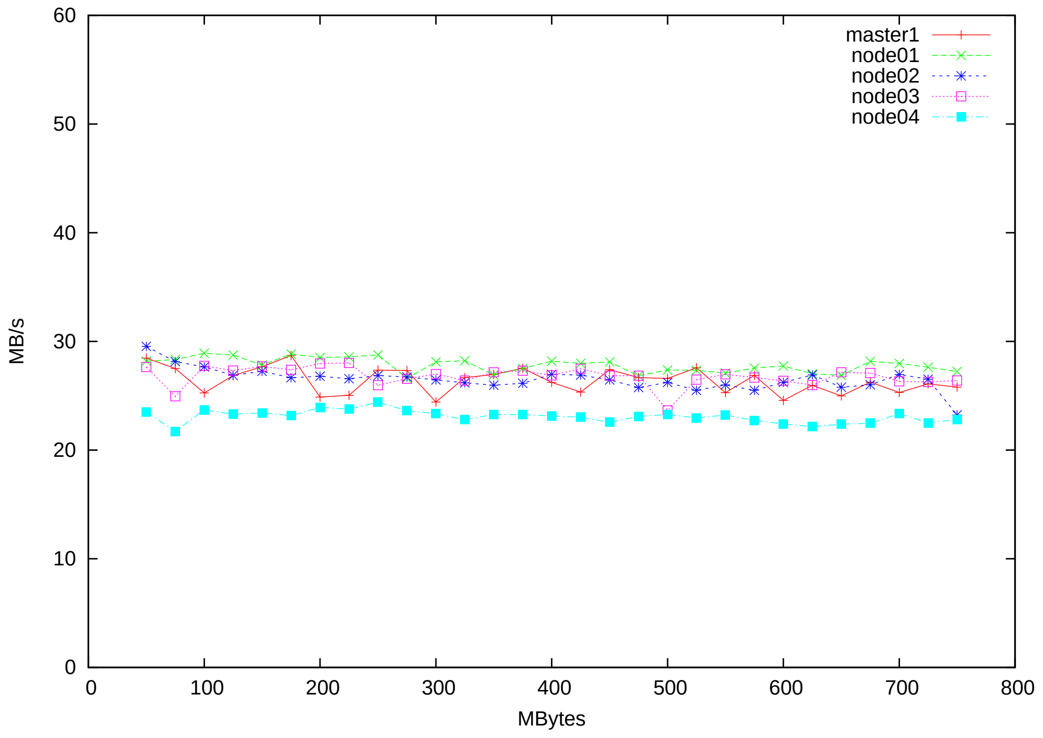

Figure A.10: *Writing* to PVFS2, simple stripe, fitted strip size

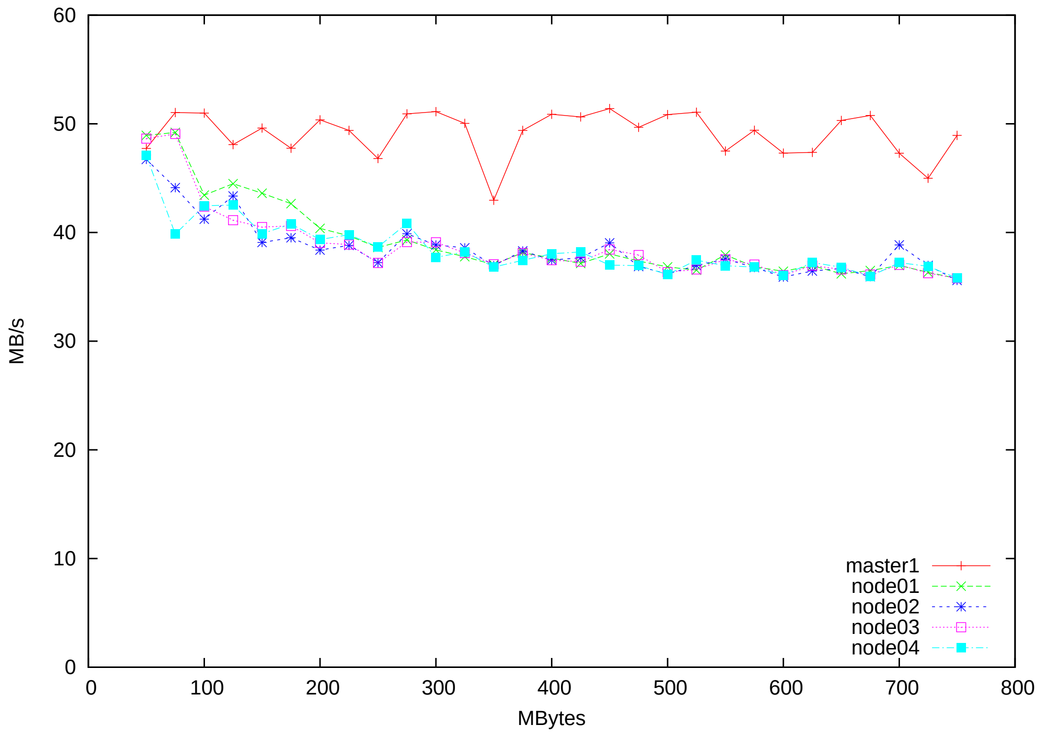

Figure A.11: *Reading* from PVFS2, simple stripe, fitted strip size

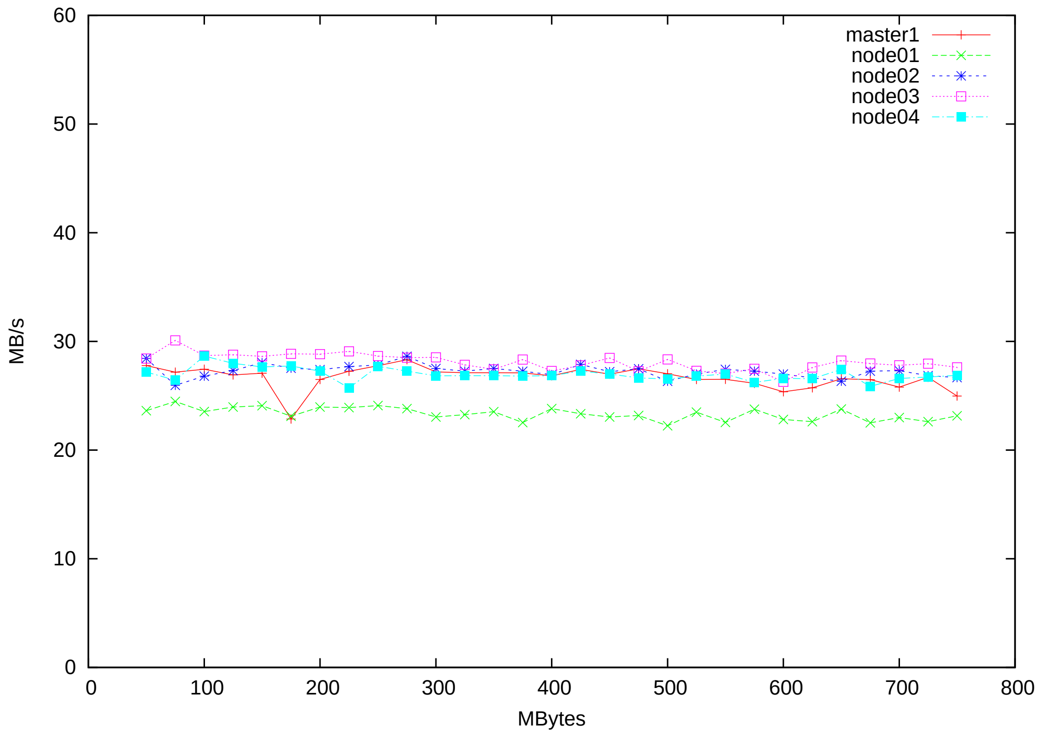

Figure A.12: *Writing* to PVFS2, varstrip distribution

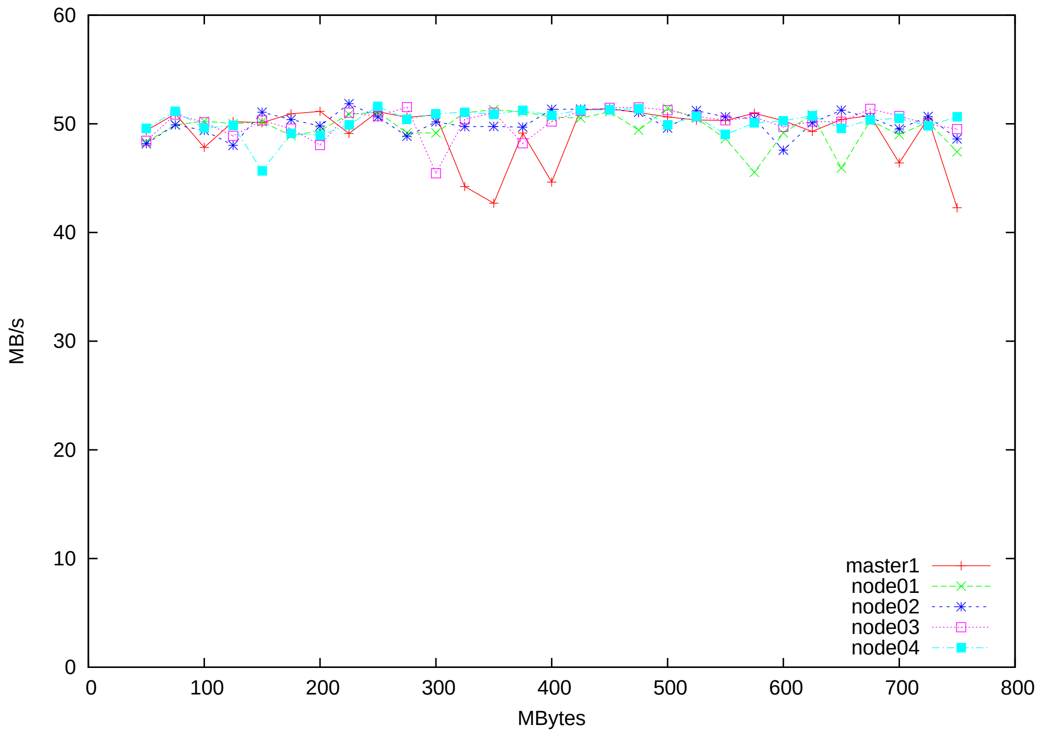

Figure A.13: *Reading* from PVFS2, varstrip distribution

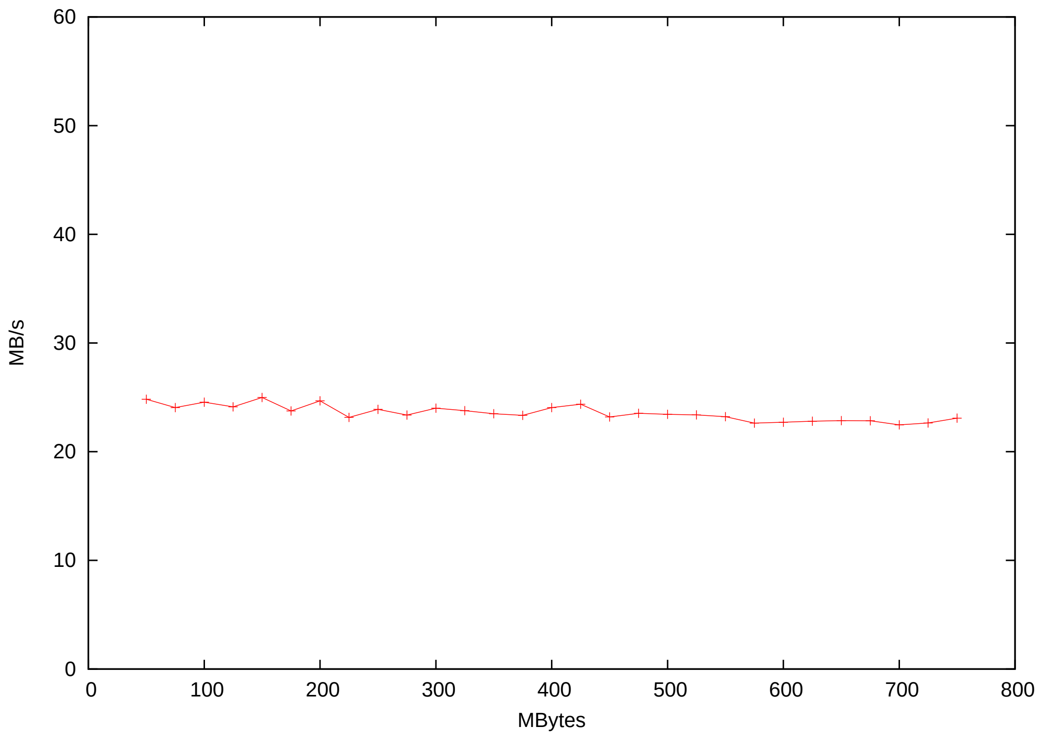

Figure A.14: `mpi-io-test` *writing* to PVFS2, simple stripe, 64KB

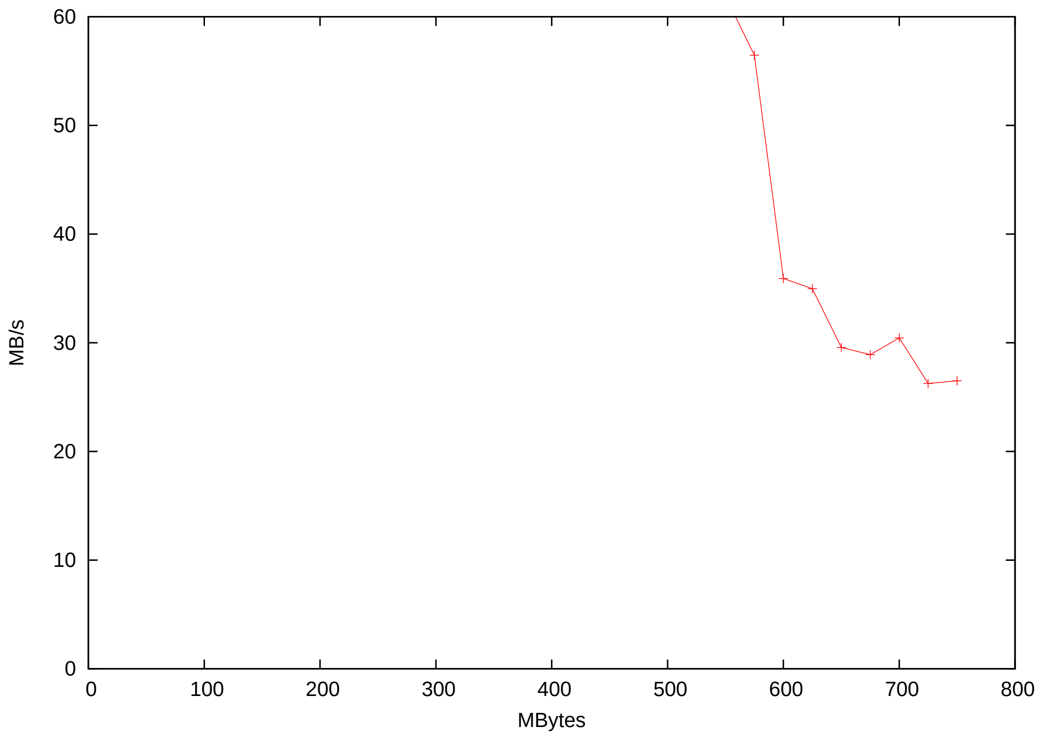

Figure A.15: `mpi-io-test` *reading* from PVFS2, simple stripe, 64KB

B Patches

B.1 Applying patches

To apply a patch go into the source directory of what you like to patch and call

```
patch -p1 < /path/file.patch
```

B.2 Available patches

- `mpich2-1.0.1_striping_unit+distribution_name+distribution_parameter.patch`
  This patch contains all changes I made to MPICH2 version 1.0.1. There are the `striping_unit` and `distribution_name` hints and the `distribution function parameter add-on`.
- `pvfs2-cvs-20050408_varstrip_distribution.patch`
  This patch contains the varstrip distribution.
- `pvfs2-cvs-20050408_print-datafile-io-server-assignment.patch`
  Using this patch you can print the assignment of the data files to the I/O-servers to `stderr` at file creation time.

Bibliography

[1] *MPI-2 website.* http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[2] *MPICH2 website.* http://www-unix.mcs.anl.gov/mpi/mpich2/.

[3] *PVFS2 website.* http://www.pvfs.org/pvfs2/.

[4] *ROMIO website.* http://www-unix.mcs.anl.gov/romio/.

[5] Sadleder, Philipp: *Bachelor-Thesis: Änderung der Datenverteilungsfunktion im parallelen Dateisystem PVFS2*, December 2004.