The GENESIS Distributed-Memory Benchmark Suite Release 3.0

Ian Glendinning

High Performance Computing Centre
University of Southampton

July 1994

1 Introduction

The motivation behind the development of the GENESIS benchmark suite has been to assemble benchmarks that can be used to evaluate the performance and the relative strengths and weaknesses of distributed-memory MIMD systems on a range of important scientific and engineering problems [1]. The benchmarks were written within ESPRIT Project P2702 – GENESIS by a number of project partners including Liverpool and Southampton Universities, PALLAS GmbH, the European Centre for Medium Range Weather Forecasting and the Universitat Politechnica de Catalunya. The suite was originally prepared for public release by Southampton Novel Architecture Research Centre, and is now being developed and distributed by the Southampton High Performance Computing Centre.

The GENESIS benchmarks are written as far as possible in standard FORTRAN 77 using `DOUBLE PRECISION` floating point arithmetic, but also `REAL` floating point arithmetic (LPM1 benchmark). Both message-passing and subset HPF versions exist for most of the codes in addition to the standard FORTRAN 77 versions. In Release 2.3 of the GENESIS suite, task creation and message-passing were expressed portably across target systems using the PARMACS macros (see section 2.1). This release features additional versions of the low-level and kernel benchmarks, which use PVM [2] instead of PARMACS. There are also four new low-level benchmarks to measure saturation bandwidth and memory and communication bottlenecks, COMMS3, POLY1, POLY2 and POLY3. The new codes have been written in coordination with the

PARKBENCH parallel benchmarking initiative [3], and most of them also form part of that benchmark suite. As well as feeding into PARKBENCH, this release of GENESIS draws from it, by adopting the SOLVER benchmark. SOLVER replaces the QCD2 benchmark, which was in the previous release. Descriptions of each benchmark and instructions on how to run them are given in individual `ReadMe` files.

The GENESIS distributed-memory benchmark suite is free software developed for academic purposes. The authors take no responsibility for maintenance and support, though the High Performance Computing Centre will continue to develop existing benchmarks and to incorporate new ones into the suite as appropriate. It is intended that a future release will include MPI versions of the codes [4].

We would very much appreciate receiving measurements, obtained with the GENESIS benchmarks, as well as any comments or queries concerning the suite. Correspondence should be addressed to the authors at:

HPC Centre, Computing Services
University of Southampton
Southampton
S017 4BJ
United Kingdom

Email: support@par.soton.ac.uk

2 Portable Message Passing and Task Creation

Portable message passing and task creation is achieved through the use of PARMACS or PVM. The low-level and kernel message-passing benchmarks (see section 3.1) are available in two versions, one using the PARMACS macros, and the other using the PVM library (with the exception of POLY3, which is only available in PVM). The compact applications are only available in PARMACS versions.

2.1 PARMACS

The PARMACS macros were developed at GMD and Argonne National Laboratories [5], to allow the same FORTRAN code to be run without modification on a number of parallel systems. A package of macros is used to express functions such as process creation, message passing and synchronisation. These macros are invoked from the user FORTRAN program and passed through the standard UNIX "m4" macro expander, which replaces the macros with FORTRAN text appropriate to the system on which the program is to be run. Some macros expand into a set of FOR-

TRAN statements whereas others expand into calls to library functions, which are supplied with the PARMACS package for each system. In the UNIX version of the suite expansion of the macros is handled by the `make`-files.

The basic programming model is based on message passing but this can easily be adapted to allow implementations on shared-memory machines as well as distributed-memory systems.

The macros are built on top of the native message passing software for each machine, but in most cases the communications overheads associated with the extra layer is small. The benchmarks that are provided to measure low-level communications parameters can easily be adapted to use native message passing for comparison with the values obtained using PARMACS. Alternatively the authors of the benchmark suite have comparisons for various systems.

This release of the benchmark suite uses version 5.1 of the PARMACS macros [6]. The most important feature of version 5.1 is the addition of the MSG_FORMAT macro which handles different data representations when sending messages between host and node. The use of this macro is essential on heterogeneous systems such as the nCUBE 2 or the iPSC/860 with remote hosting on a Sun where the data representation on the host is different from that on the nodes.

The definitions of the PARMACS macros are public domain, and implementations may be written by anyone. To date, there have been very few public domain implementations, but implementations for a wide range of parallel machines are commercially available from:

PALLAS GmbH
Hermülheimer Straße 10
D-50321 Brühl
Germany

Tel: +49 2232 18960
Email: karls@pallas-gmbh.de

2.2 PVM

PVM stands for Parallel Virtual Machine. It is a software package that allows a heterogeneous netowrk of parallel and serial computers to appear as a single concurrent computational resource. PVM consists of two parts: a daemon process that any user can install on a machine, and a user library that contains routines for initiating processes on other machines, for communicating between processes, and changing the configuration of machines. The current release of the benchmark suite uses PVM 3.x.

In contrast to PARMACS, the reference implementation of PVM is public domain. Proprietary versions, which may have been optimised for better performance, also exist. The current version can be obtained by sending a request to the netlib mail daemon. Mail (in the Subject: line or message body) to `netlib@ornl.gov` (or `research.att.com`):

```
send index from pvm3
```

will get you info for pvm3.x. Similar "send" messages will return parts of PVM. Alternatively, use anonymous ftp to `netlib2.cs.utk.edu` and look in the pvm3 directory. There is also a World Wide Web page from which PVM documentation and source code is available, with URL `http://hebe.nectar.cs.cmu.edu:8001/`.

3 Organisation and Contents of Benchmark Suite

3.1 Currently Available Benchmarks

The benchmarks are divided into three categories according to their level of complexity/realism as follows:

1. **Low-level benchmarks.** These measure basic machine parameters connected with communication, synchronisation and vectorisation.

   TICK1 Measures resolution of system clock
   Versions F-77, Subset HPF

   TICK2 Measures accuracy of system wall-clock time
   Versions F-77, Subset HPF

   RINF1 Measurement of vectorization parameters r_∞ and $n_{1/2}$
   Versions F-77, Subset HPF

   COMMS1 Pingpong benchmark
   Versions F-77, PARMACS, PVM

   COMMS2 Message exchange benchmark
   Versions F-77, PARMACS, PVM

   COMMS3 Saturation bandwidth benchmark
   Versions F-77, PARMACS, PVM

   SYNCH1 Barrier synchronisation cost
   Versions F-77, PARMACS, PVM

POLY1 In-cache memory bottleneck benchmark
Versions F-77

POLY2 Out-of-cache memory bottleneck benchmark
Versions F-77

POLY3 Communication bottleneck benchmark
Versions PVM

2. **Kernel benchmarks.** These are larger, more complex codes that represent the computationally intensive kernels common to a variety of different applications. Analysis of the benchmark performance is generally possible in terms of the basic machine parameters. They provide valuable insight into the likely behaviour of full applications.

   TRANS1 Matrix transpose using naive algorithm
   Versions F-77, PARMACS, Subset HPF, PVM

   IO1 Measurement of parallel input/output transfer rate
   Versions F-77, PARMACS, Subset HPF, PVM

   FFT1 One-dimensional FFT
   Versions F-77, PARMACS, Subset HPF, PVM

   SOLVER Conjugate gradient kernel from QCD
   Versions F-77, PARMACS, PVM

   PDE1 Three-dimensional Poisson solver using red-black relaxation
   Versions F-77, PARMACS, Subset HPF, PVM

   PDE2 Two-dimensional multi-grid Poisson solver
   Versions F-77, PARMACS, Subset HPF, PVM

3. **Compact applications.** These are typical of medium-sized research codes where effects due to non-parallelizable sections of code, and memory or I/O bottlenecks may become important.

   QCD1 Quantum chromodynamics simulation using a Monte Carlo method
   Versions F-77, PARMACS

   GR1 General relativity application
   Versions F-77, PARMACS

   LPM1 Electronic device simulation
   Versions F-77, PARMACS

   MD1 Molecular dynamics code
   Versions F-77, PARMACS, Subset HPF

3.2 UNIX Directory Structure

The GENESIS benchmark suite is available as a single UNIX directory structure. Apart from the benchmarks themselves, the top level directory includes a plain ASCII file `ReadMe.first`, which serves as a first introduction, and explains that this `ReadMe.ps` file, and its LaTeX source are located in the `ReadMe` subdirectory. There are also two auxiliary directories, `Incl` and `Lib`. The `Incl` directory contains various makefiles, whilst `Lib` contains FORTRAN source code for various parallel platforms, for the host and node libraries that encapsulate all calls to machine-specific routines. Every benchmark directory contains a `ReadMe` file, which contains a description and set of operating instructions. Depending on which versions of the benchmark exist, some or all of the following subdirectories will also be present:

d77 - distributed memory PARMACS code. The files use the following naming convention. If <name> is the name of the benchmark (e.g. <name>=comms1) then:

- **<name>.f** - the host program that loads the node code.
- **node.f** - the node program that is spawned and executed on all processors taking part in the test. One of the nodes usually acts as a master controlling several slaves, and outputs the results.
- **header.f** - subroutine to print a header at the top of the result file.
- **result** - the result file.
- **<name>** - the executable program to run the benchmark.

pvm3 - distributed memory PVM 3.x code. There is a similar set of files to those found in the `d77` directory, except that the node code is called <name>n.f, the header subroutine is <name>h.f, and the result file is <name>.res. The result file may also be written to your home directory when using PVM.

s77 - standard FORTRAN 77 code for single-node execution. The file naming convention for the single processor sequential code will be the same as that for `d77` for old code or `pvm3` for new code, except that there is no node program.

hpf - subset HPF code. The file naming convention is the same as for the `d77` directory. As for `s77`, there is no node code.

StdRes - a set of sample result files from the benchmark. The file names are typically chosen to convey information about which version of the benchmark was run, and on how many processors. For example `d4` means results from a run of the PARMACS version on four processors, `p8` means a run of the PVM version on eight processors, and `s1` is from a run of the single processor standard Fortran version. In each case the initial letter is the same as that of `d77`, `pvm3` or `s77`.

4 Installation Procedure

In order to run the distributed version of the benchmarks, it is necessary that one or both of PARMACS 5.1 and PVM 3.x is installed on the system to be benchmarked. Before running individual benchmarks, the user also has to set a few parameters describing the installation of PARMACS and/or PVM on the system in question. There is also a set of machine-specific routines which are called by all the benchmarks to supply information such as current date, elapsed time, system's details, etc. Source code for these routines is supplied with this release for a number of systems, and installation onto such a system is relatively straightforward. If the routines are not supplied for the system you are installing onto, then a little more work is required. The following sections explain the installation procedures specific to PARMACS and PVM, and how to create the machine-specific routines if they do not already exist.

4.1 PARMACS

The first step is to edit the file `Incl/make.def`, which contains information on the installation of PARMACS on the system under evaluation, and is referenced by the individual makefiles for each benchmark. The user has to specify three system-specific macro definitions - PARALLEL_PLATFORM, FRONT_END, PARALLEL_RESOURCE and the path to the PARMACS base-directory, USR_PALLAS. First find out this path for your PARMACS installation. It is usually `/usr/pallas`, but it can be different. Information on the use and installation of the PARMACS macros is contained in [7]. Next look in the file

`$(USR_PALLAS)/include/generic/makefile`

where the "$" and parentheses surrounding the macro USR_PALLAS mean that its value, in this case the path, should be substituted. The header comments in this file should contain a list of valid platform names for the PARALLEL_PLATFORM macro. It specifies the general programming model, and allowed values include `ipsc`, `cstools`, `suprenum`, `meiko` etc. Corresponding lists of valid options for FRONT_END and PARALLEL_RESOURCE are also to be found in the header. FRONT_END specifies the system on which the PARMACS host processor runs, and can be `sun4OS4`, `srm`, etc. The macro PARALLEL_RESOURCE defines the hardware architecture on which the node-processes are to be executed, e.g. `860p`, `sun4OS4`, `x8`, etc.

The message length or the buffer length in bytes has to be explicitly specified in the SEND/RECV PARMACS macros. Hence, the user program should incorporate a simple mechanism to adjust the message length and all relevant constants to the length of currently used data types. For example, a message to send an integer scalar is usually 4 bytes long, but it must be 8 bytes long on the Cray. In order to adjust the data lengths for the target machine it is also necessary to edit the file `Incl/datlen.inc`, which contains the corresponding parameters.

Machine-specific routines for the system specified by PARALLEL_PLATFORM are stored in the directory `Lib/SUITELIB/SOURCE.$(PARALLEL_PLATFORM)`. If no such directory exists in the release, you must create it and supply the machine-specific routines, plus some makefiles, as explained in section 4.3. In any case you should edit the file `setdtl.f` to include your own name as the benchmarker, and to check that compiler and other details are correct. You may then proceed to compile the routines into libraries to be linked with the host and node programs by changing directory to Lib/SUITELIB and typing `make`.

4.2 PVM

As in the case of PARMACS, the first step is to edit the file `Incl/make.def`. There is no need to define USR_PALLAS if you are not also installing PARMACS, but you will need to define PARALLEL_PLATFORM, FRONT_END and PARALLEL_RESOURCE, as explained in section 4.1. For the public domain release of PVM, you should set PARALLEL_PLATFORM to `pvm`. You must also define PVM_DIR, PVM_ARCH, ARCHLIB and LDFLAGS. PVM_DIR will normally be `$(HOME)/pvm3`, and PVM_ARCH can be determined for most systems by running the `pvmgetarch` script. ARCHLIB and LDFLAGS are there to allow optional flags to be passed to the linker, and their definition can be left blank for most systems.

Next you should check that the include file `Incl/fpvm3.h` is the same as the one in the PVM include directory, `$(PVM_DIR)/include/fpvm3.h`. If it is not, you should overwrite `Incl/fpvm3.h` with a copy of the other file.

Analagous to the `Incl/datlen.inc` file used by the PARMACS versions of the benchmarks is one called `Incl/dattyp.inc`, which defines the PVM type codes to be used for integer, real and double precision data in messages. You should check that the definitions are correct for the system you are benchmarking.

The PVM versions of the benchmarks use the machine-specific routines stored in `Lib/SUITELIB/SOURCE.$(PARALLEL_PLATFORM)`, which are the same ones used by the PARMACS versions. If no such directory exists in the release, you must create it and supply the machine-specific routines, plus some makefiles, as explained in the next section. In any case you should edit the file `setdtl.f` to include your own name as the benchmarker, and to check that compiler and other details are correct. You may then proceed to compile the routines into libraries to be linked with the host and node programs by changing directory to Lib/SUITELIB and typing `make`.

4.3 If Machine-Specific Routines do not Exist

The machine specific routines which must be supplied are `setdtl`, `setdat` and `timer`. They can be modelled on the code provided with this release for other systems.

setdtl writes installation specific information, such as compiler and operating system version, to the results file.

setdat returns current date and time.

timer returns the elapsed time in seconds since its first call.

The file `Lib/SUITELIB/SOURCE.$(PARALLEL_PLATFORM)/makefile`, the makefile for the machine-specific routines, must be supplied in addition to the routines themselves. `Lib/SUITELIB/SOURCE.suprenum/makefile` can be used as a template. It will have to be modified if any auxilliary routines are needed in addition to `timer.f`, `setdat.f` and `setdtl.f` for the platform in question. Examples can be found in the source directories that come with this release for other platforms.

Finally, two more files are required. `Incl/makefile.arch` should be copied from `$(USR_PALLAS)/include/generic/makefile.arch` if you have a PARMACS installation, or else you should use the version which comes with the benchmarks. It may be necessary to add a rule with the target `$(PARALLEL_PLATFORM)`. The other file, called `Incl/make.$(PARALLEL_PLATFORM)`, has to contain a rule to allow the sequential benchmarks to be compiled by typing `make slave`. This is only necessary for the PARMACS versions of the benchmarks. To find the correct rule for your platform, look in `$(USR_PALLAS)/include/$(PARALLEL_PLATFORM)/make.parmacs` for a line of the form:

```
<PATH>$(SLAVE): <PATH>$(SLAVE).o <other dependencies>
```

The required rule is then:

```
slave:  <PATH>$(SLAVE)
```

For example, for the CS-2, `<PATH>` is null, and the correct content of the file is:

```
slave:  $(SLAVE)
```

as it is for the cray and ipsc. For cstools and pvm `<PATH>=$(PARALLEL_RESOURCE)`, and the required rule is:

```
slave:  $(PARALLEL_RESOURCE)/$(SLAVE)
```

5 Compiling and Running the Benchmarks

Unix makefiles are supplied with each benchmark to handle compilation and linking, plus expansion of the PARMACS macros where appropriate. Compiling the distributed-memory benchmarks is as simple as typing `make` in the `d77` or `pvm3` directories. This is also true of the single-node versions of the low-level benchmarks, in their `s77` directories, however to compile the single node versions of the kernels and compact applications you must type `make slave`. The command `make clean` removes all executable files and directories. On some systems such as the iPSC an adequate make mechanism is not available and GNU make must be used instead. GNU make will normally be supplied with the PARMACS macros if its use is necessary on the system in question. The currently available `make`-files support the following platforms: Cray Y-MP, Intel iPSC machines, Meiko CS-2, SUPRENUM, and systems using Meiko's CSTools or PVM. The user will have to extend the existing makefiles or to write new ones in order to run the benchmarks on any other platform.

In some of the benchmarks the problem size or number of processors is changed by editing PARAMETER statements in an include file. In this case the routines referencing the include file will need to be recompiled after every change. The relevant files are identified automatically in the makefile and so using `make` after any such change will recompile only the appropriate files. The input/output unit numbers are also assigned by PARAMETER statements, allowing these to be adjusted easily if necessary.

On some systems it may be necessary to use system-specific commands to reserve the appropriate number of processors before running the benchmark, eg. `getcube` on the iPSC. It may also be necessary to set certain environmental variables such as on the Meiko Computing Surface where the clock-tick time and the type of processor configuration can be set by such variables. These details are best determined from somebody with knowledge of the system in question.

To run a benchmark simply type the benchmark name after compiling and linking the code. On some platforms the single node executables have to be loaded by a special command (`load` on the iPSC, `mrun` on Meiko, etc.). The suggested procedure for running the benchmark suite is as follows:

1. Run synthetic code fragments

   The TICK1 benchmark should be run first to check the resolution of the system clock. The most accurate clock available should normally be used. To compensate for insufficient resolution, most of the synthetic code fragments and applications kernels are timed over a repeat loop. The number of repeats being specified by the user. If the benchmark timing is comparable with the clock tick-time the number of repeats should be increased. Normally the measured time in the benchmark should be greater than 100 clock ticks so as to give at least 1% accuracy.

Most computers measure time by counting the number of clock ticks, then use a multiplier to convert this number to seconds. The TICK2 benchmark tests the validity of this multiplier, by measuring the same time interval with both the computer TIMER subroutine, and an external wall-clock, which can be your own wrist-watch. The time interval is started and ended by a keyboard input. We suggest you time an interval of 2 minutes.

The remaining synthetic code fragments can then be run to measure performance parameters related to vectorization, communication and synchronisation [8]. These can be useful in interpreting the results obtained from the more complex benchmarks.

2. Run sequential FORTRAN77 versions of kernels

   Measure single node performance over a range of problem sizes upto the maximum possible on a single node. Also investigate effect of compiler options and any vectorization facilities on single node performance.

   Note: Non-standard precision compiler flags should NOT be used.

3. Run distributed FORTRAN77 benchmarks

   Use optimum compiler options determined from single node measurements. To obtain a clear idea of how the benchmark performance varies, measurements should be made over a range of problem sizes for a different numbers of nodes.

A result file is produced, which is intended to be kept as a permanent record of the benchmark run. The name of the file is `result` for the PARMACS versions of the benchmarks, and `<name>.res` for the PVM versions, where <name> is the name of the benchmark in question. The PARMACS codes write the file into the directory containing the executable, but with PVM this seems to be system-dependent, and the file may be written to your home directory instead. The result file is overwritten by subsequent runs of the program and so should be copied to another suitably named file before the next run. As well as the result file just described, some of the benchmarks write informative messages to standard output, so that the progress of the run can be interactively monitored. The COMMS1 and COMMS2 benchmarks also produce files called `plot.dat`, which contain the raw timing data in the form required by `gnuplot` to plot a graph of time per message versus message length. Samples of result files produced by the sequential and distributed versions of all benchmarks are included in the respective `StdRes` directories.

6 Methodology of Measurement and Interpretation

Definitive benchmark measurements should be made on a quiet system with only one user. If this is not possible the fact should be carefully noted and more care than normal should be taken to ensure that the measured results are reproducible. I/O operations or processes loading from other users inevitably affects the system performance. While performance degradation resulting from this effect is of some interest it is important first to be able to obtain reproducible baseline measurements in which such effects are absent.

6.1 Time Measurement

The fundamental measurement made in any benchmark is the time to complete some specified task. All other performance figures (e.g. Mflop/s) are derived from this basic timing measurement. It is a principle of the GENESIS benchmarks [9] that these timing measurements must either be published as tables of numbers with at least three significant decimal figures, or be derivable to the same accuracy from performance tables, using a given formula.

The benchmark time, $T(N;p)$, will be a function of the number of processors, p, and the problem size, here represented by the vector variable, N, which stands for a set of parameters characterising the size of the problem, e.g. the number of mesh points in each dimension, and the number of particles in a particle-mesh simulation. Benchmark problems of different sizes can be created by multiplying all the size parameters by suitable powers of a single scale factor, thereby increasing the spatial and particle resolution in a sensible way.

6.2 Floating-Point Operation Count

In order to express benchmark performance in millions of floating-point operations per second, it is necessary to assess the number of floating-point operations, $F(N)$, needed to solve the benchmark problem. For simple problems (e.g. matrix multiply) it is sufficient to use a theoretical value for the floating-point operation count (in this case $2n^3$ flop, for nxn matrices) obtained by inspection of the code or theoretical consideration of the arithmetic in the algorithm. For more complex problems containing data-dependent conditional statements, an empirical method may have to be used. The sequential version of the benchmark code defines the problem and the algorithm to be used to solve it. Counters can be inserted into this code or a hardware monitor used to count the number of floating-point operations. In either case a decision has to be made regarding the number of flop that are to be credited for different types of floating-point operations, and we see no good reason to deviate from those chosen by McMahon when the Mflop/s measure was originally defined. These are:

add, subtract, multiply	1 flop
divide, square-root	4 flop
exponential, sine etc.	8 flop
IF(X .REL. Y)	1 flop

We distinguish two types of operation count. The first is the nominal benchmark floating-point operation count, $F_B(N)$, which is found in the above way from the defining FORTRAN 77 sequential code. The other is the actual number of floating-point operations performed by the hardware when executing the distributed multi-node version, $F_H(N)$, which may be greater than the nominal benchmark count, due to the distributed version performing redundant arithmetic operations.

6.3 Performance Metrics

Given the time of execution $T(N;p)$ and the flop-count $F(N)$ several different performance measures can be defined. Each metric has its own uses, and gives different information about the computer and algorithm used in the benchmark. It is important therefore to distinguish the metrics with different names, symbols and units, and to understand clearly the difference between them [10]. The principal performance metrics are:

6.3.1 Temporal Performance

If we are interested in comparing the performance of different algorithms for the solution of the same problem, then the correct performance metric to use is the *temporal performance*, R_T, which is defined as the inverse of the execution time

$$R_T(N;p) = T^{-1}(N;p) \tag{1}$$

The units of temporal performance are, in general, solutions per second (sol/s), or some more appropriate absolute unit such as timesteps per second (tstep/s). With this metric we can be sure that the algorithm with the highest performance executes in the least time, and is therefore the best algorithm. We note that the number of flop does not appear in this definition, because the objective of algorithm design is not to perform the most arithmetic per second, but rather it is to solve a given problem in the least time, regardless of the amount of arithmetic involved.

6.3.2 Benchmark Performance

In order to compare the performance of one benchmark with another, account must be taken of the different amounts of work (measured in flop) that the different problems require for their solution. Using the flop-count for the benchmark, defined in the last paragraph, we can define the *benchmark performance* as

$$R_B(N;p) = F_B(N)/T(N;p) \tag{2}$$

The units of benchmark performance are Mflop/s (benchmark name), where we include the name of the benchmark in parentheses to emphasise that the performance may depend strongly on the problem being solved, and also to emphasise, that benchmark Mflop/s may be nominal (rather than actual) values.

6.3.3 Simulation Performance

A special case of temporal performance occurs for simulation programs in which the benchmark problem is defined as the simulation of a certain period of physical time, rather than a certain number of timesteps. In this case we speak of the *simulation performance* and use units such as *simulated days per day* (written sim-d/d or 'd'/d) in weather forecasting where the apostrophe is used to indicate 'simulated', or *simulated pico-seconds per second* (written sim-ps/s or 'ps'/s) in electronic device simulation. It is important to use simulation performance rather than timestep/s if one is comparing different simulation algorithms which may require different sizes of timestep for the same accuracy (for example an implicit scheme that can use a large timestep, compared with an explicit scheme that requires a much smaller time step). For such schemes the simulation performance falls off dramatically as the problem size is increased by introducing more meshpoints in order to refine the spatial resolution.

6.3.4 Presentation

In the GENESIS benchmarks, the performance is presented both in tabular form and as graphs of $R(N;p)$ for constant N plotted against p on log/log axes. The use of logarithmic axes allows the inclusion of a wide range of processor number and performance on a single graph. Within one benchmark, it also permits the vertical axis to be interpreted as either time or performance by reading the appropriate scale. In this respect we plot the performance the *right-way-up* with the highest performance at the top not the bottom.

6.4 Optimization Issues

When assessing the strengths and weaknesses of a design we are not only interested in the optimum performance that can be achieved for a particular application, but also in the performance that can be obtained from the initial port of a code, the subsequent optimization that can be performed, and how difficult it is to implement this optimization. The last assessment must invariably be subjective and will depend on the skill of the benchmarker, as will the degree of optimization obtained.

The position of the authors of the suite on optimization is thus similar to that of the PERFECT club: optimization of the benchmark codes for a given system is permitted and even encouraged but the measurements for the standard unoptimized benchmark should always be made and published with the optimized measurements. The PERFECT club idea of quantifying the effort involved in the optimization is viewed as desirable but impractical.

Acknowledgments

First and foremost, I would like to thank Vladimir Getov, Tony Hey, Roger Hockney and Ivan Wolton, from whom I inherited release 2.2 of this benchmark suite, and have received significant input into this release. Thanks are also due to Stephen Booth, who was responsible for the SOLVER benchmark, and Sami Saarinen for many useful comments on the codes. Amongst many other people who have helped, I would like to mention Mark Hill, Mark Debbage, Matthew Peters, Klaus-Dieter Oertel and Mark Baker for their useful and detailed comments on the benchmark codes and for porting the suite to the transputer based systems running the Virtual Channel Router, the cluster of IBM RS/6000s running PVM and the Meiko Computing Surface based on transputers. Werner Krotz-Vogel has helped significantly to improve and simplify the `make`-interface to the PARMACS installation. We would also wish to thank Craig Chase who put the benchmarks on the ftp-server of Cornell University and Robert Allan for making the suite available from the info-server of Daresbury Laboratory.

References

[1] A.J.G. Hey. The GENESIS distributed memory benchmarks. *Parallel Computing*, **17**(10 - 11):1275–1283, 1991.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1993.

[3] R.W. Hockney and M. Berry (Editors). Public International Benchmarks for Parallel Computers. *Scientific Programming*, To appear, 1994. PARKBENCH Committee: Report-1, February 7, 1994.

[4] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Computer Science Dept. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, April 1994. (To appear in the International Journal of Supercomputer Applications, Volume 8, Number 3/4, 1994).

[5] L. Bomans, D. Roose, and R. Hempel. The Argone/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, **15**(1-3):119–132, 1990.

[6] GMD mbH. *The Argone/GMD Macros in FORTRAN for Portable Parallel Programming using the Message Passing Programming Model*, November 1991. Version 5.1.

[7] PALLAS GmbH. *PARMACS - Geting Started*, October 1991.

[8] R.W. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, **17**(10 - 11):1111–1130, 1991.

[9] C. Addison, J. Allwright, N. Binsted, N. Bishop, B. Carpenter, P. Dalloz, D. Gee, V. Getov, A.J.G. Hey, R. Hockney, M. Lemke, J. Merlin, M. Pinches, C. Scott, and I. Wolton. The GENESIS distributed-memory benchmarks. Part I: Methodology and general relativity benchmark with results for the SUPRENUM computer. *Concurrency: Practice & Experience*, **5**(1):1–22, 1993.

[10] R.W. Hockney. A framework for benchmark performance analysis. *Supercomputer 48*, **IX**(2):9–22, March 1992.