The CSC Benchmarks (Spring 1993)

by Sami Saarinen

Center for Scientific Computing
P.O. Box 405
Tietotie 6
FIN-02101 Espoo
Finland

e-mail: sbs@csc.fi

April 9, 1993

Table of contents

1. Introduction ....

2. Contents of the benchmarks ....

2.1 A subset of the SPEC-benchmarks ....
2.1.1 Reporting the SPEC-results ....
2.1.2 Accessing the SPEC-benchmarks ....
2.2 The CSC Benchmark Suite, version 2 ....
2.2.1 Benchmark program characteristics ....
2.2.2 Run rules ....
2.2.3 Reporting the CSC Benchmark Suite results ....
2.3 Parallel benchmarks ....
2.4 Miscellaneous benchmarks ....

3. Performance metrics ....
3.1 Wall clock time ....
3.1.2 Sum of running times ....
3.2 Ratio of wall clock times ....
3.3 Nominal flop count ....
3.4 Application performance (nominal Mflop/s) ....
3.5 Benchmark performance ....
3.6 Benchmark averages ....
3.6.1 Geometric mean ....
3.6.2 Arithmetic mean ....
3.6.3 Harmonic mean ....
3.7 Benchmark instability ....
3.8 Parallel program measures ....
3.8.1 Parallel speedup ....
3.8.2 Parallel efficiency ....
3.9 Measuring communication overhead ....
3.9.1 Computation to communication ratio ....
3.9.2 Modeling communication time ....

4. Execution instructions ....
4.1 Reading the benchmark tape ....
4.2 Directories, files and utility functions ....
4.2.1 CSCSUITE_2 ....
4.2.2 PARALLEL ....
4.2.3 MISC ....
4.2.4 Timer functions ....
4.2.5 Controlling the number of processors ....
4.3 Compiling and linking the applications ....
4.3.1 The CSC Benchmark Suite ....
4.3.2 Parallel benchmarks ....
4.3.3 Miscellaneous benchmarks ....
4.4 Running the benchmarks ....
4.4.1 The CSC Benchmark Suite ....
4.4.2 Parallel benchmarks ....
4.4.3 Miscellaneous benchmarks ....

5. Contact information ....

Appendices:
A: Configuration of the CSC reference computer system ....
B: A short description of the SPEC-benchmarks ....
C: Guidelines to report system configuration ....
D: Description of the CSC Benchmark Suite, version 2 ....
E: PARMACS macro calls ....

1. Introduction

This document describes the CSC Benchmarks to be run by selected vendors in Spring 1993 during computer evaluation project.

First, I will go through the contents of the benchmarks. After that I will talk about some benchmark metrics that we are interested in. Finally, the detailed benchmark instructions are followed.

This benchmark has been very carefully planned and should be easy port to any UNIX-based systems having proper Fortran-77 and C-compilers. I have included things like 'run script' and 'makefile' -generators to facilitate the installation, executable building and running the benchmarks. Also, detailed information about the programs' characteristics is included. This contains description of each individual program, the reference results on our reference computer system, the nominal megaflop-count collected by help of Cray Hardware Performance Monitor (hpm) and some additional information that would categorize the benchmark programs.

Our reference computer system is Silicon Graphics Indigo R4000. More detailed information about the system is found in appendix A.

2. Contents of the benchmarks

The benchmark is divided into the four parts:

(1) A subset of the Standard Perfomance Evaluation Cooperative (SPEC) benchmarks
(2) The CSC benchmark suite, version 2
(3) Parallel benchmarks
(4) Miscellaneous benchmarks

Most of the benchmarks are applied to all vendors, if not otherwise stated.

The following subsections will explain these benchmarks. Where appropriate, the appendices are used to list more detailed information.

2.1 A subset of the SPEC-benchmarks

The aim of this benchmark is to report the per processor performance of the proposed system by following the run rules specified by Standard Perfomance Evaluation Cooperative (SPEC) cooperative.

Because most of the computer manufacturers are currently members of the SPEC organization (or have access to their benchmarks), this benchmark does not (in average) require any explicit running of the benchmarks since the results are reported in SPEC's newsletter quarterly.

We are specifically interested in SPEC Floating Point Suite 92 (SPECfp92 or CFP92) and SPEC Integer Suite 92 (SPECint92 or CINT92) break down or per program results.

The SPEC CFP92 benchmark suite consists of CPU intensive benchmarks that are intended to be meaningful samples of applications which perform floating point logic and computations in a technical computing environment.

The SPEC CINT92 benchmark suite consists of CPU intensive benchmarks that are intended to be meaningful samples of applications which perform non-floating point logic and computations in a technical computing environment.

Note that neither CFP92 nor CINT92 does not assess the ability of a system under test to handle disk, graphics or any form of networking or communication.

Many of the SPEC benchmarks have been derived from publicly-available application programs, and they are intended to be portable to as many current and future hardware platforms as possible.

Appendix B contains a brief description of each SPEC-benchmark program.

2.1.1 Reporting the SPEC-results

We are interested in individual turn-around times for each program. Note however, that unlike SPEC, we will compare these times to our reference computer.

The vendors should report the SPEC-results in the following form (wall clock times in the table indicate reference computer results reported by the Silicon Graphics Inc. in the SPEC-newsletter Vol.4, No.3, September 1992, page 28 and 18):

SPECfp92:

Program	Wall clock time (seconds)
013.spice2g6	541.5
015.doduc	36.3
034.mdljdp2	88.0
039.wave5	89.1
047.tomcatv	40.8
048.ora	101.6
052.alvinn	101.5
056.ear	272.7
077.mdljsp2	76.8
078.swm256	359.2
089.su2cor	176.7
090.hydro2d	182.0
093.nasa7	230.8
094.fpppp	171.5

SPECint92:

Program	Wall clock time (seconds)
008.espresso	41.1
022.li	95.0
023.eqntott	14.4
026.compress	66.9
072.sc	61.9
085.gcc	124.8

In addition to this, the vendor should report the configuration of the benchmarked system as specified in appendix C.

2.1.2 Accessing the SPEC-benchmarks

In case some vendor(s) have currently no access to the benchmarks source codes, they can be purchased from the following address (attn. Dianne Dean):

SPEC [Standard Performance Evaluation Corporation]
c/o NCGA [National Computer Graphics Association]
2722 Merrilee Drive
Suite 200
Fairfax, VA 22031
USA

Phone: +1-703-698-9600 Ext. 318
FAX: +1-703-560-2752
E-Mail: spec-ncga@cup.portal.com

The prices for CINT92 and CFP92 release 1.1 QIC 24 -tapes are 425USD and 575USD, respectively.

2.2 The CSC Benchmark Suite, version 2

This benchmark comprises of a set of programs that represents the average load of the current computer systems at CSC. The benchmark consists of 14 floating point intensive programs. All except one C-code are written in Fortran-77. A description of each program is found in appendix D.

2.2.1 Benchmark program characteristics

Each program in this benchmark set has been carefully analyzed by Cray's Hardware Performance Monitor (hpm) and using some general tools available under Unix operating system. In addition to that reference timings are provided for the reference system, Silicon Graphics Indigo R4000.

Following table provides some static information about the programs:

Program	Prec. (bits)	Source lines	Text (KB)	+ Data+Bss (KB)	= TotalSize (KB)	DiskSpace (KB)
ARCTWOD	64	3759	344	2788	3132	942
CASTEP	64	13337	548	3680	4228	2642
FREQUENCY	32	276	212	25777	25989	628
GRSOS	32	316	196	19613	19809	457
INVPOW93	64	486	220	2930	3150	6408
MOPAC	64	22093	676	26824	27500	1294
NASKER	64	1101	200	2877	3077	351
NBODYOPT	32	1459	152	1224	1376	299
RIEMANN (*)	64	439	48	74	122	86
SIMCZO	64	2069	252	35960	36212	837
WHY12M	64	996	220	23742	23962	18201
MD1	32	1129	168	9230	9398	327
PDE1	64	207	144	37041	37185	287
QCD1	64	2641	252	7763	8015	448

(*) Written in C-language.

All programs should confirm the precision shown above. Note, that GRSOS contains some REAL*8 (DOUBLE PRECISION) operations that should not be removed, but the data-arrays are still INTEGERs and REALs. Note also, that all 64-bit codes, except CASTEP contain explicit DOUBLE PRECISION definition (either by IMPLICIT-statement or variable-wise). This means, the vendor has to activate DOUBLE PRECISION via appropriate compiler flag or, if such does not exist, manually include IMPLICIT DOUBLE PRECISION -statements at the beginning of each routine in CASTEP. Furthermore, CASTEP seems to treat all variables that begin with letter 'C' as COMPLEX.

CASTEP may also require some intrinsic functions (CEXP, AIMAG, AMOD) to be changed (EXP, IMAG, MOD). This change can be made by hand or running CASTEP sources through the '/lib/cpp -P' with appropriate definitions (-DCEXP=EXP -DAIMAG=IMAG -DAMOD=MOD). If the C-preprocessor is invoked by the Fortran-compiler as default, include these definitions as 'FFLAGS' (see chapter 4.2).

Some of the codes also require automatic SAVE-statements to be activated. I rely on the fact that all vendors have such a compiler flag available. I recommend to use "auto-SAVE" for all programs, but especially CASTEP, MOPAC, SIMCZO, MD1, PDE1 and QCD1.

Some of the programs are suitable for fine grain parallelism. We have found that for example ARCTWOD, FREQUENCY, GRSOS, INVPOW93, NASKER may benefit from parallel processors.

The programs can also be divided by the application area. Following table tries to give to each program a typical application area(s):

Program	Application area(s)
ARCTWOD	Fluid Dynamics, Engineering
CASTEP	Chemistry, Physics
FREQUENCY	Engineering, Mathematics
GRSOS	Physics
INVPOW93	Engineering, Mathematics, Eigenvalues
MOPAC	Chemistry
NASKER	Fluid Dynamics, Engineering, Mathematics
NBODYOPT	Astrophysics, Mathematics
RIEMANN	Mathematics
SIMCZO	Structural Mechanics, Fluid Dynamics
WHY12M	Mathematics, Sparse Matrices
MD1	Chemistry
PDE1	Mathematics, Partial Differential Equations
QCD1	Quantum Mechanics

In the next table approximate nominal megaflop-counts are derived from Cray's 'hpm' Group-0 and Group-3 executions and reference timings for the reference computer system. Nominal megaflop-count and performance are described more detailly in the 'Performance metrics' chapter later on.

Table also provides additional information gathered from 'hpm':

- o Vectorization percentage, which is calculated from the ratio of vector floating point operations to total number of floating point operations (vector & scalar) on Cray X-MPEA.
- o Average vector length (Avg.VL), which is expressed in modulo 64 since Cray X-MPEA's vector registers length is 64.
- o Memory references tell how many accesses to 64-bit precision words have been occurred during the execution of the program.

The application performance for the reference system is calculated by dividing the nominal megaflop-count on Cray X-MPEA by the wall clock time on the reference system. This is NOT the REAL Mflop/s, only a fairly good approximation in most cases. However, very high values (reaching or exceeding the theoretical peak) may indicate that the compiler and/or the pre-processor has done a good job or the program takes advantage of the 32-bit arithmetic that is not available on Cray X-MPEA.

Program	Floating point Nom.Ops.	Vector	Avg.VL	Memory Refs.	Wall clock time (sec)	Application Performance
ARCTWOD	555M	100%	53	692M	81.2	6.8
CASTEP	3211M	69%	43	3880M	450.7	7.1
FREQUENCY	517M	100%	63	1405M	84.2	6.1
GRSOS	9162M	97%	59	3192M	446.3	20.5
INVPOW93	1014M	100%	57	1521M	143.8	7.0
MOPAC	1256M	58%	11	1573M	162.1	7.7
NASKER	2149M	100%	53	2169M	294.5	7.3
NBODYOPT	824M	86%	10	547M	64.2	12.8
RIEMANN	248M	0%	0	2527M	72.2	3.4
SIMCZO	1356M	94%	32	2311M	659.4	2.1
WHY12M	164M	3%	0	1017M	124.7	1.3
MD1	837M	22%	63	576M	92.4	9.1
PDE1	494M	100%	60	522M	92.3	5.4
QCD1	1314M	94%	64	1792M	268.7	4.9
Total	23101M	66%	41	23724M	3036.7	7.6

From these reference results following additional information is derived:

Statistic	Reference system value
Benchmark performance	7.6
Geometric mean performance	5.9
Arithmetic mean performance	7.2
Harmonic mean performance	4.6
Benchmark instability	15.8

Please refer chapter 3 to understand these metrics.

2.2.2 Run rules

Performance results should be given for two versions of the codes:

- o Baseline
- o Optimized

CSC provides the Baseline source codes. It is up to vendor to optimize Baseline source codes in order produce the Optimized versions. We will consider this as a credit to the vendor.

In addition to this, varying number of processors must be used to accomplish the tasks. For the Baseline and Optimized runs we following number of processors must be used:

- o single processor results
- o P-processor results
- o P/2-processor results [P/2+1 is P is odd]
- o P_opt-processor results

```
where P = maximum number of proposed processors.
      P_opt = number of processors that will give optimal performance
              each individual program.
```

This means that in principle maximum of 2x4 executions of each benchmark program is required in order to run complete benchmark suite. However, it is left to vendor to decide whether to optimal number of processors differs from P-processor results.

When running any of the benchmark programs, the following general run rules must be followed:

- o All times reported must be for runs that produce correct results.
- o All information necessary for replication of the results should be disclosed and available at request.
- o Single-user mode is allowed, but must be reported.
- o Use of benchmark specific software (preprocessors etc.) is not allowed. Note, that this does NOT prevent use regular preprocessors provided by vendors nor use of KAP, VAST etc. However, all performance improvers must be included in the proposed system, too.

To obtain the Baseline results, the vendor must obey following additional rules:

- o Source code may not be modified unless it is required for portability. This includes manual insertion of compiler directives. All changes must be reported.
- o No use of scientific libraries that would replace the original code is allowed unless this is done automatically by the compiler and does not increase the compilation/linking time dramatically.

Thus, the general rule for obtaining the Baseline results is to give compiler and preprocessors full freedom to optimize as much as possible without need of any making source code modifications ny hand.

To obtain the Optimized results, the vendor CAN do the following things:

- o Modify by hand the source as required, but still solving the same problem and providing the same output (final and intermediate) using the same input files.
- o Insert compiler directives.
- o Insert calls to the scientific libraries

2.2.3 Reporting the CSC Benchmark results

Before reporting the actual run times, the vendor should report the compilation and linking times (wall clock time) for each application and case. We will consider it as a credit if the compiler and linker do not spend too much time in optimizing codes.

For the Baseline and the Optimized results, the vendor should report the CSC Benchmark Suite results in the following form:

Program	# of CPUs = 1	# of CPUs = P/2	# of CPUs = P
	Wall clock time (seconds)	Wall clock time (seconds)	Wall clock time (seconds)
ARCTWOD			
CASTEP			
FREQUENCY			
GRSOS			
INVPOW93			
MOPAC			
NASKER			
NBODYOPT			
RIEMANN			
SIMCZO			
WHY12M			
MD1			
PDE1			
QCD1			

To report the "P_opt-processor" results, create similar table than above, but specify also the number of processors used to run each of the programs. This means that the actual value of "P_opt" may vary from program to program.

2.3 Parallel benchmarks

2.3.1 CSCSUITE_2

This item was already covered in 2.2.3.

2.3.2 DM

DM or distributed memory benchmarks consists of 4 programs, of which one belong to miscellaneous tests (COMMS1). The rest three are DM-versions of MD1, PDE1 and QCD1. They are coded in Fortran, but contain system dependent PARMACS message passing macros (see Appendix E) to facilitate process to process communication.

The following cases should be at least run (2, 4, 8, 16 processors):

- o MD1 - NC=9 i.e. 4*9^3 (2916) & NC=11 i.e. 4*11^3 (5324) atoms
- o PDE1 - NN=6 & NN=7
- o QCD1 - 16*4^3 and 8*8^3 systems

The brief instructions to these tests are found in chapter 4.

2.4 Miscellaneous benchmarks

The brief instructions to these tests are found in chapter 4.

3. Performance metrics

In this chapter some crucial performance metrics used in this report are described.

3.1 Wall clock time

This is known also as real elapsed time or turn-around time or time-to-solution or running time. It is measured from the beginning of program to the end of application. It is the only relevant measure while comparing parallel applications with each other.

In any benchmark context CPU-times are often mentioned. During this benchmark we are not very much interested in these figures.

3.1.2 Sum of running times

Given a set of wall clock times for different benchmark applications, the sum of running times is a simply sum of wall clock times. During this benchmark it has no special meaning, but is included for for the sake of completeness.

3.2 Ratio of wall clock times

Assuming that wall clock times have been measured for a particular application. The ratio of wall clock times is application wall clock time over reference computer system's wall clock time.

3.3 Nominal flop count

Nominal flop count are gathered from Cray's Hardware Performance Monitor (hpm) by counting the Cray hardware multiplies, adds and reciprocals for a particular application and converting them to nominal megaflop count using following formula:

$$\text{Nominal flop count} = \text{Multiplies} + \text{Adds} - 2 * \text{Reciprocals}$$

The formulr stems from the fact that one divide on Cray hardware requires 1 reciprocal and 3 multiplies (or 2 multiplies and one add).

Although the REAL flop count varies from system to system, this value is probably the best value that we can reliably get. And it stays constant once obtained.

3.4 Application performance (nominal Mflop/s)

Dividing the program Nominal flop count (expressed normally in megaflops) by the wall clock time in a particular machine, we get a fairly good measure for application performance that resembles the famous Mflop/s metric:

$$\text{Application performance} = \frac{\text{Cray's Nominal flop count}}{\text{Application wall clock time on any machine}}$$

This can also be misleading, if interpreted incorrectly. Namely smart compiler may optimize away a lot of floating point operations that current Cray compiler was not able to. Also 32-bit codes may perform "unexpectedly" fast.

We recommend to call it as an "application performance", NOT a Mflop/s.

3.5 Benchmark performance

Benchmark performance is defined by the ratio of sum of all applications' nominal flop counts to sum of all applications' wall clock times:

$$\text{Benchmark performance} = \frac{\text{Sum of applications' nominal flop counts}}{\text{Sum of applications' wall clock times}}$$

3.6 Benchmark averages

Various summaries can be drawn from the results of several individual program executions. These summaries are sometimes characterized by different averages.

Although we don't rely on averages as much as individual program results, it is still worth to mention them. One of the most popular and less misleading average is the Geometric mean. It is not very sensitive to large variations in data values. Sometimes very misleading ones are the Arithmetic and Harmonic means. The former one will give unexpectly good averages if only one data value gets high. The latter one in turn works in the other way around: if there is one bad result (low value), it will destroy the whole data set average.

3.6.1 Geometric mean

Geometric mean is defined as a Nth root of a product of N data values:

$$\text{Geometric mean} = \left(\prod_{i=[1..N]} \{data\_i\} \right)^{1/N}$$

3.6.2 Arithmetic mean

Arithmetic mean is defined as an average of N data values:

$$\text{Arithmetic mean} = \left(\sum_{i=[1..N]} \{data\_i\} \right) / N$$

3.6.3 Harmonic mean

Harmonic mean is defined as a reciprocal average of N reciprocal data values:

$$\text{Harmonic mean} = \frac{N}{\sum_{i=[1..N]} \{1 / data\_i\}}$$

3.7 Benchmark instability

Benchmark instability of a tested machine is defined by taking the ratio of maximum attained application performance to minimum attained application performance of the programs included in the benchmark set:

$$\text{Benchmark instability} = \frac{\text{Maximum attained application performance}}{\text{Minimum attained application performance}}$$

3.8 Parallel program measures

Parallel program measures stem from the fact that run time for any application is essentially composed of two parts: sequential and parallel portion's time, T_s and T_par, respectively. Thus, one-processor wall clock time is expressed in form:

$$T(1) = T_s + T_{par}$$

Running the same application in parallel with P-processors will result the run time to be:

$$T(P) = T_s + T_{par} / P$$

This is, however, only an approximation, since increasing the number of processors will also require synchronization and communication to be included. Thus, in reality the run time for parallel application is higher:

$$T(P) = T_s + T_{par} / P + T_{sc}(P)$$

where T_sc is the time spent in synchronization and communication between processors and is typically a monotonically increasing function of the number of processors P.

The latter formula is the reason why we want the vendor to run certain applications with "P_opt-processors". In reality, if we exclude so called embarrasingly parallel applications, no continuous performance increase is gained by increasing the number of processors. This is because typical parallel application suffers from synchronization and communication bottlenecks and after certain number of processors this throuphput time becomes larger, thus making application to run slower eventhough more processors are put to work together.

In the distributed memory benchmarks we would also like to see the size effect to the solution time. If the problem size is N (defined appropriately), then solution time varies according to following formula:

$$T(P,N) = T_s(N) + T_{par}(N) / P + T_{sc}(P,N)$$

3.8.1 Parallel speedup

In order to measure parallel speedup, it is currently recommended by the benchmark authorities, that the speedup itself should not be used to compare results between different architectures. However, using is to test application speedup within a single architecture is not a bad idea. Parallel speedup is defined:

$$\text{Parallel speedup} = \frac{\text{One processor wall clock time}}{\text{P-processor wall clock time}}$$

Note, that if the code has been optimized, then comparing the speedups of baseline results to optimized is illegal. This is because the one processor version performs now much better than in baseline resulting lower parallel speedups for optimized results. A general rule is: the better attained single processor performance the lower parallel speedup.

Note also, that parallel speedup may exceed actual number of processors if the problem does not fit properly into the memory of one-processor system. This results too high one-processor execution than what would be expected if there were enough memory available.

3.8.2 Parallel efficiency

This is defined by the following formula and expressed in percentages:

$$\text{Parallel efficiency} = \frac{\text{One processor wall clock time}}{(\text{P-processor wall clock time}) * P} \times 100\%$$

3.9 Measuring communication overhead

In distributed memory applications which use mainly message passing to exchange data between processors, a communication becomes an important aspect. During data exchange processes are normally busy with sendign or receiving the data, thus creating serious bottlenecks to parallel code.

3.9.1 Computation to communication ratio

One way to measure the goodness of message passing application is to keep track of the time spent in communication. The computation to communication ratio is thus defined:

$$\text{Comp.Comm. ratio} = \frac{\text{Wall clock time} - \text{Communication time}}{\text{Communication time}}$$

The larger the ratio, the better performance and less communication.

3.9.2 Modeling communication time

Communication itself can be modeled by the following linearized formula:

```
Communication time = Latency_time + Transfer_Speed * Number_of_Bytes
```

The Latency_time is essentially the time spent while sending zero length message. The Transfer_Speed is the speed of communication network. This is usually also a weak function of number of bytes transferred.

4. Execution instructions

4.1 Reading the benchmark tape

The benchmark streamer tape (cartridge) is written on Sun Sparcstation's tape drive using 'tar'-command. Before reading it to disk, specify benchmark root directory. I will refer it here through the environment variable BENCH:

```
% setenv BENCH /my/benchmark/root/directory
% mkdir $BENCH
% cd $BENCH
```

To unload the tape, type:

```
% tar xv
```

In case of alternate tape drive, use command like this:

```
% tar xvf /dev/other_tape_drive
```

Some systems may not recognize the tape format. In such case you may have to use 'dd'-command with bytes swaps:

```
% dd if=/dev/tape_drive ibs=obs conv=swab | tar xvf -
```

After this run following command to install the benchmark files properly:

```
% install
```

This will do the rest; for example create additional directories, uncompress few compressed tar-files found in tape, create some useful symbolic links.

In case of serious problems in unloading the tape, I will put the tar-file also into our anonymous ftp nic.funet.fi (128.214.6.100), under directory pub/csc/benchmark. Use binary transfer more to retrieve the benchmark tar-file named 'spring93.tar'.

4.2 Directories, files and utility functions

Once you have unloaded the tape, you will find following subdirectories under your benchmark root directory:

- CSCSUITE_2
- PARALLEL
- MISC

These refer to the test sets described in the chapter 2. Naturally there are no files for the SPEC-benchmarks.

In the following subsections I will use next two abbreaviations quite often:

- <applic> - refers to application name (ARCTWOD, CASTEP, etc.)
- <arch>   - refers to computer architecture (SGI, CRAY, etc.)

Also, there are two important files that appear time to time during use of command scripts:

- program.list   - contains list of application codes
- <arch>.make_flags - contains default modifications to standard makefile settings

For example, file 'CRAY.make_flags' may look like this:

```
ARCTWOD - Fortran    # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
CASTEP - Fortran     # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
FREQUENCY - Fortran  # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
GRSOS - Fortran      # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
INVPOW93 - Fortran   # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
MOPAC - Fortran      # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
NASKER - Fortran     # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
NBODYOPT - Fortran   # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
RIEMANN - C          # CC=cc
SIMCZO - Fortran     # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
WHY12M - Fortran     # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
MD1 - Fortran        # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
PDE1 - Fortran       # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
QCD1 - Fortran       # FC=cf77 ;LD=cf77 ;FFLAGS=-Zv -Wf'-dp -a static'
```

A brief explanation to this. Take for example ARCTWOD:

- ARCTWOD is written in Fortran
- override default FC-setting (FC=f77) in makefile with cf77
- override default LD-setting (LD=f77) with cf77
- use new flags for Fortran compilation: -Zv -Wf'-dp -a static'

The format is simply the following (one program per line):

- Program name
- dash ('-')
- language (keywords 'Fortran' or 'C')
- hash ('#')
- overrides: NAME=new_value
- separator between overrides is semicolon (';')

This file is parsed by using 'awk' during generation of application specific makefiles.

4.2.1 CSCSUITE_2

CSCSUITE_2 contains several shell scripts and also subdirectories for each application to run the tests described in section 2.2. In addition to that a subdirectory called "lib" is found. This is a depository for timer-function routines and library.

Each application directory contains files and directories organized in the following way:

```
        Directories:
          o <applic>/src     - fsplit'ted Fortran or C-source codes
          o <applic>/results - reference results directory (<applic>.out)

        Files:
          o <applic>/<applic>.exe - the executable for the application
          o <applic>/<applic>.in  - standard input file (if any)
          o <applic>/<applic>.out - standard output file (if any)
          o <applic>/*            - other files: miscellaneous input
          o <applic>/src/Makefile.<arch> - Architecture dependent makefile
```

Following shell scripts are found directly under CSCSUITE_2:

```
        o Makefile - A driver makefile for some scripts
        o program.list - List of application programs. Used by scripts.
        o <arch>.make_flags - Default compiler/linker etc. flags used upon
                              generation of architecture dependent makefile
        o build_script - to build Bourne-shell run script for a single
                         application. See also 4.2.5 for limiting CPU-number.
        o build_all_scripts - to build all scripts in one shot
        o build_make - to build architecture dependent makefile for
                       specific application
        o build_all_makes - to build all architecture dependent makefiles
        o make_program - to invoke architecture dependent makefile for
                       specific application
        o make_all_programs - to make all programs, including libraries
        o run_program - to run specific program
        o run_all_programs - to run all programs after each other
        o get_times - a utility to collect times from <applic>/<applic>.out's
        o get_diffs - a utility to compare results with reference system
```

4.2.2 PARALLEL

PARALLEL directory contains two subdirectories CSCSUITE_2 and DM.

CSCSUITE_2 is essentially a duplicate of CSCSUITE_2-directory that has been created during installation of the tape. Its role is to provide similar environment than CSCSUITE_2, but for running 14 application codes now in parallel, P > 1. I added also few new scripts to facilitate this:

```
o run_parallel - to run application in parallel with varying
                 number of processors
o get_parallel_times - to get tabulated list of parallel times
                       for a single application
```

DM is a directory for distributed memory benchmarks COMMS1, MD1, PDE1 and QCD1 as well as depository for the PARMACS - PVM 2.4.2 -interface. Interface calls are documented briefly in appendix E.

The test program COMMS1 does not actually belong to DM-applications, but miscellaneous tests. As it contains PARMACS macros, it should be run under this directory. It is explained briefly under 4.2.3, MISC.

4.2.3 MISC

MISC contains subdirectories and files for miscellaneous tests:

```
o MEMTEST - Several memory tests:
        - CACHEMISS to test hardware behaviour during cache conflicts
        - LARGEMEM to test how large single array can be allocated
        - MEMSCAN to scan arrays with stride one or ramdomly
o IOTEST - I/O-subsystem -tests
        - IOZONE to write/read file with several block/filesizes
o KERNEL - Kernel operations test
        - BRE to run 15 different kernels tests
```

Although MISC directory does not contain communication test (COMMS1), such is included under PARALLEL/DM -directory. Purpose of this test is to test communication speed between two nodes when message size varies from 1 to 40000 bytes.

4.2.4 Timer functions

Throughout the benchmark subdirectory 'lib/' contains timer-functions. As we are not interested in CPU-times in this benchmark, but wall clock times, a utility function to get this has been coded. Times are obtained by using C-routines that are called by Fortran-routines. Fortran-routines in turn are called by user program.

Some minor (or none) modficications are needed to link properly the timer-interface.

Fortran-routines ('lib/src/timer.f') are:

```
- SUBROUTINE INITIM(IDUMMY) to initialize timer (once a run).
  Done implicitly by the library when any of the Fortran-routines
  found in 'lib/src/timer.f' are called.
- SUBROUTINE SHOTIM(IDUMMY) to print out current wall clock time
  since initialization of timer.
- SUBROUTINE TIMER(T) , DOUBLE PRECISION T
  to store wall clock time since initialization to the variable 'T'
- DOUBLE PRECISION FUNCTION CPUTIM()
  returns actually the wall clock time since initialization.
```

Among the C-routines ('lib/src/times.c') that Fortran-routines above call, the most important is 'waltim'. As this is called by the Fortran-routines, some computer systems may need the underscore to be appended after routine name ('waltim_') or capitalized expression be used ('WALTIM').

Not all benchmarks use these timer routines. In case the first time any benchmark program is executed and the execution fails, I recommend to the 'lib/' directory.

4.2.5 Controlling the number of processors

In some systems, especially shared memory multiprocessors, it is possible explicitly to specify the number of processors to be used in runtime. In order to be sure that a proper number of processors is in use, modify 'build_script' command procedure before trying to generate run scripts at any time. For example, Silicon Graphics requires environment variable 'MP_SET_NUMTHREADS' to be set to one (1) when one-processor results are needed.

4.3 Compiling and linking the applications

4.3.1 CSCSUITE_2

In order to successfully compile and link applications, change to CSCSUITE_2 directory and go through following steps:

(1) Run 'build_all_scripts' to create Bourne-shell run-scripts. Modify, if necessary, files 'build_script' or 'build_all_scripts'.

NOTE: Be sure that your application really uses ONE processor in this context. Modify 'build_script' accordingly.

(2) Create file <arch>.make_flags to contain default settings for your system <arch> for subsequent makefile-generation step. (Hint: Use file 'SGI.make_flags' as example)

(3) Generate makefile's under <applic>/src -directories using command 'build_all_makes'.

(4) Check validity of just created makefiles, <applic>/src/Makefile.<arch>.

(5) Make the executables and record the compilation & linking time for each application. Use command 'make_all_programs'. If make fails, lower the optimization level for particular source by modifying corresponding <applic>/src/Makefile.<arch>, and re-run make. Change the source code only in extreme cases. Report the changes.

Here is what I did when I built the executables for SGI:

```
% build_all_scripts
% emacs SGI.make_flags        # See beginning of section 4.2
% build_all_makes
% make_all_programs SGI clean # Clean all possible junk
% make_program lib SGI        # Be sure 'timer.a' exist. See 4.2.4
% make_all_programs SGI -n    # Trial run; don't make anything yet
% make_all_programs SGI       # Make really now!
```

After that I had to lower the optimization level in one of the CASTEP routine. So I modified CASTEP's makefile manually and re-ran the make:

```
% emacs CASTEP/src/Makefile.SGI
% make_program SGI CASTEP clean # Clean first to remove garbage
% make_program SGI CASTEP
```

To record the compilation & linking time for each application, you can make each program in the following manner:

```
% time make_program <arch> <applic>
```

and write down the time.

4.3.2 Parallel benchmarks

4.3.2.1 CSCSUITE_2

Follow the same rules as in 4.3.1, but make changes to <arch>.make_flags to activate parallel compilation. Be sure that while running any of the applications, the number of processors is set either to 'P/2' or 'P' (or

P_opt !). Thus you must check 'build_script' before building scripts and refer the result table in the chapter 2.2.3

4.3.2.2 DM

Generally, follow the same rules as in 4.3.1, but make changes to <arch>.make_flags. As this benchmark contains 4 tests for distributed memory computing that use PARMACS macros presented in Appendix E, some site depended changes are needed.

Source files for this tests are written in Fortran (extension .f) or using m4-macro processors language (.m4 files). Before actual Fortran compilation the m4-macro files containing PARMACS-macros must be preprocessed to get the Fortran equivalents. To facilitate this "preprocessing", we provide the vendor with CSC-developed PARMACS -- PVM 2.4.2 interface.

In order to successfully link the DM-application with our PVM-interface, following steps must be performed:

- o Go to 'pvm2.4.2/' directory, read the Postscipt-instructions how install pvm for your machine (in case you are not familiar with it), and replace libraries 'libpvm.a', 'libfpvm.a' and PVM-daemon executable 'pvmd' with your equivalents.

- o Go to 'macrolib/' and check the Fortran-interface that is called by the application program once the PARMACS calls have been substituted. The source files under 'macrolib/src' use some C-routines, like 'getarg()', 'iargc()', 'getcwd()', sleep(). Check that your system accepts these to be called from Fortran.

In case you wish to use vendor specific PARMACS-routines, some other path must be followed.

As these, rather difficult, steps have been completed, you may have to modify following files:

- o build_make - to re-organize libraries
- o <arch>.make_flags - to provide additions/substitutes to compiler flags

Once all seems to work, do as in 4.3.1.

Note that each application will has two executables: 'host' and 'node'. They are also fixed size master and slave process executables.

4.3.3 Miscellaneous benchmarks

4.3.3.1 Memory tests

This contains three separate tests: CACHEMISS, LARGEMEM and MEMSCAN. These are located in the directories MISC/MEMTEST/CACHEMISS, MISC/MEMTEST/LARGEMEM and MISC/MEMTEST/MEMSCAN, respectively. Please refer the 'Readme' files there for detailed information.

The CACHEMISS is a C-program that performs a full matrix multiply with increasing stride. We plot a curve where X-axis is the problem size (matrix dimension or effective stride) and Y-axis contains nominal Mflop/s rate calculated by the program. The vendor is encouraged to provide equivalent curve for an optimized version, that probably is written in Fortran or uses library routines to perform the task.

To create CACHEMISS, go to its directory, modify 'Makefile' and create executable.

LARGEMEM is a small program that is used to test how large single array can be allocated in Fortran and in what circumstances. It contains four cases, in each case the array to be allocated is in DOUBLE PRECISION:

- o mem1.m4 - array is to be allocated (probably) from stack
- o mem2.m4 - array is put to a named COMMON-block
- o mem3.m4 - array is put to a unnamed COMMON-block
- o mem4.m4 - array is SAVE'd ("static")

To create LARGEMEM, go to its directory, and run 'generate.csh'. This actually executes the actual programs, too.

MEMSCAN is a "memory scanner" that performs one dimensional array operations, like summing, scaling, saxpy, assigment and so on. Four different versions exist:

- o sequential and random scan (gather/scatter -like)
- o single and double precision for both above

To create MEMSCAN, go to its directory, modify 'Makefile.SGI' and run it.

4.3.3.2 I/O-tests

This consists of modified IOZONE-test. Purpose of this test is to check file buffer caching effect. Program writes sequentially N byteblocks of data into a file with size X megabytes. Then file is closed and opened again, but this time for reading.

A version that explicitely does 'fsync()' before 'close()' is run against version that does not 'fsync()'. The purpose of 'fsync()' is enforce write to disk and then return back to application. I hope all vendor can provide similar functionality, if 'fsync()' itself does not exist.

To create IOZONE test programs, IOTEST/IOZONE directory, modify 'Makefile.SGI' and run it.

4.3.3.3 Computational kernel test

This test contains 15 BLAS-1 or -2 operations that are supposed to test single processor performance in kernel operations.

The test is located in KERNELS/BRE directory. To create the 'BRE.exe', modify makefile and run it.

4.4 Running the benchmarks

In order to run any of the benchmarks, be sure that there's enough disk space available in the planned run directory. This directory can be located in another disk than the benchmark directory. Also, the root directory for the runs must exist.

The run scripts do not refer to any particular run directory, other than specified when program was invoked. In the following sections we refer the run directory as <rundir>.

4.4.1 The CSC Benchmark Suite

To run the CSC Benchmark Suite, you can use the following scripts:

- o run_program - to run a single program
- o run_all_programs - to run all programs after each other

Consider 'run_program'. It will run a single program in the following manner (assume working directory CSCSUITE_2):

- o Checks that <rundir> exist. If not, tries to create it.
- o Creates application run directory <rundir>/<applic>. If this already exist (from previous trial run, for example), rename it to <rundir>/old.<applic>. And if in turn <rundir>/old.<applic> already exist, delete all files under it.
- o Copies all regular files (not directories or files under directories) found under <applic> to <rundir>/<applic>.
- o Changes directory ('cd') to <rundir>/<applic> and start running the application.
- o Upon completion, file <rundir>/<applic>/<applic>.out is copied back to <applic>/<applic>.out.

Use 'get_times' and 'get_diffs' to collect timing information and differences compared to reference results. 'get_times' applies directly to <applic>/<applic>.out files for each application creating a tabulated timing summary for application. 'get_diffs' similarly compares differences on output files <applic>/<applic>.out and reference result files <applic>/results/<applic>.out.

A sample execution:

```
% run_program RIEMANN /tmp/bench  # Runs RIEMANN under /tmp/bench
% run_all_programs /tmp/bench     # Runs all program in sequence
% get_times  >  <arch>.summary    # A tabulated list of timings
% get_diffs  >  <arch>.diff       # A list of differences
```

4.4.2 Parallel benchmarks

4.4.2.1 CSCSUITE_2

Generally, follow the rules in sequential CSCSUITE_2. Check also that the processor number is correct. Following scripts are new or modified since reading the sequential CSCSUITE_2:

- build_script
- get_parallel_times
- get_parallel_diffs
- run_parallel

A sample execution:

```
% run_parallel ARCTWOD /tmp/paral 2 4 # Run ARCTWOD using 2 and 4 procs
                                        under /tmp/paral
% get_parallel_times  >  <arch>.summary  # A tabulated list of timings
% get_parallel_diffs  >  <arch>.diff     # A list of differences
```

4.4.2.2 DM

All DM-benchmarks consists of several cases to be run per application. The input files are found in <applic>/<applic>.cases* files. Each case label can be recognized in the input file name. For example, the case '11_8x1x1' refers to file 'MD1/MD1.case11_8x1x1'. The case label means: run MD1 with 4 times 11 cubed atoms, use processor topology 8 by 1 by 1 processors (8 processors in ring).

It is up to vendor to choose the to choose the topology. Only number of processors matters. For instance, the case above can be replaced with '11_2x2x2' if the vendor think this provide faster turn-around time. But also input file 'MD1/MD1.case11_2x2x2' with the corresponding changes must be embedded.

If the PVM-interface is acceptable, following scripts may found helpful:

- run_dm - runs one DM-application with variable number of "cases".
- run_all_dms - runs all required DM-applications in sequence.
- run_pvm - invoked by 'run_dm'. Check whether PVM-daemon is already running and prevents accidental starting of additional PVM-daemon(s).
- kill_pvm - kills currently active PVM-daemon which in turn kills all processes that communicate with this daemon.

A sample execution:

```
% run_dm MD1 /tmp/dm 11_2x1x1 11_8x1x1  # Runs two cases of MD1
                                          under /tmp/dm
% kill_pvm                             # Be sure that PVM-daemon is down
% run_all_dms  /tmp/bench              # Runs all DM-programs in sequence
% kill_pvm                             # Be sure that PVM-daemon is down
% get_dm_times  >  <arch>.summary  # A tabulated list of timings
```

4.4.3 Miscellaneous benchmarks

All the following run scripts are found under corresponding program directory.

Run CACHEMISS by typing 'cachemiss.csh'. Apply 'matlabgen.csh' to get Matlab-suitable data file for curve plotting.

Run LARGEMEM by typing 'generate.csh'.

Run MEMSCAN by typing 'run_memscan'.

Run IOZONE by typing 'run_iozone'. And 'get_times' applied to logfile.

Run BRE kernel test by typing 'run_bre'. Modify BRE.dat -file before that.

Run communication test 'COMMS1' as a part of DM-application.

5. Contact information

Any questions about the benchmark should be directed to me or Klaus Lindberg. I will be in holiday between April 13 and 25. Here is more information:

Sami Saarinen (or Klaus Lindberg)
Center for Scientific Computing (CSC)
Tietotie 6
P.O.Box 405
FIN-02101 Espoo
Finland

Tel: Int + 358 - 0 - 457 2713 (Sami, direct)
Int + 358 - 0 - 457 4050 (Klaus, direct)
Int + 358 - 0 - 457 1 (switch board)
Fax: Int + 358 - 0 - 457 2302

And as stated in section 4.1, the benchmark file is obtainable via anonymous ftp in case of tape reading problems.

The results should be send to us in printed format and, if possible, in a tar-file using streamer tapes, reels or so. We can even provide you a ftp access to some specific place at our center to send us the result data.

Appendix A

Configuration of the CSC reference computer system:

(1) Hardware Configuration:

```
Manufacturer:           Silicon Graphics Inc.
Model number:           INDIGO R4000
CPU:                    MIPS R4000 Processor Chip Revision: 2.2
FPU:                    MIPS R4010 Floating Point Chip Revision: 0.0
Speed:                  50 MHZ IP20 Processor
Peak performance:       50 Mflop/s (64-bit)
Number of CPUs:         1
Data cache size:        8 Kbytes
Instruction cache size: 8 Kbytes
Secondary unified instruction/data cache size: 1 Mbyte
Main memory size:       96 Mbytes
Disk subsystem:         1.6GB + 1.2GB + 0.4GB SCSI
Other Hardware:         None
Network Interface:      Integral Ethernet: ec0, version 1
```

(2) Software Configuration:

```
O/S & Version:          IRIX 4.0.5F
Compilers & Version:    SGI Fortran 77, 3.4.1
                        SGI Ansi C, 1.1
Compiler flags:         -O2 -static -sopt,-so=3,-r=3,-ur=8 -jmpopt -lfastm
                        (except in programs CASTEP, GRSOS, MOPAC, NBODYOPT,
                         RIEMANN, SIMCZO, WHY12M and MD1 where
                         "-O2 -static" was used)
Other Software:         Fortran 77 Fopt Scalar Optimizer (KAP)
File system type:       SGI efs
```

(3) System Environment:

```
System state:           Multi-user
Tuning Parameters:      None
Background load:        None
```

Appendix B

CFP92, current release: Rel. 1.1:

This suite contains 14 benchmarks performing floating-point computations. 12 of them are written in Fortran, 2 in C. The individual programs are:

013.spice2g6	Simulates analog circuits (double precision).
015.doduc	Performs Monte-Carlo simulation of the time evolution of a thermo-hydraulic model for a nuclear reactor's component (double precision).
034.mdljdp2	Solves motion equations for a model of 500 atoms interacting through the idealized Lennard-Jones potential (double precision).
039.wave5	Solves particle and Maxwell's equations on a Cartesian mesh (single precision).
047.tomcatv	Generates two-dimensional, boundary-fitted coordinate systems around general geometric domains (vectorizable, double precision).
048 ora	Traces rays through an optical surface containing spherical and planar surfaces (double precision).
052.alvinn	Trains a neural network using back propagation (single precision).
056.ear	Simulates the human ear by converting a sound file to a cochleogram using Fast Fourier Transforms and other math library functions (single precision).
077.mdljsp2	Similar to 034.mdljdp2, solves motion equations for a model of 500 atoms (single precision).
078.swm256	Solves the system of shallow water equations using finite difference approximations (single precision).
089.su2cor	Calculates masses of elementary particles in the framework of the Quark Gluon theory (vectorizable, double precision).
090.hydro2d	Uses hydrodynamical Navier Stokes equations to calculate galactical jets (vectorizable, double precision).
093.nasa7	Executes seven program kernels of operations used frequently in NASA applications, such as Fourier transforms and matrix manipulations (double precision).
094.fpppp	Calculates multi-electron integral derivatives (double precision).

CINT92, current release: Rel. 1.1:

This suite contains 6 benchmarks performing integer computations, all of them are written in C. The individual programs are:

008.espresso	Generates and optimizes Programmable Logic Arrays.
022.li	Uses a LISP interpreter to solve the nine queens problem, using a recursive backtracking algorithm.
023.eqntott	Translates a logical representation of a Boolean equation to a truth table.
026.compress	Reduces the size of input files by using Lempel-Ziv coding.
072.sc	Calculates budgets, SPEC metrics and amortization schedules in a spreadsheet based on the UNIX cursor-controlled package "curses".
085.gcc	Translates preprocessed C source files into optimized Sun-3 assembly language output.

Appendix C

Use following guidelines to report system configuration under test:

(1) Hardware Configuration:

A description of the system (cpu/clock, fpu/clock), number of processors, relevant peripherals, etc. are included in this space. The amount of information supplied should be sufficient to allow duplication of results by another party.

The following checklist is provided to show examples of hardware features which may affect the performance of benchmarks. The checklist is not intended to be all-inclusive, nor is each feature in the list required to be described. The rule of thumb is: "if it affects performance or the feature is required to duplicate the results, describe it":

- o Manufacturer
- o Model number
- o CPU component ID and clock speed
- o Floating Point Unit (FPU) ID and clock speed
- o Theoretical peak performance in Mflop/s (32- and 64-bit floating point operations) per processor
- o Number of CPUs, FPUs and vector units
- o Cache Size (per CPU), description and organization
- o Memory (Amount and description)
- o Disk subsystem configuration
    - Number of active connected disks
    - Disk controllers: ID, number and type
    - Disk: ID, number and type
- o Other Hardware
- o Network Interface

(2) Software Configuration:

The description of the software configuration used should be detailed enough so that another party could duplicate the results reported. The following is a list of items that should be documented:

- o Operating System Type and Release level (with revision/update level if appropriate)
- o Compiler release levels used. If multiple compilers are available, specify which ones were used and when
- o Compiler flags used per test program/source file
- o Other Software required to reproduce results
- o File system type used
- o Firmware level

(3) System Environment:

This section should document the overall systems environment used to run the benchmark. The following is a list of possible items that might be documented:

- o Single or multi-user state
- o System tuning parameters
- o Process tuning parameters (e.g. stack size, time slice)
- o Background load, if any

Appendix D

CSC Benchmark Suite: Version 2

ARCTWOD

Program solves the Euler equations in generalized curvelinear coordinates using implicit finite-difference method with approximated factorizations and a diagonalization of the implicit operators. The test case involves with 0.8-Mach flow over upper of a biconvex airfoil in a stretched 165x48 rectangular mesh.
The original code (ARC2D) was developed by Thomas Pulliam at the NASA Ames Research Center.

CASTEP

Program stands for CAmbridge Serial Total Energy Package and is an Ab Initio Molecular Dynamics program with Conjugate Gradient minimization for the electronic energy.
The Al-Si-Al trimer, arranged originally in linear geometry, is calculated using Car-Parrinello-type first principles density functional method. The wave functions are expanded using plane waves, and the Fourier coefficients are solved using Conjugate Gradient technique. The atomic positions are solved by minimizing the total energy with respect to the atomic coordinates.
The example program does 25 iterations.
CASTEP is maintained and partially developed by Victor Milman at the Cambridge University, UK. The input file for the test runs was prepared by Ari Seitsonen at the Helsinki University of Technology.

FREQUENCY

Program creates frequency dependent complex transfer function matrix and calculates magnitude and phase angle for a given excitation in a given degree of freedom. To accomplish the task, the program uses 50 unevenly distributed excitation frequencies in a range of (0,24] Hz and ten lowest eigenvectors, that represent dynamic behaviour of a given structure in that range.
The original code was written as a part of Sami Saarinen's Master Thesis on "A Numerical Study of the Harmonical Vibrations on the Off-Machine Coater".

GRSOS

Program is a restricted version of the solid-on-solid (SOS) growth model. It is basically an algorithm that models the deposition of particles on a substrate. The restriction is that the deposition occurs only of the difference in height between the site and its nearest neighbors is less than one. The SOS model itself does not allow overhangs or bubbles. The model can be used to study the temporal development of growth processes.
The problem size is set to 1000 and program makes 5 iterations through the Monte-Carlo loop.
GRSOS was developed by Tuomo Ala-Nissila at the Helsinki University of Technology while visiting the Brown University.

INVPOW93

Program extracts 4 lowest eigenvalues and eigenvectors of a given symmetric 605x605 matrix. It uses Inverse Power method with Deflation after each converged (tolerance is 1.0E-6) eigenvalue. The Deflation process is carried out by using temporary disk storage and contains relatively large amount of sequential binary I/O.
INVPOW93 was developed by Sami Saarinen at the CSC.

MOPAC

Program is a general purpose semi-empirical molecular orbital package for the study of chemical structures and reactions.

In the test case the thermodynamic quantities of anisole (internal energy, heat capacity, partition function, and entropy) are calculated by using the AM1-method for translation rotation and vibrational degrees of freedom for the default temperature range 200-400 K.
Version of the program is 5.0. MOPAC was developed by James J. P. Stewart at the US Air Force Academy.
The input file was prepared by Raimo Uusvuori at the CSC.

NASKER — Program executes 7 program kernels used frequently in NASA and other scientific computing applications. These kernel operations are: MXM to perform matrix product on two input matrices, CFFT2D to perform radix 2 FFT on a two dimensional input array, CHOLSKY to perform a Cholesky decomposition in parallel on a set of input matrices, BTRIX to perform a block tridiagonal matrix solution, GMTRY to set up arrays for a vortex method solution and to perform Gaussian elimination on the resulting array, EMIT to create new vortices according to certain boundary conditions, and VPENTA to simultaneously invert three matrix pentadiagonals in a highly parallel fashion.
NASKER was collected from other NASA programs by David Bailey and John Barton at the NASA Ames Research Center.

NBODYOPT — This program computes the relative locations of ten planets by integrating equations of motions with the 7th order Adams-Stoermer -difference method. Program contains a lot of trigonometric and other intrinsic function calls.
The original code was written in Algol and converted to Fortran-77 by Hannu Karttunen at the CSC. The current version contains some obvious code optimization constructs that may remove unnecessary cache memory conflicts.

RIEMANN — Program calculates period matrices of real algebraic curves. Given a set of generators and fixed points of the generators, the algorithm calculates the period matrices using a partial sum approximation (using elements of the group up to a given word length). The algorithm is recursive and uses complex-number arithmetic programmed in C. Numerical under and overflow situations are not checked for, so one should use normalized matrices as input and limit the word length to a small enough number.
RIEMANN was developed and input file prepared by Juha Haataja at the CSC.

SIMCZO — Program is a development version of a numerical simulation for the Czochralski crystal growth. Based on the Finite Element Method, program simulates the crystal growth in the axisymmetric case including the free boundaries between the solid and liquid and liquid and gas, respectively. The fluid flow in the liquid is governed by the coupled Navier-Stokes and diffusion-convection equations. The temperature distribution in the crystal is governed by the diffusion-convection equation. SIMCZO was developed and input file prepared by Jari Jarvinen at the CSC.

WHY12M — Program solves a system of linear equations by Gaussian elimination using certain sparse matrix techniques. The program is designed to solve efficiently problems which contain only one system with a single right hand side.
The order of the matrix is set to 4500 and the population of non-zero elements is less than 1%. Program writes a checkpoint-file to make an efficient restarts possible.
The original code (Y12M) was developed at the Regional Computing Center at the University of Copenhagen.
The input file was created by Sami Saarinen at the CSC.

MD1

This Molecular dynamics program essentially involves solution of the equations of motion for a system of a large number of interacting particles. The problem is solved numerically by calculating approximate solutions of the second order differential equations at a large number of time steps in a given interval 'dt'. The particles are considered to interact through an effective pair potential which is adjusted to reproduce experimental results and is used to model the exact many-body potential. For monatomic neutral atoms the form of pair potential most often employed is the Lennard-Jones potential.
A typical MD calculation for N particles involves for each time step calculation of the forces on the N particles and their new positions, and calculation of energies and radial distribution functions.
In this test the number of atoms is 4 times NC cubed, where NC is set to 11 in a non-distibuted memory version, thus simulation is carried out with 5324 atoms. The number of time steps is set to 600.
MD1 was developed by Mark Pinches at the University of Southampton, UK. Sequential and distributed memory versions are part of the European parallel benchmarking effort, Genesis.

PDE1

Program solves the Poisson-Equation on a 3-dimensional grid by parallel red-black relaxation. It is an extreme example of the class of PDE solvers, as due to the simplicity of the discretization of Poisson's equation the number of floating point operations per gridpoint is quite small relative to more complex PDEs. The ratio of computation to communication is thus rather low.
The parallelization is performed by grid splitting. A part of the computational grid is assigned to each processor. After each computational step, values at the boundary of the subgrids are exchanged with nearest neighbours.
The sequential version automatically produces results for a range of problem sizes. The problem size is determined by the grid size, which is related to the parameter N. The number of grid points in each direction is 2**N + 1.
For the non-distibuted memory version N is set to 7 and standard implementation of the red-black relaxation is used.
PDE1 was developed by J. Klose and M. Lemke at the PALLAS GmbH. Sequential and distributed memory versions are part of the European parallel benchmarking effort, Genesis.

QCD1

Program is based on a 'pure gluon' SU(3) lattice gauge theory simulation, using the Monte-Carlo heatbath technique. It uses the 'quenched' approximation which neglects dynamical fermions. The simulation is defined on a four-dimensional lattice which is a discrete approximations to continuum space-time. The basic variables are 3 by 3 complex matrices. Four such matrices are associated with every lattice site. The lattice update is performed using a multi-hit Metropolis algorithm.
In the parallel version of the program, the lattice can be distributed in any one or more of the four lattice directions. The lattice size is based on a 4-dimensional space-time lattice of size: N = NS **3 * NT, where NT & NS are even integers.
For the non-distibuted memory version NT & NS are set to 8 and start configuration is disordered.
QCD1 was developed by Eckardt Kehl at PALLAS GmbH. Sequential and distributed memory versions are part of the European parallel benchmarking effort, Genesis.

```
Appendix E

PARMACS macro calls used in this benchmarks:
[Ref: Parallel Computing 15 (1990) 199-132, North-Holland]

ENVHOST     Declaration macros for host and node program. Put before the
ENVNODE     first data or executable statement in Fortran-77 source.

INITHOST    The first executable statement in host and node main program.
INITNODE

TORUS(nx,ny,nz,slave,process_file)
            Sets up a (nx,ny,nz) grid of (logical) processors and defines
            the executable to be named 'slave'. Information to be used by
            'REMOTE_CREATE' is passed to the temporary file 'process_file'
            (not used by the PVM-interface).

REMOTE_CREATE(process_file,process_id_vector)
            A macro for process creation. Read information from
            'process_file' that was written by TORUS-macro. In return
            'process_id_vector' contains instance numbers (thread/process
            id's) of just created 'slave's.

MYPROC      A variable to identify own process id.
HOSTID      A variable to identify host process id.

SEND(target_id,buffer,buffer_len,msgtype)
            Asyncroniously send 'buffer_len' amount of bytes pointed by
            'buffer' to the process id 'target_id'. The message
            identification type is 'msgtype'.
            (Note: My PVM-interface encodes the 'msgtype' to be internally
                   (during send) 'msgtype * MAXNODES + target_id'.
                   This made the PARMACS RECV easier to implement)

SENDR(target_id,buffer,buffer_len,msgtype)
            Same as SEND(...), but syncronious. (PVM-interface: SEND).

RECV(buffer,buffer_len,actual_len,sender,msgtype,condition)
            Receives 'buffer' of data which length cannot exceed
            'buffer_len' bytes. The actual length will be stored in
            'actual_len'. The sender instance was 'sender' and the message
            type was actually 'msgtype'. Receive is made after 'condition'
            is fulfilled:

              o MATCH_ID(select_sender) : Receives any message that comes
                                          next from the selected sender.
              o MATCH_TYPE(select_type) : Receives any message that has
                                          the selected type.
              o MATCH_ID_AND_TYPE(select_sender,select_type) : Both of the
                                          previous conditions must be true.
              o MATCH_ANY : Receive any message that arrives.

             (PVM-interface: RECV polls receive buffer until respective
              UDP-message that fulfills the 'condition' has been arrived)

RECVR(buffer,buffer_len,actual_len,sender,msgtype,condition)
            Same as RECV(...), but syncronious. (PVM-interface: RECV).

CLEAN_UP    This macro MUST BE the last statement to be executed before
            leaving the host or node program.
            (PVM-interface: Nodes stay in 'barrier()' until all nodes have
                            reached the barrier. When host exits, it tries
                            to 'terminate()' possible hanging nodes.)
```