PUBLIC INTERNATIONAL BENCHMARKS FOR PARALLEL COMPUTERS

PARKBENCH Committee: Report-1

assembled by Roger Hockney (chairman) and
Michael Berry (secretary)

Computer Science Department
University of Tennessee

CS-93-213 November 1993

Public International Benchmarks
for Parallel Computers

PARKBENCH Committee: Report-1
assembled by Roger Hockney (chairman) and Michael Berry (secretary)

November 17, 1993

Contents

Chapter 1

Introduction*

1.1 Background and Objectives

The PARKBENCH (PARallel Kernels and BENCHmarks) committee, originally called the Parallel Benchmark Working Group, PBWG, was founded at Supercomputing'92 in Minneapolis, when a group of about 50 people interested in computer benchmarking met under the joint initiative of Tony Hey (University of Southampton, UK) and Jack Dongarra (University of Tennessee/Oak Ridge National Laboratory). Most of the key players were present, from the Universities, Laboratories and industries, representing both computer manufacturers and computer users from both sides of the Atlantic. Roger Hockney (University of Southampton) chaired the meeting, and the objectives of the group were:

1. To establish a comprehensive set of parallel benchmarks that is generally accepted by both users and vendors of parallel system.
2. To provide a focus for parallel benchmark activities and avoid unnecessary duplication of effort and proliferation of benchmarks.
3. To set standards for benchmarking methodology and result-reporting together with a control database/repository for both the benchmarks and the results.
4. To make the benchmarks and results freely available in the public domain.

The first year's work was to produce a report and an initial set of benchmarks for release at Supercomputing'93 in Portland, Oregon, November 1993. The committee has met at the University of Tennessee Knoxville on March 1–2, 1993, May 24, 1993 and August 23, 1993 to discuss the evolving draft of this report. The document reproduced here is the final result of these meetings, and is the first official publication of the PARKBENCH committee. It was distributed at a public *Birds of a Feather* meeting at Supercomputing'93, Portland, on 17th November 1993, together with the first release of the PARKBENCH parallel benchmarks.

The initial focus of the parallel benchmarks is on the new generation of scalable distributed-memory message-passing architectures for which there is a notable lack of existing benchmarks. For this reason the initial benchmark release concentrates on Fortran77 message-passing codes using the widely available PVM [1] message passing interface for portability. Future versions will undoubtedly adopt the proposed MPI [2] interface, when this is fully defined and becomes generally accepted. The committee's aim, however, is to cover all parallel architectures, and this is expected to be achieved by producing versions of the benchmark codes using Fortran90 and High Performance Fortran (HPF). Many shared-memory architectures provide efficient native implementations of PVM message-passing and are planning HPF compilers. They will be covered by these routes.

1.2 Procedures

The PARKBENCH committee has agreed to divide its work between five subcommittees, corresponding to the five substantive chapters in the report, each with a leader (shown in parentheses) who is responsible for assembling the contents of his chapter and its benchmarks for the committee's approval.

1. Methodology (David Bailey):
2. Low-level benchmarks (Roger Hockney):
3. Kernel benchmarks (Tony Hey):
4. Compact applications (David Walker):
5. Compiler benchmarks (Tom Haupt):

In order to facilitate discussion and exchange of information, the following e-mail addresses were set up.

1. `pbwg-comm@cs.utk.edu` for the Whole committee
2. `pbwg-method@cs.utk.edu` for the Methodology subcommittee
3. `pbwg-lowlevel@cs.utk.edu` for the Low level subcommittee
4. `pbwg-kernel@cs.utk.edu` for the Kernel subcommittee
5. `pbwg-compactapp@cs.utk.edu` for the Compact applications subcommittee

Recent practice, however, has been to send all mail to pbwg-comm so that all members may see it. All mail is being collected and can be retrieved by sending email to `netlib@ornl.gov` and in the mail message typing:

1. send comm.archive from pbwg
2. send lowlevel.archive from pbwg
3. send compactapp.archive from pbwg
4. send method.archive from pbwg
5. send kernel.archive from pbwg
6. send index from pbwg

We have setup a mail reflector for correspondence, it is called `pbwg-comm@cs.utk.edu`. Mail to that address will be sent to the mailing list and also collected in `netlib@ornl.gov`. All PARKBENCH correspondence and benchmarks may be retrieved via anonymous ftp to `netlib2.cs.utk.edu`. Alternatively, one can collect PARKBENCH mail by sending email to `netlib@ornl.gov` and in the mail message type:

---

[*] written by Roger Hockney for whole committee

send comm.archive from pbwg

The PARKBENCH committee is open without charge to anyone interested in computer benchmarking and operates similarly to the HPFF (High Performance Fortran Forum). Anyone interested in joining in the discussion or preparing benchmarks should send e-mail to that effect to:

`dongarra@cs.utk.edu`

PARKBENCH is operating in a very tight budget (in reality, it has no budget at all). Several participants are supported by the companies they represent, and support for several European participants is being provided by ESPRIT.

1.3 Vendor's Commitment

The PARKBENCH committee is anxious that its parallel benchmarks do not put undue demands on computer vendors by way of man power and resources, in a way that would prejudice the wide acceptance and use of the benchmarks. Initially it is felt reasonable to expect that most vendors should have little difficulty in running the low-level and kernel benchmarks, since these either involve basic hardware and software tests (such as COMMS1) that vendors would wish to perform in any case, or involve scientific library subroutines (such as FFT) that they would be required to produce and optimise. In the latter case, they would no doubt be pleased to show the superior performance of their library routine compared with that of the standard Fortran provided in the PARKBENCH benchmark suite.

The case of compact applications is more difficult, because these codes might require substantial effort to optimise, and in some cases even to run satisfactorily. For these reasons, it is not expected that vendors would initially run all these codes, or indeed any of them. They might, however, choose to run a selection of them from subject areas of interest to their current potential customers, in order to demonstrate their computer's capability on some standard and relevant tests. In this way, and over a period of time, it is hoped that most of the compact applications would be run in a natural way and without extra effort.

Chapter 2

Methodology*

2.1 Philosophy

One might ask why anyone should care about developing a standardized, rigorous and scientifically tenable methodology for studying the performance of high-performance computer systems. There are several reasons why this is an important undertaking:

1. To establish and maintain high standards of honesty and integrity in our profession.
2. To improve the status of supercomputer performance analysis as a rigorous scientific discipline.
3. To reduce confusion in the high-performance computing literature.
4. To increase understanding of these systems, both at a low-level hardware or software level and at a high-level, total system performance level.
5. To assist the purchasers of high-performance computing equipment in selecting systems best suited to their needs.
6. To reduce the amount of time and resources vendors must expend in implementing multiple, redundant benchmarks.
7. To provide valuable feedback to vendors on bottlenecks that can be alleviated in future products.

It is important to note that researchers in many scientific disciplines have found it necessary to establish and refine standards for performing experiments and reporting the results. Many scientists have learned the importance of standard terminology and notation. Chemists, physicists and biologists long ago discovered the importance of *controls* in their experiments. The issue of repeatability proved crucial in the recent *cold fusion* episode. Medical researchers have found it necessary to perform *double-blind* experiments in their field. Psychologists and sociologists have developed highly refined experimental methodologies and advanced data analysis techniques. Political scientists have found that subtle differences in the phrasing of a question can affect the results of a poll. Researchers in many fields have found that environmental factors in their experiments can significantly influence the measured results; thus they must carefully report all such factors in their papers.

---

*assembled by David Bailey for methodology subcommittee

If supercomputer performance analysis and benchmarking is ever to be taken seriously as a scientific discipline, certainly its practitioners should be expected to adhere to standards that prevail in other disciplines. This document is dedicated to promoting these standards in our field.

2.2 Fundamental Metrics

The conclusions drawn from a benchmark study of computer performance depend not only on the basic timing results obtained, but also on the way these are interpreted and converted into performance figures. The choice of the performance metric, may itself influence the conclusions. For example, do we want the computer that generates the most megaflop per second (or has the highest Speedup), or the computer that solves the problem in the least time? It is now well known that high values of the first metrics do not necessarily imply the second property. This confusion can be avoided by choosing a more suitable metric that reflects solution time directly, for example either the Temporal, Simulation or Benchmark performance, defined below. This issue of the sensible choice of performance metric is becoming increasing important with the advent of massively parallel computers which have the potential of very high megaflop rates, but have much more limited potential for reducing solution time.

2.3 Time Measurement

Before other issues can be considered, we must discuss the measurement of run time. In recent years a consensus has been reached among many scientists in the field that the most relevant measure of run time is actual wall-clock elapsed time. This measure of time will be required for all PARKBENCH results that are posted to the database.

Elapsed wall-clock time means the time that would be measured on an external clock that records the time-of-day or even Coordinated Universal Time (UTC), between the start and finish of the benchmark. We are not concerned with the origin of the time measurement, since we are taking a difference, but it is important that the time measured would be the same as that given by a difference between two measurements of UTC, if it were possible to make them. It is important to be clear about this, because many computer clocks (e.g., Sun Unix function ETIME) measure elapsed CPU time, which is the total time that the process or job which calls it has been executing in the CPU. Such a clock does not record time (i.e. it stops ticking) when the job is swapped out of the CPU. It does not record, therefore, any wait time which must be included if we are to assess correctly the performance of a parallel program. On some systems, scientists have found that even for programs that perform no explicit I/O, considerable *system* time is nonetheless involved, for example in fetching certain library routines or other data.

Only timings actually measured may be cited for PARKBENCH benchmarks (and we strongly recommend this practice for other benchmarks as well). Extrapolations and projections, for instance to a larger number of nodes, may not be employed for any reason. Also, in the interests of repeatability it is highly recommended that timing runs be repeated, several times if possible.

Two low-level benchmarks are provided in the PARKBENCH suite to test the precision and accuracy of the clock that is to be used in the benchmarking. These should be run first, before any benchmark measurements are made. They are:

1. TICK1 - measures the precision of the clock by measuring the time interval between ticks of the clock. A clock is said to tick when it changes its value.

2. TICK2 - measures the accuracy of the clock by comparing a given time interval measured by an external wall-clock (the benchmarker's wrist watch is adequate) with the same interval measured by the computer clock. This tests the scale factor used to convert computer clock ticks to seconds, and immediately detects if a CPU-clock is incorrectly being used.

The fundamental measurement made in any benchmark is the elapsed wall-clock time to complete some specified task. All other performance figures are derived from this basic timing measurement. The benchmark time, $T(N;p)$, will be a function of the problem size, N, and the number of processors, p. Here, the problem size is 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, and reducing the size parameters to a single size factor (usually called α).

We believe that it is most important to regard execution time and performance as a function of at least the two variables (N,p), which define a parameter plane. Much confusion has arisen in the past by attempts to treat performance as a function of a single variable, by taking a particular path through this plane, and not stating what path is taken. Many different paths may be taken, and hence many different conclusions can be drawn. It is important, therefore, always to define the path through the performance plane, or better as we do here, to study the shape of the two-dimensional performance hill. In some cases there may even be an optimum path up this hill. The following discussion of units and metrics is based on that of [11].

2.4 Units and Symbols

A rational set of units and symbols is essential for any numerate science including benchmarking. The following extension of the internationally agreed SI system of physical units [3] is made to accommodate the needs of computer benchmarking.

The value of a variable comprises a pure number stating the number of units which equal the value of the variable, followed by a unit symbol specifying the unit in which the variable is being measured. A new unit is required whenever a quantity of a new nature arises, such as the first appearance of vector operations, or message sends. Generally speaking a unit symbol should be as short as possible, consistent with being easily recognised and not already used. The following have been found necessary in the characterisation of computer and benchmark performance in science and engineering. No doubt more will have to be defined as benchmarking enters new areas.

New unit symbols and their meaning:

1. flop: floating-point operation [latex \flop]
2. inst: instruction of any kind [latex \inst]
3. intop: integer operation [latex \inop]
4. vecop: vector operation [latex \vecop]
5. send: message send operation [latex \send]
6. iter: iteration of loop [latex \iter]
7. mref: memory reference (read or write) [latex \mref]

8. barr: barrier operation [latex \barr]
9. b: binary digit (bit) [latex \bit]
10. B: byte (groups of 8 bits) [latex \B]
11. sol: solution or single execution of benchmark [latex \sol]
12. w: computer word. Symbol is lower case (W means watt) [latex \w]

When required a subscript may be used to show the number of bits involved in the unit. For example: a 32-bit floating-point operation flop_{32}, a 64-bit word w_{64}, also we have $\mathrm{b} = \mathrm{w}_1$, $\mathrm{B} = \mathrm{w}_8$, $\mathrm{w}_{64} = 8\mathrm{B}$.

Note that flop, mref and other multi-letter symbols are inseparable four or five-letter symbols. The character case is significant in all unit symbols so that e.g. Flop, Mref, W_{64} are incorrect. Unit symbols should always be printed in roman type, to contrast with variables names which are printed in italic. To aid in the use of roman type, especially within LaTex's math mode, LaTex commands have been defined for each unit, these commands being a backslash followed by the unit symbol (except for 'intop' and 'b' whose names are changed in the command to avoid a clash with already defined system commands). Such commands will print in roman type wherever they occur. Because 's' is the SI unit for seconds, unit symbols like 'sheep' do not take 's' in the plural. Thus one counts: one flop, two flop, ..., one hundred flop etc. This is especially important when the unit symbol is used in ordinary text as a useful abbreviation, as often, quite sensibly, it is.

SI provides the standard prefixes:

1. k : kilo meaning 10^3
2. M : mega meaning 10^6
3. G : giga meaning 10^9
4. T : tera meaning 10^{12}

This means that we cannot use M to mean 1024^2 (the binary mega) as is often done in describing computer memory capacity, e.g. 256 MB. We can however introduce the new prefix:

1. K : meaning 1024, then use a subscript 2 to indicate the binary versions
2. M_2 : binary mega 1024^2
3. G_2 : binary giga 1024^3
4. T_2 : binary tera 1024^4

In most cases the difference between the mega and the binary mega (4%) is probably unimportant, but it is important to be unambiguous. In this way one can continue with existing practice if the difference doesn't matter, and have an agreed method of being more exact when necessary. For example, the above memory capacity was probably intended to mean $256\mathrm{M}_2\mathrm{B}$.

As a consequence of the above, an amount of computational work involving 4.5×10^{12} floating-point operations is correctly written as 4.5 Tflop. Note that the unit symbol Tflop is never pluralised with an added 's', and it is therefore incorrect to write the above as 4.5 Tflops which could be confused with a rate per second. The most frequently used unit of performance, millions of floating-point operations per second is correctly written Mflop/s, in analogy to km/s. The slash is necessary and means 'per', because the 'p' is an integral part of the unit symbol 'flop' and cannot also be used to mean 'per'.

2.5 Floating-Point Operation Count

Although we discourage the use of millions of floating-point operations per second as a performance metric, it can be a useful measure if the number of floating-point operations, $F(N)$, needed to solve the benchmark problem is carefully defined.

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 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. The latter is the procedure followed by the PERFECT Club [4]. 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 [5] 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 (this figure will be adjusted)
IF(X .REL. Y)	1 flop

Some members of the committee felt that these numbers, derived in the 1970s, no longer correctly reflected the situation on current computers. However, since these numbers are only used to calculate a nominal benchmark flop-count, it is not so important that they be accurate. The important thing is that they do not change, otherwise all previous flop-counts would have to be renormalised. In any case, it is not possible for a single set of ratios to be valid for all computers and library software. I (rwh) suggest the committee stays with the above ratios until such time as they become wildly wrong and extensive research provides us with a more realistic set.

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 Fortran77 sequential code. The other is the actual number of floating-point operations performed by the hardware when executing the distributed multi-node version, $F_H(N,p)$, which may be greater than the nominal benchmark count, due to the distributed version performing redundant arithmetic operations. Because of this, the hardware flop count may also depend on the number of processors on which the benchmark is run, as shown in its argument list.

2.6 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. Much confusion and wasted work can arise from optimising a benchmark with respect to an inappropriate metric. The principal performance metrics are discussed in the following subsections.

2.6.1 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

$$R_T(N;p) = T^{-1}(N;p) \tag{2.1}$$

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. For this reason the temporal performance is also the metric that a computer user should employ to select the best algorithm to solve his problem, because his objective is also to solve the problem in the least time, and he does not care how much arithmetic is done to achieve this.

2.6.2 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 step). In order to maintain numerical stability, explicit schemes also require the use of a smaller timestep as the spatial grid is made finer. For such schemes the simulation performance falls off dramatically as the problem size is increased by introducing more mesh points in order to refine the spatial resolution: the doubling of the number of mesh-points in each of three dimensions can reduce the simulation performance by a factor near 16 because the timestep must also be approximately halved. Even though the larger problem will generate more Megaflop per second, in forecasting, it is the simulated days per day (i.e. the simulation performance) and not the Mflop/s, that matter to the user.

As we see below, benchmark performance is also measured in terms of the amount of arithmetic performed per second or Mflop/s. However it is important to realise that it is incorrect to compare the Mflop/s achieved by two algorithms and to conclude that the algorithm with the highest Mflop/s rating is the best algorithm. This is because the two algorithms may be performing quite different amounts of arithmetic during the solution of the same problem. The temporal performance metric, R_T, defined above, has been introduced to overcome this problem, and provide a measure that can be used to compare different algorithms for solving the same problem. However, it should be remembered that the temporal performance only has the same meaning within the confines of a fixed problem, and no meaning can be attached to a comparison of the temporal performance on one problem with the temporal performance on another.

2.6.3 Benchmark Performance

In order to compare the performance of a computer on one benchmark with its performance on 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, $F_B(N)$, we can define the *Benchmark Performance* as

$$R_B(N;p) = F_B(N)/T(N;p) \tag{2.2}$$

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 to emphasise that the values are based on the nominal benchmark flop-count. In other contexts such performance figures would probably be quoted as examples of the so-called *sustained* performance of a computer. We feel that the use of this term is meaningless unless the problem being solved and the degree of code optimisation is quoted, because the performance is so varied across different benchmarks and different levels of optimisation. Hence we favour the quotation of a selection of benchmark performance figures, rather than a single sustained performance, because the latter implies that the quoted performance is maintained over all problems.

Note also that the flop-count $F_B(N)$ is that for the defining sequential version of the benchmark, and that the same count is used to calculate R_B for the distributed-memory (DM) version of the program, even though the DM version may actually perform a different number of operations. It is usual for DM programs to perform more arithmetic than the defining sequential version, because often numbers are recomputed on the nodes in order to save communicating their values from a master processor. However such calculations are redundant (they have already been performed on the master) and it would be incorrect to credit them to the flop-count of the distributed program.

Using the sequential flop-count in the calculation of the DM programs benchmark performance has the additional advantage that it is possible to conclude that, for a given benchmark, the implementation that has the highest benchmark performance is the best because it executes in the least time. This would not necessarily be the case if a different $F_B(N)$ were used for different implementations of the benchmark. For example, the use of a better algorithm which obtains the solution with less than $F_B(N)$ operations will show up as higher benchmark performance. For this reason it should cause no surprise if the benchmark performance occasionally exceeds the maximum possible hardware performance. To this extent benchmark performance Mflop/s must be understood to be nominal values, and not necessarily exactly the number of operations executed per second by the hardware, which is the subject of the next metric. The purpose of benchmark performance is to compare different implementations and algorithms on different computers for the solution of the same problem, on the basis that the best performance means the least execution time. For this to be true $F_B(N)$ must be kept the same for all implementations and algorithms.

2.6.4 Hardware Performance

If we wish to compare the observed performance with the theoretical capabilities of the computer hardware, we must compute the actual number of floating-point operations performed, $F_H(N;p)$, and from it the actual *Hardware Performance*

$$R_H(N;p) = F_H(N,p)/T(N;p) \tag{2.3}$$

The hardware performance also has the units Mflop/s, and will have the same value as the benchmark performance for the sequential version of the benchmark. However, the hardware performance may be higher than the benchmark performance for the distributed version, because the hardware performance gives credit for redundant arithmetic operations, whereas the benchmark performance does not. Because the hardware performance measures the actual floating-point operations performed per second, unlike the benchmark performance, it can never exceed the theoretical peak performance of the computer.

Assuming a computer with multiple-CPUs each with multiple arithmetic pipelines, delivering a maximum of one flop per clock period, the theoretical peak value of hardware performance is

$$r^* = \frac{fl.pt.pipes/CPU}{clock.period} \times number.CPUs \tag{2.4}$$

with units of Mflop/s if the clock period is expressed in microseconds. By comparing the measured hardware performance, $R_H(N;p)$, with the theoretical peak performance, we can assess the fraction of the available performance that is being realised by a particular implementation of the benchmark.

2.6.5 Speedup, Efficiency and Performance per Node

Parallel speedup is a popular metric that has been used for many years in the study of parallel computer performance. However, its definition is open to ambiguity and misuse because it always begs the question "speedup over what?"

Speedup is usually defined as

$$\frac{T_1}{T_p} \tag{2.5}$$

where T_p is the p-processor time to perform some benchmark, and T_1 is the one-processor time. There is no doubt about the meaning of T_p — this is the measured time $T(N;p)$ to perform the benchmark. There is often considerable dispute over the meaning of T_1: should it be the time for the parallel code running on one processor, which probably contains unnecessary parallel overhead, or should it be the best serial code (possibly using a different algorithm) running on one processor? Many scientists feel the latter is a more responsible choice, but this requires research to determine the best practical serial algorithm for the given application. If at a later time a better algorithm is found, current speedup figures might be considered obsolete. An additional difficulty with this definition is that even if a meaning for T_1 is agreed to, there may be insufficient memory on a single node to store an entire large problem. Thus in many cases it may be impossible to measure T_1 using this definition.

One principal objective in the field of performance analysis is benchmarking: to compare the performance of different computers. It is generally agreed that the best performance corresponds to the least wall-clock execution time. In order to adapt the speedup statistic for benchmarking, it is thus necessary to define a single reference value of T_1 to be used for all calculations. It does not matter how T_1 is defined, or what its value is, only that the same value of T_1 is used to calculate all speedup values used in the comparison.

However, defining T_1 as a reference time unrelated to the parallel computer being benchmarked unfortunately has the consequence that many properties that many people regard as essential to the concept of parallel speedup are lost:

1. It is no longer necessarily true that the speedup of the parallel code on one processor is unity. It may be, but only by chance.

2. It is no longer true that the maximum speedup using p-processors is p.

3. Because of the last item, efficiency figures computed as speedup divided by p are no longer a meaningful measure of processor utilization.

There are other difficulties with this formulation of speedup. If we use T_1 as the run time on a very fast single processor (currently, say, a Cray C90 or a NEC SX-3), then manufacturers of highly parallel systems will be reluctant to quote the speedup of their system in the above way. For example, if the speedup of a 100 processor parallel system over a single node of the same system is a respectable factor of 80, it is likely that the speedup computed from the *standard* T_1 would be reduced to 10 or less. This is because a fast vector processor is typically at least ten times faster than the RISC processors used in many highly parallel systems of a comparable generation.

Thus it appears that if one sharpens the definition of speedup to make it an acceptable metric for comparing the performance of different computers, one has to throw away the main properties that have made the concept of speedup useful in the past.

Accordingly, the PARKBENCH committee has decided the following:

1. No speedup statistic will be kept in the PARKBENCH database.

2. Speedup statistics based on PARKBENCH benchmarks must never be used as figures of merit when comparing the performance of different systems. We further recommend that speedup figures based on other benchmarks not be used as figures of merit in such comparisons.

3. Speedup statistics may be used in a study of the performance characteristics of an individual parallel system. But the basis for the determination of T_1 must be clearly and explicitly stated.

4. The value of T_1 should be based on an efficient uniprocessor implementation. Code for message passing, synchronization, etc. should not be present. The author should also make a reasonable effort to insure that the algorithm used in the uniprocessor implementation is the best practical serial algorithm for this purpose.

5. Given that a large problem frequently does not fit on a single node, it is permissible to cite speedup statistics based on the timing of a smaller number of nodes. In other words, it is permissible to compute speedup as T_p/T_m, for some m, $1 < m < p$. If this is done, however, this usage must be clearly stated, and full details of the basis of this calculation must be presented. As above, care must be taken to insure that the unit timing T_m is based on an efficient implementation of appropriate algorithms.

2.7 Performance Database

The process of gathering, archiving, and distributing computer benchmark data is a cumbersome task usually performed by computer users and vendors with little coordination. Within Xnetlib [6] there is a mechanism to provide Internet-access to a performance database server (PDS) which can be used to extract current benchmark data and literature. PDS [7] provides an on-line catalog of public-domain computer benchmarks such as the LINPACK Benchmark [8], Perfect Benchmarks [4], and the NAS Parallel Benchmarks [9]. PDS does not reformat or present the benchmark data in any way that conflicts with the original methodology of any particular benchmark; it is thereby devoid of any subjective interpretations of machine performance. PDS is providing a more manageable approach to the development and support of a large dynamic database of published performance metrics.

The PDS system was developed at the University of Tennessee and Oak Ridge National Laboratory and is an initial attempt at performance data management. This on-line database of computer benchmarks is specifically designed to provide easy maintenance, data security, and data integrity in the benchmark information contained in a *dynamic* performance database.

PDS was designed with a simple tabular format that involves displaying the data in rows (machine configuration) and columns (numbers). Graphical representations of tabular data, such as the representation by SPEC [10] with the obsolescent *SPECmarks*, are straightforward.

2.7.1 Design of a Performance Database

Because of the complexity and volume of the data involved in a performance database, it is natural to exploit a database management system (DBMS) to archive and retrieve benchmark data. A DBMS will help not only in managing the data, but also in assuring that the various benchmarks are presented in some reasonable format for users: table or spreadsheet where machines are *rows* and benchmarks are *columns*.

Of major concern is the organization of the data. It seems logical to organize data in the DBMS according to the benchmarks themselves: a LINPACK table, a Perfect table, etc. It would be nearly impossible to *force* these very different presentation formats to conform to a single presentation standard just for the sake of reporting. Individual tables preserve the display characteristics of each benchmark, but the DBMS should allow users to query all tables for various machines. Parsing benchmark data into these tables is straightforward provided a customized parser is available for each benchmark set. In the parsing process, constructing a raw data file and building a standard format ASCII file eases the incorporation of the data into the database.

The functionality required by PDS is not very different from that of a standard database application. The difference lies in the user interface. Financial databases, for example, typically involve specific queries like

$$\text{EXTRACT ROW ACCT\_NO} = \text{R103049}$$

in which data points are usually discrete and the user is very familiar with the data. The user, in this case, knows exactly what account number to extract, and the format of retrieved data in response to queries. With our performance database, however, we would expect the contrary: the user does not really know (*i*) what kind of data is available, (*ii*) how to request/extract the data, and (*iii*) what form to expect the returned data to be in. These assumptions are based on the current lack of coordination in (public-domain) benchmark management. The number of benchmarks in use continues to rise with no standard format for presenting them. The number of performance-literate users is increasing, but not at a rate sufficient to expect proper queries from the performance database. Quite often, users just wish to see the best-performing machines for a particular benchmark. Hence, a simple rank-ordering of the *rows* of machines according to a specific benchmark *column* may be sufficient for a general user.

Finally, the features of the PDS user interface should include

(1) the ability to extract specific machine and benchmark combinations that are of interest,

(2) the ability to search on multiple keywords across the entire dataset, and

(3) the ability to view cross-referenced papers and bibliographic information about the benchmark itself.

We include (3) in the list above to address the concern of proliferating numbers without any benchmark methodology information. PDS would provide abstracts and complete papers related to benchmarks and thereby provide a needed educational resource without risking improper interpretation of retrieved benchmark data.

2.7.2 PDS Features

PDS provides the following retrieval-based functions for the user:

(1) a *browse* feature to allow casual viewing and point-and-click navigation through the database,

(2) a *search* feature to permit multiple keyword searches with Boolean conditions,

(3) a *rank-ordering* feature to sort and display the results for the user, and

(4) a few additional features that aid the user in acquiring benchmark documentation and references.

As discussed in [7], the `Rank Ordering` option in PDS allows the user to view a listing of machines that have been ranked by a particular performance metric such as megaflop/s or elapsed CPU time. Both `Rank Ordering` and `Papers` options are menu-driven data access paths within PDS. With the `Browse` facility in PDS, the user first selects the vendor(s) and benchmark(s) of interest, then selects the large `Process` button to query the performance database. The PDS client then opens a socket connection to the server and, using the query language (rdb), remotely queries the database. The `Search` option in PDS permits user-specified keyword searches over the entire performance database. `Search` utilizes literal case-insensitive matching along with a moderate amount of aliasing. Multiple keywords are permitted, and a Boolean flag is provided for more complicated searches. Using `Search`, the user has the option of entering vendor names, machine aliases, benchmark names, or specific strings, or producing a more complicated Boolean keyword search. Since any retrieved data will be displayed to the screen (by default), the `Save` option allows the user to store any retrieved performance data to an ASCII file. Finally, the `Bibliography` option in PDS provides a list of relevant manuscripts and other information about the benchmarks. Future enhancements to PDS include the use of more sophisticated two-dimensional graphical displays for machine comparisons. Additional serial and parallel benchmarks will be added to the database as formal procedures for data acquisition are determined. The `Browse` and `Search` facilities available in the current version of PDS are illustrated in Appendix B.

2.7.3 PDS Availability

To receive Xnetlib with PDS support for Unix-based machines, send the electronic mail message *send xnetlib.shar from xnetlib* to `netlib@ornl.gov`. You can *unshar* the file and compile it by answering the user-prompted questions upon installation. Use of *shar* will install the full functionality of Xnetlib along with the latest PDS client tool. Questions concerning PDS should be sent to `utpds@cs.utk.edu.` The University of Tennessee and Oak Ridge National Laboratory will be responsible for gathering and archiving additional (published) benchmark data.

At present each benchmark measurement for a particular problem size N and processor number p, is represented by one line in the database with variable length fields chosen by the benchmark writer as suitable and comprehensive to describe the conditions of the benchmark run. The fields separated by a marker include, benchmarkers name and e-mail, computer location and date, hardware specification, compiler date and optimisation level, N, p, $T(N, p)$, $R_B(N, P)$ and other metrics as deemed appropriate by the benchmark writer. Ideally, the line for the database would be produced automatically as output by the benchmark program itself.

2.8 Interactive Graphical Interface

The Southampton Group has agreed to provide an interactive graphical front end to the PARKBENCH database of performance results. To achieve this, the basic data held in the Performance Data Base should be values of $T(N; p)$ for at least 4 values of problem size N, each for sufficient p-values (say 5 to 10) to determine the trend of variation of performance with number of processors for constant problem size. It is important that there be enough p-values to see Amdahl saturation, if present, or any peak in performance followed by degradation. A graphical interface is really essential to allow this multidimensional data to be viewed in any of the metrics defined above, as chosen interactively by the user. The user could also be offered (by suitable interpolation) a display of the results in various scaled metrics, in which the problem size is expanded with the number of processors.

In order to encompass as wide a range of performance and number of processors as possible, a log-scale on both axes is unavoidable, and the format and scale range should be kept fixed as long as possible to enable easy comparison between graphs. A three-cycle by three-cycle log-log graph with range 1 to 1000 in both p and Mflop/s would cover most needs in the immediate future. Examples of such graphs are to be found in [11, 12].

A log/log graph is also desirable because the size and shape of the Amdahl saturation curve is the same wherever it is plotted on such a graph. That is to say there is a universal Amdahl curve that is invariant to its position on any log/log graph. Amdahl saturation is a two-parameter description of any of the performance metrics, R, as a function of p for fixed N, which can be expressed by

$$R = \frac{R_\infty}{(1 + p_{\frac{1}{2}}/p)} \tag{2.6}$$

where R_∞ is the saturation performance approached as $p \rightarrow \infty$ and $p_{\frac{1}{2}}$is the number of processors required to reach half the saturation performance. The graphical interface should allow this universal Amdahl curve to be moved around the graphical display, and be matched against the performance curves. The changing values of the two parameters (R_∞,$p_{\frac{1}{2}}$)should be displayed as the Amdahl curve is moved.

As more experience is gained with performance analysis, that is to say the fitting of performance data to parameterised formulae, it is to be expected that the graphical interface will allow more complicated formulae to be compared with the experimental data, perhaps allowing 3 to 5 parameters in the theoretical formula. But, as yet, we do not know what these for parameterised formula should be.

2.9 Benchmarking Procedure and Code Optimisation

Manufacturers will always feel that any benchmark not tuned specifically by themselves, is an unfair test of their hardware and software. This is inevitable and from their viewpoint it is true. NASA have overcome this problem by only specifying the problems (the NAS paper-and-pencil benchmarks [13]) and leaving the manufacturers to write the code, but in many circumstances this would require unjustifiable effort and take too long. It is also a perfectly valid question to ask how a particular parallel computer will perform on existing parallel code, and that is the viewpoint of PARKBENCH .

The benchmarking procedure is to run the distributed PARKBENCH suite on an *as-is* basis, making only such non-substantive changes that are required to make the code run (e.g. changing the names of header files to a local variant). The as-is run may use the highest level of automatic compiler optimisation that works, but the level used and compiler date should be noted in the appropriate section of the performance database entry.

After completing the as-is run, which gives a base-line result, any form of optimisation may be applied to show the particular computer to its best advantage, up to completely rethinking the algorithm, and rewriting the code. The only requirement on the benchmarker is to state what has been done. However, remember that, even if the algorithm is changed, the official flop-count, $F_B(N)$ that is used in the calculation of nominal benchmark Mflop/s, $R_B(N, p)$, does not. In this way a better algorithm will show up with a higher R_B, as we would want it to, even though the hardware Mflop/s is likely to be little changed.

Typical steps in optimisation might be:

1. explore the effect of different compiler optimisations on a single processor, and choose the best for the as-is run.
2. perform the as-is run on multiple processors, using enough values of p to determine any peak in performance or saturation.
3. return to single processor and optimise code for vectorisation, if a vector processor is being used. This means restructuring loops to permit vectorisation.
4. continue by replacement of selected loops with optimal assembly coded library routines (e.g. BLAS [14] where appropriate).
5. replacement of whole benchmark by a tuned library routine with the same functionality.
6. replace whole benchmark with locally written version with the same functionality but using possibly an entirely different algorithm that is more suited to the architecture.

Chapter 3

Low-Level Benchmarks*

3.1 Introduction

The first step in the assessment of the performance of a massively parallel computer system is to measure the performance of a single processing node of the multi-node system. There exist already many good and well-established benchmarks for this purpose, notably the LINPACK benchmarks and the Livermore Loops. These are not part of the PARKBENCH suite of programs, but PARKBENCH recommends that these be used to measure single-node performance, in addition to some specific low-level measurements of its own (see Section 3.2). There follows a brief description of existing benchmarks that are recommended for measuring single-node performance, with a discussion of their value.

3.1.1 Most Reported Benchmark: LINPACKD (n=100)

This well-known standard benchmark is a Fortran program for the solution of (100×100) dense set of linear equations by Gaussian elimination. It is distributed by Jack Dongarra of the University of Tennessee [8]. The results are quoted in Mflop/s and are regularly published and available by electronic mail. The main value of this benchmark is that results are known for more computers than any other benchmark. Most of the compute time is contained in vectorisable DO-loops such as the DAXPY (scalar times vector plus vector) and inner product. Therefore one expects vector computers to perform well on this benchmark. The weakness of the benchmark is that it tests only a small number of vector operations, but it does include the effect of memory access and it is solving a complete (although small) real problem.

3.1.2 Performance Range: The Livermore Loops

These are a set of 24 Fortran DO-loops (The Livermore Fortran Kernels, LFK) extracted from operational codes used at the Lawrence Livermore National Laboratory [5]. They have been used since the early seventies to assess the arithmetic performance of computers and their compilers. They are a mixture of vectorisable and non-vectorisable loops and test rather fully the computational capabilities of the hardware, and the skill of the software in compiling efficient code, and in vectorisation. The main value of the benchmark is the range of performance that it demonstrates, and in this respect it complements the limited range of loops tested in the LINPACK benchmark. The benchmark provides the individual performance of each loop, together with various averages (arithmetic, geometric, harmonic) and the quartiles of the distribution. However, it is difficult to give a clear meaning to these averages, and the value of the benchmark is more in the distribution itself. In particular, the maximum and minimum give the range of likely performance in full applications. The ratio of maximum to minimum performance has been called the *instability* or the *speciality* [15], and is a measure of how difficult it is to obtain good performance from the computer, and therefore how specialised it is. The minimum or worst performance obtained on these loops is of special value, because there is much truth in the saying that "the best computer to choose is that with the best worst-performance".

3.2 Single-Processor Benchmarks

The single-processor low-level benchmarks provided by PARKBENCH , aim to measure performance parameters that characterise the basic architecture of the computer, and the compiler software through which it is used. For this reason, such benchmarks have also been called appropriately *basic architectural benchmarks*. Following the methodology of Euroben [16], the aim is that these hardware/compiler parameters will be used in performance formulae that predict the timing and performance of the more complex kernels (see Chapter 4) and compact applications (see Chapter 5). They are therefore a set of *synthetic* benchmarks contrived to measure theoretical parameters that describe the severity of some overhead or potential bottleneck, or the properties of some item of hardware. Thus RINF1 characterises the basic properties of the arithmetic pipelines by measuring the parameter $(r_\infty, n_{\frac{1}{2}})$, and POLY1 and POLY2 characterise the severity of the memory bottleneck by measuring the parameters $(\hat{r}, f_{\frac{1}{2}})$.

The fundamental measurement in any benchmarking is the measurement of elapsed wall-clock time. Because the computer clocks on each node of a multi-node MPP are not synchronised, all benchmark time measurements must be made with a single clock on one node of the system. The benchmarks TICK1 and TICK2 have, respectively, been designed to measure the resolution and to check the absolute value of this clock. These benchmarks should be run with satisfactory results before any further benchmark measurements are made.

3.2.1 Timer resolution: TICK1

TICK1 measures the interval between ticks of the clock being used in the benchmark measurements. That is to say the resolution of the clock. A succession of calls to the timer routine are inserted in a loop and executed many times. The differences between successive values given by the timer are then examined. If the changes in the clock value (or ticks) occur less frequently than the time taken to enter and leave the timer routine, then most of these differences will be zero. When a tick takes place, however, a difference equal to the tick value will be recorded, surrounded by many zero differences. This is the case with clocks of poor resolution; for example most UNIX clocks that tick typically every 10 ms. Such poor UNIX clocks can still be used for low-level benchmark measurements if the benchmark is repeated, say, 10,000 times, and the timer calls are made outside this repeat loop.

With some computers, such as the CRAY series, the clock ticks every cycle of the computer, that is to say every 6ns on the Y-MP. The resolution of the CRAY clock is therefore approximately one million times better than a UNIX clock, and that is quite a difference! If TICK1 is used on such a computer the difference between successive values of the timer is a very accurate measure of how long it takes to execute the instructions of the timer routine, and therefore is never zero. TICK1 takes the minimum of all such differences, and all it is

---

* assembled by Roger Hockney for Low-Level subcommittee

possible to say is that the clock tick is less than or equal to this value. Typically this minimum will be several hundreds of clock ticks. With a clock ticking every computer cycle, we can make low-level benchmark measurements without a repeat loop. Such measurements can even be made on a busy timeshared system (where many users are contending for memory access) by taking the minumum time recorded from a sample of, say, 10,000 single execution measurements. In this case, the minimum can usually be said to apply to a case when there was no memory access delay caused by other users.

TICK1 exists and forms part of the Genesis benchmarks [17].

3.2.2 Timer value: TICK2

TICK2 confirms that the absolute values returned by the computer clock are correct, by comparing its measurement of a given time interval with that of an external wall-clock (actually the benchmarker's wristwatch). Parallel benchmark performance can only be measured using the elapsed wall-clock time, because the objective of parallel execution is to reduce this time. Measurements made with a CPU-timer (which only records time when its job is executing in the CPU) are clearly incorrect, because the clock does not record waiting time when the job is out of the CPU. TICK2 will immediately detect the incorrect use of a CPU-time-for-this-job-only clock. An example of a timer that claims to measure elapsed time but is actually a CPU-timer, is the returned value of the popular Sun UNIX timer ETIME. TICK2 also checks that the correct multiplier is being used in the computer system software to convert clock ticks to true seconds.

TICK2 exists and forms part of release 2.2 and later of the Genesis benchmarks [17].

3.2.3 Basic Arithmetic Operations: RINF1

This benchmark takes a set of common Fortran DO-loops and analyses their time of execution in terms of the two parameters (r_∞,$n_{\frac{1}{2}}$)[18, 19, 20, 21, 22, 23]. r_∞ is the asymptotic performance rate in Mflop/s which is approached as the loop (or vector) length, n, becomes longer. $n_{\frac{1}{2}}$(the half-performance length) expresses how rapidly, in terms increasing vector length, the actual performance, r, approaches r_∞. It is defined as the vector length required to achieve a performance of one half of r_∞. This means that the time, t, for a DO-loop corresponding to q vector operations (i.e. with q floating-point operations per element per iteration) is approximated by

$$t = q * (n + n_{\frac{1}{2}})/r_\infty \ . \tag{3.1}$$

Then the performance rate is given by

$$r = \frac{q * n}{t} = \frac{r_\infty}{(1 + n_{\frac{1}{2}}/n)} \ . \tag{3.2}$$

We can see from Eqn.(3.1) that $n_{\frac{1}{2}}$is a way of measuring the importance of vector startup overhead (=$n_{\frac{1}{2}}/r_\infty$) in terms of quantities known to the programmer (loop or vector length). In the benchmark program, the two parameters are determined by a least-squares fit of the data to the straight line defined by Eqn.(3.1). A useful guide to the significance of $n_{\frac{1}{2}}$is to note from Eqn.(3.2) that 80 percent of the asymptotic performance is achieved for vectors of length $4 \times n_{\frac{1}{2}}$. Generally speaking, $n_{\frac{1}{2}}$values of upto about 50 are tolerable, whereas the performance of computers with larger values of $n_{\frac{1}{2}}$is severely constrained by the need to keep vector lengths significantly longer than $n_{\frac{1}{2}}$. This requirement makes the computers difficult to program efficiently, and often leads to disappointing performance, compared to the asymptotic rate advertised by the manufacturer.

RINF1 has been used extensively for about ten years as part of the Hockney and EuroBen benchmarks (module MOD1AC) [24]. It is also included in the Genesis benchmarks [17].

3.2.4 Memory-Bottleneck Benchmarks: POLY1 and POLY2

Even if the vector lengths are long enough to overcome the vector startup overhead, the peak rate of the arithmetic pipelines may not be realised because of the delays associated with obtaining data from the cache or main memory of the computer. The POLY1 and POLY2 benchmarks quantify this dependence of computer performance on memory access bottlenecks. The computational intensity, f, of a DO-loop is defined as the number of floating-point operations (flop) performed per memory reference (mref) to an element of a vector variable [23]. The asymptotic performance, r_∞, of a computer is observed to increase as the computational intensity increases, because as this becomes larger, the effects of memory access delays become negligible compared to the time spent on arithmetic. This effect is characterised by the two parameters ($\hat{r}$,$f_{\frac{1}{2}}$), where $\hat{r}$is the peak hardware performance of the arithmetic pipeline, and $f_{\frac{1}{2}}$is the computational intensity required to achieve half this rate. That is to say the asymptotic performance is given by:

$$r_\infty = \frac{\hat{r}}{(1 + f_{\frac{1}{2}}/f)} \tag{3.3}$$

If memory access and arithmetic are not overlapped, then $f_{\frac{1}{2}}$can be shown to be the ratio of arithmetic speed (in Mflop/s) to memory access speed (in Mword/s) [23]. The parameter $f_{\frac{1}{2}}$, like $n_{\frac{1}{2}}$, measures an unwanted overhead and should be as small as possible. In order to vary f and allow the peak performance to be approached, we choose a kernel loop that can be computed with maximum efficiency on any hardware. This is the evaluation of a polynomial by Horner's rule, in which case the computational intensity is the order of the polynomial, and both the multiply and add pipelines can be used in parallel. To measure $f_{\frac{1}{2}}$, the order of the polynomial is increased from one to ten, and the measured performance for long vectors is fitted to Eqn.(3.3).

The POLY1 benchmark repeats the polynomial evaluation for each order typically 1000 times for vector lengths upto 10,000, which would normally fit into the cache of a cache-based processor. Except for the first evaluation the data will therefore be found in the cache. POLY1 is therefore an *in-cache* test of the memory bottleneck between the arithmetic registers of the processor and its cache.

POLY2, on the other hand, flushes the cache prior to each different order and then performs only one polynomial evaluation, for vector lengths from 10,000 upto 100,000, which would normally exceed the cache size. Data will have to be brought from off-chip memory, and POLY2 is an *out-of-cache* test of the memory bottleneck between off-chip memory and the arithmetic registers.

The POLY1 benchmark exists as MOD1G of the EuroBen benchmarks [24]. POLY2 exists as part of the Hockney benchmarks.

3.3 Multi-Processor Benchmarks

The PARKBENCH suite of benchmark programs provide low-level benchmarks to characterise the basic communication properties of an MPP by measuring the parameters (r_∞,$n_{\frac{1}{2}}$)for communication (COMMS1, COMMS2, COMMS3). The ratio of arithmetic speed to communication speed (the hardware+compiler parameter $f_{\frac{1}{2}}$for communication) is measured by the

POLY3 benchmark. The ability to synchronise the processors in a large MPP, in an acceptable time, is a key characteristic of such computers, and the SYNCH1 benchmark measures the number of barrier statements that can be executed per second as a function of the number of processors taking part in the barrier.

3.3.1 Communication Benchmarks: COMMS1 and COMMS2

The purpose of the COMMS1, or *Pingpong*, benchmark [25, 15] is to measure the basic communication properties of a message-passing MIMD computer. A message of variable length, n, is sent from a master node to a slave node. The slave node receives the message into a Fortran data array, and immediately returns it to the master. Half the time for this *message pingpong* is recorded as the time, t, to send a message of length, n. In the COMMS2 benchmark there is a message exchange in which two nodes simultaneously send messages to each other and return them. In this case advantage can be taken of bidirectional links, and a greater bandwidth can be obtained than is possible with COMMS1. In both benchmarks, the time as a function of message length is fitted by least squares using the parameters $(r_\infty, n_{\frac{1}{2}})$[20, 23] to the following linear timing model:

$$t = (n + n_{\frac{1}{2}})/r_\infty \tag{3.4}$$

when the communication rate is given by

$$r = \frac{r_\infty}{1 + n_{\frac{1}{2}}/n} = r_\infty \mathrm{pipe}(n/n_{\frac{1}{2}}) \tag{3.5}$$

$$\text{where} \qquad \mathrm{pipe}(x) = \frac{1}{1 + 1/x} \tag{3.6}$$

and the startup time is

$$t_0 = n_{\frac{1}{2}}/r_\infty \tag{3.7}$$

In the above equations, r_∞ is the *asymptotic bandwidth* of communication which is approached as the message length tends to infinity (hence the subscript), and $n_{\frac{1}{2}}$ is the message length required to achieve half this asymptotic rate. Hence $n_{\frac{1}{2}}$ is called the *half-performance message length.*

The importance of the parameter $n_{\frac{1}{2}}$ is that it provides a yardstick with which to measure message-length, and thereby enables one to distinguish the two regimes of short and long messages. For long messages ($n > n_{\frac{1}{2}}$), the denominator in equation 3.5 is approximately unity and the communication rate is approximately constant at its asymptotic rate, r_∞

$$r \approx r_\infty \tag{3.8}$$

For short messages ($n < n_{\frac{1}{2}}$), the communication rate is best expressed in the algebraically equivalent form

$$r = \frac{\pi_0 n}{(1 + n/n_{\frac{1}{2}})} \tag{3.9}$$

$$\text{where} \qquad \pi_0 = t_0^{-1} = r_\infty/n_{\frac{1}{2}} \tag{3.10}$$

For short messages, the denominator in equation 3.9 is approximately unity, so that

$$r \approx \pi_0 n = n/t_0 \tag{3.11}$$

In sharp contrast to the approximately constant rate in the long-message limit, the communication rate in the short message limit is seen to be approximately proportional to the message length. The constant of proportionality, π_0, is known as the *specific performance*, and can be expressed conveniently in units of kilobyte per second per byte (kB/s)/B or 'k/s'. Unfortunately since an SI prefix, such as k, cannot stand alone without a unit symbol, this unit must be written either as 10^3/s or as kHz, where Hz is a special unit name for per second (s^{-1}).

Thus, in general, we may say that r_∞ characterises the long-message performance and π_0 the short-message performance. The COMMS1 benchmark computes all four of the above parameters, $(r_\infty, n_{\frac{1}{2}}, t_0, \text{and} \pi_0)$, because each emphasises a different aspect of performance. However only two of them are independent. In the case that there are different modes of transmission for messages shorter or longer than a certain length, the benchmark can read in this breakpoint and perform a separate least-squares fit for the two regions. An example is the Intel iPSC/860 which has a different message protocol for messages shorter than and longer than 100 byte.

Because of the finite (and often large) value of t_0, the above is a *two-parameter* description of communication performance. It is therefore incorrect, and sometimes positively misleading, to quote only one of the parameters (e.g. just r_∞, as is often done) to describe the performance. The most useful pairs of parameters are $(r_\infty, n_{\frac{1}{2}})$, $(\pi_0, n_{\frac{1}{2}})$ and (t_0, r_∞), depending on whether one is concerned with long vectors, short vectors or a direct comparison with hardware times. Note also that, although $n_{\frac{1}{2}}$ is defined as the message length required to obtain half the asymptotic rate r_∞, the two parameters $(r_\infty, n_{\frac{1}{2}})$ are sufficient to calculate the communication rate for any message length via equation 3.5, or equivalently using π_0 instead of r_∞ via 3.9.

The COMMS1 and COMMS2 benchmarks exist as part of the Genesis benchmarks [17].

3.3.2 Total Saturation Bandwidth: COMMS3

To complement the above communication benchmarks, there is a need for a benchmark to measure the total saturation bandwidth of the complete communication system, and to see how this scales with the number of processors. A natural generalisation of the COMMS2 benchmark is made as follows, and called the COMMS3 benchmark: Each processor of a p-processor system sends a message of length n to the other $(p-1)$ processors. Each processor then waits to receive the $(p-1)$ messages directed at it. The timing of this generalised *pingpong* ends when all messages have been successfully received by all processors; although the process will be repeated many times to obtain an accurate measurement, and the overall time will be divided by the number of repeats. The time for the generalised pingpong is the time to send $p(p-1)$ messages of length n and can be analysed in the same way as COMMS1 and COMMS2 into values of $(r_\infty, n_{\frac{1}{2}})$. The value obtained for r_∞ is the required total saturation bandwidth, and we are interested in how this scales up as the number of processors p increases and with it the number of available links in the system.

COMMS3 is a new benchmark written specifically for PARKBENCH .

3.3.3 Communication Bottleneck: POLY3

POLY3 assesses the severity of the communication bottleneck. It is the same as the POLY1 benchmark except that the data for the polynomial evaluation is stored on a neighbouring processor. The value of $f_{\frac{1}{2}}$ obtained therefore measures the ratio of arithmetic to communication performance. Equation 3.3 shows that the computational intensity of the calculation must be significantly greater than $f_{\frac{1}{2}}$ (say 4 times greater) if communication is not to be a bottleneck. In this case the computational intensity is the ratio of arithmetic performed on a processor to words transferred to/from it over communication links. In the common case that the amount of arithmetic is proportional to the volume of a region, and the data communicated is proportional to the surface of the region, the computational intensity is increased as the size of the region (or granularity of the decomposition) is increased. Then the $f_{\frac{1}{2}}$ obtained from this benchmark is directly related to the granularity that is required to make communication time unimportant.

POLY3 is a new benchmark written specifically for PARKBENCH .

3.3.4 Synchronisation Benchmarks: SYNCH1

SYNCH1 measures the time to execute a barrier synchronisation statement as a function of the number of processes taking part in the barrier. The practicability of massively parallel computation with thousands or tens of thousands of processors depends on this barrier time not increasing too fast with the number of processors. The results are quoted both as a barrier time, and as the number of barrier statements executed per second (barr/s).

The SYNCH1 benchmark exists as part of Genesis v2.1.1 [17].

3.4 Summary of Benchmarks

Table-3.1 summarises the current low-level benchmarks, and the architectural properties and parameters that they measure.

Table 3.1: Current Low-Level benchmarks and the Parameters they measure. Note we abbreviate performance (perf.), arithmetic (arith.), communication (comms.), operations (ops.).

Benchmark	Measures	Parameters
SINGLE-PROCESSOR		
TICK1	Timer resolution	tick interval
TICK2	Timer value	wall-clock check
RINF1	Basic Arith. ops.	$(r_\infty, n_{\frac{1}{2}})$
POLY1	Cache-bottleneck	$(\hat{r}, f_{\frac{1}{2}})$
POLY2	Memory-bottleneck	$(\hat{r}, f_{\frac{1}{2}})$
MULTI-PROCESSOR		
COMMS1	Basic Message perf.	$(r_\infty, n_{\frac{1}{2}})$
COMMS2	Message exch. perf.	$(r_\infty, n_{\frac{1}{2}})$
COMMS3	Saturation Bandwidth	$(r_\infty, n_{\frac{1}{2}})$
POLY3	Comms. Bottleneck	$(\hat{r}, f_{\frac{1}{2}})$
SYNCH1	Barrier time and rate	barr/s

3.4.1 Arithmetic Benchmark Results

As an indication of the type of results given by the proposed low-level arithmetic benchmarks, Table-3.2 gives measurements made on a number of workstations, and microprocessor chips that are used as processing nodes in multiprocessor MIMD computers. The measurements shown represent the state of affairs on the date of the measurements, and both hardware and software improvements since that time should have significantly improved the results. They are presented here only to illustrate the type of results to be expected from the low-level benchmarks. They should not be taken as representing the current state of competitive performance in the very rapidly changing workstation and chip market. Such a comparison will only be possible if these benchmarks are routinely run on new hardware and software and the results stored in the PARKBENCH interactive performance database, which would then contain an up-to-date comparison of competitive hardware and software. Notwithstanding these caveats, we feel it is helpful to give these examples of low-level benchmark measurements that happen to be available, even though some are a few years old and therefore probably seriously outdated. In this small table we have not room to give the full specification of the conditions for each measurement (full and exact description hardware, and of compiler and options used, etc.), but this information would be an essential and required component of an entry into the PARKBENCH database of benchmark results.

Table 3.2: Examples of low-level benchmark measurements on some common workstations and microprocessor chips used in MIMD computers. Measurements were made with the highest level of optimisation that ran, and are in Mflop/s for 64-bit precision. The units of $n_{\frac{1}{2}}$ are vector length, and $f_{\frac{1}{2}}$ are flop/mref. Results are for the best generally available compiler on the date shown. The RINF1 benchmark gives values of the $(r_\infty, n_{\frac{1}{2}})$ parameters for the kernel A=B*C (vector = vector × vector) for contiguously stored vectors.

Benchmark	Intel i860XP 50MHz	IBM RS/ 6000-530 25MHz	DEC α 133MHz
d/m/y	12/10/93	14/6/90	13/1/93
Linpackd n=100	14.7	9.54	20.7
Livermore Maximum	28.8	31.8	46.6
Livermore Minimum	2.62	1.34	4.47
RINF1			
r_∞	7.64		26.4
$(n_{\frac{1}{2}})$	(2.58)		(5.6)
POLY1			
$\hat{r}$	13.50	25.85	88.9
$(f_{\frac{1}{2}})$	(0.44)	(0.34)	(0.71)
POLY2			
$\hat{r}$	13.48	25.65	
$(f_{\frac{1}{2}})$	(1.12)	(0.91)	

3.4.2 Example Results for the COMMS1 benchmark

We report below results for the COMMS1 benchmark on the SUPRENUM , and Intel iPSC/860 [15], Touchstone Delta [26], Intel Paragon XP/S and Meiko CS-2 computers.

Table-3.3 gives the values obtained for the communication parameters, in the version of the benchmark using the native SUPRENUM extensions to the Fortran90 language. These include a SEND and RECEIVE language statement with a syntax similar to that of the Fortran READ and WRITE statement. The asymptotic stream rate, or bandwidth, (r_∞) shows considerable variation on the Suprenum, depending on how the data to be transferred is specified in the I/O list of the SEND statement. A variable length array in Fortran90 syntax in single precision achieves 0.67 MB/s, whereas the same statement specified in double precision achieves 4.8 MB/s. This double-precision rate is about twice that observed on the iPSC/860 with their CSEND Fortran subroutine, which sends an array whose length is specified in bytes. The principal difference between the two computers is the magnitude of the startup time, t_0, which is $73\mu s$ on the iPSC/860 compared with about 3ms on the Suprenum. Since the startup time, via π_0, determines the transfer rate for short messages (say < 100B), we see that the Suprenum is 45 times slower than the iPSC/860 for short messages. On the other hand the Suprenum has almost twice the stream rate for long messages (as seen by the value of r_∞), provided the most favourable format (i.e. double precision or 64-bit) is used in the I/O list. One may compute from these numbers that the iPSC/860 is faster at transferring messages for all message lengths less than 16,481 Byte. The longer startup time on Suprenum results in larger values of $n_{\frac{1}{2}}$, showing that longer messages are needed to achieve any given fraction of the asymptotic rate.

The results for the Touchstone Delta show that this computer has the fastest short and long message performance, judged respectively by the values of π_0 and r_∞. However the improvement of short message performance over the iPSC/860 is only marginal, and the long message performance is only about one quarter of the advertised bandwidth of 25MB/s. However hardware and software improvements made since the measurements were made should have improved the results.

If we compare the new generation of production computers, the Intel Paragon XP/S and the Meiko CS-2, we find, on the dates stated, the CS-2 to have a higher communication performance than the Paragon for both short (π_0) and long messages (r_∞), and therefore for all message lengths. However both computers are at an early state of the hardware and software development, and both have considerable development potential. The COMMS1 benchmark will continue to be used to track this competition in communication performance, and the success of both manufacturers to achieve a high performance for both short and long messages.

Table 3.3: Values of (r_∞,$n_{\frac{1}{2}}$, t_0, π_0) for the communication of messages between two nodes of the same cluster on the Suprenum and neighbouring nodes on the Intel iPSC/860, Touchstone Delta, Intel Paragon and Meiko CS-2 computers. The Delta measurements were made on 17 Jan. 1992, and should have been improved by subsequent hardware and software changes. Paragon measurements were made at ORNL 25-28 May, 1993, and the CS-2 measurements were made at Southampton University 9 July, 1993

Specification	Range B*	r_∞ MB/s	$n_{\frac{1}{2}}$ B	t_0 ms	π_0 kHz
SUPRENUM					
sp SEND A(1:N)		0.67	2041	3.05	0.328
dp SEND A(1:N)		4.82	12740	2.64	0.378
INTEL iPSC/860					
CSEND (,A,N,,)	$N < 100$	2.36	179	0.074	13.5
	$N > 100$	2.80	560	0.200	5.0
INTEL Delta					
CSEND (,A,N,,)	$N < 512$	3.48	213	0.061	16.3
	$N > 512$	6.76	892	0.132	7.57
INTEL Paragon XP/S					
CSEND (,A,N,,)	$N < 40000$	23.5	4044	0.172	5.80
Meiko CS-2					
PARMACS	$N < 40000$	43.0	3747	0.087	11.5

* B - byte

Chapter 4

Kernel Benchmarks[*]

4.1 Introduction and Rationale

The low-level benchmark codes are designed to measure the basic architectural features of parallel machines. Full application codes obviously measure the performance of a parallel system on the full problem and this is ultimately what the user wants. However, in many instances, the full application codes are complex, contain many 100s of thousands of lines of Fortran, and are not available in a suitable parallel version. In order to obtain a guide to the performance of any given parallel system on a particular application something less complex than the full application is useful. A profile of the sequential version of the application enables the compute intensive portions of the program to be identified. It is these compute-intensive sections of an application that we wish to model with the introduction of parallel kernel benchmarks.

The popular kernel benchmarks that have been used for traditional vector supercomputers, such as the Livermore Loops [5], the LINPACK benchmark [8] and the original NAS kernels [27], are clearly inappropriate for the performance evaluation of highly parallel machines. First of all, the tuning restrictions of these benchmarks rule out many widely used parallel extensions. More importantly, the computation and memory requirements of these programs do not do justice to the vastly increased capabilities of the new parallel machines, particularly those that will be available by the mid 1990's. For these reasons we believe that a new, widely accepted set of kernel benchmarks is desirable as a step on the way to more sensible and scientific performance reporting of parallel systems.

The kernel codes are typically up to a few thousand lines of Fortran and are sufficiently simple that the performance of a given parallel machine on this program may be related to the underlying architectural parameters. It must be acknowledged, however, that the performance on kernels alone is insufficient to completely assess the performance potential of a parallel machine on full scientific applications. The chief difficulty is that a certain data structure may be very efficient on a certain system for one of the isolated kernels, and yet this data structure would be inappropriate if incorporated into a larger application. For example, the performance of a real CFD application on a parallel system is critically dependent on data motion between different computational kernels. In addition, full applications typically have initialization phases, I/O and so on, so complete reproduction of these features can be of critical importance for a realistic guide to performance.

For these reasons the PARKBENCH suite introduces a level of complexity above kernel

---

[*] assembled by Tony Hey for Kernel subcommittee

codes which is called *compact applications*. These are full but perhaps simplified application codes that contain all the necessary features of the full problem but are sufficiently simple to run and analyse. These are described in the Compact Application chapter of this document.

4.2 The Kernel Benchmarks

The kernels attempt to span a reasonably wide range of application areas by including the most frequently encountered computationally intensive types of problems. We have tentatively grouped them into four sections. Some of the benchmark codes are taken from existing parallel benchmark suites (NAS [28], Genesis [12], etc). In order to avoid duplication and redundancy, we have attempted to list some of the attributes of the parallel system tested by each kernel benchmark.

4.2.1 Matrix benchmarks

For the past 15 years or so, there has been a great deal of activity in the area of algorithms and software for solving linear algebra problems. The linear algebra community has long recognized the need for help in developing algorithms into software libraries, and several years ago, as a community effort, put together a *de facto* standard for identifying basic operations required in linear algebra algorithms and software. The hope was that the routines making up this standard, known collectively as the Basic Linear Algebra Subprograms (BLAS), would be efficiently implemented on advanced-architecture computers by many manufacturers, making it possible to reap the portability benefits of having them efficiently implemented on a wide range of machines. This goal has been largely realized.

The key insight of this approach to designing linear algebra algorithms for advanced architecture computers is that the frequency with which data are moved between different levels of the memory hierarchy must be minimized in order to attain high performance. Thus, our main algorithmic approach for exploiting both vectorization and parallelism in our implementations is the use of block-partitioned algorithms, particularly in conjunction with highly-tuned kernels for performing matrix-vector and matrix-matrix operations (the Level 2 and 3 BLAS). In general, the use of block-partitioned algorithms requires data to be moved as blocks, rather than as vectors or scalars, so that although the total amount of data moved is unchanged, the latency (or startup cost) associated with the movement is greatly reduced because fewer messages are needed to move the data.

A second key idea is that the performance of an algorithm can be tuned by a user by varying the parameters that specify the data layout. On shared memory machines, this is controlled by the block size, while on distributed memory machines it is controlled by the block size and the configuration of the logical process mesh.

The way in which an algorithm's data are distributed over the processors of a concurrent computer has a major impact on the load balance and communication characteristics of the concurrent algorithm, and hence largely determines its performance and scalability. The block scattered (or block cyclic) decomposition provides a simple, yet general-purpose, way of distributing a block-partitioned matrix on distributed memory concurrent computers. In the block scattered decomposition, described in detail in [29], a matrix is partitioned into blocks of size $r \times s$, and blocks separated by a fixed stride in the column and row directions are assigned to the same processor. If the stride in the column and row directions is P and Q blocks respectively, then we require that $P\,Q$ equals the number of processors, N_p. Thus, it is useful to imagine the processors arranged as a $P \times Q$ mesh, or template. Then the processor at position (p, q) $(0 \le p < P,\ 0 \le q < Q)$ in the template is assigned the blocks indexed by,

$$(p + i\,P,\ q + j\,Q), \tag{4.1}$$

where $i = 0, \ldots, \lfloor (M_b - p - 1)/P \rfloor$, $j = 0, \ldots, \lfloor (N_b - q - 1)/Q \rfloor$, and $M_b \times N_b$ is the size of the matrix in blocks.

Blocks are scattered in this way so that good load balance can be maintained in algorithms, such as LU factorization [30, 31], in which rows and/or columns of blocks of a matrix become eliminated as the algorithm progresses. However, for some of the distributed Level 3 BLAS routines a scattered decomposition does not improve load balance, and may result in higher concurrent overhead. The general matrix-matrix multiplication routine xGEMM is an example of such a routine for which a pure block (i.e., nonscattered) decomposition is optimal when considering the routine in isolation. However, xGEMM may be used in an application for which, overall, a scattered decomposition is best.

The underlying concept of the implementations we have chosen for dense matrix computations is the use of block-partitioned algorithms to minimize data movement between different levels in hierarchical memory. The ideas discussed here for dense linear algebra computations are applicable to any computer with a hierarchical memory that (1) imposes a sufficiently large startup cost on the movement of data between different levels in the hierarchy, and for which (2) the cost of a context switch is too great to make fine grain size multithreading worthwhile. These ideas have been exploited by the software packages LAPACK [14] and ScaLapack [32]. The PARKBENCH suite includes five matrix kernels.

1. Dense matrix multiply. Communication involves broadcast of data along rows of mesh, and periodic shift along column direction (or vice versa).
2. Transpose. Matrix transpose is an important benchmark because it exercises the communications of computer heavily on a realistic problem where pairs of processors communicate with each other simultaneously. It is a useful test of the total communications capacity of the network.
3. Dense LU factorization with partial pivoting. Searching for a pivot is basically a reduction operation within one column of the processor mesh. Exchange of pivot rows is a point-to-point communication. Update phase requires data to be broadcast along rows and columns of the processor mesh.
4. QR Decomposition. In this benchmark parallelization is achieved by distribution of rows on a logical grid of processors using block interleaving.
5. Matrix tridiagonalization, for eigenvalue computations of symmetric matrices.

There have been many implementations of matrix multiplication algorithms on distributed memory machines [33, 34, 35]. Many of them are limited in their use since they are implemented with a pure block (non-scattered) distribution, or specific (not general-purpose) data distribution, and/or on square processor configurations with a specific number of processors (column and/or row numbers of processors are powers of 2). The software contained in this benchmark eliminates all of these constraints.

Our matrix multiplication algorithm is a block scattered variant of that of Fox, Hey, and Otto [33], that deals with arbitrary rectangular processor templates.

Suppose the matrix $\mathbf{A}$ has M_b block rows and L_b block columns, and the matrix $\mathbf{B}$ has L_b block rows and N_b block columns. Block (I, J) of $\mathbf{C}$ is then given by

$$C(I, J) = \sum_{K=0}^{L_b-1} A(I, K) \cdot B(K, J) \tag{4.2}$$

```
DO $K = 0, L_b - 1$
    [Columncast one block of $\mathbf{B}$ ($B(I, \mathrm{MOD}(I+K, N_b))$, $I = 0 : L_b$)
     along each column across template]
    PARDO $I = 0, M_b - 1$
        $KP = \mathrm{MOD}(K + I, L_b)$
        PARDO $J = 0, N_b - 1$
            $\mathrm{C}(I,J) = \mathrm{C}(I,J) + \mathrm{A}(I,KP) \cdot \mathrm{B}(KP,J)$
        END PARDO
    END PARDO
    [Roll $\mathbf{A}$ leftwards]
END DO
```

Figure 4.1: A distributed block scattered matrix multiplication algorithm. The PARDO's indicate over which indices the data are decomposed. All indices refer to blocks of elements. Communication phases are indicated in square brackets.

where $I = 0, 1, \ldots, M_b - 1$, $J = 0, 1, \ldots, N_b - 1$. In Equation 4.2 the order of summation is arbitrary.

Fox et al. initially considered only the case of square matrices in which each processor contains a single row or a single column of blocks. That is, the blocks that start the summation lie along the diagonal. The summation is started at a different point for each block row of **C** so that in the phase of the parallel algorithm corresponding to summation index K, $A(I,K)$ and $B(K,J)$ can be multiplied in the processor to which $C(I,J)$ is assigned.

This requires each processor containing a block of **B** to be multiplied in step K to broadcast that block along the column of the processor template at the start of the step. Also **A** must be rolled leftwards at the end of the step so that each column is overwritten by the one to the right, with the first column wrapping round to overwrite the last column. The pseudocode for this algorithm is shown in Figure 4.1. Another variant of this algorithm involves broadcasting blocks of **A** over rows, and rolling **B** upwards.

In Figure 4.1 a *columncast* is a communication phase in which one data item (typically a block, or set of blocks) is taken from each block column of the matrix and is broadcast to all the other processors in the same column of the processor template. A *rowcast* is similar, but broadcasts a data item from each block row of the matrix to all processors in the same row of the template.

The kernels for LU, QR and the reduction of a symmetric matrix to tridiagonal form in preparation for eigenvalue computations all use block-partitioned algorithms. They rely on the BLAS for most of the computational performance and the BLACS for communication.

4.2.2 Fourier Transforms

The computational of the fast Fourier transforms (FFTs) is the cornerstone of many supercomputer applications. These include not only the predictable digital signal processing, speech recognition, image processing, and petroleum seismic analysis, but also other less obvious applications, such as in computational fluid dynamics, medical technology, multiple precision arithmetic and computational number theory. Computations worthy of a highly parallel supercomputer generally fall into four categories: (1) one or a few very long 1-D FFTs; (2) many small or moderate-sized 1-D FFTs; (3) one or a few large 2-D FFTs; or (4) one or a few large 3-D FFTs. The PARKBENCH suite includes two FFT test kernels, one for a large 1-D FFT, and one for a large 3-D FFT.

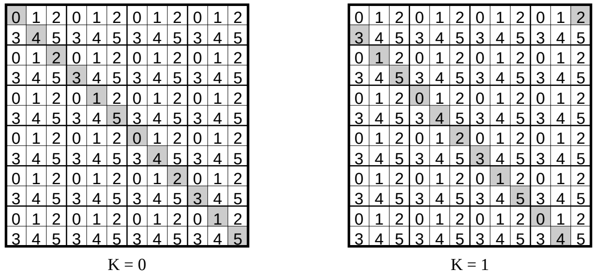


Figure 4.2: Snapshot of SDB algorithm. The blocks of the matrix **B** communicated in the first two stages of the matrix multiplication algorithm are shown shaded. In this case $P = 2$ and $Q = 3$. In each stage, only one wrapped diagonal is columncast. The total number of stages is L_b.

1. 1-D FFT. In this kernel, two sequences of integers x_i and y_i are generated, with length $n = 2^m$ and values in the range $0 \leq x_i,\ y_i < M$. The standard value of M is 1024. These sequences are generated using the same uniform pseudo-random number generator as is used in the 3-D FFT kernel and the embarrassingly parallel kernel. Then the linear convolution of these two sequences is computed using a complex-number FFT, i.e. by padding x and y with zeroes to length $2n$, then performing a forward FFT on x and y, multiplying the two resulting sequences of complex numbers, and finally performing an inverse FFT on the result. The result sequence should have exclusively integer values, which permits a straightforward validity check.

   No restriction is placed on the FFT technique used to perform this convolution, except that it be based on a complex-number FFT rather than, for example, a number-theoretic FFT. It is expected, however, that efficient implementations will employ techniques, such as Edson's algorithm and real-to-complex FFTs, that take advantage of the purely real nature of the input and output data to reduce the computational cost. The usage of vendor-supplied library FFT routines is permitted. The serial implementation program includes a reasonably efficient 1-D FFT suitable for computation on a workstation or single processor vector system.

2. 3-D FFT. The PARKBENCH 3-D FFT kernel is the *3-D FFT PDE* benchmark from the NAS Parallel Benchmark suite [28]. It performs the essence of many *spectral* codes and is a rigorous test of long-distance communication performance. A brief description of this benchmark is as follows.

Consider the partial differential equation (PDE)

$$\frac{\partial u(x,t)}{\partial t} = \alpha \nabla^2 u(x,t)$$

where x is a position in three-dimensional space. When a Fourier transform is applied to each side, this equation becomes

$$\frac{\partial v(z,t)}{\partial t} = -4\alpha\pi^2 |z|^2 v(z,t)$$

where $v(z,t)$ is the Fourier transform of $u(x,t)$. This has the solution

$$v(z,t) = e^{-4\alpha\pi^2|z|^2 t} v(z,0)$$

In this benchmark problem, one starts with a 3-D complex array U, which represents u above, which is filled with pseudorandom data generated by the same scheme as used in the embarrassingly parallel kernel. Then one computes V, the result of a forward 3-D FFT of U. For each of several iterations, one multiplies V by the appropriate exponential factors and performs an inverse 3-D FFT on the result.

Any complex FFT algorithm may be used for the computation of the 3-D FFTs mentioned above, and vendor-supplied library routines may be employed.

4.2.3 PDE Kernels

In these PDE kernels communication is basically exchange with neighbors and the convergence check is a reduction. A variety of methods and update stencils may be used. The following two PDE solvers have been proposed for inclusion in the parallel benchmark suite:

1. Successive Over-Relaxation (SOR) kernel. The PARKBENCH SOR kernel is based on the PDE1 benchmark from the GENESIS distributed memory benchmark suite [17]. This benchmark solves the Poisson equation on a 3-dimensional grid by parallel red-black relaxation with Chebyshev acceleration. In this method the mesh points are divided into two groups according to whether the sum of indices is odd ('red') or even ('black'). The method proceeds in half iterations, during each of which only half the points are adjusted (alternately the 'red' and 'black' set of points). Thus all the 'red' points can be adjusted in parallel during one half iteration, and similarly all the 'black' points in parallel during the next half iteration. The problem is discretized using the ordinary 7-point difference stencil in a regular cubic grid. The value of the relaxation factor (ω) changes at each half iteration according to:

   $$\begin{aligned} \omega^{(0)} &= 1 \\ \omega^{(1/2)} &= 1/(1 - 1/2\rho^2) \\ \omega^{(t+1/2)} &= 1/(1 - 1/4\rho^2\omega^{(t)}), \qquad t = 0, 1/2, 1, ..., \infty, \end{aligned} \tag{4.3}$$

   where ρ is the convergence factor of the corresponding Jacobi iteration and the superscript t designates the iteration number. For large number of iterations, ω tends to the constant relaxation factor that is used throughout the traditional SOR procedure. The asymptotic convergence factor is therefore the same for both algorithms.

   In order to map the problem onto a parallel multiprocessor system the 3-dimensional grid is divided into cuboidal subgrids. Each subgrid is assigned to a node in such a way that neighbouring subgrids are mapped on neighbouring nodes. The grid variables in each subgrid are exclusively computed by its associated node. At the inner boundaries of the subgrid the nodes need values at points which are contained in the neighbouring subgrid. Rather then transferring these values exactly at the time when they are needed – this would prevent vector processing within the node – they are stored in so-called overlap areas. After each iteration the values in the overlap areas are exchanged and updated via the message-passing communication mechanism. The introduction of overlap areas needs a strict synchronization following each iteration step in order to ensure the correct execution of the benchmark.

   Since the Chebyshev SOR method requires no extra arithmetic over the traditional SOR algorithm yet has more favourable initial error decay properties, it is one of the most efficient PDE kernels. Note, however, that in this benchmark only nearest neighbour interactions are required and the number of floating point operations per grid point is very small when compared to more complex PDEs.

2. Multigrid kernel.

   The PARKBENCH multigrid kernel is the multigrid benchmark from the NAS Parallel Benchmarks [28]. It requires highly structured long distance communication and tests both short and long distance data communication.

   This kernel performs a V-cycle multigrid algorithm to obtain an approximate solution u to the discrete Poisson problem

   $$\nabla^2 u = v$$

   on a $256 \times 256 \times 256$ grid with periodic boundary conditions.

   One starts out with the array $v = 0$, except at a 20 particular points where $v = \pm 1$. The iterative solution starts with $u = 0$. Each iteration consists of the following two steps, where $k = 8 = \log_2(256)$:

   $$\begin{aligned} r &= v - A\,u & &\textit{(evaluate residual)} \\ u &= u + M^k\,r & &\textit{(apply correction)} \end{aligned}$$

   Here M^k denotes a *V-cycle* multigrid operator, and A denotes a trilinear finite element discretization of the Laplacian ∇^2.

4.2.4 Other

1. Embarrassingly Parallel. The Parkbench *embarrassingly parallel* kernel is taken from the NAS Parallel Benchmarks [28]. It provides an estimate of the upper achievable limits for floating point performance, i.e. the performance without significant interprocessor communication.

   In this benchmark, one first generates pairs $(x_j,\ y_j)$ of uniformly distributed pseudo-random floating point values generated using the linear congruential generator

   $$\begin{aligned} z_{k+1} &= a z_k \pmod{2^{46}} \\ r_{k+1} &= 2^{-46} z_{k+1} \end{aligned}$$

   For each pair $(x_j,\ y_j)$, one tests to see if $t_j = x_j^2 + y_j^2 \leq 1$. If not, this pair is rejected. If this inequality holds, then one sets $X_k = x_j\sqrt{(-2\log t_j)/t_j}$ and $Y_k = y_j\sqrt{(-2\log t_j)/t_j}$.

Then X_k and Y_k are independent Gaussian deviates with mean zero and variance one. The benchmark problem is to count the number of these Gaussian deviates that lie in various square annuli around the origin.

2. Large Integer Sort.

   Although sorting has traditionally been thought of as of importance primarily in non-scientific computing, this operation is increasingly important in advanced scientific applications. In *particle method* fluid simulations, for example, sorting is the dominant cost.

   The PARKBENCH integer sort benchmark is taken from the NAS Parallel Benchmarks [28]. The kernel tests both integer computation speed and communication performance.In this benchmark, a vector of integer data is generated using the same pseudorandom number generator that is used in the embarrassingly parallel kernel. This data is initially mapped according to a particular scheme. The benchmark problem is to sort this data by the most efficient scheme for a particular architecture. Vendor-supplied sort routines may be used to perform the sort operation.

3. Input/Output. We propose a *pencil and paper* style benchmark – not tied to any particular parallel platform or application but just measuring some key fundamental I/O parameters of the system. A standard Fortran-77 version complements the detailed description given in the individual `ReadMe` file. The I/O performance is tested by writing and then reading different sized data sets to and from disk. The read and write buffer sizes are varied so that estimates of disk I/O start-up time, bandwidth and data transference times may be made.

4.3 Benchmark Implementation

The PARKBENCH kernel benchmarks are written as far as possible in standard Fortran 77 using 64-bit floating point arithmetic (`DOUBLE PRECISION` on most systems), unless otherwise stated. Both PVM/MPI [1, 2] and subset HPF versions exist for most of the codes in addition to the standard Fortran-77 versions. A description of each benchmark and instructions on how to run it are given in individual `ReadMe` files. They also contain a specification of the three problem sizes agreed upon for each code: (1) test problem (2) moderate size and (3) grand challenge size. A formula should be given in the `ReadMe` files to produce flop counts for the kernel benchmarks along with precalculated figures for each standard problem size. `Make`-files are supplied with each benchmark to handle compilation and linking in a Unix environment.

4.4 Concluding Remarks

The contents of the PARKBENCH kernel benchmark suite should map reasonably well onto any parallel library supplied by the vendors. This will allow comparative performance measurements across different platforms using the PARKBENCH kernels but also performance comparisons to the functionally similar and highly-optimized library routines on every particular parallel system. Another advantage of the use of kernel benchmarks is that they should not involve an unreasonable amount of labor on the part of vendors.

Chapter 5

Compact Applications*

5.1 Introduction

While kernel applications, such as those described in Chapter 3, provide a fairly straightforward way of assessing the performance of parallel systems they are not representative of scientific applications in general since they do not reflect certain types of system behavior. In particular, many scientific applications involve data movement between phases of an application, and may also require significant amounts of I/O. These types of behavior are difficult to gauge using kernel applications.

One factor that has hindered the use of full application codes for benchmarking parallel computers in the past is that such codes are difficult to parallelize and to port between target architectures. In addition, full application codes that have been successfully parallelized are often proprietary, and/or subject to distribution restrictions. To minimize the negative impact of these factors we propose to make use of compact applications in our benchmarking effort.

Compact applications are typical of those found in research environments (as opposed to production or engineering environments), and usually consist of up to a few thousand lines of source code. Compact applications are distinct from kernel applications since they are capable of producing scientifically useful results. In many cases, compact applications are made up of several kernels, interspersed with data movements and I/O operations between the kernels.

In this chapter the criteria for selecting compact applications for the PARKBENCH suite will be discussed. In addition, the general research areas that will be represented in the suite are outlined.

5.2 Criteria for Selection

The three main criteria for inclusion of a parallel code in the Compact Applications suite are,

1. The code must be a complete application and be capable of producing results of research interest. These two points distinguish a compact application from a kernel. For example, a code that only solves a randomly-generated, dense, linear system by LU factorization should be considered a kernel. Even though the code is complete, it does not produce results of research interest. However, if the LU factorization is embedded in an application that uses the boundary element method to solve, for example, a two-dimensional elastodynamics problem, then such an application could legitimately be considered a

---

*assembled by David Walker for Compact Applications subcommittee compact application. Compact applications and full production codes are distinguished by their software complexity, which is difficult to quantify. Software complexity gives an indication of how hard it is to write, port and maintain an application, and may be gauged very roughly by the length of the source code. However, there is no hard upper limit on the length of a code in the Compact Applications suite. It is expected that the source code (excluding comments and repeated common blocks) for most compact applications will be between 2000 and 10000 lines, but some may be longer.

2. The code must be of high quality. This means it must have been extensively tested and validated, preferably on a wide selection of different parallel architectures. The problem size and number of processors used must not be hard-coded into the application, and should be specified at runtime as input to the program. Ideally, the parallel code should not impose restrictions on the problem size that are not applicable for the corresponding sequential code. Thus, the parallel code should not require that the problem size be exactly divisible by the number of processors, or that the number of processors be a power of two. In some cases this latter requirement may have to be relaxed. For example, most parallel fast Fourier transform routines require the number of processors to be a power of two. It is preferable that the code be written so that it works correctly for an arbitrary one-to-one mapping between the logical process topology of the application and the hardware topology of the parallel computer. This is desirable so that the assignment of a location in the logical process topology to a physical processor can be easily adjusted when porting the application between platforms. For example a Gray code assignment may be best for a hypercube, and a natural ordering for a mesh architecture.

3. The application must be well documented. The source code itself should contain an adequate number of comments, and each module should begin with a comment section that describes what the routine does, and the arguments passed to it. In addition, there should be a *Users' Guide* to the application that describes the input and output, the parameterization of the problem size and processor layout, and details of what the application does. The Users' Guide should also contain a bibliography of related papers.

In addition, to the three criteria discussed above, there are a number of other desirable features that a PARKBENCH Compact Application should have. These are discussed in the following subsections.

5.2.1 Self Checking Applications

The application should be self-checking. That is, at the end of the computation the application should perform a check to validate the results of the run. The application may also output a summary of performance results for the run, such as the Mflop rate, and other pertinent information.

5.2.2 Programming Languages

The code should be written in Fortran 77, Fortran 90, High Performance Fortran, or C. Data should be passed between processors by explicit message passing. PARKBENCH does not specify which message passing system should be used, but one that is available on a number of parallel platforms is preferable. Eventually it is expected that MPI will become the message passing system of choice, but in the meantime portable systems such as PVM, PICL, Express, PARMACS, and P4 are acceptable alternatives. The codes in the Compact Applications suite should not contain any assembly coded portions, although assembly code may be used in optimized versions of the code.

5.3 Proposed Compact Application Benchmarks

At the time of writing (October 1993) the PARKBENCH organization is in the process of soliciting submission of applications for inclusion in the Compact Applications suite. Thus, the applications that comprise the suite cannot yet be listed here. However, in this section the main application areas that are expected to be in the suite are outlined. The intention is that these areas should be representative of the fields in which parallel computers are actually used. The codes should exercise a number of different algorithms, and possess different communication and I/O characteristics. Initially the Compact Applications suite will consist of no more than ten codes. This restriction is imposed so that the resources needed to manage and distribute the suite can be assessed. The suite may be enlarged in the future if this seems manageable. Below is a list of the application areas that are expected to be represented in the suite. This is not meant to be an exclusive list; submissions from other application areas will be considered for inclusion in the suite.

- Climate and meteorological modeling
- Computational fluid dynamics (CFD)
- Finance, e.g., portfolio optimization
- Molecular dynamics
- Plasma physics
- Quantum chemistry
- Quantum chromodynamics (QCD)
- Reservoir modeling

5.4 Submitting to the Compact Application Suite

The procedure for submitting codes to the PARKBENCH Compact Applications suite is as follows.

1. Complete the submission form in Appendix A, and email it to David Walker at `walker@msr.epm.ornl.gov`. The data on this form will be reviewed by the PARKBENCH Compact Applications Subcommittee, and the submitter will be notified if the application is to be considered further for inclusion in the PARKBENCH suite.

2. If PARKBENCH Compact Applications Subcommittee decides to consider the application further the submitter will be asked to submit the source code and input and output files, together with any documentation and papers about the application. Source code and input and output files should be submitted by email, or ftp, unless the files are very large, in which case a tar file on a 1/4 inch cassette tape. Wherever possible email submission is preferred for all documents in man page, Latex and/or Postscipt format.

These files documents and papers together constitute the application package. The application package should be sent to the following address, and the subcommittee will then make a final decision on whether to include the application in the PARKBENCH suite.

David W. Walker
Oak Ridge National Laboratory
Bldg. 6012/MS-6367
P. O. Box 2008
Oak Ridge, TN 37831-6367
(615) 574-7401/0680 (phone/fax)
`walker@msr.epm.ornl.gov`

3. If the application is approved for inclusion in the PARKBENCH suite an authorized person from the submitting organization will be asked to complete and sign a form giving PARKBENCH authority to distribute, and modify (if necessary), the application package.

Chapter 6

HPF Compiler Benchmarks[*]

6.1 Objectives

For most users, the performance of codes generated by a compiler is what that actually matters. This can be inferred from running HPF version of PARKBENCH codes described in chapter 4 and 5. For HPF compiler developers, however, *an additional* benchmark suite may be very useful: the benchmark suite that can evaluate specific HPF compilation phases and the compiler runtime support. For that purpose, the relevant metric seems to be the ratio of execution times of compiler generated to hand coded programs as a function of the problem size and number of processors engaged in the computation.

The compilation process can be logically divided into several phases, and each of them influence the efficiency of the resulting code. The initial stage is parsing of a source code which results in an internal representation of the code. It is followed compiler transformations, like data distribution, loop transformations, computation distribution, communication detection, sequentialization, insertion of calls to a runtime support, and others. This we will call a HPF-specific phase of compilation. The compilation is concluded by code generation phase. For portable compilers that outputs f77+message passing code, the node compilation is obviously factorized out and the efficiency of the node compiler can be evaluated separately.

This benchmark suite addresses the HPF-specific phase only. Thus, it is well suited for performance evaluation of both translators (HPF to F77+message passing) and genuine HPF compilers. The parsing phase is an element of the conventional compiler technology and it is not of interest in this context. The code generation phase involves optimization techniques developed for sequential compilers (in particular, Fortran 90 compilers) as well as micro-grain parallelism or vectorization. The object codes for specific platforms may be strongly architecture dependent (e.g., may be very different for processors with vector capabilities than for those without it). Evaluation of performance of these aspects require different techniques that these proposed here.

It is worth noting, that the HPF-phase strongly affect the possibility of optimization of the node codes. For example, insertions of calls to the communication library may prohibit the node compiler to perform many standard optimizations without expensive interprocedural analysis. Therefore, capability to exploit opportunities for optimizations at HPF level and to generate the output code that way it can be further optimized by the node compiler is an important element of evaluation of HPF compilers. Nevertheless, evaluation of the HPF-phase separately is very valuable since the hand coded programs face the same problems. We will

---

[*] assembled by Tom Haupt for Compiler Benchmarks subcommittee

address these issues in the future releases of the benchmark suite.

Compilers for massively parallel and distributed systems are still object of a research and laboratory testing rather than commercial products. The parallel compiler technology as well as methods of evaluating it is not mature yet. Nevertheless, the advent of the HPF standard gives opportunity to develop systematic benchmarking techniques.

The current definition of HPF cannot be recognized as an ultimate solution for parallel computing. Its limitations are well known, and many researchers are working on extensions to HPF to address a broader class of real life, commercial and scientific applications. We expect new language features to be added to the HPF definition in future versions of HPF, and we will extend the benchmark suite accordingly. On the other hand, new parallel languages based on languages other than Fortran, notably C++, become more and more popular. Since the parallelism is inherent in a problem and not its representation, we anticipate many commonalities in the parallel languages and corresponding compiler technologies, notably sharing the runtime support. Therefore, we decided to address this benchmark suite to these aspects of the compilation process that are inherent to parallel processing in general, rather than testing syntactic details of the HPF.

6.2 Low Level HPF Compiler Benchmarks

6.2.1 Overview

The benchmark suite comprises several simple, synthetic applications which test several aspects of the HPF compilation. The current version of the suite addresses the basic features of HPF, and it is designed to measure performance of early implementations of the compiler. They concentrate on testing parallel implementation of explicitly parallel statements, i.e., array assignments, FORALL statements, INDEPENDENT DO loops, and intrinsic functions with different mapping directives. In addition, the low level compiler benchmarks address problem of passing distributed arrays as arguments to subprograms.

The language features not included in the HPF subset are not addressed in this release of the suite. The next releases will contain more kernels that will address all features of HPF, and also they will be sensitive to advanced compiler transformations.

The codes included in this suite are either adopted from existing benchmark suites, NAS suite [27], Livermore Loops [5], and the Purdue Set [36], or are developed at Syracuse University.

6.2.2 FORALL statement - kernel FL

FORALL statement provides an convenient syntax for simultaneous assignments to large groups of array elements. Such assignments lie at the heart of the data parallel computations that HPF is designed to express. The idea behind introducing FORALL in HPF is to generalize Fortran 90 array assignments to make expressing parallelism easier. Kernel FL provides several examples of FORALL statements that are difficult or inconvenient to write using Fortran 90 syntax.

6.2.3 Explicit template - kernel TL

Parallel implementation of the array assignments, including FORALL statements, is a central issue for an early HPF compiler. Given a data distribution, the compiler distributes computation over available processors. An efficient compiler achieves an optimal load balance with minimum interprocessor communication.

Sometimes, the programmers may help the compiler to minimize necessary interprocessor communication by suitable data mapping, in particular by defining a relative alignment of different data object. This may be achieved by aligning the data objects with an explicitly declared template. Kernel TL provides an example of this kind.

6.2.4 Communication detection in array assignments - kernels AA, SH, ST, and IR

Once the data and iteration space is distributed, the next step that strongly influences efficiency of the resulting codes is communication detection and code generation to execute data movement. In general, the off-processor data elements must be gathered before execution of an array assignment, and the results are to be scattered to destination processors after the assignment is completed. In other words, some of the array assignments may require a preprocessing phase to determine which off-processor data elements are needed and execute the gather operation. Similarly, they may require postprocessing (scatter). Many different techniques may be used to optimize these operations. To achieve a high efficiency, it may be very important that compiler is able to recognize structured communication patterns, like shift, multicast, etc. Kernels AA, SH, and ST introduce different structured communication patterns, and kernel IR is an example of an array assignment that require unstructured communication (because of indirections).

6.2.5 INDEPENDENT assertion - kernel EP

In addition to array assignments and FORALL statments, parallelism may be expressed by using INDEPENDENT assertions. The EP kernel test performance of INDEPENDENT DO construct with NEW variables.

6.2.6 Non-elemental intrinsic functions - kernel RD

Fortran 90 intrinsics and HPF functions offer yet another way to express parallelism. Kernel RD tests implementation of several reduction functions.

6.2.7 Passing distributed arrays as subprograms' arguments - kernels AS, IT, IM and EI

The last group of kernels, demonstrate passing distributed arrays as subprograms' arguments. They represents four typical cases:

1. a known mapping of the actual argument is to be preserved by the dummy argument (AS).
2. the mapping of the dummy argument is to be inherited from the actual argument, thus no remapping is necessary. The mapping is known at compile time (IT).
3. the mapping of the dummy argument is to be identical to that of the actual argument, but the mapping is not known at the compile time (IM).
4. a specific mapping of the dummy argument is forced, regardless the mapping of the actual elements (EI).

6.3 Summary

The synthetic compiler benchmark suite described here is an addition to the benchmark kernels and applications described in chapter 4 and 5. It is not meant as a tool to evaluate the overall performance of the compiler generated codes. It has been introduced as an aid for compiler developers to address some selected aspect of the HPF compilation process. In the current version, the suite does not comprise a comprehensive sample of HPF codes. Actually, it addresses only the HPF subset. Hopefully, this way, we will contribute to establishment of a systematic compiler benchmarking methodology. We intend to continue our effort to develop a complete, fully representative HPF benchmark suite.

Chapter 7

Conclusions*

The PARKBENCH benchmark suite comprises codes that vary from low-level benchmarks measuring basic machine parameters, through important application kernels, to compact research applications. This hierarchical structure allows information derived from the simpler codes to be used in explaining the performance characteristics of the more complicated codes. Thus the benchmark suite can be used to evaluate performance on a range of levels from simple machine parameters to full applications where effects due to non-parallelisable sections of code, and memory, communication or I/O bottlenecks may become important.

Acknowledgements

Special thanks to all the contributors to this report: David Bailey (NASA Ames Research Center), Michael Berry (University of Tennessee), Jack Dongarra (University of Tennessee/Oak Ridge National Laboratory), Vladimir Getov (University of Southampton), Tom Haupt (Syracuse University), Tony Hey (University of Southampton), Roger Hockney (University of Southampton), and David Walker (Oak Ridge National Laboratory).

The following PARKBENCH participants were instrumental in defining/promoting the effort, attending meetings, and providing helpful comments and suggestions: Ed Brocklehurst (National Physical Laboratory), Koushik Ghosh (Cray Research), Charles Grassl (Cray Research), Ed Kushner (Intel SSD), Brian LaRose (Hewlett Packard), Todd Letsche (University of Tennessee), David Mackay (Intel SSD), Joanne Martin (IBM), Ramesh Natarajan (IBM, Yorktown Heights), Bodo Parady (Sun Microsystems), Robert Pennington (Pittsburgh Supercomputing Center), Philip Tannenbaum (NEC), Pearl Wang (George Mason University/US Geological Survey), and Patrick Worley (Oak Ridge National Laboratory).

---

*assembled by Roger Hockney for whole committee

Bibliography

[1] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated pvm framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–175, April 1993.

[2] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Computer Science Dept. Technical Report CS-93-214, University of Tennessee, Knoxville, Tennessee, November 1993.

[3] *Quantities, Units and Symbols*. The Royal Society, London, 1975.

[4] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT Club Benchmarks: Effective Performance Evaluation of Computers. *Intl. J. Supercomputer Appls.*, 3(3):5–40, 1989.

[5] F. H. McMahon. The Livermore Fortran Kernels test of the Numerical Performance Range. In J. L. Martin, editor, *Performance Evaluation of Supercomputers*, pages 143–186. Elsevier Science B.V., North-Holland, Amsterdam, 1988.

[6] J. Dongarra, T. Rowan, and R. Wade. Software Distribution Using XNETLIB Database Server. Computer Science Dept. Technical Report CS-93-191, University of Tennessee, Knoxville, Tennessee, March 1993.

[7] B. H. LaRose. The Development and Implementation of a Performance Database Server. Computer Science Dept. Technical Report CS-93-195, University of Tennessee, Knoxville, Tennessee, August 1993.

[8] J. J. Dongarra. Performance of various Computers using Standard Linear Equations Software in a Fortran Environment. Computer Science Dept. Technical Report CS-89-85, University of Tennessee, Knoxville, Tennessee, March 1990.

[9] D. Bailey, J. Barton, T. Lasinski, and H. (editors) Simon. The NAS parallel benchmarks. Technical Report 103863, NASA Ames Research Center, Moffett Field, CA 94035, July 1993.

[10] Joseph Uniejewski. SPEC Benchmark Suite: Designed for Today's Advanced Systems. SPEC Newsletter, Fall 1989. Volume 1, Issue 1.

[11] R. W. Hockney. A Framework for Benchmark Analysis. *Supercomputer*, 48(IX-2):9–22, 1992.

[12] C. Addison, J. Allwright, N. Binsted, N. Bishop, B. Carpenter, P. Dalloz, D. Gee, V. Getov, A. Hey, R. Hockney, M. Lemke, J. Merlin, M. Pinches, C. Scott, and I. Wolton. The Genesis Distributed-Memory Benchmarks. Part 1: methodology and general relativity benchmark with results for the SUPRENUM computer. *Concurrency: Practice and Experience*, 5(1):1–22, 1993.

[13] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. *Int. J. of Supercomputer Applications*, 5(3):63 – 73, 1991.

[14] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992.

[15] R. W. Hockney. Performance Parameters and Benchmarking of Supercomputers. *Parallel Computing*, 17:1111–1130, 1991.

[16] A. Friedli, W. Gentzsch, R. Hockney, and A. van der Steen. A European Supercomputer Benchmark Effort. *Supercomputer 34*, VI(6):14–17, 1989.

[17] A. J. G. Hey. The Genesis Distributed-Memory Benchmarks. *Parallel Computing*, 17:1275–1283, 1991.

[18] R. W. Hockney. Super-Computer Architecture. In F. Sumner, editor, *Infotech State of the Art Conference: Future Systems*, pages 277–305. Infotech, Maidenhead, 1977.

[19] Roger W. Hockney and Christopher R. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger, Bristol, 1981.

[20] R. W. Hockney. Characterization of Parallel Computers and Algorithms. *Computer Physics Communications*, 26:285–29, 1982.

[21] R. W. Hockney. Characterizing Computers and Optimizing the FACR(l) Poisson-Solver on Parallel Unicomputers. *IEEE Trans. Comput.*, C32:933–941, 1983.

[22] R. W. Hockney. Parametrization of computer performance. *Parallel Computing*, 5:97–103, 1987.

[23] Roger W. Hockney and Christopher R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms*. Adam Hilger/IOP Publishing, Bristol & Philadelphia, second edition, 1988. Distributed in the USA by IOP Publ. Inc., Public Ledger Bldg., Suite 1035, Independence Square, Philadelphia, PA 19106.

[24] A. J. van der Steen and P. P. M. de Rijk. Guidelines for use of the EuroBen Benchmark. Technical Report TR3, EuroBen, The EuroBen Group, Utrecht, The Netherlands, February 1993.

[25] R. W. Hockney. Synchronization and Communication Overheads on the LCAP Multiple FPS-164 Computer System. *Parallel Computing*, 9:279–290, 1988.

[26] R. W. Hockney and E. A. Carmona. Comparison of Communications on the Intel iPSC/860 and Touchstone Delta. *Parallel Computing*, 18:1067–1072, 1992.

[27] D. Bailey and J. Barton. The NAS Kernel Benchmark Program. Technical Report 86711, NASA Ames Technical Memorandum, 1985.

[28] D. H. Bailey, J. Barton, T. Lasinski, and H. (editors) Simon. The NAS parallel benchmarks. Technical Report RNR-91-02, NASA Ames Research Center, Moffett Field, CA 94035, January 1991.

[29] J. Choi, J. J. Dongarra, and D. W. Walker. The design of scalable software libraries for distributed memory concurrent computers. In *Proceedings of Environment and Tools for Parallel Scientific Computing Workshop, (Saint Hilaire du Touvet, France)*. Elsevier Science Publishers, September 7-8, 1992.

[30] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of Fourth Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginia)*. IEEE Computer Society Press, Los Alamitos, California, October 19-21, 1992.

[31] J. J. Dongarra, R. van de Geijn, and D. Walker. A look at scalable linear algebra libraries. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, pages 372–379. IEEE Press, 1992.

[32] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In IEEE, editor, *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, McLean Virginia*, pages 120–127. IEEE Publishers, October 1992.

[33] G. C. Fox, S. W. Otto, and A. J. G. Hey. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel Computing*, 4:17–31, 1987.

[34] S. Huss-Lederman, E. M. Jacobson, A. Tsao, and G. Zhang. Matrix multiplication on the Intel Touchstone Delta. Technical report, Supercomputing Research Center, 1993. in preparation.

[35] C. Lin and L. Snyder. A matrix product algorithm and its comparative performance on hypercubes. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, pages 190–194. IEEE Press, 1992.

[36] J. Rice. Problems to Test Parallel and Vector Languages. Technical Report CSDTR 516, Purdue University, West Lafayette, Indiana, October 1990.

Appendix A

Compact Applications Submission Form

This appendix gives the form to be completed when submitting a compact application for inclusion in the PARKBENCH suite. For an electronic version of this form send email to `walker@msr.epm.ornl.gov` or obtain a copy from netlib under pbwg (see Chapter 1). The completed form should be emailed to the same address.

Name of Program :

Submitter's Name :
Submitter's Organization :
Submitter's Address :

Submitter's Phone Number :
Submitter's Fax Number :
Submitter's Fax Email :

Cognizant Expert(s) :
CE's Organization :
CE's Address :

CE's Phone Number :
CE's Fax Number :
CE's Fax Email :

Extent and timeliness with which CE is prepared to respond to questions and bug reports from PARKBENCH :

Major Application Field :
Minor Application Field :

Application "pedigree" (origin, history, major ports and modifications) :

May this code be freely distributed (if not specify restrictions) :

Give length in bytes of integers and floating-point numbers that should be used in this application:

Integers: bytes
Floats: bytes

Documentation describing the implementation of the application (at module level, or lower) :

Research papers describing sequential code and/or algorithms :

Research papers describing parallel code and/or algorithms :

Other relevant research papers:

Application available in the following languages (give message passing system used, if applicable, and machines application runs on) :

Total number of lines in source code :
Number of lines excluding comments :
Size in bytes of source code :

List input files (filename, number of lines, size in bytes, and if formatted) :

List output files (filename, number of lines, size in bytes, and if formatted) :

Brief, high-level description of what application does :

Main algorithms used :

Skeleton sketch of application :

Brief description of I/O behavior :

Brief description of load balance behavior :

Describe the data distribution (if appropriate) :

Give parameters of the data distribution (if appropriate) :

Give parameters that determine the problem size :

Give memory as function of problem size :

Give number of floating-point operations as function of problem size :

Give communication overhead as function of problem size and data distribution :

Give three problem sizes, small, medium, and large for which the benchmark should be run (give parameters for problem size, sizes of I/O files, memory required, and number of floating point operations) :

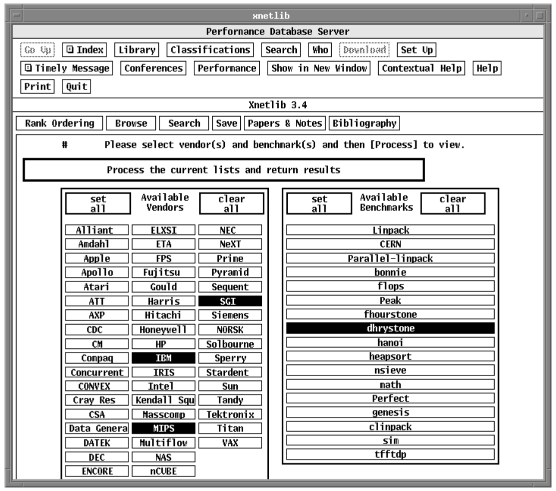
How did you determine the number of floating-point operations (hardware monitor, count by hand, etc.) :

Other relevant information :

Appendix B

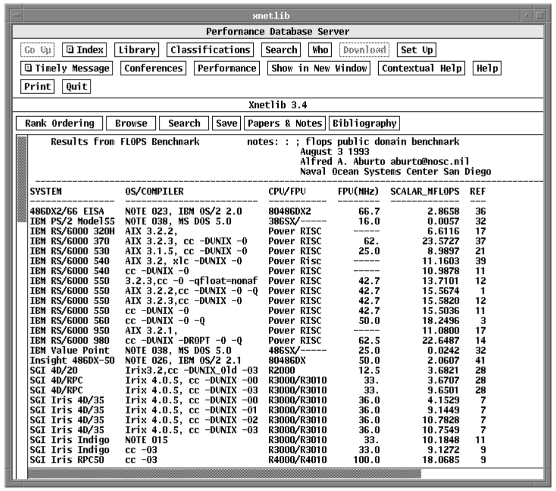
Sample Xnetlib/PDS Screens

With the `Browse` facility in PDS (see Figure B.1), the user first selects the vendor(s) and benchmark(s) of interest, then selects the large `Process` button to query the performance database. The PDS client then opens a socket connection to the server and, using the query language (rdb), remotely queries the database. The format of the returned result is shown in Figure B.2. Notice that the column headings which will vary with each benchmark. The returned data is displayed as an ASCII widget with scrollbars when needed.



```
#       Please select vendor(s) and benchmark(s) and then [Process] to view.
```

Figure B.1: The browse facility provided by PDS



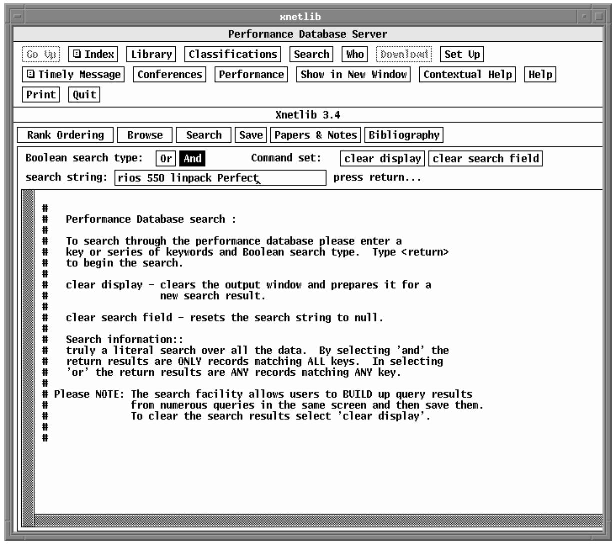
```
    Results from FLOPS Benchmark        notes: : ; flops public domain benchmark
                                                August 3 1993
                                                Alfred A. Aburto aburto@nosc.mil
                                                Naval Ocean Systems Center San Diego
  ---------------------------------------------------------------------------------------
SYSTEM           OS/COMPILER                CPU/FPU      FPU(MHz)  SCALAR_MFLOPS  REF
---------------  -------------------------  -----------  --------  -------------  ---
486DX2/66 EISA   NOTE 023, IBM OS/2 2.0     80486DX2         66.7         2.8658   36
IBM PS/2 Model55 NOTE 038, MS DOS 5.0       386SX/-----      16.0         0.0057   32
IBM RS/6000 320H AIX 3.2.2,                 Power RISC      -----         6.6116   17
IBM RS/6000 370  AIX 3.2.3, cc -DUNIX -O    Power RISC        62.        23.5727   37
IBM RS/6000 530  AIX 3.1.5, cc -DUNIX -O    Power RISC       25.0         8.9897   21
IBM RS/6000 540  AIX 3.2, xlc -DUNIX -O     Power Risc      -----        11.1603   39
IBM RS/6000 540  cc -DUNIX -O               Power RISC      -----        10.9878   11
IBM RS/6000 550  3.2.3,cc -O -qfloat=nomaf  Power RISC       42.7        13.7101   12
IBM RS/6000 550  AIX 3.2.2,cc -DUNIX -O -Q  Power RISC       42.7        15.5674    1
IBM RS/6000 550  AIX 3.2.3,cc -DUNIX -O     Power RISC       42.7        15.5820   12
IBM RS/6000 550  cc -DUNIX -O               Power RISC       42.7        15.5036   11
IBM RS/6000 560  cc -DUNIX -O -Q            Power RISC       50.0        18.2496    3
IBM RS/6000 950  AIX 3.2.1,                 Power RISC      -----        11.0800   17
IBM RS/6000 980  cc -DUNIX -DROPT -O -Q     Power RISC       62.5        22.6487   14
IBM Value Point  NOTE 038, MS DOS 5.0       486SX/-----      25.0         0.0242   32
Insight 486DX-50 NOTE 026, IBM OS/2 2.1     80486DX          50.0         2.0607   41
SGI 4D/20        Irix3.2,cc -DUNIX_Old -O3  R2000            12.5         3.6821   28
SGI 4D/RPC       Irix 4.0.5, cc -DUNIX -O0  R3000/R3010       33.         3.6707   28
SGI 4D/RPC       Irix 4.0.5, cc -DUNIX -O3  R3000/R3010       33.         9.6501   28
SGI Iris 4D/35   Irix 4.0.5, cc -DUNIX -O0  R3000/R3010      36.0         4.1529    7
SGI Iris 4D/35   Irix 4.0.5, cc -DUNIX -O1  R3000/R3010      36.0         9.1449    7
SGI Iris 4D/35   Irix 4.0.5, cc -DUNIX -O2  R3000/R3010      36.0        10.7828    7
SGI Iris 4D/35   Irix 4.0.5, cc -DUNIX -O3  R3000/R3010      36.0        10.7549    7
SGI Iris Indigo  NOTE 015                   R3000/R3010       33.        10.1848   11
SGI Iris Indigo  cc -O3                     R3000/R3010      33.0         9.1272    9
SGI Iris RPC50   cc -O3                     R4000/R4010     100.0        18.0685    9
```

Figure B.2: Sample data returned by the PDS `Browse` facility

The `Search` option in PDS is illustrated in Figures B.3 and B.4. This feature permits user-specified keyword searches over the entire performance database. `Search` utilizes literal case-insensitive matching along with a moderate amount of aliasing. Multiple keywords are permitted, and a Boolean flag is provided for more complicated searches. Notice the selection of the Boolean `And` option in Figure B.3. Using `Search`, the user has the option of entering vendor names, machine aliases, benchmark names, or specific strings, or producing a more complicated Boolean keyword search. The benchmarks returned from the Boolean `And` search

rios 550 linpack Perfect

are shown in Figure B.4. The alias terms *rios 550* are associated with the IBM RS/6000 Model 550 series of workstations. The specification of *linpack* and *Perfect* will limit the search to the LINPACK and Perfect benchmarks only. Since any retrieved data will be displayed to the screen (by default), the `Save` option allows the user to store any retrieved performance data in an ASCII file.

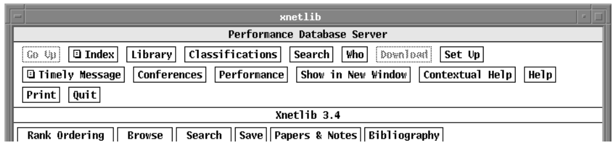


Boolean search type: Or [✓ And]

search string: rios 550 linpack Perfect

```
#
#   Performance Database search :
#
#   To search through the performance database please enter a
#   key or series of keywords and Boolean search type.  Type <return>
#   to begin the search.
#
#   clear display - clears the output window and prepares it for a
#                   new search result.
#
#   clear search field - resets the search string to null.
#
#   Search information::
#   truly a literal search over all the data.  By selecting 'and' the
#   return results are ONLY records matching ALL keys.  In selecting
#   'or' the return results are ANY records matching ANY key.
#
# Please NOTE: The search facility allows users to BUILD up query results
#              from numerous queries in the same screen and then save them.
#              To clear the search results select 'clear display'.
#
#
```

Figure B.3: Specifying a keyword search using the PDS `Search` facility


xnetlib
Performance Database Server
Go Up
☐ Index
Library
Classifications
Search
Who
Download
Set Up
☐ Timely Message
Conferences
Performance
Show in New Window
Contextual Help
Help
Print
Quit
Xnetlib 3.4
Rank Ordering
Browse
Search
Save
Papers & Notes
Bibliography