The Genesis Distributed Memory Benchmarks - I Methodology and General Relativity benchmark with results for the SUPRENUM Computer

Cliff Addison *, James Allwright, Norman Binsted, Nigel Bishop †, Bryan Carpenter, Peter Dalloz, David Gee, Vladimir Getov, Tony Hey, Roger Hockney, Max Lemke ‡, John Merlin, Mark Pinches, Chris Scott, Ivan Wolton

Department of Electronics and Computer Science
Southampton University, Southampton SO9 5NH, UK
email contact: ajgh@ecs.soton.ac.uk

To appear in *Concurrency: Practice and Experience*, John Wiley and Sons

SUMMARY

This is the first of a series of papers on the Genesis distributed-memory benchmarks, which were developed under the European ESPRIT research program. The benchmarks provide a standard reference Fortran77 uniprocessor version, a distributed memory MIMD version, and in some cases a Fortran90 version suitable for SIMD computers. The problems selected all have a scientific origin (mostly from physics or theoretical chemistry), and range from synthetic code fragments designed to measure the basic hardware properties of the computer (especially communication and synchronisation overheads), through commonly used library subroutines, to full application codes. This first paper defines the methodology to be used to analyse the benchmark results, and gives an example of a fully analysed application benchmark from General Relativity (GR1). First suitable absolute performance metrics are carefully defined, then the performance analysis treats the execution time and absolute performance as functions of at least two variables, namely the problem size and the number of processors. The theoretical predictions are compared with, or fitted to, the measured results, and then used to predict (with due caution) how the performance might scale for larger problems and more processors than were actually available during the benchmarking. Benchmark measurements are given primarily for the German SUPRENUM computer, but also for the IBM 3083J, Convex C210, and a Parsys Supernode with 32 T800-20 tansputers.

1 Introduction

During the European Esprit-2 project called Genesis, a number of benchmark programs were developed to evaluate the performance of the German SUPRENUM distributed-memory (DM)

*Liverpool University, Liverpool, UK
†Department of Mathematics, University of South Africa, Pretoria 0001, South Africa
‡Pallas GmbH, 5040 Bruhl, Germany

computer [1] compared to other parallel computers (including multiple-vector and shared memory architectures). Preliminary results for the so-called Genesis pre-study have already been published in outline [2].

The Genesis benchmarks are required to be written in standard Fortran77 except where this is not reasonably practicable (perhaps because of the origin of the code), and both a sequential (base) version and a distributed-memory version are supplied. The normal arithmetic precision is 64-bits, although 32-bit codes may be included if there is good reason. Some of the benchmarks also include a version with explicit vector instructions expressed in Fortran90 syntax [3]. This version will be of particular use in benchmarking SIMD computers, such as the Connection machine, which have adopted this language, and for comparing the performance of different Fortran90 compilers as they become available. Optimisation of the benchmark codes is permitted – indeed encouraged in order to show the full potential of the hardware – provided that the performance of the standard codes is measured first to provide a common point of reference, and the results of both are published. In the case of optimised codes, the steps taken during optimisation should be stated, but a statement of the effort involved is not required. The codes which we report on here in release-2 of the Genesis benchmarks, have been optimised to various degrees for the SUPRENUM computer, but we do not claim that they cannot be improved upon.

The Fortran77 sequential version of the benchmark defines the problem and algorithm to be used, and with minor modifications and tuning should run on vector computers and shared-memory (SM) parallel computers, using the currently available auto-vectorising and auto-parallelising (or auto-tasking) compilers. The distributed-memory version of the benchmark, on the other hand, is a fundamentally different program in which the work to be performed is explicitly distributed to multiple processors which may only communicate with each other by sending messages over communication channels using I/O (send/receive) statements. The translation of the sequential version to an efficient DM version without the help of software tools is usually a non-trivial task measured in man-months of effort. Currently a number of automatic software tools are being developed to ease this task, and it is expected that these will significantly reduce the labour involved, nevertheless significant human programming effort is still likely to be required. The value of the Genesis Benchmarks is that they avoid this work by providing distributed-memory message-passing (DMMP) programs for a number of important problems in physics.

Another important aspect of the Genesis Benchmarks is that they provide an approximate timing model which allows the variation of performance with both the problem size and the number of processors to be estimated. It is believed that this feature of the benchmarks will prove particularly valuable for predicting the performance of future computers, and in understanding how their performance scales.

A reader might well ask what is the justification for introducing yet another suite of benchmarks — after all the PERFECT Club [4], SPEC [5] and Euroben [6] have already established substantial and growing suites of programs covering similar scientific applications. The reason is that none of the benchmarks mentioned above yet provide DMMP programs. Their programs are sequential and suitable for testing single CPUs, or shared-memory multiple CPUs through the use of auto-tasking. They cannot be run on DMMP computers. The only other DM benchmarks known to us are those from the CalTech Performance Evaluation Project (CPEP [7]), which indeed were assembled at about the same time as the Genesis benchmarks and for similar reasons. Our work therefore has some overlap with that of CPEP. Their work covers, at present, a much wider range of computers, but does not provide such a full timing analysis as that given here. At the present early state in the evolution of DM benchmarks we feel there is justification for this and other independent initiatives.

It is not practical to report on the Genesis benchmarks in a single paper of manageable size and, in any case, the benchmarks and results will be produced over a period of time. It is therefore proposed to publish the results as a series of papers entitled "The Genesis Distributed Memory Benchmarks I, II etc.", with a subtitle giving the computers and benchmarks included. This first paper of the series discusses the problems selected, the methodology used (techniques of time measurement, definitions of performance metrics), and gives results and timing analysis for the SUPRENUM computer [1] on the General Relativity (GR1) benchmark. Having established the methodology and style of presentation, subsequent papers will be able to concentrate on reporting benchmark results. Because of their origin, most of the initial results will be for the SUPRENUM computer, however the intention is to obtain benchmark measurements as soon as possible on the new generation of DMMP computers exemplified by the iPSC/860 and Touchstone Delta, the NCUBE2 and 3, and the European i860 based Genesis-P and Transputer-based T9000 GP-MIMD machines.

We hope we have listed as authors on the title page all those who have made a significant contribution to the Genesis benchmark project. Individual papers will also state the authorship of sections so that readers may know who is responsible for the opinions therein which may not necessarily be shared be all members of the group, and who to contact with any queries or criticisms. In this paper, Roger Hockney is responsible for sections 1, 2, 3 and 4.5, and Nigel Bishop for the remaining parts of section 4 on the General Relativity benchmark. Professor Tony Hey is in overall charge of the project.

The Genesis benchmarks are offered as open benchmarks to the parallel computing community, and will be made available as a series of releases over netlib. Enquiries regarding current availability, or problems with the benchmarks, should be sent by email to 'icw@ecs.soton.ac.uk'. In the absence of an accepted programming standard for expressing message passing, there is no way currently of making DMMP programs easily portable across all manufacturers computers. However, our policy is to seek the maximum portability by making available a version of each benchmark using the Argonne/GMD PARMACS communication macros [8], which have now been implemented on a wide range of different computers. In this method, macro statements describing the required communication are expanded into Fortran calls to the native communication primitives by a preprocessor. Currently the PARMACS macro preprocessor is available on the SUPRENUM , Cray Y-MP, Intel iPSC/2 and iPSC/860, nCUBE2, and Meiko Transputer systems and networks of Suns running CS-Tools. In addition implementations are planned for the Alliant FX/2800, networks of IBM RS/6000, and PARSYTEC computers.

Because it is implemented on top of the native communication libraries of the above computers, PARMACS carries with it an overhead of varying severity. Consequently some benchmarks are also available using the native communication facilities of the computers. These mostly use the SUPRENUM extension of Fortran-90 which adds SEND and RECEIVE statements to the language with a syntax similar to the Fortran READ and WRITE, CS-Tools for Meiko systems, or the Intel iPSC communication subroutine library. It is hoped that PARMACS and these native versions will cover most needs, or be relatively easily modified to the requirements of the benchmarker.

2 Problems selected

The problems and algorithms selected for the distributed-memory benchmarks range from kernel measurements to full applications, and were conditioned also by what was readily available to the researchers involved. As a consequence they are primarily research programs developed at Universities for applications in physics and chemistry. The benchmarks are subdivided into three categories according to their complexity, and are:

1. Synthetic code fragments representing frequently used kernels, or designed to measure basic hardware properties:

   COMMS1 unidirectional single message transfer (Pingpong Benchmark)

   COMMS2 bidirectional exchange of two messages (Pingping Benchmark)

   TRANS1 Transpose using naive algorithm

   TRANS2 Transpose using divide and conquer algorithm

   SYNCH1 Barrier synchronisation cost

2. Application kernels, or frequently used library subroutines:

   MATMUL1 Matrix Multiply

   LU1 Matrix LU Factorisation

   QR2 QR decomposition using Householder transformations

   FFT1 One-dimensional FFT

   PDE1 Three-dimensional Poisson solver using parallel red-black relaxation

   PDE2 Two-dimensional multigrid Poisson solver using a range of fine and coarse grids

   QCD2 Conjugate Gradient Kernel from QCD

3. Full application codes:

   HELM1 Meteorology: Psuedo-spectral Helmholtz solver

   HELM2 Meteorology: Multigrid Helmholtz solver

   QCD1 Quantum Chromodynamics (Monte-Carlo)

   MD1 Molecular Dynamics (particle-particle)

   GR1 General Relativity

   LPM1 Local Particle-Mesh simulation (electro-magnetic)

A fundamental aspect of a multi-node DMMP computer is the latency and speed of communication for messages between different nodes of the multi-node network. The success of a DM computer depends on keeping the latency (message startup time) low and the speed (asymptotic transfer bandwidth) high. If this is not done, one can easily find that the time of execution is completely dominated by message communication time, and that the expected arithmetic speed is not realised in practice. The first kernel benchmark (the COMMS1 or Pingpong benchmark) is designed to measure directly the communication speed of the system for the transfer of a single message between a pair of nodes. The results are interpreted in terms of the (r_∞,$n_{\frac{1}{2}}$)parameters [9], which give respectively the asymptotic bandwidth and the message length required to obtain one half of the asymptotic rate. The COMMS2 (or Pingping) benchmark is similar but measures the time to exchange two messages between two nodes, such as occurs frequently when exchanging boundary information during the solution of PDEs by domain decomposition. If bidirectional links are available between the processing nodes, then the exchange can take place faster than the equivalent unidirectional single message transfer measured by COMMS1. Matrix Transpose is included as a further test of communication, because it is an important application in its own right, and loads the communication network heavily with both short and long range messages, without confusing the interpretation with the presence of arithmetic. The two versions relate to a naive algorithm (TRANS1) and one based on a divide-and-conquer strategy (TRANS2). The two formulations lead to different communication patterns.

The global synchronisation of all processors in a highly parallel network of nodes is a costly operation, because all processors must communicate globally with a master process. For this reason, global synchronisation is best avoided in DMMP systems and replaced by the much looser control of local synchronisation between neighbours in the computational network of nodes. However, in practice, some global synchronisation is often found to be either necessary or, at least, the most convenient way of organising a calculation. An example from simulation is the need to ensure that all plotted points on the screen refer to the same timestep when the processors themselves may not be computing exactly in step due to the use of only local synchronisation in the main timestep loop. The most fundamental global synchronisation operation is the *Barrier* statement, which when placed in the program executed by every node, prevents any processor passing the barrier statement until all processors have reached it. Many systems provide special synchronisation hardware or efficiently coded library software to implement a barrier statement, and the SYNCH1 benchmark is designed to measure the effectiveness of these. It measures the maximum number of barrier statements that can be executed per second as a function of the number of processors participating in the barrier.

Matrix manipulations lie at the heart of many scientific codes, and their efficient implementation on parallel computers is an essential requirement for the success of any parallel architecture. The Genesis suite provides four benchmarks to test this, namely matrix-multiplication (MATMUL1), LU decomposition (LU1), and Householder QR factorisation (QR2). Many scientific algorithms are based on the use of the Fast Fourier Transform (FFT), and this also is an algorithm that requires both short and long range communication across the physical system under investigation. In fact, the reason that FFT based algorithms are more effective in solving some classes of partial differential equations than e.g. relaxation methods that require no long-range communication, is precisely that the FFT itself provides such communication. Similarly, hypercubes are favoured networks for DM computers precisely because they provide in hardware the long-range connections needed by the FFT. The performance of the FFT is therefore also a good test of the communications and also of the balance of arithmetic and communication performance of the computer. Two further kernel applications concern the solution of Poisson's equation by iterative methods. The first, PDE1, uses red-black or odd-even relaxation, and the second, PDE2, uses a multigrid iteration. The conjugate gradient algorithm (CG) for the iterative solution of a set of linear equations, also appears at the heart of many physics programs, and this has been selected as another important kernel to test performance. The benchmark QCD2 is a CG kernel taken from a quantum chromodynamics program.

The remaining six benchmarks are complete application codes from meteorology (HELM1 and HELM2), quantum chromodynamics (QCD1), molecular dynamics (MD1), general relativity (GR1), and electronic device simulation (LPM1). QCD1 and GR1 have been chosen because they are ideally suited for execution on a DMMP computer, and should enable such computers to display their best performance. This is because their solution requires only short-range communication between immediate neighbours on a multi-dimensional mesh of processors.

The local particle-mesh benchmark (LPM1) is of intermediate difficulty because although the electro-magnetic fields are advanced in time using only near-by values, the electrons are simulated as particles that move through the field mesh. This is also known as a particle-in-cell or PIC-type calculation. If each processor is assigned to separate regions of space in addition to exchanging the boundary field values at each timestep, it is also necessary to detect any particles that are moving between the regions calculated by neighbouring processors, and to exchange their coordinates. Thus the calculation is still local but the exchanges necessary are more complex.

The problems of greatest difficulty for DMMP computers are global problems in which the solution at any place depends on information everywhere else in the system. The MD1, HELM1 and HELM2 benchmarks are of this nature. Molecular Dynamics using direct particle-particle interaction is a global problem because all the atoms in the simulated system interact with each other at every timestep. Thus all coordinates (which are distributed amongst the processors) must be passed to all processors at every step. This clearly is a heavy load on the communication system, and a good test of it. The solution of the Helmholtz equation is also a global problem because, even though the problem is stated as a local differential equation, the *solution* at any point depends on source values from every other mesh point, via the interaction or Green's function. Hence long-distance communication must take place before the solution can be obtained. Note that the global nature of these problems is inherent in their physics, and cannot be avoided.

The above problems are those currently being prepared for inclusion in the Genesis benchmarks, which will be issued as a sequence of releases.

3 Methodology

3.1 Time Measurement

The fundamental measurement made in any benchmark is the time to complete some specified task. All other performance figures (e.g. Mflop/s) are derived from this basic timing measurement. It is a principle of the Genesis Benchmarks that these timing measurements must either be published as tables of numbers with at least three significant decimal figures, or be derivable to the same accuracy from performance tables, using a formula given in the paper. This is to enable other workers to make alternative analyses of the data to full accuracy. With the availability of certain graphical packages on PCs, it has become popular to present benchmark results as bar charts (e.g. the Livermore Loop results are now often shown in this way); however it is our view that such a presentation, whilst it may be pretty, is not very informative because it is impossible to read such charts with any numerical accuracy.

The next decision concerns the definition of time — whether one should use the elapsed time, CPU time, user time, total time, or some other time offered by the computer system. In common with the PERFECT Club we have decided that, where possible, the benchmarks should be run as a sole user on a dedicated system, and that the elapsed time (as would be measured by an external clock-on-the-wall) should be used. Fortunately most DM computers are relatively inexpensive workstation computers, which are intended for dedicated use, so that sole-user access does not usually present a problem.

The so-called CPU or user time (as provided by Unix systems) should not normally be used for time measurement, because these are derived quantities provided by the operating system. The problem with these measurements is that different operating systems may include different components of the total time into them, leading to an unfair comparison between computers. Furthermore in a multi-processor system, the CPU time recorded for a job is usually the *sum* of the CPU time used by all the processors. In this case the CPU time for a parallelised job is always greater than the same job running on a single CPU, because of the extra CPU cycles used by the parallelised version for synchronisation and data communication. Thus the use of CPU time, defined in this way, completely defeats the purpose of parallel processing, and cannot be used to measure its effectiveness.

The objective of parallel processing is to reduce the elapsed time for a job by spreading the work over multiple processors, and it is clear that the correct time to use is the elapsed time measured by a single clock, usually on the master or host processor. The use of a single clock avoids any problems of clock synchronisation between the multiple clocks that exist on a multi-processor system.

In kernel benchmarks such as the measurement of message transfer times, we may be trying to measure startup times of a few microseconds or less. This is most satisfactorily done if the clock used ticks once every clock period (or cycle) of the computer itself, because then the startup time will represent several hundred ticks. Such cycle-timers are available on computers like the Cray series, but generally not on DM workstations, where commonly the clocks provided by Unix tick only every 10ms. Clocks with such a poor resolution may still be used for kernel measurements if a repeat loop (say 10,000 times) is placed around the kernel and within the timing measurement. In this case care must be taken to ensure that an optimising compiler does not remove the loop, after noticing that it does not alter the calculation. This can usually be achieved by placing the working variables in COMMON and making a call to a dummy subroutine within the repeat loop. Since Fortran subroutines are separately compiled, the compiler must assume that the dummy subroutine might alter the working variables through COMMON, and the compiler cannot safely remove the repeat loop. If a repeat loop has to be used, it is important to measure the time taken by the repeat loop control and the call to the dummy subroutine. This can be done by timing the loop with the kernel commented out. This timing overhead must then be subtracted from the measurement taken with the kernel inserted, in order to obtain the time for the repeated kernel alone. In the timing of an application benchmark, the length of the run is likely to be long enough to be timed satisfactory with a 10ms clock without repeating the calculation.

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

3.2 Floating-Point Operation Count

In order to express benchmark performance in millions of floating-point operations per second, it is necessary to assess the number of floating-point operations, $F(N)$, needed to solve the benchmark problem. It is also important to make a clear distinction between the quantity of arithmetic performed and the rate at which these operations are performed. This is best done by a proper definition of units and symbols for the units. The quantity of arithmetic performed, floating-point operation count or work, has the unit symbols 'flop' or 'Mflop'(for 10^6 flop), and the arithmetic performance or rate at which these operations are performed has the unit symbols 'flop/s' or 'Mflop/s'. This distinction is analogous to that made in mechanics between 'work' which has the unit 'joule', and 'power' which has the unit 'joule/s or watt'. For this reason, we abbreviate floating-point operation(s) to 'flop', but never pluralise it with an added 's', in order to avoid any possible confusion with a performance rate (which is written 'flop per second' or 'flop/s'). Note that the 'p' is an integral part of the name of the unit 'flop' and does not mean 'per' which has the standard symbol '/', as in other physical units, e.g. cm/s etc. In this way we follow the normal conventions for naming units in physics as recommended by the Royal Society [10], and avoid the ambiguity which can arise with such notations as Mflops (is this the plural of Mflop or is it a rate 'per second'?), or the popular

MFLOPS and MFLOPs which are similarly open to misinterpretation.

For simple problems (e.g. matrix multiply) it is sufficient to use a theoretical value for the floating-point operation count (in this case $2n^3$ flop, for nxn matrices) obtained by inspection of the code or theoretical consideration of the arithmetic in the algorithm. For more complex problems containing data-dependent conditional statements, an empirical method may have to be used. The sequential version of the benchmark code defines the problem and the algorithm to be used to solve it. Counters can be inserted into this code or a hardware monitor used to count the number of floating-point operations. 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 [11] when the Mflop/s measure was originally defined. These are:

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

We distinguish two types of operation count. The first is the nominal benchmark floating-point operation count, $F_B(N)$, which is found in the above way from the defining Fortran77 sequential code. The other is the actual number of floating-point operations performed by the hardware when executing the distributed multi-node version, $F_H(N)$, which may be greater than the nominal benchmark count, due to the distributed version performing redundant arithmetic operations.

3.3 Performance Metrics

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

3.3.1 Temporal Performance

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

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

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

As we see below, 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.

The good example of the danger of using Mflop/s to judge the speed of an algorithm is given by Shadid and Tuminaro [12], who compare the solution of a convection-diffusion problem by a local Jacobi iterative method, with the solution by a multigrid method which provides long-distance communication via its coarse mesh. The Jacobi method generates 5.6 times the Mflop/s than multigrid, but unfortunately takes 317 times longer to solve the problem! Using the correct metric we see that the multigrid method has 317 times greater temporal performance, and is therefore the best algorithm. The Mflop/s generated are quite simply irrelevant to the question of choosing the best algorithm. 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.

3.3.2 Benchmark Performance

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

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

The units of benchmark performance are Mflop/s (benchmark name), where we include the name of the benchmark in parentheses to emphasise that the performance may depend strongly on the problem being solved, and also to emphasise, for the reasons given in the next paragraph, that benchmark Mflop/s may be nominal (rather than actual) values. In other contexts such performance figures would probably be quoted as examples of the so-called *sustained* performance of a computer. However, 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 the same for 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 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. 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.

3.3.3 Hardware Performance

If we do 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*

$$R_H(N;p) = F_H(N)/T(N;p) \tag{3}$$

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.

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

$$r^* = \frac{fl.pt.pipes/CPU}{clock.period} \times number.CPUs \tag{4}$$

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.

3.3.4 Speedup and Efficiency

Readers will note that none of the performance results is presented in terms of the popular metrics of speedup, efficiency or performance per node. This is deliberate, because all these performance measures are either easily misinterpreted or obscure important effects. The speedup of a benchmark code is defined as the ratio of the p-processor temporal performance to the single-processor temporal performance. It is a very useful and convenient measure if we are concerned with the optimisation of a particular code in isolation on a particular computer, because its value can easily be compared with the maximum possible speedup, namely the number of processors being used. We can thereby assess how much of the potential hardware performance is being utilised.

However problems can arise, and incorrect conclusions can be drawn, if we try to use speedup to compare different algorithms on the same computer, or the same algorithm on different computers. This is because speedup is a relative measure (it is defined as the ratio of two performances), and therefore all knowledge of the absolute performance has been lost. Benchmarking, however, is concerned with the comparison of the absolute performance of computers and algorithms, and therefore the use of a relative measure like speedup is not useful, and can be positively misleading. For example, we do not wish to conclude that a computer with a large number of slow processors and therefore high value of speedup, is faster than another with fewer processors and therefore with a lower speedup, if in fact the reverse is the case, because the processors on the second computer are so much faster. Only by adopting absolute measures of performance with physical units involving inverse time, can one avoid this type of false conclusion.

Speedup is not even useful for comparing the performance of one algorithm with another on the same computer, because it is not necessarily true that the algorithm with the highest speedup executes in the least time. A good example of this problem is given by Cvetanovic, Freedman and Nofsinger [13], who compare a parallel SOR with a parallel ADI algorithm for the solution of a partial differential equation. They find that ADI solves the problem in the least time by a substantial margin (i.e. has the highest temporal performance), but that its speedup is less than that of SOR. It is clear in this case that speedup is telling us nothing about the relative performance of the two algorithms.

One can only be sure that the algorithm with the highest speedup executes in the least time, if the single-processor temporal performance of both algorithms is the same. This is most unlikely because the algorithms, if genuinely different, will almost certainly perform different amounts of computational work. Speedup can be regarded as the performance measured in units of the single processor temporal performance. Therefore, if the single-processor temporal performances of the two algorithms are different, and we compare the speedups of the two algorithms, then we are comparing the performance of the two algorithms measured in different units. This is like comparing the speeds of two cars, one measured in m.p.h. and the other in cm/s. Such a comparison has no validity either for cars or algorithms. Computers and algorithms can only safely be compared in terms of their absolute performance in solving a problem. The most unambiguous measure is the temporal performance, which is the inverse of the time of execution.

The benchmark performance per node might seem to be an attractive metric because it is an absolute measure which can be related directly to the hardware performance of a node. However it has the major defect that it hides the point at which the absolute performance begins to decrease as the number of processors increases. If we plot benchmark performance against number of processors, this point is clearly visible as a maximum, however if the same data is plotted as performance per node, all we see is a very uninteresting monotonically falling line, and the important maximum has disappeared. The efficiency, which is defined as the speedup divided by the number of processors, is doubly condemned because it is both a relative measure and it hides the maximum.

3.3.5 Presentation

In the Genesis benchmarks, the performance is presented both in tabular form and as graphs of $R(N;p)$ for constant N plotted against p on log/log axes. The use of logarithmic axes allows the inclusion of a wide range of processor number and performance on a single graph. Within one benchmark, it also permits the vertical axis to be interpreted as either time or performance by reading the appropriate scale. In this respect we follow the practice of CPEP [7], but we plot the performance the *right-way-up* with the highest performance at the top not the bottom. In such log/log performance graphs, the lines of perfect linear speedup (performance directly proportional to the number of processors) are diagonal straight lines.

4 General Relativity (GR1)

4.1 Physical Problem

While the equations of general relativity are long and complicated, from a conceptual point of view the situation is similar to that of solving the wave equation. Given data describing the initial state of a gravitational field, we would like to calculate the future evolution of the field; this is done by solving a system of hyperbolic partial differential equations.

The specific problem tackled in the benchmark is the axially symmetric characteristic initial value problem (c.i.v.p.)[14, 15, 16]. This means that initial data is given on a light cone and, because light cones are natural constructs in relativity, the equations to be solved in this formulation are comparatively simple. The specific problem solved has a known exact solution, and the output of the program is the difference between the numerically calculated solution and the exact one. Axisymmetry is necessary to ensure that the problem is of manageable size. The full calculation will require of order 10^{10} flop, while for the general (no symmetry) problem about 10^{13} flop would be required. Worldwide, little work has been done on the general problem because of the computing time needed. However, as progress is made towards a teraflop machine there is likely to be a substantial growth of interest in relativity codes, which is why the code was chosen as part of the benchmark suite.

The coordinates used are (u, r, y); u is analogous to time, and r and y are spatial coordinates. At each step in the evolution, $q_0 = dq/du$ (= rate of change of the gravitational field) is calculated. The algorithm is parallel in y. The host process creates the node processes, each at a different value of y, sends initial information and collects the results. Message passing is mainly between nearest neighbours, in calculating y-derivatives. However, messages are also passed to and from the host at every step in the evolution to smooth the data at the I1 r-points nearest to the origin. The code uses double precision throughout.

The (r, y) grid has $(n \times m)$ points. For realistic problems $(n \times m)$ should be at least (100×25). The grid is partitioned in the y-direction on a grid of p processors. For good load balance, m should be a multiple of p. After n_i time-steps, an estimate of the error is output, and after n_r such reports the program terminates.

The input parameters to the code are:

n Number of grid points in r-direction

m Number of grid points in y-direction

p Number of processing nodes

n_i Number of iterations between reports.

n_r Number of reports, i,e, when data is written out to disk

i_1 Number of r-points used in data smoothing

4.2 Program Description

The code was originally written at Southampton University in occam to run on a transputer array. The code therefore used message passing but no vectorisation. During the Genesis project the following versions of the code were developed.

- Fortran 77 for a single node (1MM)
- Fortran 77 for many nodes (2MM)
- Fortran 90 for many nodes (2MM)

The figures in brackets indicate roughly how much effort was required in man-months (MM) to write the codes and implement them on the Suprenum computer. There are approximately 950 lines in the code.

There is an algorithmic difference between the single node and multi-node versions of the code. The computation requires the calculation of many intermediate numbers. In the multi-node versions these are stored as arrays, with each node storing a different section of the array.

Table 1: The number of flop, messages and bytes-transferred in the General Relativity benchmark. The numbers given are the totals for all processors per iteration, and should be multiplied by $n_i n_r$ to give the count for the complete benchmark. Only the coefficient of the leading term is given.

Code version	flop total	Messages number	Message to	Total Byte transferred	Computational Intensity, flop/B
F77					
Host		p	workers	$8p$	
Workers	$922nm$	p	host	$8p$	
		$6p$	workers	$96np$	$9.60m/p$
F90 worker					
vector	$845nm$	Messages	as above		
(length n)					
scalar	$77nm$				

Storing the data in this way permits efficient vectorization. However, this cannot be done for the single node version because of storage space. In a realistic computation the (r, y) grid is, say (200×64). The total number of arrays in the code is 56. This it would require 5.73 Mbyte of storage to run the problem on a single node, which exceeds the capacity of the Suprenum machine. Instead a marching algorithm (step by step in the r-coordinate) can be used and intermediate quantities are not stored as arrays over the whole grid. Thus vectorisation is not worthwhile and single node Fortran 90 code has not been written.

The occam program was originally written by Dr N.T. Bishop, and he has been responsible for most of the Fortran code development, together with Mr P.S. Dalloz, Mr A.C.W. Garratt and Ms Charmaine Viljoen.

4.3 Performance Analysis

There is a host program and a worker program that runs on each node. The host program initiates the tasks, and communicates with the workers at every step of the evolution to prevent instabilities near the origin. The initial data is generated at each worker and data is output, via the host, n_r times.

Table 1 shows the leading term of the operation count for the different versions of the benchmark. We note that, to leading order, there are $922nmn_in_r$ flop; in the Fortran90 version of the code, $845nmn_in_r$ flop are vectorised (92%) and the remaining 8% are performed in scalar mode. For benchmarking purposes the input parameters are set as $i_1 = 10$, $n_i = 10$, $\mathrm{DU} = 0.001$, $n_r = 5$. The grid size $(n \times m)$ is varied. Timings are taken for 50 $(=n_i n_r)$ iterations of the evolution equations. They include the time for calculation and associated message passing, and a very small amount of data output.

The performance of the code on the Suprenum computer may be estimated on the basis of the leading terms given in Table 1, and the following expected performance parameters:

Message startup time	2640 μs
Message stream rate	10 MB/s, where B is the symbol for byte
scalar arithmetic	0.188 Mflop/s
vector arithmetic [9]	
r_∞	6 Mflop/s
$n_{\frac{1}{2}}$	100

During each iteration the host processor sends a message of 8 bytes to each of p workers, and receives a message of 8 bytes from each worker. On the Suprenum these messages are sent over a shared serial bus and their per-byte times, arising from the stream rate, add. The startup times for the messages sent by the host also add because they are all performed on the single host processor. The startup times for the worker-to-host messages, however, are executed in parallel on the p workers, and therefore the total startup time is divided by p. During each iteration, each worker also sends to its logical neighbour 6 messages of length 16 bytes, giving a total of $96p$ bytes. These messages are also sent over the shared bus and the per-byte times add. However the startup times for these worker-to-worker messages are performed in parallel on the p workers, and the total startup time is divided by p. Similarly the arithmetic is performed in parallel on the workers, and the time to perform it is $(1/p)$th of the time that would be taken to perform it on one processor, because each processor is performing $(1/p)$th of the work simultaneously with the others. In short, items that are performed serially on some shared resource (such as bus), or on a single processor, contribute a time calculated directly from the entry in Table 1, whereas those performed in parallel contribute $(1/p)$th of this time. We note, therefore, that the estimated time will be different on other architectures with different overlapping properties. For example, the same benchmark running on a transputer network would be able to execute in parallel the per-byte times of the worker-to-worker messages, because these would occur simultaneously over different links in the network, and not serially over a shared bus, as on the Suprenum. Taking these effects into account, we estimated the time for the benchmark, on the Suprenum, to be

$$T(m,n;p) = \left[a(n)\ \frac{nm}{p} + b\ np + c\ (7+p) \right] n_i n_r \tag{5}$$

where the coefficients a, b, c are:

$$\begin{aligned}
a(n) &= 922/0.188\ \ \mu\text{s} && \text{Fortran77 scalar version} && (6)\\
a(n) &= \left\{ \frac{77}{0.188} + \frac{845}{6}\left(1+\frac{100}{n}\right) \right\}\ \mu\text{s} && \text{Fortran90 vector version} && (7)\\
b &= 96/10\ \ \mu\text{s} && \text{both versions} && (8)\\
c &= 2640\ \ \ \mu\text{s} && \text{both versions} && (9)
\end{aligned}$$

The first term in equation 5 is the time to perform $(1/p)$th of the arithmetic in parallel on each of the p processors, the second term is the time to transfer the total number of bytes between neighbours at the stream rate over the serial bus, and the third term is the time for message startup, which is substantial on the Suprenum. The first term in the parentheses is from the $7p$ startups that are executed in parallel on the workers, and the second term is for the p startups executed serially on the host. In the Fortran90 version, the arithmetic term must be divided into a scalar and vector part and the remainder of the expression is the same.

Table 2: The measured benchmark performance, $R_B(m,n;p)$, (in Mflop/s) of the General Relativity benchmark, with the predicted performance in parentheses. The number of nodes is $p=m$. All figures are total Mflop/s, i.e. not per node.

n	$p=m$				
	8	10	12	14	16
Fortran 77 on many nodes					
101	1.30	1.62	1.94	2.24	2.55
	(1.37)	(1.69)	(2.01)	(2.31)	(2.61)
Fortran 90 on many nodes					
101	5.53	6.71	7.73	8.60	9.24
	(6.37)	(7.50)	(8.50)	(9.40)	(10.21)
151	6.78	8.35	9.93	10.93	12.02
	(7.51)	(8.89)	(10.14)	(11.27)	(12.30)
201	8.25	10.14	12.04	13.76	15.49
	(8.25)	(9.81)	(11.23)	(12.52)	(13.71)
251	8.51	10.54	12.46	14.26	15.99
	(8.77)	(10.46)	(12.00)	(13.42)	(14.72)

The decrease in time (or increase in performance) as the vector length increases is taken into account by the factor in parentheses within $a(n)$.

The predicted benchmark performance can now be calculated, using the floating-point operation count

$$\begin{aligned} F_B(m,n) &= g\ nmn_in_r \qquad (10)\\ \text{where} \qquad g &= 922 \quad flop \end{aligned}$$

and we obtain,

$$\begin{aligned} R_B(m,n;p) &= \frac{F_B(m,n)}{T(m,n;p)} \qquad (11)\\ &= \frac{g}{\left[a(n)\ \frac{1}{p} + b\ \frac{p}{m} + c\ \frac{(7+p)}{nm}\right]} \qquad (12) \end{aligned}$$

4.4 Suprenum Performance

In all cases the single-node Fortran77 performance was measured to be approximately 0.188 Mflop/s. For many nodes, Table 2 compares the actual floating-point performance of the code with that predicted by the formulae above. In the Fortran77 case, the variation of performance with n is not particularly significant or interesting, and we give results only for one value $n = 101$. Considering that the predicted performance is based only on leading terms, the actual and predicted performance correlate reasonably well (within 15%). Figure 1 shows the measured values from Table 2 compared to the theoretical formulae $R_B(p,n;p)$ from equation 12, where we have taken into account the fact that the problem size is scaled-up with the number of processors by setting $m = p$. Thus the lines in figure 1 are not lines of constant problem size, but of constant vector length n. This explains why all the measured results for the Fortran77 version, which is unvectorised, give virtually the same floating-point performance. In this case only scalar instructions are used to execute the loops over n, and scalar loops have the property that performance is independent of loop length n (or put another

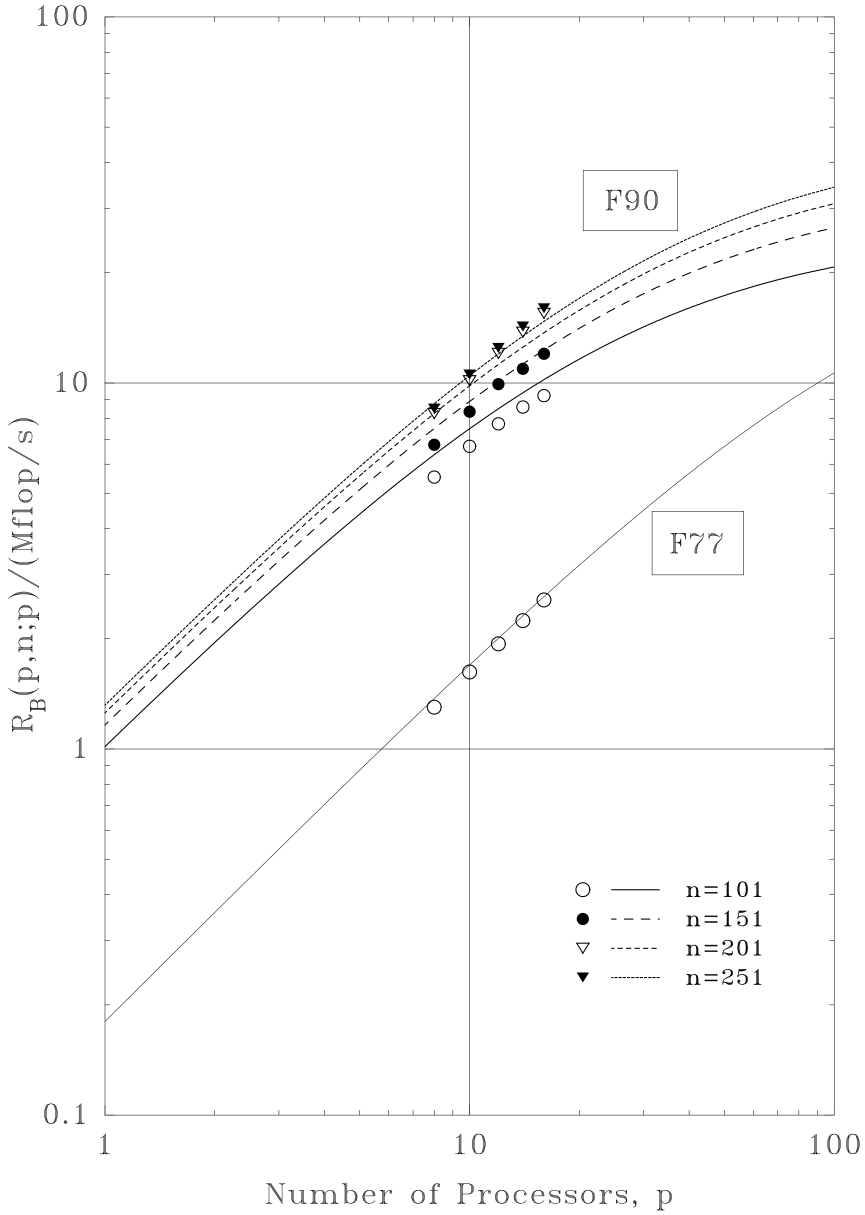

Figure 1: Measured Benchmark performance of the GR1 benchmark, $R_B(p,n;p)$, as a function of the number of processors, p, for different values of vector length, n. The plot symbols are measured values, and the lines are the predicted values from the performance analysis (see text).

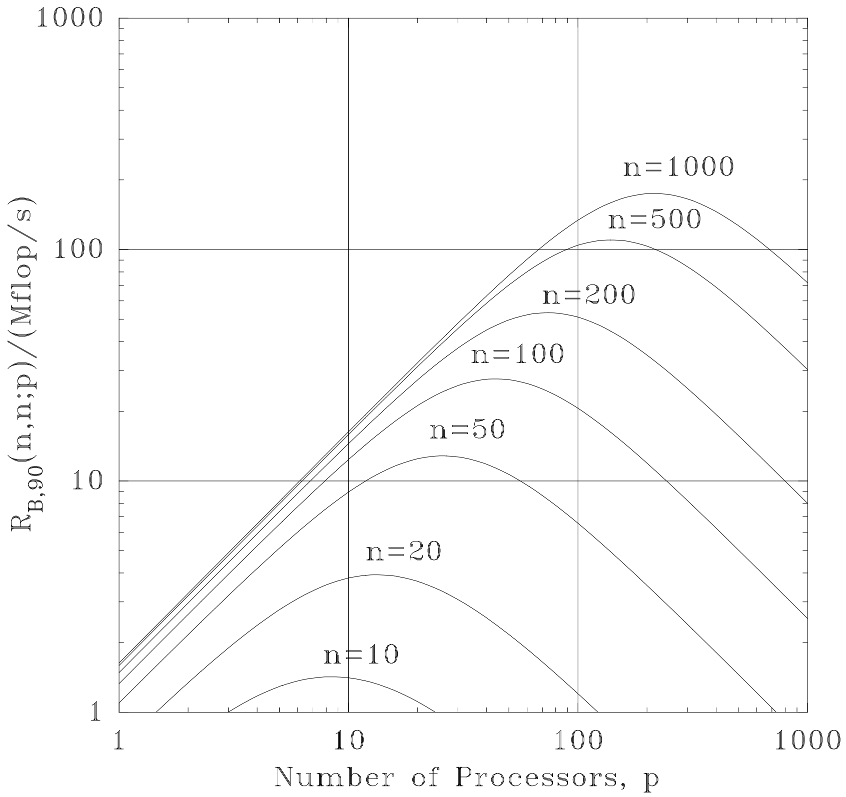

Figure 2: Variation of the theoretical benchmark performance of the Fortran90 version of the GR1 benchmark for an $(n \times n)$ square problem, $R_{B,90}(n,n;p)$, with number of processors p. Each line is for the given constant value of n. Note that, for each value of n, there is a peak in performance, $\tilde{R}_{B,90}(n)$, at an optimum number of processors,$\tilde{p}_{90}(n)$.

way $n_{\frac{1}{2}} \approx 0$ for scalar operations). The variation of performance with n for the Fortran90 version is clear from Fig. 1 and this is primarily a vector-length effect. It is substantial on Suprenum because of the large value of $n_{\frac{1}{2}}$(=100), and the variation is well accounted for by the factor $(1 + n_{\frac{1}{2}}/n)$ in the coefficient $a(n)$ (see equation 7). The computational intensity, $f = 9.6m/p$ flop/B (see Table 1), is the ratio of arithmetic operations performed on a node to data transferred to or from it, so that with the scaling $m = p$, this has the constant value of $f = 9.6$ flop/B for all cases run in the benchmark.

4.5 Theoretical Performance Extrapolations

One of the purposes of a theoretical performance analysis is to permit an extrapolation of the measured performance on existing computers to larger numbers of processors and larger problem sizes than exist at present. We consider that the agreement shown in the last paragraph, validates the timing model reasonably well for the range of variables and parameters used. We will now assume that this model is generally true, and derive a number of useful conclusions about the expected performance of the GR1 benchmark on larger computer systems of a similar type. Whilst such extrapolations must be viewed with caution, because the timing model may well omit important effects that only arise for larger numbers of processors and larger problem sizes than were used in the validation. Nevertheless, we believe that they are worth making, because they increase our understanding of the likely behaviour of the benchmark and the computer system on which it runs.

In particular the timing model is for a bus-connected DM system, like a single cluster of the SUPRENUM computer. Although we extend the theoretical graphs to many hundreds of processors, it must be remembered that the SUPRENUM itself has a maximum of 16 processors in a cluster, after which communication takes place between clusters over token rings. The curves past $p = 16$ therefore do not apply to this computer, but are predictions for a computer with similar parameters but a much larger number of processors per cluster. As has already been mentioned, the timing model would be different for a mesh-connected DM system, like a transputer array, and the predictions below would not apply.

Figure 2 shows the theoretical benchmark performance as a function of number of processors and problem size. In order to limit the number of variables, we specialise the study to a square problem on an $(n \times n)$ mesh (i.e. we take $m = n$, but similar effects would be seen for any other fixed ratio of mesh dimensions). The performance, $R_B(n, n; p)$, is now only a function of the two variables (n, p). In Fig. 2 we do show the dependence as a series of performance curves, each for a fixed problem size n. It is immediately clear that, for constant problem size, there is a sharp peak in performance, after which the performance decreases as the number of processors increases. For each problem size, therefore, there is an optimum number of processors that should be used (which we name $\tilde{p}(n)$), and an optimum or maximum performance (which we name $\tilde{R}(n)$). There is obviously no point in using more processors than $\tilde{p}$ because this would decrease rather than increase the performance. This is an example of a common phenomenon, that whereas the theoretical hardware performance can often be scaled to large values by increasing the number processors, the performance of a fixed size problem usually reaches a maximum at a number of processors which is related to the size of the problem, and not to size (i.e. number of processors) in the computer. This dependence is shown in Figs. 3 and 4 for both the Fortran77 and Fortran90 versions.

By setting the p-derivative of the timing expressions 5 to zero, we can obtain the following theoretical expressions for the optimal values, which apply to the general case of an $(m \times n)$ mesh

$$\tilde{p}(m, n) = \sqrt{\frac{a(n)\ mn}{b\ n + c}} \tag{13}$$

$$\tilde{R}(m, n) = R(m, n; \tilde{p}) \tag{14}$$

For the case of the square problem, for which the graphs are drawn, we abbreviate the arguments, and write for short

$$\begin{aligned} \tilde{p}(n) &= \tilde{p}(n, n) \\ \tilde{R}(n) &= \tilde{R}(n, n) \end{aligned} \tag{15}$$

4.6 Comparison with Other Computers

The GR1 benchmark has been run on three other computers, described below, and performance figures are given in Table 3.

IBM 3083J This is a common mainframe computer, without facilities for running parallel or vectorised code. The computer has a nominal performance of 38.5 Mips, and the

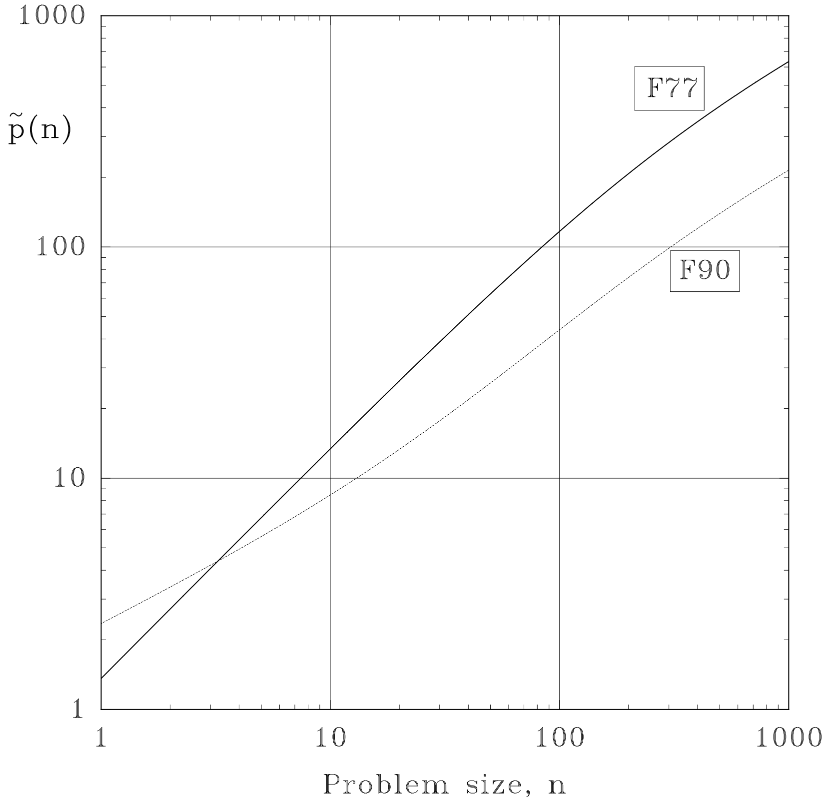

Figure 3: Theoretical variation of the optimum number of processors, $\tilde{p}$, with the problem size n, for the $(n \times n)$ square GR1 benchmark, on a bus-connected DM computer.

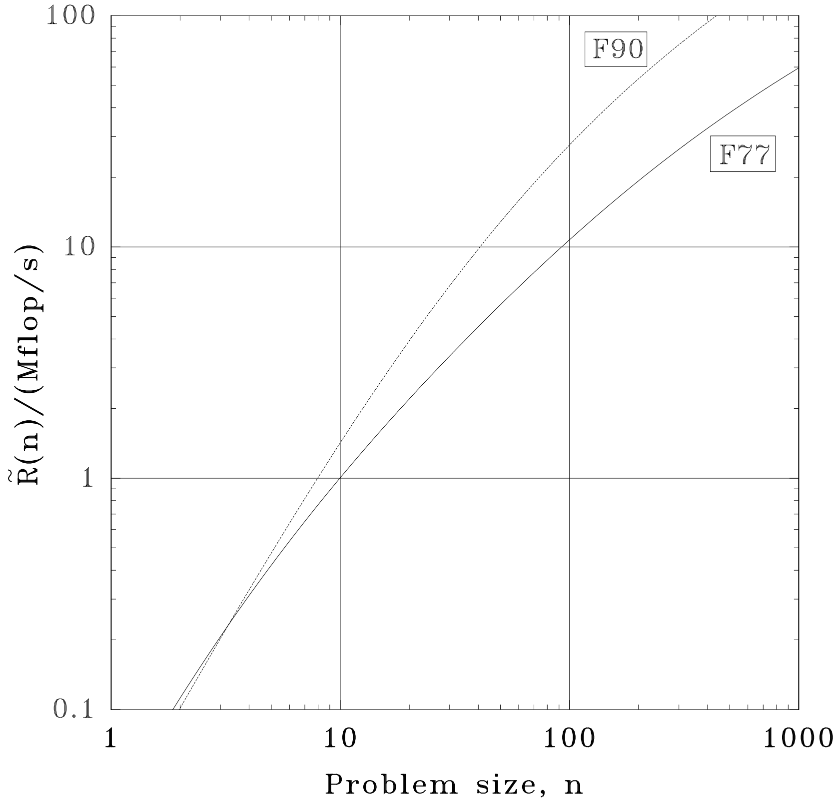

Figure 4: Theoretical variation of the optimum benchmark performance $\tilde{R}_B(n)$, with the problem size n, for the $(n \times n)$ square GR1 benchmark, on a bus-connected DM computer.

Table 3: The measured benchmark performance (in Mflop/s) of the General Relativity benchmark on other computers. The number of nodes is $p = m$. All figures are total Mflop/s, i.e. not per node. The performance is not sensitive to n, whose value is taken here as $n = 101$

COMPUTER	$p = m$					
	8	10	12	14	16	24
IBM 3083J	4.86	4.91	4.88	4.86	4.75	
Convex C210	18.24	22.10	25.86	29.45	31.31	
Parsys Supernode	6.65				13.26	19.65

benchmark was run under the VM/XA operating system. The compiler used was Fortran version 2, release 3, with the optimisation flag set to 3 (the highest level).

Convex C210 This computer runs vectorised, but not parallel, code. It has a nominal performance of 50 Mflop/s, and runs under ConvexOS. The single-node Fortran77 code was compiled using the Convex Fortran compiler, version 6, with the following flags: -O3 (highest level of optimisation), -r1 (to unroll the loops), and -uo (unsafe optimisation, but the results satisfy the accuracy test). This version of the code has vector constructs in the y-direction of length m. Because $n_{\frac{1}{2}}$ is of the order 9 on the Convex, vectorisation is worth while for m in the range 8 to 16. This is not the case for Suprenum, on which $n_{\frac{1}{2}}$ is about 100.

Transputer Array The computer used was a Parsys Supernode with 32 T800-20 transputers and one T414 transputer as host. The code is parallel but not vectorised, and was written in occam. The T800-20 transputer has a peak on-chip register-to-register hardware performance of about 1.5 Mflop/s, but the realised performance with off-chip data is not likely to exceed 1 Mflop/s. We should comment that occam is particularly efficient on transputers, and that equivalent code generated by Fortran compilers would probably run at about one-third the speed.

5 Conclusions

Having setout a methodology for analysing the performance of complex application benchmarks in terms of a small number of parameters, we have shown its application to the General Relativity benchmark. Specific timing and performance formulae have been given for this benchmark showing the variation with both problem size and number of processors. We conclude that the agreement between the predicted and measured performance of about 20 percent or better, is satisfactory for most purposes.

Our approach differs from many others, in that we discuss only performance measured in absolute units (e.g. Mflop/s or timestep/second) and reject as confusing or irrelevant relative measures such as Speedup and Efficiency which dominate most other work. Many previous benchmarks only give spot values of performance in the multidimensional performance space, in that they are defined for a fixed problem size and run on a fixed or very small variation in the number of processors. We believe that, since parallel computer performance is strongly dependent on both problem size and number of processors, $(N; p)$, it is important to treat performance as a function of many variables, and produce predictive formulae showing this dependence. Performance can artificially be treated as a function of a single variable by defining a particular path through the $(N;p)$ plane, and this is done in the definition of 'scaled speedup' where $N \propto p$. However we feel that this hides the complex nature of the performance surface $R(N;p)$, which should be studied properly as a function of many variables (here two). In fact, in our example we find that for every problem size, there is an optimum number of processors to use, and therefore a 'best path' to take through the performance plane (defined by equation 14), which is not the same as the 'scaled speedup' path. This concept of the optimum path may be a new way of viewing benchmark performance results. We conclude from the example in this paper that it is possible to treat performance properly as a function of several independent variables, and to give predictive formulae of acceptable accuracy.

Acknowledgments

The authors wish to acknowledge the help received from the staff at the computer centres used in the benchmarking, in particular those at the SUPRENUM installations at the University of Liverpool, England and at GMD Bonn, Germany. The funding received from the European ESPRIT Genesis research program is also gratefully acknowledged. The figures in the paper were produced with the Jandel Scientific Sigma-Plot4 graphics package run on an IBM PS/2 with output to a laser printer. This package proved especially useful for fitting the timing and performance formulae to the benchmark results, because of its ability to fit a non-linear function with upto 10 independent variables to given data.

References

[1] Proceedings 2nd International SUPRENUM Colloquium, 30 Sept.-2 Oct. 1987, Bonn, FRG. Special Issue (U. Trottenberg ed.), *Parallel Computing*,**7** (3) (1988)

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

[3] M. Metcalf and J. Reid, *Fortran-90 Explained*, Oxford Science Publications/OUP, Oxford and New York, (1990) Chapter 6.

[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, J. Martin, The PERFECT Club benchmarks: effective performance evaluation of computers, *Intl. J. Supercomputer Appls.* **3**(3) (1989) 5-40.

[5] SPEC Benchmarks Suite Release 1.0, *SPEC Newslett.* **2**(3) (1990) 3-4. publ. Systems Performance Evaluation Cooperative, Waterside Associates, Fremont, California.

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

[7] P. Messina, C. Baillie, E. Felten, P. Hipes, R. Williams, A. Alagar, A. Kamrath, R. Leary, W. Pfeiffer, J. Rogers, D. Walker, Benchmarking advanced architecture computers, *Concurrency: Practice and Experience* **2**(3) (1990) 195-255.

[8] L. Bomans, D. Roose and R. Hempel, The Argonne/GMD Macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2, *Parallel Computing* **15** (1990) 119-132.

[9] Hockney R.W., C.R. Jesshope, *Parallel Computers 2: Architecture, Programming and Algorithms*, Adam Hilger/IOP Publishing, Bristol & New York, (1988). (Distributed in USA by American Inst. Phys., c/o AIDC, 64 Depot Road, Colchester, VT 05445).

[10] Symbols Committee of the Royal Society, *Quantities, Units and Symbols* (The Royal Society, London, 1975).

[11] F. H. McMahon, The Livermore Fortran Kernels test of the numerical performance range, in: J. L. Martin,ed., *Performance Evaluation of Supercomputers* (Elsevier Science B.V., North Holland, Amsterdam, 1988) 143-186.

[12] J. N. Shahid,and R. S. Tuminaro, Iterative Methods for Nonsymmetric Systems on MIMD Machines, *Proc. Fifth SIAM Conf. Parallel Processing for Scientific Computing* (1991).

[13] Z. Cvetanovic, E. G. Freedman and C. Nofsinger, Efficient Decomposition and Performance of Parallel PDE, FFT, Monte-Carlo Simulations, Simplex, and Sparse Solvers, *Proc. Supercomputing'90* (IEEE, New York, 1990) 465-474.

[14] N. T. Bishop, C. J. S. Clarke and R. A. d'Inverno, *Classical and Quantum Gravity*,**7** (1990) L23-L27.

[15] R. A. Isaacson, J. S. Welling and J.Winicour, *J. Math. Phys.* **24** (1983) 1824-1834.

[16] J. M. Stewart and H. Friedrich, *Proc. Roy. Soc.* **A384** (1982) 427-454.