Benchmarking for Distributed Memory Parallel Systems: Gaining Insight from Numbers

C. A. Addison[a] V. S. Getov[b] A. J. G. Hey[b] R. W. Hockney[b] I. C. Wolton[b]

[a]Institute of Advanced Scientific Computation
University of Liverpool, Liverpool L69 3BX, U.K.

[b]Department of Electronics and Computer Science
University of Southampton, Southampton SO9 5NH, U.K.

Accepted for publication in *Parallel Computing*

Abstract. The inherent difficulties of benchmarking parallel computers and the motivation behind the development of the GENESIS benchmark suite introduce the paper. Some of the broader issues that must be taken into account with any rigorous benchmarking effort on a distributed memory system are then discussed together with a relevant example. The initial version of the GENESIS benchmarks was used to evaluate the SUPRENUM computer. The later releases of the suite have been developed to achieve portability of codes across different distributed memory systems by means of the PARMACS macros. Strict portability is however infeasible due to the relative immaturity of current distributed memory computers and the presence of so many different architectures. Therefore some aspects of transportable software are also described, which were felt important when porting the benchmarks to different parallel platforms.

1 Introduction

Benchmarking is one of those activities that everyone claims to know something about, but which few people have actually attempted in a methodical and careful manner. Benchmarking generally is hard work and its results can be made meaningless by ill-considered interpretations or comparisons. If the benchmarking is on a distributed memory (DM) system potential difficulties are greatly magnified. Firstly, there is the inherent difficulty of dealing with parallelism. In addition to all of the constraints required to obtain good performance on a single node, a DM system benchmarker must also contend with the fact that there are many parallel processes working cooperatively via message-passing on a system that consists of several self contained nodes connected together by some form of communications network.

To make things even worse, there is little standardisation across systems. Potential users are faced with a bewildering variety of node and communication performance. The differences are sufficiently wide that good performance on one system is not a guarantee of even reasonable performance on another system. Such variations have a doubly negative effect – they oblige users to have a keen interest in cross system comparisons, which is a dangerous game at the best of times, and they make portability, in the sense of ensuring a reasonable degree of performance across platforms with minimal change, extremely difficult to obtain. Therefore, even given carefully optimised single node code, DM system benchmarks may be worthless. This may happen, for example, because:

- logical processes were mapped poorly onto physical nodes, creating avoidable contention on the communications network;
- the benchmark was unnecessarily biased towards a programming model inappropriate for the current target.

Note the words "avoidable" and "unnecessarily". Portability is never far from the centre of difficulties, even with the best of intentions. A range of such examples became evident while developing the GENESIS Benchmark Suite. Several of the original benchmarks exploited "fine grain" levels of parallelism available from transputer systems. This limited their usefulness in assessing the performance of the SUPRENUM system since it was not always clear whether the SUPRENUM programming model was inherently inappropriate to the application or not. Similar difficulties arose later when benchmarkers were given access to new (to them) systems for limited periods of time in which they were to run as many benchmark codes as possible. Under such circumstances, numbers were obtained, but because time pressures precluded being able to address portability properly, little meaningful insight was gained.

In order to obtain a better understanding of the issues that revolve around benchmarking DM systems, it is important that the reasons for the benchmarking exercise are clear at the outset and that the strengths and weaknesses of the benchmark suite are well understood. The usual reasons for benchmarking include:

1. Estimating performance potential of an application on a particular architecture.
2. Exploring the relative strengths and weaknesses of several computer systems.
3. Estimating the porting effort required to move a particular application onto a target system and what steps need to be considered in this port.
4. Identification and analysis of bottlenecks for a particular architecture.

Cynics may also add that benchmarking often provides yet more numbers for vendors to quote and to tune for.

Clearly these reasons extend to DM systems but complications arise. The biggest of these is the fact that few sequential codes run efficiently on DM systems without some redesign. Therefore, users of sequential codes need to first establish what a reasonable parallel potential for their code is, and then to see how well that parallel potential can be realised.

But what is the parallel potential of a code? What really matters to the users? Are they interested in solving the largest possible problem within a fixed amount of time, or a more complicated problem within a fixed amount of time, or in solving a particular problem as quickly as possible? Other additional constraints may be imposed; for instance there may be an established maintenance team for a software product, which means that the vendor is understandably reluctant to consider a parallel code that is too different from the sequential one. There is probably also a limited time window and budget for moving the code onto the parallel platform. This retargeting effort has to be accounted for when pricing a DM system. Some form of performance / price index, where price includes hardware and porting costs, is therefore highly desirable.

Sometimes a theoretical study, together with some testing performed on a single processor environment can lead to a good estimate of the parallel potential of an application and the cost to obtain it. More often than not, however, such studies do not provide enough insight. Results from a previously compiled benchmark suite can improve this insight if some of the benchmark codes are loosely equivalent to important parts of the user's code. The optimisations required in such a suite can also provide an idea of the effort required to realise this potential.

Therefore, an ideal benchmark suite for a DM system would:

- assemble benchmarks for a range of important scientific and engineering problems;
- provide a useful estimate of best performance on a range of platforms;
- identify which code transformations produce best performance gains;
- estimate the relative ease / difficulty of various transformations;
- be easily transported among different DM systems.

It is our belief that such an ideal is extremely hard to obtain.

2 What is benchmarked?

When any code is run on any system, it is essential to ask what (or who) is really being benchmarked. For instance, what sort of optimisations are allowed? If, as with the PERFECT Club [3], the scope is wide open provided that the optimisations are recorded, then in a sense the benchmarkers, their knowledge of the tested system and their knowledge of the code are being assessed along with the code itself.

If optimisations are restricted to compiler options only, as is the case for the LINPACK 100 by 100 problem [7], then the strength of the compiler's runtime library is being tested as much as the code performance. This is reasonable, except that several systems now incorporate sophisticated preprocessors to rework code, and the compiler options needed to optimise a certain benchmark can most kindly be described as convoluted. LINPACK makes extensive use of Level 1 BLAS [19]. Some systems have trip-wires for BLAS-codes, whereby these calls are replaced with in-lined assembler, and any included source code is ignored [7]. Indeed, the SUPRENUM compiler had an intrinsic `daxpy` routine primarily to improve the LINPACK 100 performance. Such specific tuning may mean that the LINPACK 100 performance levels are not attained on similar, or even just larger, problems. Indeed, with the proliferation of hierarchical memory systems, high sustained performance rates may require vector operations (Level 1 BLAS) to be blocked into Level 2 BLAS (matrix-vector) [8] or Level 3 BLAS (matrix-matrix) [9]. This fact was a motivating factor in the LAPACK development [6].

As with the LINPACK 100 benchmark, the SPEC benchmarks [21], are carefully written, portable codes. Users are not allowed to make optimisations to the code, but compiler optimisation is allowed. However, the routines are sufficiently varied to make narrow "SPEC-limited" compiler optimisations impractical. Therefore, because a widely ranging suite of codes is involved, allowing only compiler optimisations can lead to good insight into the performance possible without modifying the code – the traditional definition of portability. Unfortunately, with the range of DM systems currently on the market, taking this narrow view of portability creates as many problems as it solves. A parallel code's portability depends strongly on the mapping used to go from the parallel model to the hardware. Some programming models are wholly unsuitable for particular architectures, so there is a tendency to adapt a model consisting of "lowest common denominator" features. Such a choice means that the portable version of a DM benchmark code may be sub-optimal on all targets.

To illustrate these points, consider the "simple" operation of transposing a square matrix. All commercial DM systems support communication over a 2-D grid, where node (i, j) communicates only with its immediate neighbours. Suppose that the $n \times n$ matrix is logically distributed over a $p \times p$ node grid and that p divides n. Conceptually, the transpose is simple. Node (i, j) just exchanges its block with node (j, i) and both transpose the data they receive. But how is this communication performed? The portable option only allows communication between grid neighbours. Therefore the data between nodes $(1, p)$ and $(p, 1)$ must traverse at least $2 \times (p - 2) + 1$ other nodes while these other nodes are attempting to perform other communications. It is easy to devise an optimal store and forward strategy to handle these communications. This would be extremely portable, but systems that provide a richer range of communications than simple store and forward over a grid would be severely disadvantaged. Matrix transpose is sufficiently regular that nearly any additional communication connectivity could be exploited.

This example arose while evaluating the SUPRENUM system with the GENESIS suite. A 4×4 grid of T800 processors would perform this transpose with a complicated store and forward organisation while a cluster of 16 SUPRENUM nodes could perform this same operation with a single send / receive operation between the appropriate off-diagonal nodes. This difference in underlying hardware and hence, the appropriate programming model had a second effect, which made the T800-based system appear worse than it should have. Since T800 systems have extremely good support for threads, an optimal T800 solution would have employed a parallel thread for each of the four communication engines within the processor. However, such a use of threads is clearly not portable so could not be used in a "portable" benchmark suite. In other words, by attempting to have a code that could run on a range of platforms, there is a danger that all of the systems involved will be shown at a disadvantage.

A similar situation could arise with the current generation of machines. The configuration of the Intel Paragon as a physical grid might lead one to believe that the store and forward solution is the best one to employ. It is not. Again, the regularity of the transpose, the presence of through routing and fast communication chips on the Paragon mean that a carefully crafted communication based on nodes (i, j) and (j, i) communicating directly, with special routing information, would be significantly faster.

We therefore return to the question of what is being benchmarked? The code itself, or how appropriate the programming model is for a particular system, or the cleverness of the benchmarker and of the compiler writer are all possible answers. These may all be valid points of study in a benchmark, but, it must be reiterated, the relative importance of these must be clear before any benchmarking exercise begins.

Despite the potential shift of emphasis onto assessing the benchmarker, we firmly believe that strict portability in the context of DM systems is infeasible. The best that can be hoped for therefore, is transportable software, where some, possibly significant, changes are necessary to a code before good performance can be assured across platforms. Interestingly enough, this was also the conclusion from the LAPACK effort [6], where a range of single processor hierarchical memory systems could only be supported by including a performance parameter whose value was system dependent.

3 The GENESIS benchmark suite – a good start

3.1 An evaluation suite

Many of the comments of the previous section were only partially understood by the GENESIS partners before the project began. However, enough was understood that no-one had any illusions that developing a benchmark suite was going to be easy. A great deal was learned in the exercise despite time pressures that forced some pragmatic solutions. However, because of the considerable degree of collective experience and because there was genuine desire to achieve credible results, the GENESIS benchmark suite and its underlying philosophy can provide a good basis for future benchmark developments.

An initial version of the GENESIS benchmark suite had been developed in a preliminary six-month pre-study [16]. At the outset of the next phase of the GENESIS project, we were charged with the task of evaluating the SUPRENUM computer [20, 1], to verify that it was indeed a DM supercomputer, to identify the system's strengths and weaknesses and to perform some extrapolations as to the performance of the proposed GENESIS machine. We were given about 10 months. There were 8 partners in this part of the GENESIS project so the potential for disaster was considerable. Fortunately this part of the project demonstrated how ESPRIT collaborative efforts can succeed – a synergy was achieved in which the whole was considerably greater than the sum of its parts.

One of the first results of this collaboration was to identify the broad classes of benchmarks required in order to present a comprehensive view of the performance available on a DM system and to make sensible predictions about the performance of future DM systems. These classes include:

- Benchmarks that can exploit almost arbitrary amounts of both fine-grain and coarse-grain parallelism;
- Benchmarks that have proven performance on large, vector machines with small numbers of processors;
- Benchmarks that can exploit large amounts of coarse-grain parallelism but have limited use of fine-grain parallelism;
- Computationally intensive benchmarks with a controllable level of interprocessor communications;
- Computationally intensive benchmarks with a controllable amount of disk I/O;
- Benchmarks that have potentially poor vector / parallel performance;
- Benchmarks that involve extensive communications and data movement with relatively little computation;
- Single node benchmarks that can exploit fine-grain parallelism.

All of the above classes of benchmarks were covered by the GENESIS benchmarks, although time and system difficulties prevented extending certain benchmarks to perform disk I/O. [1]

Consistent with the above classes, the problems and algorithms selected for the DM benchmarks range from kernels to full applications and were inevitably partly constrained by what

[1] This important aspect of benchmarking DM systems has been covered recently in the latest public release of the suite by the IO1 benchmark.

was available to the researchers involved. As a consequence, most of the benchmark codes were primarily research programs developed at Universities for applications in physics and chemistry.

The benchmarks used in the evaluation are given below. Notice that not all of these are present in the current publicly available benchmark suite, whose contents are detailed in [2].

1. Single node benchmarks and synthetic code fragments
   - BLAS 1, 2 and 3
   - LINPACK benchmarks
   - NASA benchmark kernels and Livermore loops
   - Single node FFTs
   - Inter-node communication
   - Matrix transpose
2. Parallel numerical kernels
   - Parallel LU factorisation, triangular solve and matrix-vector product
   - Matrix multiply
   - QR decomposition using Givens or Householder transformations
   - 1-D parallel FFTs
   - 3-D Red-Black Relaxation for Poisson's equation
   - 2-D multigrid Poisson solver
   - CG (Conjugate Gradient) solver for QCD (Quantum Chromodynamics)
3. Compact applications
   - Meteorology: A pseudo-spectral model for the shallow water problem
   - QCD: the kernel of a lattice gauge theory Monte Carlo simulation
   - TOMCATV: a mesh generation program
   - Molecular dynamics
   - General relativity

3.2 Benchmarking guidelines

Recall that performance prediction as well as measurement was an important focus of GENESIS work. It was therefore important that the benchmark suite should reveal the weakest points of a computer's design. It was also important to be able to estimate performance if underlying characteristics were changed to overcome these weaknesses. In effect, we needed to know how sensitive the benchmarks were to changes in hardware characteristics. It was also desirable to estimate performance on machines that were not yet built. The need for timing models for such predictions focused attention on what aspects of a benchmark should be timed and how.

The following guidelines were agreed upon in the evaluation:

- Benchmarks generally should be run on a quiet system with only one user; this includes the host machine if communication with the host is a part of the benchmark. The equivalent of elapsed wall-clock time should be measured.

- When a benchmark represents only a portion of an application code, task creation time must not be included in overall benchmark times. In this case, the total time is defined to be the maximum elapsed time per node measured after an initial synchronisation of all tasks.

- The available clock is often only accurate to a fifth of a millisecond. This means that small events cannot be measured directly. If small events must be timed, then there must be a separate benchmark for these small events that can be run together with the main benchmark.

- Benchmarks should not only measure the time of major modules, but whenever possible, they should also give the number of "useful" floating point operations [2] performed within that time as well. Mega-flop rates can then be determined, which help to measure processor utilisation. Times should be presented along with mega-flop rates because the latter alone are too easily open to abuse.

- Test results must be reproducible so that different evaluators can produce similar results. This has certain implications for routines that use random number generators.

- Two general measures of performance will be used: the performance improvement as the number of processors increases with a fixed problem size and the performance improvement as the problem size increases for a fixed number of processors. Scale-up, the performance when the problem size is scaled proportionately to the number of processors is of secondary importance and can be computed from sets of runs where either the problem size or number of processors is fixed.

The presentation of performance in terms of relative measures, such as speed-up or efficiency, was strongly discouraged as it was felt that such measures can obscure detail.

It was felt that both single node and parallel versions of a benchmark were necessary, even when realistically sized problems could not be run on a single node. The same basic algorithms were to be employed in both, although different optimisations might have been used.

The algorithms for DM benchmarks are often selected for their parallel potential and may not be widely used in single processor environments. In such cases it was agreed that "credibility runs" should be included, which consist of running a single processor version of the relevant benchmark and a compatible version of the "best" single processor algorithm on the same test problem. The machine picked for this credibility run should be sufficiently powerful that realistically sized test problems could be used. Examples of suitable processors for such runs would be single processors of a Cray Y/MP or an IBM ES 9000 – machines that have vector processors and a relatively large amount of main memory.

Several other aspects of the evaluation methodology benefited significantly from the work of the PERFECT Club:

- As mentioned earlier, optimisation is actively encouraged, but the optimisations (and the resulting changes in performance) should be recorded. The idea of recording the time each optimisation took to implement was viewed as desirable, but impractical.

- Optimisations may alter a program's performance so that similar, but not identical, results are obtained. It is therefore important to have some easy way for a tester, who is not the developer, to see if a version is acceptably accurate. If each benchmark has a small set (say fewer than 10) of values that can be compared among different versions of

[2] For instance, redundant floating point operations performed to avoid communications should not be included.

the program, then a reliability test can be made automatically by computing the relative difference of these values when compared with a baseline version. A benchmark run would be acceptable only if these differences were all smaller than pre-defined thresholds.

- Benchmarks must be specified by a low dimensional parameter space. This is necessary to obtain a complete picture of a benchmark's performance on a target machine with a "small" number of test runs. A benchmark would be run at predetermined values within this parameter space in order to obtain a clear picture of how performance changes with varying parameters (such as problem size or number of nodes). As part of the specification, the developer must also indicate the legal ranges for these input parameters, including how these parameter ranges are inter-related.

Despite some severe difficulties with hardware and software, the evaluation task was deemed successful. Several of the lessons learned have been reflected in earlier discussions, in particular the need for a representative multi-processor model. The need to take the application expert off the critical path for optimisation of an application was also not fully appreciated beforehand. This tends to reinforce the need for tuning parameters that are easily accessible by a benchmarker. All the standard sort of catch phrases for maintainable software – modular approach, layered design, good documentation and the like tend to make it easier for optimisations to be made by someone other than the expert.

The evaluation exercise brought single processor performance into sharp focus. A SUPRENUM node was quite coarse grained. The achievable chained vector performance was respectable but the performance ramped slowly. There was also a significant gap between scalar and vector performance and it did not cope well with gather / scatter type operations. These limitations were made worse by the fact that many of the application level codes came from simple serial processor systems rather than vector machines. There was a vectoriser in the SUPRENUM compiler, but it was not as useful on application codes as we needed. As a consequence, several benchmark contributors would not or could not optimise their code to take advantage of what SUPRENUM had to offer.

Consistent with the PERFECT Club methodology, the effort and steps required to obtain optimised performance were often as interesting as the actual performance itself. They provided the basis of some sort of "contortion" metric – how much time and how many code changes are required to obtain certain levels of performance. They also provided the basis for an optimiser's guide for a particular machine. This sort of pragmatic information is invaluable when benchmarkers are dealing with new systems. It should also be useful to users and software developers using the systems in the future.

3.3 Portability of the GENESIS suite

In order to get full potential from the SUPRENUM system, it was necessary to employ Fortran 90 array syntax and SUPRENUM specific communication routines. With the second phase of the GENESIS project the emphasis shifted to a portable programming model based on the PARMACS macros [4], and Fortran 77. The benchmark routines were modified to reflect this change. This was in keeping with the initial desire to have a portable benchmark suite.

While functional portability was now obtained – in that the codes would run on the wide range of systems for which PARMACS was available – this was not sufficient for meaningful comparisons to be made without local optimisations. There were also instances where changes were necessary to the top-levels of the codes because of variations between the way host and node systems were organised at different sites.

Under certain circumstances, the communications options provided under PARMACS were found to be insufficiently rich. For example, it does not provide a non-blocking receive operation. In other instances, some PARMACS operations, such as synchronised communication, could not be implemented efficiently on certain systems.

Consistent with our earlier discussion, some of these difficulties were due to a mismatch in the programming model assumed by PARMACS and that supported efficiently by certain systems. However, even when there was a reasonable match between the operations PARMACS supplied and those which a system supported, additional optimisations were required. These were often necessary because of the wide range of speeds of computation and communication among different systems.

4 What is needed – truly transportable software

4.1 Communication characteristics

Consider the matrix transpose problem. Assume that the programming model includes arbitrary process to process communication and that the target systems support this well. Further, assume that the benchmark code is written so that contention for the communications network is not a difficulty with any of the target systems. This is a convenient, but optimistic assumption.

Nodes (i,j) and (j,i) need to swap matrix areas. How should this be written? One approach might be for (i,j) to send its matrix and then receive, while (j,i) receives the matrix from (i,j) and then sends its matrix. Notice that storage is required to store both parts of the matrix. This means that the maximum sized problem that can be dealt with in the benchmark is half what the memory could allow.

Sending a large buffer full of data at once is biased towards systems with high bandwidth. This bias can be made worse by picking sufficiently large problems. Conversely, large buffers and large problem sizes can effectively mask slow message start-up times. The following example illustrates the point. Assume that communications can be modeled by two parameters: the message start-up time, α and the byte transfer time, β, so the time to communicate n bytes is:

$$t_{comm} = \alpha + \beta \times n. \tag{1}$$

Consider two computers, **A** and **B**, where $\alpha_A = 75$ μs, $\beta_A = 5 \times 10^{-3}$ μs/byte (equivalent to 200 Mbyte/s bandwidth) and $\alpha_B = 10$ μs, $\beta_B = 0.2$ μs/byte (equivalent to 5 Mbyte/s bandwidth). These times roughly match what some new DM systems are able to deliver for communications between arbitrary processes. If a 500×500 double precision matrix, corresponding to 2×10^6 bytes, were to be communicated, then one message on system **A** requires about 0.01 seconds, while it requires about 0.4 seconds on system **B**. However, as the message is broken into more and more pieces, corresponding to using smaller and smaller buffers, the time on system **B** does not rise as rapidly as on system **A**. If the original message was decomposed into 6000 parts, the communication requires roughly the same amount of time on both systems. Table 1 gives the single message time for both systems and the number of messages required before the time on both systems is roughly the same. While greatly simplified, the analysis does show the importance of bandwidth and the potential difficulties benchmarkers face if the bandwidth on target systems varies widely. [3]

Table 1 also shows the importance of low start-up time (α), by the fact that system **A** with a

[3] This was a problem with the SUPRENUM machine when more than 1 cluster of 16 nodes was involved. The bandwidth within a cluster was good but between clusters it was severely limited.

Table 1:

Problem size (bytes)	System **A** Single Message Time (μs)	System **B** Single Message Time (μs)	Number of Partitions for equality
10^6	5.08×10^3	2×10^5	3000
10^5	575	2.001×10^4	300
10^4	125	2.01×10^3	30
10^3	80	210	3
10^2	75.5	30	-
10	75.1	12	-
1	75.0	10.2	-

forty times higher communication bandwidth than system **B**, is actually slower than **B** for all message lengths less than 333 bytes, because system **B** has the shorter message start-up time. This emphasises the fact that improving the bandwidth has little influence on the transfer time for short messages, which can only be improved by a reduction in message start-up time.

Any code submitted for inclusion in a benchmark suite is almost certain to reflect characteristics of the particular system used for development. If it was developed as an application code to solve real problems and not an artificial benchmark, it will have been optimised for that system. If the system had high bandwidth but high message start-up time, as SUPRENUM did, the largest possible message granularity will have been used. However, the benchmark authorisation team must insert a more system neutral communications if the code is to be a "fair" benchmark. The transpose problem has been deliberately chosen because its communications are transparent. This may not be true of a proper application code. Finally, a good benchmark code must work with a range of problem sizes. What were sensible problem sizes when porting the application to the target platform may be too restrictive when viewed as input to a potential benchmark code.

Recall our assumption that contention on the communication network was not an issue. If there is a considerable amount of runtime dependent communications, which can arise when dealing with sparse data structures, for example, contention may be a problem. Alternatively, design decisions may be made for the application that removed contention on the target platform, but which are not necessary for other systems. Bias in such situations is extremely difficult to remove from proper application codes because the communication and computation patterns will tend to be intricately entwined. If bias cannot be removed, it must be clearly documented and the results flagged as possibly being misleading.

4.2 Blocked algorithms for computation and communication

Let us now assume a hierarchical memory model for each node, which implies that exploiting locality of reference is a key aspect of performance on such systems. Recall that optimisations of this sort spurred on the LAPACK project, which attempted to merge the functionality of LINPACK and EISPACK into a single transportable package. Better potential performance was achieved by making more extensive use of the Level 2 (matrix-vector) and Level 3 (matrix-matrix) BLAS.

Level 2 BLAS provide one way of chaining a sequence of Level 1 BLAS, so that once a vector has been loaded into vector memory, several operations can be performed on that vector. These were the optimal BLAS to use on SUPRENUM. Converting from Level 1 to Level 2 BLAS is usually straightforward – a loop of Level 1 BLAS involving the same vector and several other vectors becomes a single Level 2 call, and seldom requires extensive changes to the underlying algorithms.

Notice however, that with matrix-vector operations there are $O(n^2)$ data movements for $O(n^2)$ floating point operations. Such operations do not really help sustain performance in hierarchical memory systems. For that, Level 3 BLAS, which are matrix-matrix operations, have been defined. There are still $O(n^2)$ data movements, but now there are $O(n^3)$ floating point operations, so that, in principle, fast memory caches can be exploited fully, even when those caches are relatively small. Routines that are based on matrix-matrix operations are called blocked algorithms in the literature. The particular block-size used is machine dependent, but typically 64×64 blocks work well on a range of machines. Block size then becomes a tuning parameter.

Exploiting blocking in many applications is a difficult task and consequently optimal performance for applications employing indirect addressing or irregular access patterns is often lower than for applications involving dense matrices and regular access patterns. Efforts are underway to provide a comprehensive suite of support routines for sparse iterative linear equation solvers [12, 15]. If successful, they should provide a convenient basis on which to write reasonably efficient portable code using published standard interfaces.

As discussed earlier, blocking several small messages into one large message can improve performance considerably if the message start-up time is high relative to the bandwidth and the total amount of data being communicated is relatively large. This effect can be quantified by rewriting the message transfer timing formula (1) using Hockney's $(r_\infty, n_{1/2})$ parameters [18]:

$$t_{comm} = r_\infty^{-1}(n + n_{1/2}) \tag{2}$$

where the average transfer rate is

$$r = \frac{r_\infty}{(1 + \frac{n_{1/2}}{n})} \tag{3}$$

This shows that to achieve more than half the asymptotic bandwidth requires messages larger than $n_{1/2}$. In the above example, system **A** has a communication half-performance parameter $n_{1/2} = 15000$ byte, and system **B**, $n_{1/2} = 50$ byte. The vast difference between these two values shows why **A** is a long-message-only system, and why **B** can only be efficiently used for problems with short messages.

4.3 Towards transportable software for DM systems

There is considerable overlap with what is presented here as desirable from the point of view of benchmarking software and what is a sensible approach when developing production software that is intended to be used on more than one DM system. This is hardly accidental – nearly all the same constraints are present. It is therefore useful to look at how these ideas can be realised to develop good quality software for DM systems.

The solution of dense systems of linear equations using Gaussian elimination has deservedly received considerable attention on DM systems. The problem can usually be broken down into two stages. In the first phase, the LU decomposition, a non-singular matrix A is factored into the product of three matrices, namely $A = PLU$, where P is a permutation matrix, L is a (unit) lower triangular matrix and U is upper triangular. The matrix P is present because rows of the matrix are interchanged throughout the factorisation (partial pivoting) to improve the numerical properties of the factorisation. Given this factorisation, the solution to $Ax = b$ can be found by two triangular solves, $Ly = P^T b$, followed by $Ux = y$. Initially, experts felt that the parallelism present was too fine grain to be exploitable on DM systems. This has turned out to be wrong. The breakthrough came with the realisation that the data needed to be "scattered" across nodes in a way that kept as many nodes as possible involved in the factorisation. An effective way to achieve this is to distribute the columns or rows of the matrix in a cyclic style, so that given an n-dimensional matrix and p nodes, labelled from 0 to $p-1$, node i would have columns (rows) $i+1+k \times p$ where $k = 0, 1, 2, ...\lceil n/p \rceil$ [13].

An important aspect to consider in parallel LU decomposition is the number and size of the messages that must be communicated. When the matrix is distributed by columns, it is necessary to distribute (possibly by broadcast) the current pivot row number and the column of multipliers to all nodes. When the matrix is distributed by rows, the partial pivoting requires communication among all nodes to find the maximum eligible element in the pivot column, make the row on which this occurs the pivot row and to then communicate the entries of this row to the other nodes.

On systems where message start-up time is relatively high compared to the time required for floating point operations, there may be advantages in grouping columns (rows) together, doing a block factorisation on the appropriate submatrix and then distributing the necessary lower triangular matrix along with the group of columns (rows). This reduces the number of communications down by a factor of the block size. Blocking also has another advantage mentioned earlier – it can improve floating point performance. This type of distribution is often referred to as block-cyclic. A difficulty with this blocking is that it creates more load imbalance and reduces the potential parallelism across nodes. One way to improve this parallelism (and so make effective use of more nodes) is to treat the nodes as a logical 2-D grid and to partition the block columns of the matrix across the columns of the node grid and the block rows of the matrix across the rows of the node grid. This type of distribution can be viewed as a doubly-blocked cyclic distribution.

LU decompositions based on this distribution have been among the top "parallel" LINPACK performers on DM systems [14], but such codes can serve another purpose [10, 5]. They can be heavily parameterised, so that nearly any variant of LU decomposition can be provided by one code merely by altering the block size and the shape of the logical node grid. Thus, a distribution by rows is realised by using a $p \times 1$ grid. The result is a scalable and very flexible piece of software. It also leads to a transportable benchmark code. Solving dense systems of linear equations is specialised – it is an application kernel, not a full application. Consequently, only some performance questions can be answered and the parallelistion strategies used here are not universally applicable. However, such developments do illustrate a potential direction for DM benchmark suites.

4.4 Software standards for DM systems

One of the lessons learned from the GENESIS work was that presenting users with a consistent programming model is important, but not sufficient for portable software. High-performance parallel programs exist but usually they are machine-dependent. Indeed, the wider range of testing performed on different systems in the second year of GENESIS confirmed that strict portability was not feasible in a benchmark suite.

This is partly due to the relative immaturity of DM systems and the presence of so many different architectures. Such variety means that any programming model is biased so that some baseline codes developed on this programming model will be more highly optimised for some systems than others. Differences in the absolute and relative performance levels of communication and computation also lead to bias and introduce performance losses because one code can not be equally suitable for different DM systems.

Obviously, the choice of an adequate programming model can substantially widen the range of architectures over which a program has a good performance. There is no doubt that such a programming model has to be standard in order to ensure a direct port of a code to any newly-marketed DM system. Since the beginning of 1992 there has been great progress towards two informal programming standards aimed to provide widely acceptable solutions to the problem of portability in DM systems. The specification of High Performance Fortran (HPF) language [17] represents an extension to Fortran-90 to assist its implementation on parallel architectures, particularly for data-parallel computation. It includes directives for expressing data distribution across multiple memories, and concurrent execution features. The Message Passing Interface (MPI) defines a set of library routines for message-passing [11], and is intended to become a widely used standard for writing parallel programs with explicit message-passing, gradually replacing the vendor-specific and other interfaces, which are in use today.

5 Conclusions

We have argued in this paper that meaningful comparisons across DM systems require consistent levels of optimisation on the target systems. This is difficult to attain, but adopting an appropriate programming model – HPF or message-passing – and isolating portability concerns to well defined code regions makes the effort considerably easier. While it is difficult to get full applications to conform to these requirements, work invested in developing such parallel benchmark codes decreases the dependence on application experts.

Documentation for a benchmark suite also needs to define what aspects of a system the suite is able to assess, how this assessment is achieved, and provide guidance to users on how to interpret the results. A major difficulty in benchmarking DM systems is that their salient strengths and weaknesses have been changing rapidly, at least in comparison with the time required to assemble good benchmark codes. Generally, most reliable results will be obtained from kernels rather than full applications simply because they are easier to modify to keep up to date. This means that the development and availability of transportable application codes in DM benchmark suites deserves even more attention than it is receiving at present.

Changes are also beginning to appear in the way in which DM systems can be used. High Performance Fortran holds out the promise of achieving reasonable parallel performance for data-parallel applications from within a sequential programming model, at least for regularly distributed data objects. Several systems also provide support for virtual shared memory on DM systems: whether such systems can offer reasonable performance without detailed attention to locality is as yet unclear. These innovations mean that potential DM users have more questions than ever. Benchmarking that aims to provide insight over mere numbers is one way of giving some of the answers.

References

[1] 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, I. Wolton. The GENESIS Distributed Memory Benchmarks. Part I: Methodology and General Relativity Benchmark with Results for the SUPRENUM Computer, *Concurrency: Practice and Experience*, **5**(1) (1993) 1-22.

[2] C. A. Addison, V. S. Getov, A. J. G. Hey, R. W. Hockney and I. C. Wolton. The GENESIS Distributed-Memory Benchmarks. In: J. Dongarra and W. Gentzsch (Eds.), *Computer Benchmarks*, North-Holland 1993 (to appear).

[3] 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 Supercomputers, *Intl. J. Supercomputer Appls.*, **3**(3) (1989) 5-40.

[4] L. Bomans, D. Roose and R. Hempel. The Argone/GMD Macros in FORTRAN for Portable Parallel Programming and their Implementation on the Intel iPSC/2, *Parallel Computing*, **15**(1-3) (1990) 119-132.

[5] J. Choi, J. J. Dongarra, R. Pozo and D. Walker. *ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers*, LAPACK Working Note 55, Computer Science Technical Report CS-92-181, University of Tennessee, November 1992.

[6] J. Demmel, J. Dongarra and W. Kahan. On Designing Portable High Performance Numerical Libraries. *Proc. of the 14th Biennial Conference on Numerical Analysis*, University of Dundee, June, 1991.

[7] J. J. Dongarra. *Performance of Various Computers Using Standard Linear Equations Software*, Computer Science Technical Report CS-89-85, University of Tennessee, May, 1989.

[8] J. J. Dongarra, J. Du Croz, S. Hammerling and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms, *ACM Trans. Math. Soft.*, **14** (1988), 1-17.

[9] J. J. Dongarra, I. S. Duff, D. C. Sorensen and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, 1991.

[10] J. Dongarra, R. van de Geijn and D. Walker. A Look at Scalable Dense Linear Algebra Libraries. In: J. H. Saltz (Ed.), *Proceedings of the 1992 Scalable High Performance Computing Conference*, IEEE Press, 1992.

[11] J. J. Dongarra, R. Hempel, A. J. G. Hey and D. W. Walker. *A proposal for a user-level, message-passing interface in a distributed memory environment*, Technical Report TM-12231, Oak Ridge National Laboratories, October, 1992.

[12] I. Duff, M. Marrone and G. Radicati. *A proposal for user level sparse BLAS*, SPARKER Working Note 1, CERFACS TR/PA/92/86, November, 1992.

[13] K. Gallivan, M. Heath, E. Ng, J. Ortega, B. Peyton, R. Plemmons, C. Romine, A. Sameh and R. Voigt. *Parallel Algorithms for Matrix Computation*, SIAM, Philidelphia, 1990.

[14] R. A. van de Geijn. *Massively Parallel LINPACK Benchmark on the Intel Touchstone and iPSC/860 Systems*, Computer Sciences Technical Report TR-91-28, University of Texas, August, 1991.

[15] M. A. Heroux. *A Proposal for a Sparse BLAS Toolkit*, SPARKER Working Note 2, CERFACS TR/PA/92/90, December, 1992.

[16] A. J. G. Hey. The GENESIS Distributed-Memory Benchmarks, *Parallel Computing*, **17**(10-11) (1991) 1275-1283.

[17] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, Technical report CRPC-TR-92225, Rice University, Houston, May, 1993.

[18] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms*, IOP Publishing/Adam Hilger, Bristol and Philadelphia, 1988.

[19] C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh. Basic linear algebra subprograms for Fortran usage, *ACM Trans. Math. Soft.*, **5** (1979) 308-323.

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

[21] SPEC Benchmarks Suite Release 1.1, *SPEC Newsletter*, **4**(4), pp. 6-8, Systems Performance Evaluation Cooperation, Waterside Associates, Fremont, California.