D.U. 070 Ext. 5079
Copies 12 pages 35

U.S. DEPARTMENT OF COMMERCE • National Bureau of Standards

NBSIR 87-3580

Institute for Computer Sciences and Technology

On Parallel Processing Benchmarks

Gordon E. Lyon

U. S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Institute for Computer Sciences and Technology
Advanced Systems Division
Gaithersburg, MD 20899

FILE COPY
DO NOT REMOVE

June 1987

CMRF

COMPUTER MEASUREMENT
RESEARCH FACILITY
FOR HIGH PERFORMANCE
PARALLEL COMPUTATION

Partially sponsored by the
Defense Advanced Research Projects Agency
under ARPA order number 5520,
July 23, 1985, and July 28, 1986.

On Parallel Processing Benchmarks

Gordon Lyon

Advanced Systems Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Gaithersburg, MD 20899

Partially sponsored by the
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209
ARPA Order No. 5520, July 23, 1985

U.S. Department of Commerce, Malcolm Baldrige, Secretary

National Bureau of Standards, Ernest Ambler, Director

June 1987

TABLE OF CONTENTS

On Parallel Processing Benchmarks

Gordon Lyon

This is a summary of preliminary work and experiences at NBS on benchmarks for parallel machines. Discussion covers the several roles that benchmarks play, ideal and realistic settings, and quick reviews of several types of benchmark sets. Several conclusions can be drawn, given the rudimentary nature of parallel processor characterization: (i) That the performance metrics be embedded in more comprehensive frameworks that can be appraised and modified as needed; (ii) That one universal framework is beyond reach, since distinct clusters of use are emerging with separate emphases; (iii) That large application benchmarks are most successful when they run well on a machine, and thereby demonstrate compatibility of job and architecture; (iv) That the value of smaller metrics (fragments of code) is more diagnostic and preventive than predictive; small metric sets should encourage the parametric study of architectures and applications, and thus promote both economical hardware enhancement and suitable program design.

Key words: benchmarks; computer performance; frameworks; measurements; metrics; models.

1. Introduction

Applications of parallel processing place an emphasis upon performance which is different than that commonly seen for serial processors in commercial use. Ignoring the fault-tolerant aspect of parallel computation, the user of a concurrent processor quite often wants his program to run *faster* [NRC86]. The implication here is that the system must address, for this one user, *his* problem and provide a satisfactory turnaround service. There is less opportunity for the system to buffer its load among jobs, as is so often assumed in mono-processor throughput characterization and evaluation. This emphasis upon a single job or class of jobs, each running *stand-alone*, means that variance in any general or overall characterization will be higher: No single program can be characterized or predicted with confidence unless it has a

---

National Bureau of Standards Internal Report NBSIR 87-3580.

No recommendation or endorsement, express or otherwise, is given by the National Bureau of Standards or any sponsor for any illustrative items in the text. Partially sponsored by the Defense Advanced Research Projects Agency, 1400 Wilson Boulevard, Arlington, Virginia 22209 under ARPA Order No. 5520, July 23, 1985.

close fit to available metrics. This demands that one understand a program and architecture very well, and implies that claims from tests or benchmarks be carefully qualified.

A benchmark result for some machine is really a statement of confidence in both one's understanding of the application features that the benchmark represents and the manner in which it exercises the machine. When a high degree of uncertainty exists on the meaningfulness of a "measurement", it for practical purposes ceases to quantify anything. It is this problem, the careful and deliberate framing of benchmark questions, that burdens measurements on parallel processor machines. (As a point of illustration, Huss and Pennline examine five illustrative benchmark sets and related problems of interpretation on one machine [HUS87].) While the structures for parallel architectures have become more diverse [OMA85], understanding of the various architectural tradeoffs has not maintained pace. In addition, the once popular and relatively easy *empirical* technique of capturing instruction mix streams does not in any useful way characterize salient features of parallel executions. In particular, there is need to know communication patterns which are not captured in mixes [ETC83]. This failure of a relatively easy, if not always accurate, evaluation tool has forced a closer *analytic* examination of application characteristics.

The question that arises as a thread through any parallel architecture effort is: for a given application, what architectural features are necessary, and in what combinations and strengths? When uniprocessors were dominant the underlying model did not often differ substantially from one machine to another, at least not from the user's perspective. Nonetheless, from a micro-architectural view, at the instruction level, there has always been a clash between, say, a compiler's ideal target (application virtual instructions) and a machine's capabilities (real instruction architecture). This is, and always has been, the difficult problem of code generation in compilers. Code generation requires thorny structural reconciliations between the arrangement of operations that is wanted and that which the hardware offers.

Parallel processing elevates the structural clash to a higher level. Thus an application with a detailed set of interacting parallel pieces may or may not be architecturally suited for a certain parallel machine. The clash is higher level because it transcends instruction-order mismatches; the matching involves partitioning larger application pieces, often process aggregates, among processors, memories, and communication capabilities. It includes patterns of process fetching from memory, communication modes among processes, and levels of the communication traffic.

Ideally, one would like benchmarks that are pertinent, accepted, and simple [ETC83]. Achievement of this laudable goal does not lie in the immediate future, for it demands that the field be in stable evolution. Such is hardly true for parallel computation today. A predictive mechanism rests upon a thorough understanding of the underlying parallel systems and their relation to applications. Prediction from a benchmark set is *harder* than running actual programs, at least in the intellectual sense. It is this fact that allows many machine "measurements" to consist of runs of large, poorly characterized applications. More is said later on the issue.

2. Focus and Role of Benchmarks

There are layers of benchmark focus, with each layer addressing a distinct element of performance. Overlapping does occur. Furthermore, as will be argued later about frameworks for benchmarking, the layers are not independent of each other. Each can reinforce or undermine confidence in other measurements in other layers. Four broad strata are discussed here: applications, algorithms, process mechanisms, and instruction/hardware events.

2.1 Programming Applications

There are usually applications that define clearly the objectives that drive an architectural layout. Without these objectives, a machine has no economic reason for being. Exceptions do occur; an experimental machine may represent an *ad hoc* opportunity that has arisen because of components newly available. However, the *deliberate* commercial design is more the rule.

An application runs, computes answers, and is fully representative of the complexity of the problem it addresses. Unfortunately, application programs are often very large, with organizations and detailed performances not easily understood. Minor changes in data may yield large deviations in performance. Typically, an application job becomes important enough to be used as a benchmark because it may:

1. Compute heavily, and deprive others of machine time.
2. Epitomize a whole class of problems that are important to an organization.
3. Represent an instance whose design elements generalize to other vital methods.
4. Match a test machine architecture, and exercise the machine well.

The weaknesses in a large benchmark lie in the cost of getting it onto a machine, and in obtaining clear interpretations of its results. Outsiders are usually reluctant to invest large blocks of time to learn about other fields of application. Thus outsiders cannot readily extrapolate test results for impact in their field; indeed they may not understand enough of the program to vary meaningfully the input. Similarly, the size of a real application may discourage a detailed analysis of its instruction-level demands. Yet this is often necessary to gain insight, especially when low-level demands vary greatly with program parameter sets.

Thus production-size "codes" ask a lot of anyone who wants to use them to investigate computer performance. To be sure, large application benchmarks are often used to select service computers doing heavy, but fairly stable and well-defined, workloads. However, this use is more of a service for an organization's bureaucracy than it is a contribution to understanding parallel computation.

2.2 Algorithmic Paradigms and Programming Techniques

Algorithms in-the-large can be paradigms for solving whole subsystems. Often they are templates which describe, outside-in, how solutions should be constituted. That is, a description consists of outer layers of code, with the variable (substitution) opportunities occuring within. Examples are given in Figures 1-3; there the user defines the functions G (global), N (neighborhood), or I (independent). In contrast, programming techniques are algorithms-in-the-small, and are usually inside-out in orientation. A file access method is an example. That is, the code is wrapped with the user's application; the view is more of supplying isolated location sections. In any event, neither is an entire solution to some application. Each is an abstracted characteristic of the computation.

Algorithms and programming techniques are related to smaller benchmarks in a very important way. All strive to identify patterns of computation that are very important to large classes of applications. Paradigms and techniques seek improvements in software mechanisms. Benchmarks are similar, but emphasis is given to performance of a fixed, representative method on various machines. In all cases, an assumption is implicit that truly important software mechanisms can be improved or measured independently of their uses, and yet still reflect upon programs in which they are used. That is, pieces in the software are separable into building blocks.

Paradigms and techniques provide a benchmark focus for:

1. Ignoring programming language *as long as it is adequate.*
2. Focusing upon mechanistic details that are *truly important,* independent of a given program. (Parallel prefix and its variations is an example from the field of parallel programming.)
3. Studying mechanisms from an algorithmic viewpoint rather than from narrow implementation improvements. However, one selects alternate approaches on practical grounds. This step emphasizes actual *performance* as measured. While structural soundness (e.g. proof-of-correctness) may be very important, it is secondary to the main argument.
4. Supplying practical software design recommendations based upon actual execution experiences. This may exclude certain elegant theoretical approaches whose deficiences are clearly abundant once tried.
5. Remembering that new machinery technologies may redeem a technique that was once unworkable.

2.3 Process Mechanisms

This level is peculiar to larger-grained MIMD systems which use process-level parallelism. The focus of study at this level is how and at what costs do the various cooperating processes start, run, synchronize and exchange data. Chapter 4 is devoted to an example that explores communication costs. Generally the processes themselves can be *synthetic*, which is to say, need not compute real answers, and so can be simplified greatly. The aspects which must be real are:

1. Synchronization
2. Process creation
3. Memory allocation
4. Context switching
5. Message transmission-reception
6. Shared memory accessing
7. Multiprocessor execution

2.4 Higher-level Language Instruction and Machine Events

The instruction level of performance characterization is very important in the design of machines for commercial production. The approach often has been empirical; large, representative work loads are characterized at the instruction-stream level. Recording tapes store this information, and new designs for machines must perform well against the archives. Now questionable for serial machines, the stream approach is of much diminished utility for parallel architecture. It conveys little sense of the data movement that is of paramount importance in parallel computation [ETC83]. An alternate approach at the instruction-level of a programming language is time-stamping various events and positions in a program. This has been pursued at a number of laboratories, including our own project at NBS [MIN86]. A problem here is the fine-grained nature of the events, their number, and the sensitivity of parallel computation to perturbations. Two examples where instruction level instrumentation is revealing are the shared memory actions LOCK and UNLOCK.

Machine event detail is also critical in verifying that performances characterized at higher levels are accurately based upon actual resource utilizations. Here one may isolate and document the causes of anomalies observed at other benchmark levels. For instance, *if* a communication path is becoming saturated during some parallel processing computation, *then* performance on a machine with significantly faster communication should be much improved provided that the communication path in question is a beneficiary. A weakness with many benchmarking "measurements" is that they are not based upon a confirmed machine model. As a consequence, their conjectural basis is often inaccurate when predicting changes from a major host machine shift. The detail here depends heavily upon the architecture itself. A shared-memory bus architecture will have concerns of processor cache hits, bus and memory-bank contention, and load balancing, among others. A message-passing system's principal concern might be transmission latencies, bandwidth, error recovery and flow control.

It is at the lower levels that full characterization of the machine begins to dominate. Applying some of the insight of G. Amdahl, as interpreted by D. Hillis [HIL85], there are things that should fit together in balanced capacities to define a reasonable performance regime for a machine. The following questions address this issue, and provide a basis for examination of a machine. It is possible, of course, that the machine has some clever twist that renders the questions unsuited; in such cases further explanation by the designer is in order. The questions ask: "What are..."

1. Local and global memory sizes
2. Memory-to-processor bandwidths and significant latencies
3. Processor bandwidths
4. I/O capabilities
5. "Memory move-around" capability and need
6. Processor-to-processor bandwidth and latency

In terms of benchmarking a *machine*, the above are fundamental for parallel architectures. Together they define dimensions of elementary capability and balance of which no one feature is more important than another. For example, raw bandwidth can be very misleading when accompanied by latencies; its usefulness may be considerably less than gross capabilities would indicate, because information exchanges are always delayed in starting. Parallel computations can be quite sensitive to delays in synchronization and other perturbations of information exchange.

2.5 Establishing a Frame of Reference

The problem of a conceptual framework for benchmarking is well recognized. Only through such intellectual simplification can any approach handle the vast horde of application programs and the many machines that they can run on. NRC [86-NRC] has proposed (after ideas of Joanne Martin [SAL86]) five stages in performance evaluation:

1. Determine major application areas and solution techniques
2. Select representative programs covering these
3. Define parameters for models of architecture and the application
4. Define metrics for environment and performance of the models
5. Assess relationship between the computational and architectural models

Of course, the above can be interpreted as "Go skin a bear," something considerably easier to say than to accomplish. Discussing problems in image understanding, certainly a subculture of the overall parallel community, Etchells and Nudd remark:

> *The problem of selecting representative algorithms is itself complicated by the breadth of the field [Image Understanding], the wide range of approaches to any given IU sub-task, and by the fact that there are many areas in which there is not clear consensus as the best algorithm for performing a given task. ..Ideally, what we would like to do is to find the lowest level of program modules with the greatest degree of applicability across the entire range of IU algorithms.*

For a highest level of organization for their image understanding proposal they borrow a six-dimension taxonomy. They also mention that "...we must be particularly careful to not only represent ... application requirements... but to include as well the entire range of processing requirements, as seen by the hardware." This can be read as a requiring a solid grounding of the measurement work, conceptually at the high, applications end, and physically at the machine itself.

Our own NBS work has proceeded to some extent along similar lines. The lowest-level model for our investigations has been the forementioned Amdahl-Hillis formulation. At the highest application level one can establish other dimensions of interest: The example in Chapter 4 addresses process communication. Between the extremes of application and hardware are selected levels, as discussed earlier. Casting the levels in a slightly different form:

Applications. This layer defines requirements, and clarifies *need.*

Algorithms. Here is the organization and detail, along with actual *answers.*

Parallel (process) Structure. This model of patterns and behavior provides parallel computation *abstraction. NB:* This is, of course, for process-oriented systems. Finer grained organizations will need this level relocated as appropriate.

Instructions. The fine detail here provides *isolation* of anomalies. Special hardware may be necessary to get convenient, accurate, non-disruptive measurement results.

Hardware Resource Utilization. The stress, balance, and overall envelope seen here serve to define gross *capacities* and *limitations* of a machine.

2.5.1 The Utility of Each Layer. Discussion has skirted around the issue of where in the layering one should select a benchmark set. There is no pat answer, of course, the focus depending upon need. If the user-community is an applications group with principal interests apart from parallel computing, then it mostly wants a suitably faster machine for bigger jobs. There is a strong hint of this view in the NRC recommendation of "select representative programs.." The common terminology in the computational physics community for these programs is "codes", which are here designated *large codes.* These large codes demonstrate whether a new parallel machine can service some application.

A second view, expounded by Etchells and Nudd, seeks more fundamental building blocks, much as the earlier discussion emphasized the role of programming techniques or code fragments, here designated *small metrics.* These fragments, like a test for blood pressure, are narrow indicators that highlight potential threats. The view with metrics is that of *understanding* parallel computation via fundamental capabilities. Metrics provide looser estimates (if acceptable at all) on expected production performances.

Both large codes and small metrics have strengths (and weaknesses) that provide instructive contrasts. Ineffectiveness of a parallel host can be difficult to interpret for a large code of ten to twenty thousand lines. While the host machine may be incompatible, it can also be that the program is trying to do something in a particularly unsuitable way that could be changed *if one could identify the problem.* Similarly, a series of good showings on small metrics (e.g., the LFK set [MCM86]) indicates that a machine enjoys isolated strengths, but without reliable rules of synthesis for extrapolating the results, predictions for complex computations are uncomfortably loose and conjectural. A small table, below, summarizes some differences in utility given two outcomes from running:

	Nice Speedup	Little Improvement
Large Code	**(+) Indicates application/ architecture compatibility**	*(-) Identifying reasons often bothersome, difficult*
Small Metric	*(-) Rules of extrapolation not general at this time*	**(+) Isolates deficiencies quickly and clearly**

The larger and smaller benchmarks serve complementary roles. If things go well, then the best way to be sure is to run a very representative job. However, if performance is inferior, then metrics are far more convenient as diagnostics. The current state of parallel computing is far too chaotic to place great faith in any general synthesis from metric results, since to do so assumes that one knows how something truly works. With classes of machines emerging, it not clear there is a simple "thing." As for large codes, the best way to handle complications of any magnitude may be simply to set them aside. This recommendation is especially true for "dusty-deck" codes that hardly ever move to different architectures, and whose structures antedate most modern software engineering practices.

2.6 Timings

Almost every benchmark set involves some performance against a clock, the exception being something like a test for accuracy. This section briefly reviews some technical problems in using a clock. Beginning with a naive "calling" and correction for timer overhead, discussion expands to timings that have a random component. The importance of a measurement model is underscored. While sometimes correctable, the perturbing nature of some timing measurements can severely limit applicability: parallel processes are especially sensitive to measurement overhead. Sometimes there are software corrections that are available to resolve at fine detail through indirect means. Under other circumstances, only special hardware instrumentation can resolve with the speed, explicitness or lack of perturbation that is demanded.

2.6.1 Algebraic Expressions, Random Variables. It is sometimes suggested that the following fragment works for calling internal program timer:

```
(i)    { t0=time(); t1=time(); ... <code> ...; t2=time();}
```

If the differences *t2 - t1* and *t1 - t0* were fixed quantities (t2 - t1 for a given instance of *<code>*, of course), the time captured by (i) would be

$$\text{time} = t2 - 2*t1 + t0.$$

The simple algebra subtracts out a fixed overhead. However, for reasons given in the next paragraph, the differences *t2 - t1* and *t1 - t0* are *not* algebraic expressions, but rather, random variables. That is to say, they are defined on any given reference not by fixed values (e.g., 5 - 3), but by probabilities on a range of values, a sample space. An algebraic quantity is invariant with each reference unless its constituents are reassigned; a random variable, which is really a *function,* assumes a new value within its range upon each reference.

2.6.2 Timings Have Random Components. The first point is that computer instructions on modern machines take varying lengths of time. The same fetch and add instruction may on one execution suffer from memory contention and delay, or on another execution retrieve its operand from the fast cache memory. The time depends upon what has gone before. Similarly, a multiply time varies depending upon the significant digits in the operands. The whole complexion of the machine is probabilistic in time; while this variation is hardly noticeable in normal practice, it is there. The law of large numbers simply smooths out macro-level observations.

Resolution, Asynchrony, and More Randomness. A call to *time()* does not start the clock running at the point of call. The clock ticks asynchronously and independently of any readings of its time. This renders the call of *time()* a random variable; depending upon where in the interval between ticks the call starts, the read time will vary. If time() executes slowly relative to the clock ticking, possibilities of variance between immediate calls, as in *t1 - t0,* are fewer. However, on many systems quite the opposite holds. The *tick* is infrequent (e.g., 20 ms, 50ms, 1 s...) and clock reading can be fast (5-15 microseconds). Here individual differences *t1 - t0* will be either 0 or 1, with a mean value depending upon the disparity of speeds. The faster clock reading is relative to the ticking, the harder it will be to "catch" a tick with t1 - t0 and get a non-zero difference. A single sample of *t1 - t0* will certainly not suffice.

2.6.3 Importance of Measurement Models. The establishment of the circumstances of timing is not a distillate from collected data. There is no magical effect to be gained in collecting vast quantities of somewhat-organized measurement data. Proper experiment design requires that one establish as well as possible the structural details, the mechanism, of the measurement apparatus. Only in this way can the measurement model avoid confounding its own mechanism overhead with that of the observed. Without proper planning and structural details the measurements become much diminished in value. Conversely, if measurements as outlined in (ii), below, fail to show any variance, one is alerted to programming error *(defective methodology)* or circuitry failure *(malfunction).* The whole idea is to represent major known measurement circumstances in a tight model that accounts for them. This leaves fewer uncontrolled factors, and thereby enhances measurement accuracy and precision.

A More Complex Measurement Model. Once simple algebra is abandoned, the approach of (i) is misleading--it provides but one sample point whose value may be totally unrepresentative of the average of a number of such samples from the space of values. Consequently, (i), above, must be enclosed in some iterative sampling mechanism, as

(ii)

```
        ntest = <appropriate value>;
        time_sum = 0;
        for (i from 0 while i < ntest step 1)
           {
           t0=time(); t1=time(); ... <code> ...; t2=time();
           time_sum = time_sum + t2 - 2*t1 + t0;
           }
        time_ave = time_sum / ntest;
```

This fragment is still incomplete from a statistical standpoint. The question arises as to what *ntest* (the number of sampling tests) should be. This value depends upon the variance of the timed segments of code and the confidence and tolerance limits one wants for the final *time_ave* [NAT63]. For example, a high degree of confidence requires more samples, as does a tighter interval range. That is to say, measuring *time_ave±c* to 50% confidence is easier than claiming a 95% confidence (that the *true* distribution mean falls into the established interval).

Some extra code or preliminary estimate runs outside of (ii) must be included to establish a suitable value for *ntest*. While not difficult, the careful determination of the number of sampling trials can be tedious. This determination is also part of the measurement method, whether it is explicit code or simply trial runs prior to the true measurement. See [NAT63] for details on such estimates. In addition, the statistical model in [NAT63] used is good, but only approximate. It assumes many factors that yield normal distributions. Modern "distribution-independent" methods of computational statistics could be employed at some greater effort [EFR78]. This seems unnecessary.

An assumption in all of this is that the sampling is from one distribution. Such is not true if the first execution takes longer than subsequent passes because an appropriate machine state (cache, etc.) must be established. An F-ratio test can often be used to separate bimodalities (or worse).

In conclusion, the fragment of (i) is wholly inadequate. There can be a considerable amount of computation and thought beyond what one sees in the instrumentation.

2.6.4 Measurement Perturbation and Parallel Processes. Supervisor calls, *SVCs*, often mentioned in timing examples, are very slow methods of time collection. Usually two decimal orders of magnitude can be saved with some more direct approach if this is available. Direct methods may include a user-accessible direct clock-to-register instruction, or special incorporation of independent run-time collection hardware [MIN86].

Perturbation becomes an especially sensitive issue with concurrent processes. A slow clock call made for instrumentation may inadvertently serve as a point for synchronization. (Printout statements for debugging are also gross perturbations.) Hazardous conditions may thereby vanish during program testing and timing [GAI86], only to surface again upon removal of timer invocations. This problem is the focus of the "Trace" and "Resource" measurement systems that are being developed within our facilities [CAR86]. An eventual objective is to support unperturbed programs that can be nonetheless measured in parallel execution. As an improvement to parallel performance methodology, this goal is very attractive.

3. Practical Aspects

Discussion has covered how the principal role for many benchmark metric collections is to show weaknesses. Such an approach is not deliberately chosen, but rather reflects the fragmentary nature of the understanding of parallel processing; it is an *ad hoc* adaptation to realities of the moment. Thus passing a test well cannot show definitively that an architecture *is* truly suitable for an application, since there are many, many factors to be considered, but failure of a test certainly indicates where caution must be employed. Again, the converse, that of passing easily, is more meaningful (without synthesis rules) if the benchmark is as close to the real program as can be.

Returning to the case of poor performance, suppose for example a host machine cannot handle many short messages, but it can handle fewer ones of medium length. Then all "fine-grained" algorithms are going to be likely sources of problems or impossibilities on the host. Certain applications may have characteristic sets of solution techniques that are dependent upon specific programming features--these applications will be easier to test for than others with complex sets of interactions. In short, if there are well-defined dimensions to the computational needs, the benchmarks can at least check for some minimal capability, without which feasibility is much in doubt. The popularity of LFK, the Livermore FORTRAN Kernels (or "Loops"), a set of twenty-four short but representative computations from physics, owes much to this observation. While much abused in interpretation, the "Loops" are nonetheless extremely useful. The report on the LFK does an excellent job of explaining their designed role [MCM86].

Larger benchmark codes test production capabilities well enough, but for reasons already discussed can be unattractive. The very number of applications is one reason why a full repertoire of larger codes is simply hard to attain. A collection will lack economy of size and thought, and because of this, entail a maintenance burden. And since parallel processing is by definition not "doing the same old thing", there is always a lurking suspicion with old programs that things might be better if only they were rewritten in a more contemporary manner.

3.1 Elements in Realistic Design

3.1.1 Functional Level. The level of design that a benchmark addresses will affect its overall utility to various people. A low-level benchmark is simpler to interpret, gives reproducible results and is easy to code. It may, unfortunately, tell one little about germane parallel interactions. Another weakness in low level benchmarks is that they may reflect either that structure visible to the programmer or that necessarily understood for the target machine to run well; that is, they are close to hardware.

However, discussion has also shown how a full application code is somewhat less than ideal. As Etchells and Nudd have indicated, some middle ground is often a most attractive target.

3.1.2 Collection Membership and Size. Membership selection for benchmark sets varies considerably. Nonetheless, two common and diametric poles exert an influence: (i) The empirical--culling from real application codes those program figurations which arise frequently, and (ii) The axiomatic--working within known organizational relationships which must be satisfied. The Livermore FORTRAN Kernels [MCM86] convey a strong feeling of (i), whereas the example in Chapter 4 is more (ii). Circumstance and choice will always determine some proportional mix of the two.

Ideally, one would like somehow to isolate *all* important programming and algorithmic factors and test for them with measurement benchmarks. This is not possible, of course, since a general set will be very large. Eventually, a method of synthesis from important benchmark "bases" may solve some of the size-of-set limitation. In this ideal case, a set of metrics would cover all pertinent dimensions, with more complex cases being built up from the basis-set. However, at this time, accurate predictions of interaction results are likely to be more difficult than simply writing an *ad hoc* benchmark to measure interactions of interest directly. Parallel processing is simply not that well understood; exactly which facets are major and which are minor is not clear.

3.1.3 Specification Choice. Benchmark design, as with any other design, involves tradeoffs. There is considerable latitude in the description of a benchmark. It may be the most simply-stated question, such as "Is $2^{314959216}-1$ prime?" or it can be as detailed as loadable machine code. The first question, on primality, is a quite portable benchmark, whereas the machine code version is very easily run, *provided that a suitable system is under test.* A narrow target of use allows simplifications and assumptions that must otherwise be accounted for. An example of this accounting is the library system used with a collection of coded benchmarks. If the benchmarks will not always be run with the same support library, then arguments can be made that the library support should be bound to the benchmark set itself. This is laudable, but it means that someone will have to trouble themselves to get or write the library routines, and to maintain them. Furthermore, the library will be inferior for some architectures and implementations.

The question of specifying the benchmark can be resolved at several levels. An English description will provide the most portable, applications-oriented possibility. It is closest to application requirements, and leaves miles of latitude for implementation. Pseudo-code is noticeably more concrete. The opportunity for misinterpretation is much diminished, and the portability of the benchmark is not greatly degraded *on machines similar to the virtual machine assumed in the pseudo-code.* Real code is even more pronounced in its immediacy, the principal variabilities being the quality of the compilation and libraries, yet even here the range of performance is a factor of ten--a whole decimal magnitude. And once again, machine-loadable code removes all compiler uncertainties, but it also precludes opportunities for running on other machines. Such are specification tradeoffs.

3.1.4 Definition of Test, Scoring. Another important facet in benchmarking is in data for benchmark results, and the scoring of the run. More precisely, are all machines asked the same question, and must they give the same answers? What does this question mean? Without standard input data, one is never quite sure that a result is comparable. Many algorithms have wide changes in their execution from what seem to be minor parameter changes. The problem with output is not quite so bad. Here comparabilities are more safely made, so that 1.55556 may be close to 1.6 provided that all the other numbers are similarly related. However, if such discrepancies are not acceptable, then an "question and answer sheet" must be prepared for scoring a benchmark performance.

The allowed parametric range for a metric set is quite dependent upon the use the results must serve. For example, when checking a machine for service suitability in some application, the application determines the range. But this may or may not "stress" the machine in a manner interesting to an architect trying to broaden the regime of performance; from his view, which is naturally more bottom-up, the parameter ranges should uncover interesting and significant transitions in execution performance. Without the points of transition, he has little idea whether there is balanced--uniform-- extra capacity in the machine for the task (capability, job, feature) that the metric represents.

3.1.5 Interpretation. Overall interpretations of a benchmark score are sometimes hard to make. This is one reason that a fuller framework has been proposed. It forces the user to either tacitly agree with the framework setting, or to supply arguments on his reinterpretation of a benchmark's significance. In either case, the results are certainly much more valuable when explained; provision should be made for interpretations of all scores. Otherwise laymen and rascals will draw their own conclusions, and these may *occasionally* be correct! The report on the LFK set is an excellent example of interpretation, although it does follow introduction of the set by roughly seventeen years.

3.2 Some Example Sets

A set of benchmarks can grow in any number of ways: as a random collection; a deliberate design with specialized architectural focus and wide application; an attempt to codify a field of endeavor. Several contemporary sets illustrate these possibilities.

3.2.1 NBS Ad-Hoc Parallel Collection. Over the last year or two, NBS has been the host of a collection of contributed benchmarks. The set is largely in FORTRAN, and is scientific code. As one might imagine, the organization of the set, other than being in an electronic mail system, is not deep. Contributed benchmarks can simply overwhelm a host organization if it must to check and explain all contributions. So a genuine weakness of the set is the lack of any interpretations that might be supplied. The strength of the set seems to be that it is something of substance that can be run. The mail system has sustained a good traffic as interest has picked up. Generally, one can venture that running a variety of applications makes vendors and experimenters feel better. When nothing bad happens, they at least have increased their Bayesian confidence in their architecture. Benchmarks in the set that do not perform well may be ignored, or an expert in the field can be called in. This service, then, is a rough-

and-ready means to sample crudely and easily some representative programs from scientific computing.

3.2.2 Dhrystones (uni-processor). The Dhrystones focus narrowly upon language for systems programming. This may be secondary to parallel architecture, but almost *any* calculation uses enumeration, records and pointer types, if but indirectly through a compiler. With such a narrow focus, the set can run on just about anything. It is a contemporary set of language statement-level metrics. The set certainly has a broad audience. It measures *processor+compiler* efficiency for a typical systems program, which is defined by a statement-mix characterization [RIC87]. Both statement type and data type are accounted for. There is neither floating point nor i/o, and the code, which is synthetic, cannot be optimized well by vector processors. The operating system is not called.

R. Richardson [WEI84,RIC87], commenting over arpa-net "info-micro" on the Dhrystones, suggests their use as follows in selecting a machine:

1. *Use Dhrystones to establish* **processor+compiler** *speed.*
2. *Run other benchmarks for disk bandwidth, multiuser response time, and floating point performance (Whetstone). Add vector and matrix computations if germane.*
3. *Examine costs of purchase, operation, maintenance, and depreciation.*
4. *Try appropriate application benchmarks (large codes) on the narrowed set of machines.*

Note that Richardson's advice is a practical application of the entries in the table-of-utility discussed earlier. He uses the kernel benchmarks to winnow a field, and then suggests larger, fully realistic benchmarks to ensure the soundness of a machine selection.

3.2.3 Vision Understanding. Our NBS effort has only had a mild exposure to the many problems of vision understanding. While the original intent was to comment upon these experiences and impressions here, the effort is being extended into a somewhat longer and distinct study. Hence these notes are in passing, and highly provisional.

A major problem in vision is that the field is very poorly characterized relative to the more mature computational areas such as numerical analysis. One has difficulty getting workers to concur as to just *what* is necessary, or good, or fully wrong. Furthermore, there seem to be at least three layers: a low-level pixel oriented, frame processing stage; a middle range, object-oriented simplification; a top level with knowledge-based methods. In some versions, data flows up, and control flows down the layers. Unfortunately, there are no available machines to test an elaborate layered benchmark, so most researchers evaluate within a given layer. Even then, the tendency is to gravitate toward the pixel layer. Additionally, i/o is a major problem in vision, and has not been addressed very well in benchmarks; one does not just do *one* frame--there can be 1-50 frames per second!

3.2.4 Yet Another Set--With Framework. The NBS explorations have led to a framework for benchmarking which has initially been built around process communication. This is expanded in the next chapter, and is given as an example of how some of the ideas discussed in the first three chapter might be put to use. The framework ranges from an application level to very low level, hardware-dependent events.

4. An Example: Metrics for Process Communication

This chapter demonstrates the embedding of benchmarks in a framework for clearer interpretation. The subject is process communication. The brief preliminary study provides good evidence that no single mode of implementation is completely satisfactory over broad ranges of circumstance, while at the same time demonstrating the utility of an organized, parametric approach to benchmark design.

To illustrate, a *logical ring pipeline* test routine has been run with communication variations that include three types of synchronization, two modes of value transmission, and a spectrum of datum lengths. A sampling of runs on one shared memory machine shows that: (i) with one processor available for each ring node, busy-waiting supports a broad range of message lengths, (ii) interrupts work well for medium to long messages, but context-switching limits short message applications, (iii) polling is sensitive to problem size-- a millisecond misjudgement in polling frequency can be disastrous.

Similar results when interpreted from a fuller collection of benchmarks should assist in establishing compatible matchings of algorithm, programming technique, and architecture, perhaps identifying where extra software design care should be exercised, or highlighting special architectural strengths. A framework and its related benchmarks would provide a quick summary of communication on new parallel MIMD machines.

4.1 Communication

Process communication is absolutely necessary in parallel processing. Without mutual exchanges of data, cooperative computations are precluded. A system's capabilities in this communication exchange have important implications in algorithm design and techniques of programming. The communications programming should match logical needs of algorithms to physical hardware capabilities. Otherwise less than satisfactory performance will be realized.

There are of several aspects of process communication *granularity* on parallel machines. The term is used here very informally as an indication of size or resolution. It conveys a sense of relative scale. Thus a fast synchronization method is fine-grained, and a slower one, say five-second polling, coarse. Similarly, fine-grained transmissions have but a few bytes, whereas coarse messages are long. Granularity is important because it relates both to the cost of writing software and to the cost of the underlying hardware. Generally, coarse-grained MIMD hardware is less expensive. Opposing this, fine-grained software communication is more flexible in accommodating various algorithmic choices. (See [KRU87] for a more ambitious attack on the problem of computational grain.)

The investigations pursue process communication granularity within and across various architectures, although shared memory architecture has been given greatest initial emphasis. However, both shared memory and message passing paradigms are considered overall. *Shared memory* denotes storage mutually accessible by all processors at approximately the same cost; it supports communication *by-reference,* i.e. through pointers. Shared memory is often called *tightly coupled. Message passing* systems have memory attached privately to each processor (at least conceptually), so that processors communicate only through explicit transmissions of whole messages. The coupling in this latter case is then referred to as *loose.* From a programming view, messages are a *by-value* form of communication.

Accommodating both architectures within the same test routine requires advance planning. The code for message passing is designed first. Translation--in an informal sense--is then made to a shared memory version. This direction of designing *from* message passing *to* shared memory is very important. It can be quite difficult to cast some shared memory codes into a suitable message passing equivalent [HIK85].

4.1.1 A general framework. The overall effort is lent cohesion through a two dimensional general framework (see Figure 4) which has been chosen to emphasize communication. The abscissa delineates several broad application modes by communication dependencies, e.g.: computational objects that can be scheduled independently (*a la* radiation transport), locally-dependent calculations (fluids), scattered global dependencies (circuit simulation), and many interdependent global calculations (molecular dynamics). While these modes are vast simplifications, they do present a serviceable organization of application communication needs.

The ordinate depicts a degree of abstraction away from the physical machine. Three or four approximate layers have been identified, although the exact resolution may vary to suit measurement purposes. At the highest level are the applications, classified by their component dependencies. This top level is more of a requirements specification than anything else. Next is an algorithmic layer which establishes the functionality of the *serial process code* and yields the application's detailed communication requirements. Lower yet is a (parallel) process level in which computation loads, communication patterns, and transmission protocols dominate. And beneath this, instruction or machine event measurements should ensure that what is measured by software benchmarks is really the pattern of machine resource utilization that one thinks it is.

The general framework of Figure 4 shows a machine at the bottom. The machine should be well-balanced in its support of the software at higher levels. That is, memory size, memory move-around, and the other machine aspects discussed earlier should be equally stressed for an application [HIL85]. This balance-of-stress is an ideal which is never fully achieved, but one aim of performance measurement is to discover economical approximations.

4.2 An Example within the Framework

The scope just described is very large. To explore sections of the framework, various examples should be tried. Emphasis here will be upon the process communication mechanisms.

Within each of the application modes at the process mechanism level, one can define a logical (but hardly unique) process communications structure. The role of the logical structure is to provide an abstract model of communication divorced from fine details of (i) the original problem, or (ii) algorithmic and coding features not related to process communication. (Communication here denotes both synchronization and data transmission.) Certainly there may be numerous acceptable models for a given application. However, to study process communications, specific examples must be chosen and tested.

One synthetic test might be an example of a parallel computation with little interdependency. For this, a form of distributed event scheduling may be appropriate, the model being a collection of nodes with random communication needs. (A parallel garbage collection might use this organization [LYO86].) Such a benchmark would explore general routing performance and (perhaps) load balancing. A rudimentary version of this routine has been run.

At the other end of the interdependencies spectrum are globally interdependent, homogeneous objects. Here routing can be simple and fixed. The computational loads are similarly less variable. Illustrative results that follow focus upon an example chosen here, using interdependent processes to study communication granularity.

4.2.1 A ring. As mentioned, the selected application mode places interdependent global constraints on calculations. In the sketch of the framework, Figure 4, this corresponds to the right side of the "applications" dimension. Under actual circumstances, not all global constraints are of equal importance, so that certain "regions" can be defined to give approximations. This aspect is ignored here, the results providing measurements for an extreme case.

Ring pipelines are a commonly accepted method of communicating globally among processes that share mutual constraints. A *systolic ring* is also similar, although an SIMD architecture may be implied. However, the target architectures discussed here are MIMD, with process control proceeding more or less independently. (Whether MIMD is truly suited for such ring problems is a point not discussed, although experiments with SIMD architectures might prove eye-opening.)

The ring communication structure is implemented with parameter variations as follows:

A. Synchronization (busy-wait; polling; interrupts)
B. Mode of transmission (by-value; by-reference)
C. Message length (short to long)

Further variation is necessary within the gross parameter selections. For example, polling introduces the notion of *frequency* which must be explored. Computation per datum should be adjustable. In addition, the variance of processing each datum can be set by another parameter.

The synthetic ring benchmark for global dependencies (Figure 5) functions as follows:

1. Each of *n* nodes will originate x messages, and additionally, process all other messages passing by. The number x of messages and their length y are parameters.
2. Each message travels around the ring, it being "processed" synthetically by each node in turn.
3. A message that returns to its origination node is removed from the ring traffic. A new message is sent unless all x have been sent.
4. When all nodes have sent and received all of their messages, the ring of processes is dissolved, and the results are reported.

Communication is asynchronous, with message traffic controlled by a simple form of flow control. This keeps slower nodes from being overrun with messages. Such control is essential on systems that cannot control buffer overflows. Messages are, again for this preliminary study, acknowledged on a one-to-one basis. Thus at most a process (node) will have one waiting message. The acknowledgment parameter is, however, adjustable.

The general arrangement of the ring runs in $\Theta(n^2)$ on a serial machine. This can be seen by doubling the ring size: each message must go twice as far around the ring and hence cause twice as much processing; there are also twice as many messages. Now, an architecture such as hypercube can assign a physical processor to each ring node (process). Here one expects running in $\Theta(n)$, based upon the number of messages each node will see. The distance around the ring is not a major factor for the hypercube because each node corresponds to a real processor.

Actual observation bears out the above. Ignoring startup, hypercube performance holds fairly close to a straight line, whereas a processor-limited shared memory machine begins to slow in a nonlinear fashion. For p processors in a shared memory machine, the deviation from linear appears when the ring size reaches p nodes, assuming that interconnect and memory remain unsaturated, but that the computational load is heavy. Prior to this, each ring node has a processor and the system still has a processor to do its chores. At n=p the ring needs all p processors, but the system also needs one occasionally too. Here time-to-complete begins to take longer than linear, and each additional virtual ring node merely worsens processor contention.

4.3 Sample Results

There are three sets of illustrative data from the ring benchmark. (Most of the discussion concerns a shared memory machine with six processors.) The data plots include (1) observations on handling very short messages, (2) a comparison of interrupts, polling, and busy-waiting on long messages, and (3) an example of the influence of polling frequency. No claim is made that these sets are fundamental to all or any other parallel programming applications. What the results do show is that even so simple a program as the synthetic ring benchmark is quite capable of extracting widely differing performances *on the same machine.* Thus it and others similar in philosophy may be quite useful in quickly characterizing salient parallel performance features. This can only shorten the task of writing programs for a new machine.

4.3.1 Short messages. Message granularity is an important aspect of process communication. Short messages are especially useful if they are well supported, since they ease programming at the algorithmic level. Failure to support very short messages well means that algorithms should compute for longer times before sending (longer) messages. This is not always easy or even possible.

Experiments were conducted sending the same amount of information throughout the ring in varying degrees of "chunking". Long messages were then fewer in number. The times-to-complete in Figure 6 are plotted (log-log scales) against the message length. The overall constraint is

(message length)(number of messages)=CONSTANT*

The results in Figure 6 are for shared memory, by-reference transmission. Essentially, a pointer is passed around. As to be expected, whenever there is a real processor to assign to each node process in the ring, busy-waiting works very well. In Figure 6 one sees numerous time-to-completion curves for rings of two to seven nodes run on a six-processor shared-memory machine. For six or fewer nodes, variation in message length is not nearly as critical for busy-waiting as it is for interrupts. This is because busy-waiting does not incur the latencies of context- switching and system service that interrupts and polling have. Hence very fine grained communication is feasible. But busy-waiting squanders processor capacity, and one expects that once processes exceed processors in number, busy-waiting will be definitely inferior. Indeed, the line BW(7) in Figure 6 (a seven node ring) depicts a much worst performance than that for interrupts, INT(7). Even when the number of processors and ring (node) processes are equal, a slight degradation has set in, for reasons (mentioned earlier) of processor contention. The ring routine can do by-value message transmissions as well as by-reference. However, for very short messages this difference is not so crucial.

Some systems do not perform well on the test of Figure 6. Figure 6-a depicts the performance of an older by-value (message-passing) architecture in which the system's frame size for messages was fairly long. Any attempt to send short messages mostly swamped the communications network: Whole message frames were sent no matter how few bytes were used. If too short transmissions were attempted, the system would crash from communications overload. The sawtooth pattern of Figure 6-a reflects the influence of the message frame, and in fact fits the simple equation

$$\text{time} = a * \lceil x/1024 \rceil / x + b/(x-230) + c, \text{ where}$$

a=system message-frame constant
b=component for algorithmic message actions,
  e.g. wait for free buffer, send ACK,...
c=useful computational time, invariant for
  amount of information overall
x=message length

The divisor "x-230" represents an unreachably small "grain". Attempts to send messages of lengths just above 230 bytes brought the system down.

Drawing from the above experience, one can conclude that the ring benchmark is useful in establishing quickly which message lengths need extra concern in designing algorithms for a system. In addition, the robustness of system communications software is put to test.

4.3.2 Interrupts, polling and busy-waiting. The next example shows variations with several implementations of synchronization. To simplify, messages are very long (8 kilobytes). This removes many of the problems with short messages.

In a practical sense, interrupts probably work best. That is, for larger, computationally heavier applications with medium to long messages (both common), interrupts give the fastest executions. The primary message grain limitation arises from system overheads; interrupt handling requires operating system service, and this places a practical limit on shorter messages. The comment is especially true if briefer communications also imply more transactions. However, when messages are longer, as in Figure 7, interrupts (denoted *INT*) outperform busy-waiting *(BW),* and are not nearly as sensitive to tuning as polling (*POLL*). Note again, that these are for a specific shared-memory machine.

Figure 7 also displays time-to-completion curves for *by-value* modes of transmission. This corresponds closely to message-passing, in that no permanent storage of a message is allowed in shared memory. A message must be copied totally into a processor's private memory area, processed, and copied back to shared memory for transmission. While this mode is considerably more secure, it is understandably slower for longer messages. Milde, *et. al.,* [MIL86] commenting on their experiences with the Aachen M^{5}PS cluster machine, remark that, "The comparison reveals a significant overhead of transparent message passing as against synchronized communication via shared variables." The point is worth noting because shared memory performance declines considerably when using a by-value paradigm. Loosely and tightly coupled systems are more comparable, however, on a by-value basis. (By nature of difference, by-value offers security, whereas by-reference gives access-bandwidth.)

4.3.3 The sensitive nature of polling. For larger applications with medium to long messages, polling offers performances ranging from excellent to inferior (Figure 7). The sensitive nature of the ring benchmark regarding polling is clear from the shape of the curves in Figure 8, with their abrupt transitions. Figure 8 depicts *relative* time-to-completion against polling frequency in milliseconds. While a good polling frequency for the ring structure is easy to identify, the curve changes with problem parameters, especially ring size. Thus each distinct set of problem parameters may need a new tuning. This tuning can be very sensitive, with a millisecond change in polling frequency resulting in a nearly three-fold increase in time-to-complete. As the ring grows larger, it amplifies performance differences that result from polling frequency. Thus 20 ms polling may run ten percent faster than 19 ms polling for a ring of ten nodes, but yield a factor of two or three times faster for a ring of thirty nodes. Figure 8 shows two runs, each normalized *within their own time-scale* for time-to-completion, and plotted against polling frequency in milliseconds. The obvious "steps" are at system clock "tick" multiples. Polling at more than one but less than three clock "ticks" seems an especially poor choice on this specific shared-memory system.

Variations in polling benchmark performance may require investigation at a lower level of measurement. The randomness of each message's computation at a node does affect the time-to-completion (as expected, more variance implies longer runs), but the abrupt points of transition in Figure 8 remain whether the computations are fixed or uniformly random. Similarly, varying the ring size modifies the steps, but they remain. Initial instruction level measurements on the polling request (SELECT in Unix) show a very high variance in its service times. However, the results are inconclusive at this point. The equipment to perform these lower level characterizations is special-built at NBS for the shared-memory machine; time-stamp perturbations are only 3-5 microseconds, about two decimal orders of magnitude less than otherwise available from the system clock.

4.4 The Ring and Framework in Perspective

Preliminary studies provide evidence that no single mode of communication implementation is completely satisfactory over broad ranges of circumstance. Variations in implementation, host, or problem size can profoundly affect performance, even with essentially the same program code.

The very structured framework and the shared core of code for the ring benchmark (and others to follow) represents an effort to promote easy parametric studies of various architectures. A core benchmark and its conditional compilation features have a simple structure that can be understood well. That the performance can get interesting is encouraging, indicating that a simple approach may have utility. Certainly more experience is needed, both running the ring structure on various machines, and with other related benchmarks, as sketched earlier.

The framework and its related routines appears promising as a structure for organizing debate on what certain communication aspects imply. For example, elements of synchronous versus asynchronous communication, of SIMD versus MIMD, are not now addressed directly. As evidence is acquired, the framework and its routines can be changed and modified to suit circumstances.

5. Conclusions

This completes the summary of preliminary work and experiences at NBS on benchmarks for parallel machines. Discussion has covered several roles, ideal and realistic settings, and quick summaries for several types of benchmark sets. A detailed example demonstrates some of the observations of earlier chapters.

5.1 Recommendations

Several observations can be forwarded, given the rudimentary nature of parallel processor characterization:

1. That performance metrics should be embedded in more comprehensive frameworks *that can be evaluated and modified as needed, and*

2. That one universal framework is beyond reach, since distinct clusters of use are emerging with separate emphases, and

3. That large application benchmarks are most successful when they run well on an architecture, thereby demonstrating compatibility of job and machine, and

4. That the value of smaller metrics (fragments of code) is more diagnostic and preventive than predictive. At this time, the variety of uses and architectures render tight predictions unlikely in any simple system of metrics. Small metrics should promote easy parametric studies which can isolate system anomalies and encourage directions of enhancement.

Small metrics and the large codes can take quite useful roles that complement each other. Codes fulfill a present need to evaluate the use of parallel processing in real applications. They assure purchasers of radical architectures that, while the full envelope of performance is not understood, the new machine does demonstrate powers that are judged an advance over services otherwise available. In contrast, the metrics reflect a piecemeal basic understanding that is progressing in parallel processing. A metric is an element of a model, a useful simplification of a complex endeavor. As the metrics and their parameters become more agreed upon, the field of parallel processing itself will have become more orderly, and *vice versa*.

6. References

[86-NRC] "An Agenda for Improved Evaluation of Supercomputer Performance." Report prepared by the Committee on Supercomputer Performance and Development, National Research Council (Washington, D.C.,1986), 58pp.

[CAR86] Carpenter, R.J. "Proposed computer performance measurement hardware." Parallel Processing Lab. Note No. 20, ICST, NBS, March, 1986.

[EFR78] Efron, J. "Computers and theory of statistics: thinking the unthinkable." *SIAM Review 21,* 4(October, 1978), 460-480.

[ETC83] Etchells, R. David and Nudd, Graham R. "Software metrics for performance analysis of parallel hardware," Hughes Research Laboratory (Malibu, CA.) Report, circa 1983, under Darpa Order No. 3119. Reported at Alvery-DARPA Workshop on Benchmarking Parallel Architectures, October 10-11, 1984, London.

[FER86] Ferrari, D. "Considerations on the insularity of performance evaluation," *IEEE Trans. on Software Eng., SE-12, 6(June 1986), 678-683.*

[GAI86] Gait, J. "A probe effect on concurrent programs." *SOFTWARE--Practice and Experience 16,* 3(March, 1986), 225-233.

[HIK85] Hikita, T. and Ishihata, K. "A method of program transformation between variable sharing and message passing," *Software--Practice and Experience,* 15(7), 677-692(July 1985).

[HIL85] Hillis, D. Remarks on parallel architectural similarity, 12th Annual Int. Symp. on Computer Architecture, Boston, June, 1985.

[HUS87] Huss, J.E. and Pennline, J.A. "A comparison of five benchmarks," NASA Technical Memorandum 88956, February, 1987, 13pp.

[KRU87] Kruskal, C.P. and Smith, C.H. "On the notion of granularity," Internal report, Parallel Processing Group, NBS, *circa* January, 1987, 15pp.

[LYO86] Lyon, G. "A fast, message-based, tagless marking," *Proc., Second Hypercube Multiprocessor Conference,* Knoxville, Sept. 1986.

[MCM86] McMahon, Frank H. "The Livermore FORTRAN Kernels: A computer test of the numerical performance range." Draft report UCRL-53745, October 1986. In preparation at Lawrence Livermore National Laboratory, and to be available at a later date from: National Technical Information Service, Springfield, VA 221611.

[MIL86] Milde, J., Plueckebaum, T., and Ameling, W. "Synchronous communication of cooperating processes in the M^5SP multiprocessor," *Proc., CONPAR 86, Lecture Notes in Computer Science 237,* Springer-Verlag, 1986.

[MIN86] Mink, A. *et al.* "Simple multiprocessor measurements techniques and preliminary measurements using them." Parallel Processing Lab. Note No. 23, ICST, NBS, March, 1986.

[NAT63] Natrella, M.G. *Experimental Statistics.* NBS Handbook 91, August 1963.

[OMA85] O'Malley, S.W. and Gilmer, J.B. Jr. *Survey of High Performance Parallel Architectures,* Report BDM/ROS-85-0988-TR, BDM Corp., Dec. 1985, 55pp.

[ORG85] Organick, E. and Asbury, R. "The iPSC stress tester (Organick Rasberry)," *Proc., First Conf. on Hypercube Multiprocessors,* Knoxville, Tenn., Aug. 1985. SIAM, Philadelphia, 1986.

[PAV84] Pavelin, C. "Notes on Alvey-DARPA Workshop on Benchmarking Parallel Architectures, October 10-11, 1984, Millbank Tower, London." Rutherford Appleton Laboratory, Chilton, 1984.

[RIC87] Richardson, R. Arpa-mail, "info-micro," 2 Feb. 1987. Subject: 1/31/87 Dhrystone Results and Source. 9 pp.

[SAL86] Salazar, S.B. and Smith, C.H. "National Bureau of Standards Workshop on Performance Evaluation of Parallel Computers," NBSIR86-3395, July, 1986, 50pp. Available through National Technical Information Service (NTIS), Springfield, VA. 22161.

[WEI84] Weicker, Reinhold P. "Dhrystone: A synthetic systems programming benchmark," *Comm. ACM 27,* 10(October, 1984), 1013-1030.

Three Common Communication Paradigms

adapted to parallel execution

```
Initialize time, all local_domains;
REPEAT
        FOR ALL local_domains DO
                local_domain(time+1) = G[ all other local_domains (time)];
        INCREMENT(time); GLOBAL_SYNCHRONIZE;
UNTIL done
```

Figure 1. Global Interdependency

Among the local domains, *which might be particles in a physical model, say molecular dynamics, each depends upon all others in the ensemble. Thus any "next step" in time requires that each and every domain propagate its state to the rest. A ring architecture (systolic or otherwise) is suited for this.*

```
Initialize time, all local_domains;
REPEAT
        FOR ALL local_domains DO
                local_domain(time+1) = N[ neighboring local_domains( time)]
        INCREMENT(time); IF(SIMD) GLOBAL_SYNCHRONIZE;
UNTIL done
```

Figure 2. Neighboring Data Dependency

Models in fluid flow, such as weather prediction, often have computations in which each domain is influenced by some local set, its neighbors in the topology. These computations may fit upon a grid with node-points each performing mostly the same calculations. An SIMD machine can be appropriate, but some algorithms will converge asynchronously (chaotically) and can use MIMD. The code reflects either option.

```
FOR ALL local_domains DO
        *Initialize* local_time, local_domain;
        REPEAT
                local_domain(local_time+1)= I[local_domain(local_time)];
                INCREMENT(local_time);
        UNTIL *done;*
```

Figure 3. Computationally Independent Events

Some applications admit calculations that can proceed independently of any additional calculations that accompany them. Radiation transport is an example; each particle has more or less a life of its own. This makes load-balancing for a MIMD machine easier, since some variation of a "self-serve" paradigm (large-grain dataflow) can be used. In the dataflow context, the "FOR ALL" may imply more than just a single startup: as new local domains arise they too must be scheduled for processing.

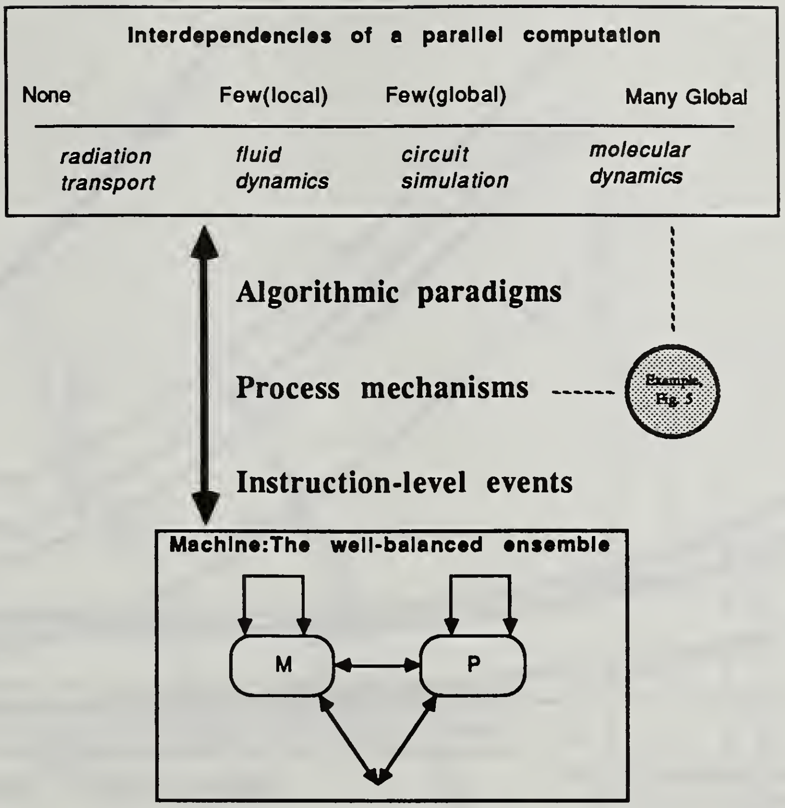

Figure 4. Framework

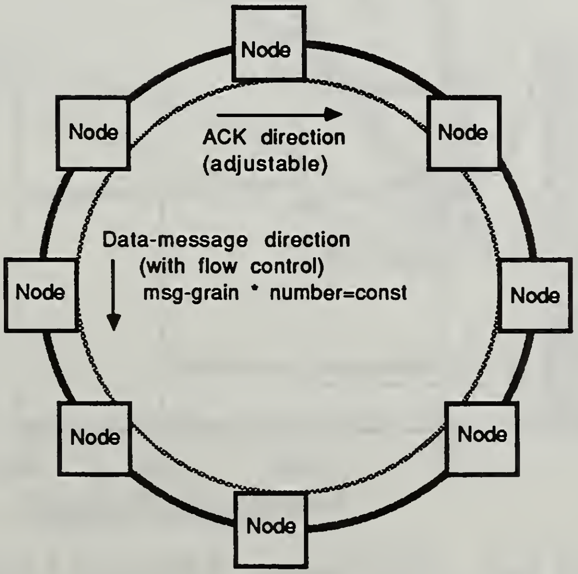
Node
Node
Node
Node
Node
Node
Node
Node
ACK direction (adjustable)
Data-message direction (with flow control) msg-grain * number=const

Figure 5. Ring Pipeline

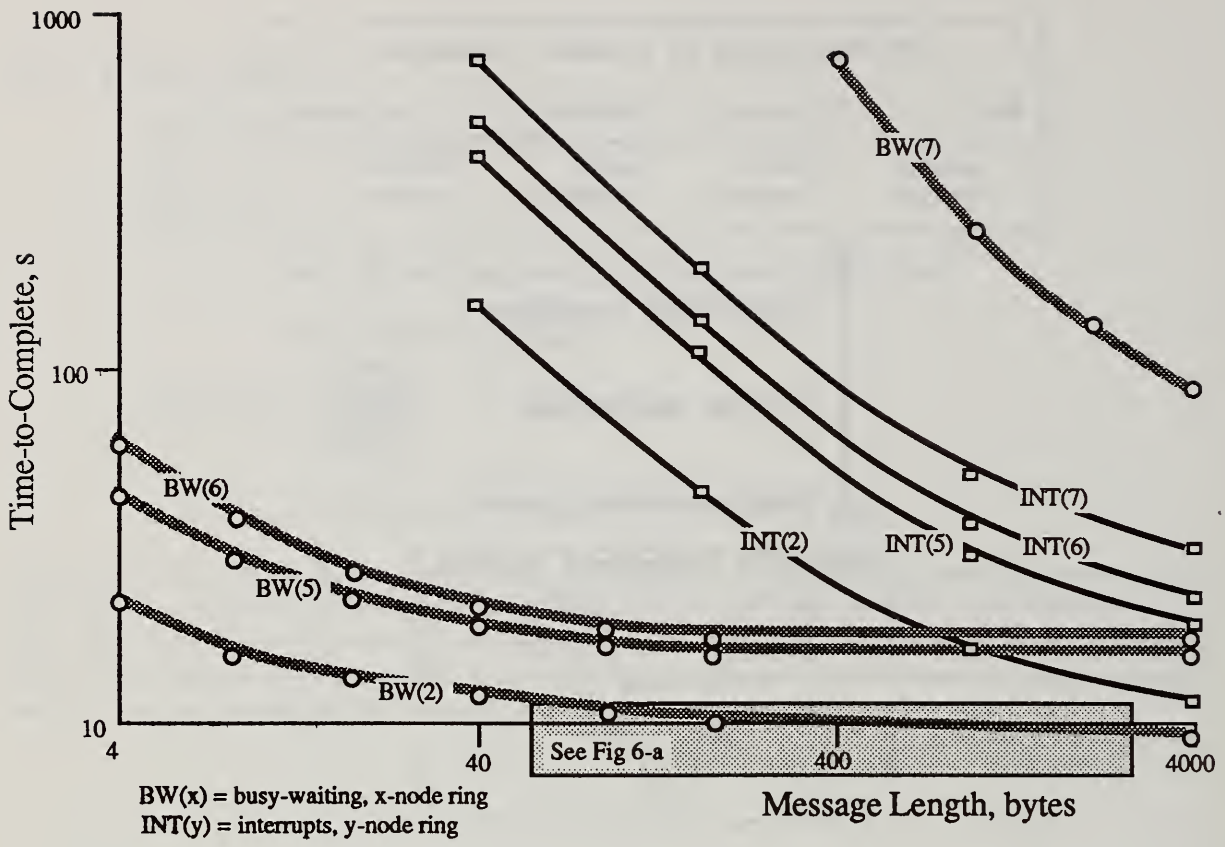

Figure 6. Time versus Message Lengths, Shared Memory

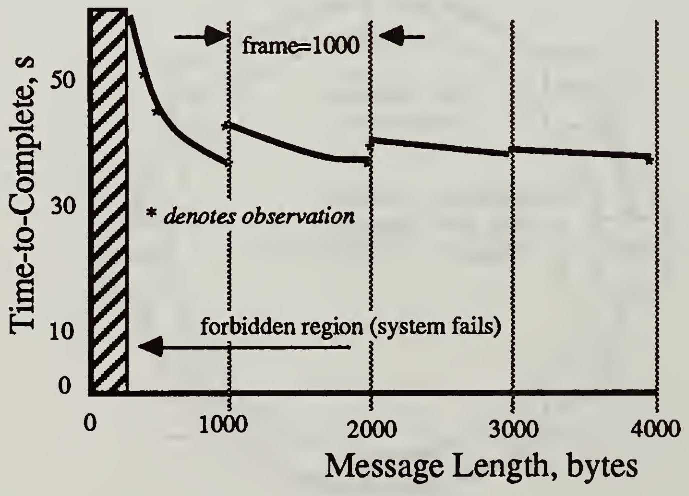

Figure 6-a. Time versus Message Lengths, BW(2) on Hypercube Variant

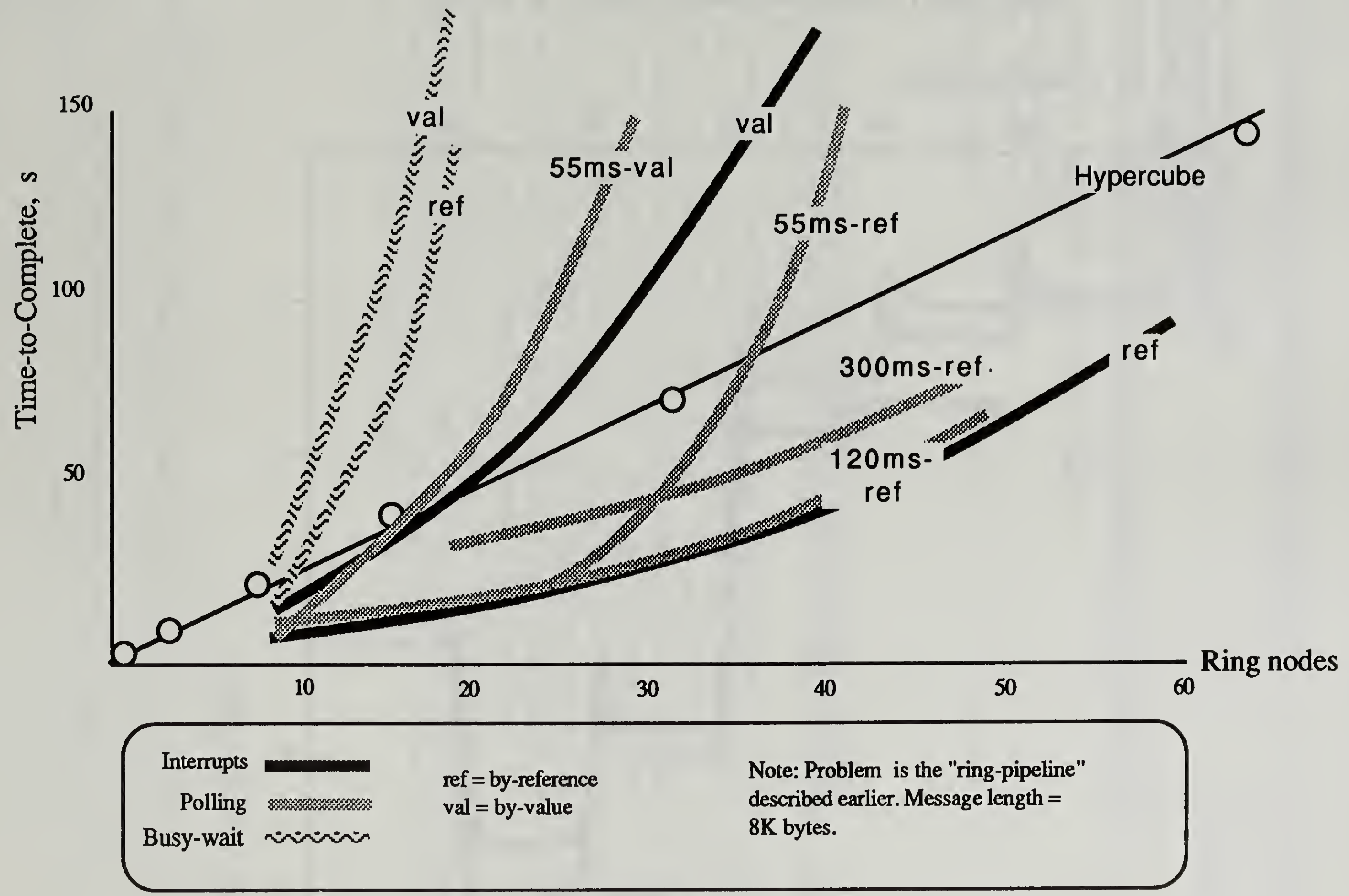

Figure 7. Time-to-Completion versus Ring Size, Shared-memory Parameters plus Hypercube

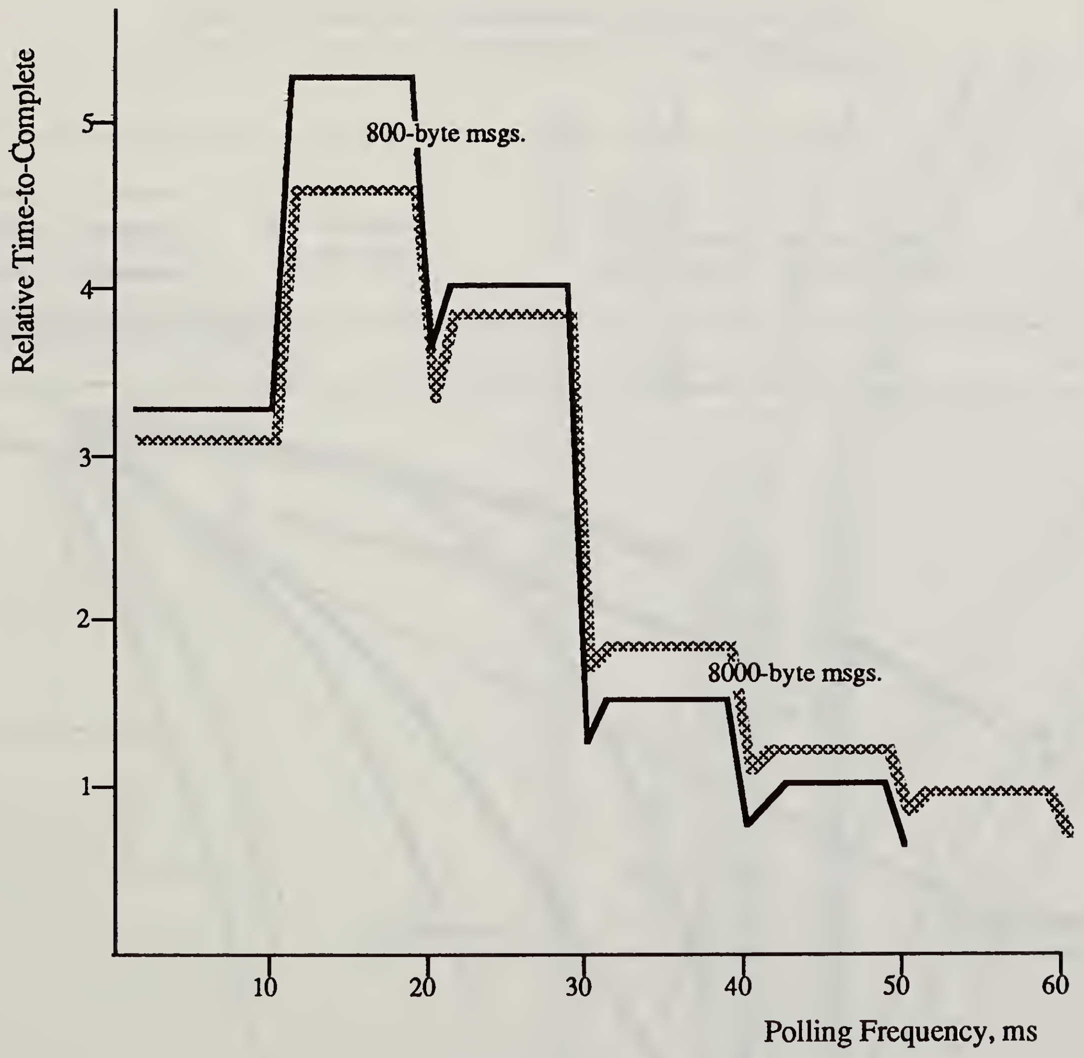

Figure 8. Typical Polling Results (30 nodes)

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBSIR 87-3580	2. Performing Organ. Report No.	3. Publication Date JULY 1987

4. TITLE AND SUBTITLE

On Parallel Processing Benchmarks

5. AUTHOR(S)

Gordon Lyon

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)* **NATIONAL BUREAU OF STANDARDS U.S. DEPARTMENT OF COMMERCE GAITHERSBURG, MD 20899**	7. Contract/Grant No.
	8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

This is a summary of preliminary work and experiences at NBS on benchmarks for parallel machines. Discussion covers the several roles that benchmarks play, ideal and realistic settings, and quick reviews of several types of benchmark sets. Several recommendations can be forwarded, given the rudimentary nature of parallel processor characterization: (i) That the performance metrics be embedded in more comprehensive frameworks that can be appraised and modified as needed; (ii) That one universal framework is beyond reach, since distinct clusters of use are emerging with separate emphases; (iii) That large application benchmarks are most successful when they run well on a machine--and thus demonstrate success for a narrow class; (iv) That to programmers, the value of smaller metrics (fragments of code) is more diagnostic and preventive than predictive. Small metric sets should encourage the parametric study of architectures and applications, and thereby promote both economical hardware enhancement and suitable program design.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

benchmarks; computer performance; frameworks; measurements; metrics; models

13. AVAILABILITY	14. NO. OF PRINTED PAGES 35
[✓] Unlimited ☐ For Official Distribution. Do Not Release to NTIS ☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402. [✓] Order From National Technical Information Service (NTIS), Springfield, VA. 22161	15. Price $11.95