

RL-TR-96-168
Final Technical Report
December 1996



BENCHMARKING METHODOLOGY FOR REAL-TIME EMBEDDED SCALABLE HIGH PERFORMANCE COMPUTING

The MITRE Corporation

Richard A. Games

19970211 019


APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

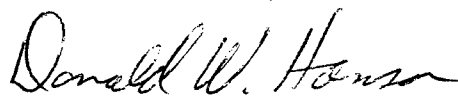
[DTIC QUALITY INSPECTED 3]

**Rome Laboratory
Air Force Materiel Command
Rome, New York**

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-96-168 has been reviewed and is approved for publication.

APPROVED: 
RALPH L. KOHLER, JR.
Project Engineer

FOR THE COMMANDER: 
DONALD W. HANSON, Director
Surveillance & Photonics Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/OCSS, 26 Electronic Pky, Rome, NY 13441-4514. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE December 1996		3. REPORT TYPE AND DATES COVERED Final Mar 94 - Jun 95
4. TITLE AND SUBTITLE BENCHMARKING METHODOLOGY FOR REAL-TIME EMBEDDED SCALABLE HIGH PERFORMANCE COMPUTING			5. FUNDING NUMBERS C - F19628-94-C-0001 PE - N/A PR - MOIE TA - 74 WU - 11	
6. AUTHOR(S) Richard A. Games				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The MITRE Corporation Center for Air Force C3 Systems 202 Burlington Road Bedford, MA 01730-1420			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/OCSS 26 Electronic Pky Rome, NY 13441-4514			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-96-168	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Ralph L. Kohler/OCSS/(315) 330-2016				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This application of scalable high performance computing to real-time embedded systems is expanding. Traditional benchmarks and notions of scalability drawn from the scientific parallel computing community are of limited relevance when timing requirements must be met within strict size, weight, and power requirements. This paper proposes a benchmarking methodology for real-time embedded applications, including a relevant notion of scalability. The essential point is to give equal emphasis to timing and functional specifications. We use a test bench on the machine itself to realistically impulse the function or system under test. Metrics that measure the steady-state performance account for limitations of non-real-time system software. The key metric is the minimum processor size required to meet the real-time specifications for a given problem size. The scalability of the embedded processor can then be characterized in terms of the rate of increase of this minimum machine size as the problem (and potentially the real-time requirement) is scaled.				
14. SUBJECT TERMS Real-time embedded systems, benchmarking methodology, test bench, embedded processor			15. NUMBER OF PAGES 36	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

DTIC QUALITY INSPECTED 3

Table of Contents

Section	Page
Introduction	1
Real-Time Embedded Benchmarking Methodology	3
2.1 Background	3
2.2 Design to Specification and Real-Time Scalability	3
2.3 Real-Time Test Bench	4
2.4 Real-Time Benchmark Specifications	4
2.5 Auxiliary Performance Metrics	5
Where Do the Benchmarks Come From?	9
3.1 Low-level Real-Time Benchmarks	9
System Clock Calls	9
Interrupts for Scheduling	11
3.2 Kernel Benchmarks and Compact Applications	12
Related Efforts	13
4.1 PARKBENCH	13
4.2 C ³ I Parallel Benchmark Suite	13
4.3 Hartstone	14
4.4 Transactions Processing Performance Council Benchmarks	14
4.5 Other Focused Efforts	15
Scalability Comparisons	17
Conclusion	19
List of References	21
Appendix A	RT_2DFFT: Real-Time Two-Dimensional FFT Benchmark Specification
	23
Appendix B	RT_CornerTurn: Real-Time Distributed Corner Turn Benchmark Specification
	27

Acknowledgments

This work was supported by the United States Air Force Electronic Systems Center and performed under MITRE Mission Oriented Investigation and Experimentation (MOIE) Project 03957426 of contract F19628-94-C-0001, managed by Rome Laboratory/OCSS.

Leonard Monk contributed the low-level real-time benchmarks (CLOPS, IACC, and IOVERHEAD) in Section 3.1. His work was supported by the Defense Advanced Research Projects Agency and performed under MITRE Project 07958785 of contract DAAB07-94-C-H601, managed by the Naval Command, Control and Ocean Surveillance Center, RDT&E Division (NRAD).

All product names, trademarks, and registered trademarks are the property of their respective holders.

Section 1

Introduction

Under previous efforts we have begun the development of a benchmarking methodology for assessing the performance of scalable high performance computers for real-time embedded applications [Brown, et al., 1994]. The purpose of this paper is to further extend this benchmarking methodology and to suggest an initial benchmarking activity, a real-time two-dimensional fast Fourier transform (FFT), to test out what we propose. The overall goal is the development of a rigorous and uniformly embraced methodology for performance assessment applicable to scalable real-time embedded computing. This is especially critical now as research papers are beginning to appear assessing the use of scalable high performance computing in real-time embedded applications. Some of these papers report meaningless or flawed performance results from the standpoint of the motivating applications.

Assessments will be needed across a wide spectrum. Initial assessments need to establish baseline processor performance using native processing and communication libraries and hand-tuned code. The focus here will be on establishing the "existence proof" that a particular processor/system software pairing can deliver desired levels of scalable sustained performance. Simple and standardized tests are needed to provide a common methodology for comparing disparate approaches (MIMD versus SIMD, meshes versus crossbars, packet switched versus circuit switched). System software assessments need to focus on relevant real-time operating system benchmarks to determine the time-scale (granularity) at which guarantees can be made.

Support tools (e.g., performance monitors) and interoperability layers (e.g., Message Passing Interface (MPI) and MessageWay implementations) need to be assessed as they become available. The focus at this stage will be on quantifying the impact on real-time embedded performance of those techniques that offer more portability. As parallel software design processes are developed for real-time applications (e.g., data flow shells, more general partitioning and mapping tools), previous baseline benchmarks need to be repeated with these higher-level tools, again to assess the impact on performance. Performance impacts of any proposed security and fault-tolerant techniques also need to be quantified. A key aspect of all these assessments will be to determine the range of applicability (scalability) of potential solutions.

Three levels of benchmark complexity are considered. Low-level benchmarks are designed to narrowly focus on the performance of some crucial component of the parallel system. Kernel-level benchmarks provide useful parametric information on the computational and communication building blocks of subsequent real-time implementations. Finally, compact application benchmarks incorporate representative system behavior and are implemented according to the design-to-specification methodology described in this paper.

Section 2 describes our proposed real-time embedded benchmarking methodology, including the use of a real-time test bench. The emphasis is on how the benchmarking methodology must change to incorporate the requirements of real-time embedded processing. A unified treatment of utilization, speed up, and efficiency for periodic real-time processing is given. Section 3 discusses where the benchmarks will come from. One of our points is that there already exist a variety of higher-level benchmarks (kernels and compact applications) that can be incorporated into our methodology. We examine by example how real-time requirements influence the design of lower-level benchmarks.

Section 4 lists a number of related benchmarking efforts. In particular a potential collaboration with the Rome Laboratory C³I Parallel Benchmark Suite [Metzger, 1994] is described. Section 5 considers how our notion of scalability for real-time embedded applications compares to traditional notions of scalability from scientific computing. Section 6 states the conclusions. Appendix A contains an initial real-time benchmark specification of a two-dimensional FFT treated as a compact application to provide a test case of the proposed methodology. Appendix B contains a kernel-level benchmark specification of a data remapping function relevant to a two-dimensional FFT.

Section 2

Real-Time Embedded Benchmarking Methodology

In this section we propose a real-time embedded benchmarking methodology for scalable high performance computing. The notion of using a software test bench to assess steady-state real-time performance is introduced. The components comprising a high-level real-time compact application benchmark are described. A unified treatment of utilization, speed up, and efficiency for periodic real time applications is given.

2.1 Background

A previous focus of our work was on creating a benchmarking methodology that supported the development of parallel software for real-time embedded applications [Brown, et al., 1994, 1995a, b]. This parallel software development process involved running benchmarks designed to assess the level of real-time performance that the scalable massively parallel processor, or more precisely its components, could deliver on processing and communication kernels. These kernels were then combined into a single implementation using a variety of parallelization approaches so that the application's timing requirements were satisfied.

The process was iterative and involved a series of benchmarks that incorporated increasingly more of the application's requirements. The approach was to make the initial benchmarking results highly predictive by including in the single component benchmarks much of the infrastructure and overhead required in a real-time implementation. This methodology has proved useful in developing scalable parallel software for real-time signal processing applications, and we propose to expand it to attack the real-time embedded benchmarking problem in general.

2.2 Design to Specification and Real-Time Scalability

The specification of the timing requirements and their impact on the benchmarking methodology is what distinguishes the proposed approach. The essential point is to treat timing and functional specifications on an equal footing. A *design-to-specification* methodology provides the most meaningful assessment of competing approaches. In this methodology the timing requirements of representative applications are extracted and specified. The objective then becomes to complete the processing within the timing specification. The smallest size of the scalable high performance computer that meets both the functional and timing requirements is determined. This *minimum machine size* metric is perhaps the most compelling overall metric for the case of embedded applications. This methodology provides an easily understandable way of comparing different approaches (e.g., tools) on the same scalable platform. Competing architectures can be compared by converting the minimum sized solutions to common measures of size, weight, power, and price.

Scalability for real-time embedded applications can now be placed on a solid footing. The primary concern is to maintain the real-time requirements as the complexity of the problem is increased. This increase in complexity often follows from an increase in the input problem size with the algorithm fixed, but one relevant alternative could be to examine a sequence of increasingly complicated algorithm enhancements that correspond to proposed upgrades of system functionality. In some settings the real-time requirements could be scaled as well. Each benchmark must clearly state how the real-time requirements are adjusted as a function of problem complexity. The rate of growth of the *minimum* machine size then becomes a metric in which to compare prospective solutions. This approach will reveal those proposed solutions that have limited scalability—solutions that can satisfy today’s requirements, but that have limited potential for the applications that are driving current investments in scalable computing. Section 5 discusses how this notion of “design-to-specification” scalability compares to traditional notions of scalability popular in the scientific computing community.

2.3 Real-Time Test Bench

A crucial component of the benchmarking methodology is to use a “test bench” to measure the real-time performance of the software under realistic steady state conditions. This is done with a dedicated *data source* responsible for providing data to the function under test and a dedicated *data sink* responsible for collecting desired results. Because I/O is often the “last frontier” for these embedded systems, early evaluations will implement the test bench on the machine itself. In other words, we “pick up” the processing after the I/O has occurred and the data has been appropriately buffered. An actual system implementation would incorporate the actual I/O interface, which can then be easily integrated with the test bench source and sink.

The test bench provides realistic stimulus to the function under test. For example, in a periodic hard real-time situation, the program is run for a long duration so that the data source generates many instances of the problem. The corresponding solutions are collected at the data sink and time stamped. A key metric in this case is the *worst case* difference between the times of successive solutions. It corresponds to a lower bound on the period that the particular parallel system can sustain. In contrast to scientific computing benchmarks, this approach correctly accounts for behavior that affects the regularity of the processing, such as unpredictable overheads encountered in non-real-time operating systems.

Soft real-time or event driven scenarios can also be incorporated in this test bench methodology. In this case the system’s response to increasing load is characterized, where the data source could initiate events to be processed according to a prescribed scenario.

2.4 Real-Time Benchmark Specifications

In general, a high-level real-time compact application benchmark specification should consist of a functional specification, a timing specification, the guidelines for conducting the scalability study (how to scale the problem complexity and the timing specifications, if desired), and implementation guidelines. Implementation guidelines describe any unique aspects of the test

bench, the acceptability tests, and the reporting requirements. Sequential code written in a high-level language (e.g., C) that runs on a workstation or personal computer with a standard operating system (e.g., UNIX) should also be supplied along with sample input and output results. This executable version of the functional specification can be used to construct more specific test sets for the parallel implementation. The sequential code implementing the function under test would not necessarily be the best starting point for the parallel implementation, and developing a more suitable sequential implementation could be the first step in the parallel software development process.

In many of the applications under consideration, the timing specification is given in terms of a periodic sequence of input data sets $X_1, X_2, X_3, \dots, X_i, \dots$. Two real-time requirements are typical in these periodic applications: the *period* is the time interval between successive inputs and the *latency* is the length of time required to process a single instance X_i , measured as the interval of time between when the data set X_i leaves the data source and the corresponding results arrive at the data sink. The requirement for the period is often easily determined from how fast data is coming off of a sensor and how much of it is to be processed. The latency is a system-specific quantity, some systems have short latency requirements, some have long latency requirements. The strictness of the latency requirement governs the difficulty of the parallel software development task and the efficiency of the parallel implementation. Because of the important role that latency plays, we often specify at least two benchmark cases corresponding to a strict and loose latency requirement.

As an illustrative example, and as a way of getting started, we provide in Appendix A a real-time benchmark specification called RT_2DFFT, a real-time two-dimensional FFT benchmark. Initial implementations of the proposed methodology using this benchmark are planned on several parallel machines, e.g., the Intel Paragon, the MasPar MP-1/2 [Koester and Rushanan, 1996], the Mercury MCV9, and on a network of workstations. Our plan is to make the resulting code, including examples of the test bench implementation, available to the community for additional optimizations and enhancements. A version of the test bench written using MPI is planned as a portable starting point. However, optimizations would be encouraged as we believe that there is a short-term need to establish the feasibility of highly efficient implementations on general purpose scalable massively parallel processors. The performance impact of software portability and ease of programming must eventually be assessed.

2.5 Auxiliary Performance Metrics

Our primary focus is on comparing scalable high performance computing alternatives by establishing the size of the smallest machine that can satisfy both the functional and timing specifications. In this section we discuss a number of other metrics that have been used to characterize the performance and scalability of high performance computers. In the present context, we are especially interested in notions that measure the efficiency of the system under test. "Efficiency" has taken on a specific meaning within the parallel processing community to mean "average speed up." We first introduce an alternative notion of efficiency in terms of

processor utilization and then relate it to traditional notions of speed up and efficiency. The discussion focuses on real-time applications that are characterized by period and latency constraints.

In embedded applications it is desirable to know how well we are using the, often limited, computing resources that are at our disposal, hence our interest in measuring some notion of processor utilization. Knowledge of this sort is certainly important in the design process as it can help determine when code optimization has reached a point of diminishing returns. It is also relevant from the standpoint of assessing the scalability of a proposed system. If there is a big drop in the processor utilization as the system is scaled, then this is an indication of trouble. This would also be reflected in the minimum machine size metric, but the utilization measure could provide additional insight. Notions of utilization alone are not sufficient to compare the performance of different platforms.

For a given periodic real-time implementation, we define the *utilization* metric to be the percentage of the peak operational rate that the high performance computer is actually able to sustain. Often single precision floating point operations per second (flop/s) are of interest, but we give the definition in terms of generic operations per second (op/s). The peak performance, denoted by $peak_op/s(n)$, must be agreed upon ahead of time and be expressible as a function of the number of processors n . Usually $peak_op/s(n) = n[peak_op/s(1)]$. The number of operations required to process a single problem instance must be specified, and is denoted by $\#op$. For consistency, all concerned must use the specified $\#op$ in the utilization calculation, regardless of the number of operations that are actually implemented by a given approach. This assumption is less controversial when the algorithms employed are fairly standardized, as is often the case in signal processing applications. Given an n processor implementation, the minimum sustainable period that can be supported, denoted by $period(n)$, must be determined. This is best done using a consistent source-sink test bench discussed previously. Then the sustained operational rate, denoted by $sustained_op/s(n)$, is given by $\#op/period(n)$. The processing utilization (as a percentage) is defined as

$$utilization(n) = [sustained_op/s(n)] / [peak_op/s(n)] \times 100 (\%).$$

Note that the processing utilization is a function of the minimum period that can be sustained for a sequence of input problems. It has nothing to do with the latency of individual solutions.

In contrast, *speed up* is concerned with the latency for a single problem instance. The speed up, $S(n)$, is usually defined as the computation time for a problem instance when a single processor is used divided by the computation time for the problem instance when n processors are used. Care must be used when establishing the single processor baseline to assure meaningful results [Sahni and Thanvantri, 1996]. Because of problems in this regard, many in the parallel processing community argue against relying on speed up to measure the performance of a parallel system.

In parallel processing, “efficiency” is usually defined for an n processor system as

$$efficiency_{parallel}(n) = S(n)/n.$$

This notion of efficiency can be interpreted as average speed up. In the embedded community the word “efficiency” is sometimes used to refer to what we have termed “utilization,” and some care must be taken to make the correct interpretation. As defined here, utilization and efficiency involve different aspects of the periodic real-time implementation, namely utilization is associated with periods, while efficiency is associated with latencies. Usually these quantities, period and latency, have quite different real-time requirements, with the case of latency greater than the period being common in signal processing applications. Although some applications (e.g., control) may require that the latency be equal to the period, the distinction should be maintained between the two concepts. A second, more minor distinction is that the efficiency metric embodies a comparison of the performance of two different sized machines (one processor versus n processors), whereas utilization compares the actual and theoretical performance of a single machine (with n processors).

Next we explore how $efficiency_{parallel}$ and $utilization$ are related (but are still not the same concept) if they are defined consistently. This is accomplished by defining speed up in terms of the period sustained as

$$S_{period}(n) = period(1)/period(n).$$

Here, as before, $period(n)$ is the minimum period that a machine with n processors can sustain, again measured using a realistic source-sink test bench. Then, assuming that $peak_op/s(n) = n(peak_op/s(1))$, a calculation gives

$$S_{period}(n) = n [utilization(n)]/[utilization(1)].$$

So that,

$$efficiency_{period}(n) = S_{period}(n) / n = [utilization(n)]/[utilization(1)].$$

Thus efficiency (as usually defined in parallel processing, but for periods instead of latencies) corresponds to the ratio of the processing utilizations obtained for n processors and for one processor. A sign of a “scalable” high performance computing implementation is that this measure of efficiency is close to 1, i.e., the utilization is close to being constant as a function of n . This corresponds to what we suggested previously. Note, though, that this notion of efficiency is just comparing two different sized implementations and is not making any statement about how “good” either of them are relative to resource use (the lay definition of efficiency useful for embedded applications). Thus the progression of the actual values of $utilization(n)$ as the number of processors n is increased is a more useful way of characterizing both the resource usage and the scalability of an embedded implementation. In Section 5 we contrast this approach with approaches that measure the rate of increase in problem size

required to keep parallel processing efficiency constant as the number of processors is increased—the so called *isoefficiency* metric [Grama, et al., 1993].

Section 3

Where Do the Benchmarks Come From?

The choice of benchmarking functions and their level of implementation complexity is crucial. Different application areas suggest their own unique real-time processing requirements. Judgments on the potential for future success of scalable massively parallel processing technology must be considered in the choice of application domains. Many of the initial insertions of this technology will focus on signal processing applications such as synthetic aperture radar (SAR) image formation or space-time adaptive processing (STAP). The SAR application was the motivation behind the real-time two-dimensional FFT benchmark specification given in Appendix A. Other motivating applications, including, for example, multiple-hypothesis inferencing and image processing, are equally compelling and are mentioned in related benchmarking efforts in Section 4.

There are many existing scientific parallel processing benchmarking efforts, some incorporating functions that are relevant to real-time embedded applications. The PARKBENCH effort [Hockney and Berry, 1994], for example, organizes its benchmarks into three levels: low-level, kernel, and compact applications. In this section we focus on low-level benchmarks that are suggested because of real-time concerns. Kernel-level benchmarks and compact applications are briefly discussed and will be derived primarily from the existing efforts listed in Section 4.

3.1 Low-level Real-Time Benchmarks

Low-level benchmarks are designed to narrowly focus on the performance of some crucial component of the parallel system. Examples of low-level benchmarks in PARKBENCH include TICK1 and TICK2, which are designed to measure the resolution and accuracy of the clock used in the benchmark measurements. The COMMS1 benchmark in PARKBENCH measures the basic communication properties of a message-passing computer. In this section we describe examples of low-level benchmarks that are suggested because of real-time issues. These particular illustrations were derived from our work developing a real-time communications scheduling service for scalable massively parallel processors [Games, et al., 1994, 1995, Brown, et al., 1996] and are only meant to be representative. The impact of the hard real-time context is evident in all of the benchmark specifications presented below. Two categories of low-level synthetic benchmarks are introduced. They concern system clock capabilities and mechanisms for effecting time-driven schedules via interrupts.

System Clock Calls

A good system clock is obviously desirable for time-driven hard real-time scheduling. More generally, the system clock is the basis of instrumentation necessary to establish other benchmarks. The contrast in how we must treat the system clock compared with

PARKBENCH is striking. For one thing, it is imperative that clock resolution and accuracy must be very good relative to the standards originally envisioned in their TICK1 and TICK2 experiments. We hereby assume very good resolution and accuracy of the underlying clocks. These qualities must be verified at some point, but other means than running benchmark code are probably more appropriate for this job at the levels of resolution and accuracy available (microseconds).

We advocate a subtle modification of what is to be measured: the accuracy and overhead of the available user-level system clock calls when being used as instrumentation or a scheduling mechanism are critical and can be evaluated by benchmarks. Typically, one will attempt to determine when, in an actual run, a particular event occurs by inserting a clock call in the code and doing something with the returned value that allows it to be retrieved later. The relation of the actual time of the event to the value returned by this call, as well as the degree of interference of these insertions, are what need to be evaluated. This is the goal of the CLOPS benchmark described below.

It is helpful to standardize the basic instrumentation insertions. The combination of obtaining a clock value, storing it, and preparing for the next measurement can be accomplished in C by something like

```
A[i++] = timer();
```

where A is an appropriately sized and typed array, and `timer()` is the best available system clock call, e.g., `dclock()` on the Paragon. A single instance of this basic instrumentation insertion will be called a *clock operation* (clop). Crudely put, one wants to determine the maximum rate of clock operations per second (clop/s). This gives an idea of how long one measurement takes and, together with bounds on the actual underlying clock accuracy, enables one to estimate the accuracy and overhead we want.

The CLOPS benchmark simply iterates clock operations in a tight loop until interrupted by Control-C and then writes the array of times to a file. Included in the specification are requirements that the writes to memory are prevented somehow from inducing page faults, the size of the array holding time values is not exceeded, and there is only one application process running.

The alternative of halting when a predetermined bound is reached requires a comparison involving the running index *i* at each timer call. This is a seemingly insignificant extra overhead, except that, when the timer call really amounts just to reading a memory location (as with `dclock`) instead of a system call, the modest extra overhead may be easily detectable.

The average clop/s, as indicated by a long run of CLOPS, can be quite misleading, especially for non-real-time operating systems. Closer examination of the data will probably indicate that operating system time-outs of various sizes are occurring, in spite of there being only one applications process and no paging. In the case of OSF1 on the Paragon, a graph of timer call

index versus consecutive timer value difference will show one low band corresponding to the values one might naively expect from this experiment, but other bands are observable. The maximum time between consecutive timer calls is radically greater than the average.

The operating system itself is being tested by CLOPS as much as the clock. Our ability to depend on clock operations for measuring system performance on other benchmarks may be severely limited unless time-outs can be eliminated or measurements can be restricted reliably to intervals between long time-outs. Even benchmark results stated with coarse granularity require rigorous bounding of the operating system time-outs over intervals.

Interrupts for Scheduling

Operating systems often provide facilities for sending interrupt signals to processes at predetermined times, often periodically. A typical example is the `ualarm` call, which can be used to send a signal every time a fixed number of microseconds has elapsed, at the first opportunity the operating system has to communicate with the process. One application of such interrupts in a real-time context is to implement time-driven scheduling. The benchmark IACC assesses the accuracy of the interrupt mechanism, and IOVERHEAD assesses its overhead. Together they can be used to establish the scheduling granularity supported by the interrupt facility in question.

IACC tests periodic interrupt accuracy on a gradually decreasing sequence of periods until the interrupt mechanism is found not to perform adequately for the period being tested. Each period is tested by having the operating system use the given interrupt mechanism to interrupt a basic work activity (incrementing a variable) a fixed number of times or until the interrupts become too inaccurate. This is judged by having the interrupt handler calculate when each interrupt should be, read the clock when it actually occurs, and check to see if the difference exceeds an error parameter. In one variant the prediction is made by adding the period to the actual time of the previous interrupt, while a more rigorous variant calculates by dead reckoning from the initial interrupt using the period and the ordinal of the interrupt.

IOVERHEAD runs a basic loop that just increments a variable a prespecified number of times, but which is interrupted at a given period, where the interrupt handler does nothing. When the interrupt period is great, the overhead is near 0, and a base, 0-overhead time to perform the specified number of increments can be determined. As the interrupt period is decreased (probably by a factor of two each time), the overhead of the interrupt mechanism is easily determined. The limiting factor may either be that below a certain period interrupts are lost (so the actual interrupts don't get any more frequent below that point), or the overhead may increase sharply. Our experience with `ualarm` on the Paragon indicates that the former happens.

This presentation is intended to indicate the general nature and some of the important themes of a more mature development of low-level benchmarks for evaluating the ability of a system to support real-time applications. One such theme is that care must be taken to avoid subtle traps

when fine time measurements are being made, even in very simple experiments. Another is the prominence of the operating system as a determining factor for the benchmarks' results.

3.2 Kernel Benchmarks and Compact Applications

Intermediate-level benchmarks involving computation and communication kernels are needed to provide an assessment of the real-time performance the hardware and system software can deliver on the building blocks of associated applications. Kernel-level benchmarks should be designed to provide useful parametric information to support subsequent real-time implementations. Examples include single processor computational timings as a function of input size or distributed data remapping times for a fixed data size as a function of the number of processing nodes. Care must be taken to specify the interface requirements of the kernel-level benchmarks so that the results are realistic. For example, if the data originates in the memory of another processing node, then the overhead required to move the data across the network needs to be included. Characterizing the variability of the kernel-level results is essential from the real-time perspective. An example of a kernel-level benchmark for a "distributed corner turn" is given in Appendix B.

At the highest level are the compact application benchmarks. These benchmarks consist of a few thousand lines of code and incorporate representative system behavior not found in kernel-level benchmarks. These broader benchmarks allow the testing of proposed real-time software design and implementation methods. A number of sources of compact applications are listed in Section 4. Some effort will be needed to add real-time requirements to these existing specifications. A template for a real-time compact application benchmark specification is given in Appendix A using the two-dimensional FFT as an example.

Section 4

Related Efforts

In this section we describe four existing benchmarking efforts/methodologies and compare and contrast them to our proposed effort. A collection of more focused efforts that also could be a source of benchmarking kernels/compact applications are also listed.

4.1 PARKBENCH

The Parallel Kernels and Benchmarks (PARKBENCH) committee was founded at Supercomputing'92 to establish a comprehensive set of benchmarks that will be generally accepted by both scientific computing users and vendors of parallel systems [Hockney and Berry, 1994]. One of the objectives of the PARKBENCH committee is to set standards for benchmarking methodology and the way results are to be reported. They are developing a rigorous and scientifically tenable methodology for studying the performance of high-performance computer systems. Where relevant, we will adopt their philosophy and terminology. The focus on scientific computing and non-real-time "batch" processing means that their methodology must be augmented along the lines described above to be applicable to real-time embedded applications.

The 1994 PARKBENCH methodology stipulates specific languages and communications software (mainly F77 with PVM message passing) for their parallel benchmarks and views them as a measure of the hardware and compiler. Thus, one can compare two different Fortran compilers for the same machine, but one cannot use their benchmarks to compare a Fortran implementation with a C implementation, or PVM message passing with another kind. Our benchmarks are intended not to be language specific (although portability issues do arise) and are to be thought of as measuring the complete combination of hardware and supporting system software. For instance, the operating system is a significant variable, especially in the real-time context. It is not mentioned in the PARKBENCH material.

4.2 C³I Parallel Benchmark Suite

The Rome Laboratory C³I Parallel Benchmark Suite is a 30-month program (beginning in October 1994) whose goal is to develop a suite of benchmarks aimed at furthering the use of high-performance computing for future C³I systems [Metzger, 1994]. Honeywell Technology Center, Minneapolis, MN, is the prime contractor and is being supported by ALPHATECH, Inc., and Vipin Kumar (Univ. of Minnesota). The benchmark suite is to be designed to assess parallel hardware architectures as well as parallel software development tools and processes. The program will select core algorithms from computationally challenging C³I applications and specify a set of derived benchmarks along the lines of the NAS Parallel Benchmarks. A key objective of the program is to disseminate the benchmarks and initial implementations as widely as possible, including the use of World Wide Web sites and netlib.

Benchmark specifications at the level of compact applications in the PARKBENCH classification will be defined. The initial set of 12 application areas include: plot-track assignment, SAR processing, hypothesis testing, route optimization, discrete-event simulation, tracking, threat analysis, decision support systems, terrain masking, map-image correlation, image understanding, and pattern recognition.

“Real time” was not included in the ground rules of the Rome Laboratory C³I Parallel Benchmark Program. Even so, there should be a great deal of synergy between our proposed real-time embedded benchmarking effort and this program. In particular we will use the infrastructure being set up under the Rome Laboratory program to widely disseminate our real-time embedded benchmarking results. We are also very interested in this program’s suggestions relating to the quantification of level of software development effort and other intangibles like “ease-of-use” and associated productivity of using tools as part of the software development process. Finally, the Rome Laboratory C³I Parallel Benchmark Suite will be a rich source of relevant benchmarking specifications.

4.3 Hartstone

Hartstone (Weiderman and Kamenoff, 1992) is a benchmarking methodology designed to assess a system’s real-time performance. This approach incrementally loads the combined hardware-software system and then assesses its ability to meet prescribed real-time constraints. In particular, the purpose of a Hartstone implementation is to measure the *breakdown* point of a real-time system. This is the point when the computational and scheduling load causes a hard deadline to be missed. The Hartstone benchmark series is a collection of synthetic benchmarks that include scenarios with periodic tasks (harmonic and nonharmonic) and aperiodic tasks with hard or soft deadlines. The complexity of these benchmarks place them at the kernel-level in the PARKBENCH classification.

The Hartstone Distributed Benchmark (Kamenoff and Weiderman, 1991) extends the Hartstone methodology to real-time distributed systems. In this setting there is a set of tasks located on different physical nodes that communicate via messages. Both the tasks and the messages between them have real time constraints. As in the uniprocessor case, the methodology increases the workload on the system until a real-time breakdown occurs. In this case, however, communication and task scheduling is considered both separately and in combination. We have done some preliminary experiments with the Distributed Hartstone methodology on the Intel Paragon [Brown, et al., 1994]. As real-time system services become available for massively parallel processors, we will adopt these ideas and develop a series of Hartstone MPP Benchmarks to assess the capacity of this system software.

4.4 Transactions Processing Performance Council Benchmarks

The major commercial applications of scalable massively parallel processors is to intensive database applications such as on-line transactions processing (OLTP) and off-line decision

support. The use of MPPs for these tasks is becoming increasingly common in commercial "command and control" applications. The notion of leveraging these trends and incorporating these same functions into scalable MPPs involved in real-time embedded applications such as military command and control systems is just beginning to be explored. As these ideas take hold there will be an increasing need to expand the scope of the real-time, embedded benchmarking effort to include such database intensive processing.

The Transactions Processing Performance Council was founded in 1988 with a goal to define domain specific benchmarks for transaction processing and database systems. It is interesting to note that the benchmark specifications contain many of the same ideas being proposed here for real-time processing; for instance, the separation of the system under test and the driver system (test bench) and the requirement to assess performance in the "steady state." We will adopt their standard methodology and benchmarks (denoted by TCP-A, TCP-B, and TCP-C) to the emerging real-time embedded database applications. A major difference is expected to be the time scales at which the processing must be completed. For instance, decision support queries will need to be completed in cycle times on the order of minutes, or even seconds, as opposed to hours.

4.5 Other Focused Efforts

There are a variety of programs that are focusing on various real-time embedded applications from which it should be possible to extract kernels and compact application benchmarks. Examples involving STAP include the DARPA/Lincoln Laboratory "Mountain Top" and the Rome Laboratory MultiChannel Airborne Radar Measurement (MCARM) programs. We have extracted a real-time STAP benchmark specification from previous off-line analysis of MCARM data. Examples involving SAR and automatic target recognition include efforts involving the Lincoln Laboratory Advanced Detection Technology Sensor and the Sandia National Laboratory's SAR and automatic target recognition test bed.

A particularly compelling future use of scalable high performance computers will be in applications that blur the distinction between signal processing and data processing, as in recent "smart" signal processing approaches. An example of this is a decision-directed SAR image formation procedure called planar subarray processing (PSAP) [Perry, et al., 1994] currently under development as part of DARPA/STO's "Clipping Service" SAR program.

Finally, there is also a need to coordinate with other complementary research efforts. For example, the ARPA-sponsored Rapid Prototyping of Application Specific Signal Processors (RASSP) program is focusing on improving the process of developing application-specific processors for many of these same real-time embedded applications. The RASSP program has a number of application demonstrations underway that could provide benchmarks to be implemented with general-purpose massively parallel processors. Our initial implementation on the Intel Paragon [Brown, et al., 1995a, b] of the RASSP SAR benchmark [Zuerndoerfer and Shaw, 1994] is a good example of this. Also, the tools that the RASSP program is developing at the front end of the design process can be applied to the parallel software development

problem, and these tools need to be compared to similar tools that are being developed by the high performance computing community.

Section 5

Scalability Comparisons

In this section we discuss previous approaches to scalability developed by the scientific computing community and contrast them to our notion of scalability for real-time embedded parallel processing. Traditionally parallel processing for scientific applications has used multiple processors to reduce the amount of time (latency) required to execute an algorithm for a single problem instance. In real-time applications of parallel processing, the goal is to meet specified real-time requirements. For example, in signal processing applications, there are usually both period and latency requirements involving multiple problem instances. In Section 2.5, we contrasted various “efficiency” measures relevant to these two domains. This section examines scalability measures in a similar way. We first review a number of scalability approaches that have been applied in scientific computing.

The progression of reduced execution times as additional processors are added is often regarded as a key descriptor of how effective parallel processing is for the particular application. A system’s *scalability* refers to the rate of reduction of run times as the number of processors is increased. The notion of “speed up”, $S(n)$ in the notation of Section 2.5, has been used to compare the run time of various sized systems. For a fixed problem size, there is a limit to the speed up that can be obtained with some applications bottoming out quickly because they contain large proportions of unavoidably sequential operation (Amdahl’s law). As a result, there have been alternative measures of scalability proposed that describe the behavior of the system as both the problem size and the number of processors are increased [Gustafson, 1988].

In the *fixed-time* approach [Gustafson, et al., 1990, 1991], an algorithm-machine combination is evaluated in terms of how much work it can perform in a fixed amount of time. In the scalable language-independent Ames Laboratory one-minute measurement (SLALOM) benchmark, the execution time is limited to one minute. The problem size grows with system size, and the largest problem size that can be accommodated is limited by the prescribed execution time. This problem size increase varies with algorithms, machines, and their combination, and this rate of growth provides a quantitative measurement of scalability.

In the *isospeed* approach [Sun and Rover, 1994], an algorithm-machine combination is scalable if the achieved average speed of the implementation (often expressed as flop/s-per-processor and computed over the duration of the run) on the given machine can remain constant with increasing numbers of processors, provided the problem size can be increased with system size. Again, the rate of growth of the problem size required to maintain a fixed average speed as the number of processor grows provides a quantitative measurement of scalability.

In the *isoefficiency* approach [Grama, et al., 1993], an algorithm-machine combination is scalable if the efficiency (defined in this setting to be $S(n)/n$) on the given machine can remain

constant with increasing numbers of processors, provided the problem size can be increased with system size. Again, the rate of growth of the problem size required to maintain a fixed efficiency as the number of processor grows provides a quantitative measurement of scalability.

The fixed time scalability approach is perhaps closest in spirit to what we are proposing for real-time embedded applications. We instead emphasize a *dual* notion: for a given problem size, determine the smallest machine size that is required to meet a prescribed real-time requirement (often including, for example, both period and latency constraints). As the problem size and potentially the real-time requirement are scaled, the rate of increase in this minimum system size is what we suggest be used to compare alternative solutions.

The isospeed concept is somewhat relevant to the present context since maintaining constant average speed is related to the notion of achieving a fixed percentage of the peak processing rate (processing utilization) as the problem and machine sizes are increased. The notion of processing utilization defined in Section 2.5 was concerned with periodic situations. Pipelining or “round robin” are common parallel processing techniques used to maintain a constant period and processing utilization independent of problem size. However, the processing latency for any single instance increases with the depth of the pipeline or the amount of round robin replication, and so this solution would not satisfy the isospeed criterion as defined for latency.

As it stands, the isoefficiency metric is less easily related to our current concerns. Maintaining a fixed average speed up appears to bear little relation to satisfying a prescribed timing requirement. In Section 2.5, a similar notion of efficiency was defined for periodic processing, and a connection was made between this measure of efficiency being near 1 and maintaining a constant level of processing utilization. There the focus was on a given n -processor machine and a fixed problem, and the minimum period that could be sustained needed to be determined.

We have proposed an alternative benchmarking perspective that explicitly treats real-time requirements (periods, latency, responsiveness) when considering the scalability of a parallel processing solution for real-time embedded applications. Rather than focusing on increasing the machine size and then determining the problem size that is needed to accomplish some scalability criterion, our proposed approach is more “problem-centric.” For a given problem, we determine the machine size required to meet a specified timing requirement, and then scale the problem complexity (including potentially the real-time requirements). For a given real-time implementation, the software development process might determine the period and latency satisfied as a function of the number of processors n . Although, such parametric studies are closer to the original motivation of parallel processing, the multiple problem instances found in periodic processing situations provide relief from Amdahl’s law. As long as the real-time latency requirement allows for it, adequate processing utilization can often be maintained by processing parts of many problem instances simultaneously.

Section 6

Conclusion

There are a plethora of benchmarks associated with large-scale scientific computing and significant controversy surrounding their use and interpretations. Many current benchmarking efforts embrace the “big is better” mentality of large-scale scientific computing, which is not relevant to embedded applications, and ignore real-time requirements or the need to characterize performance variability. Irrelevant benchmarking will become an increasing problem as researchers less familiar with requirements of real-time embedded computing are drawn in by this new application area. There is a clear need for coordination across the embedded high performance computing field to standardize an appropriate benchmarking methodology for real-time embedded applications.

This paper proposed a benchmarking methodology that treats functional and temporal performance on an equal footing and that compares solutions in terms of the minimal hardware required to meet fixed real-time specifications. The assessments are conducted using a software test bench that measures the steady-state real-time behavior. The design-to-specification methodology can be coupled with increasing problem complexity to produce meaningful real-time scalability assessments. Strict size, weight, and power requirements are overriding concerns in embedded computing, and the proposed benchmarking approach is designed to make relevant assessments in this regard.

Three levels of benchmark complexity were considered. Low-level benchmarks are designed to narrowly focus on the performance of some crucial component of the parallel system. Kernel-level benchmarks provide useful parametric information on the computational and communication building blocks of subsequent real-time implementations. Finally, compact application benchmarks incorporate representative system behavior and are implemented according to the design-to-specification methodology described in this paper. Appendix A provides a template for a real-time compact application benchmark specification using the two-dimensional FFT as an example. Appendix B describes an associated kernel-level data remapping benchmark.

List of References

- Brown, C. P., M. I. Flanzbaum, R. A. Games, J. D. Ramsdell, 1994, *Real-Time Embedded High Performance Computing: Application Benchmarks*, MTR 94B145, The MITRE Corporation, Bedford, Massachusetts.
- Brown, C. P., R. A. Games, J. J. Vaccaro, 1995a, "Real-Time Implementation of the RASSP SAR Benchmark on the Intel Paragon," *Proceedings of the 2nd Annual RASSP Conference*, ARPA, pp. 191-195.
- Brown, C. P., R. A. Games, J. J. Vaccaro, 1995b, *Real-Time Parallel Software Design Case Study: Implementation of the RASSP SAR Benchmark on the Intel Paragon*, MTR 95B95, The MITRE Corporation, Bedford, Massachusetts.
- Brown, C. P., R. A. Games, A. Kanevsky, and L. G. Monk, 1996, *Real-Time Communications Scheduling: Fiscal Year 1995 Summary*, MTR 96B8, The MITRE Corporation, Bedford, Massachusetts.
- Games, R. A., A. Kanevsky, P. C. Krupp, and L. G. Monk, 1994, *Real-Time Embedded High Performance Computing: Communications Scheduling*, MTR 94B146, The MITRE Corporation, Bedford, Massachusetts.
- Games, R. A., A. Kanevsky, P. C. Krupp, and L. G. Monk, 1995, "Real-Time Communications Scheduling for Massively Parallel Processors," *Proceedings of the Real-Time Technology and Applications Symposium, 15-17 May 1995, Chicago, IL*, IEEE Catalog Number 95TH8055, pp. 76-85.
- Grama, A. Y., A. Gupta, and V. Kumar, 1993, "Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures," *IEEE Parallel and Distributed Technology*, Vol. 1, No. 3, pp. 12-21.
- Gustafson, J. L., 1988, "Reevaluating Amdahl's Law," *Communications of the ACM*, Vol. 31, No. 5, pp. 532-533.
- Gustafson, J. L., D. T. Rover, S. Elbert, and M. Carter, 1990, "SLALOM, the First Scalable Supercomputer Computer Benchmark," *Supercomputer Review*, 3(11), 56-61.
- Gustafson, J. L., D. T. Rover, S. Elbert, and M. Carter, 1991, "The Design of a Scalable, Fixed-Time Computer Benchmark," *J. Parallel Distributed Computing*, Vol. 11.
- Hockney, R., and M. Berry, 1994, *Public International Benchmarks for Parallel Computers*, PARKBENCH Committee: Report-1.

Kamenoff, N. I., and N. H. Wiederman, 1991, Hartstone Distributed Benchmark: Requirements and Definitions, *Proceedings of the 1991 Real-Time Systems Symposium, San Antonio, TX, 4-6 November 1991*, 0-8186-2450-7/91, Institute of Electrical and Electronic Engineers, Inc., New York, NY.

Koester, D. P., and J. J. Rushanan, 1996, *Real-Time Parallel Software Design Case Study: Implementation of the RT_2DFFT Benchmark on the MasPar MP-X Architecture*, MTR 96B9, The MITRE Corporation, Bedford, Massachusetts.

Metzger, R. C., 1994, "C³I Parallel Benchmark Suite," Program Description, Rome Laboratory (<http://www.se.rl.af.mil:8001>).

Perry, R. P., R. C. DiPietro, A. Kozma, J. J. Vaccaro, April 1994, "SAR Image Formation Processing Using Planar Subarrays," *Proceedings of the SPIE OE/Aerospace Sensing, Conference on Algorithms for Synthetic Aperture Radar Imagery*, Orlando, FL.

Press, W. H., B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, 1986, *Numerical Recipes*, Cambridge University Press, New York, NY.

Sahni, S., and V. Thanvantri, 1996, "Performance Metrics: Keeping the Focus on Runtime," *IEEE Parallel and Distributed Technology*, Vol. 4, No. 1, pp. 43-56.

Shaw, G. A., 1994, "RASSP Benchmark Program Overview," *Proceedings of the 1st Annual RASSP Conference*, ARPA, pp. 33-42.

Sun, X.-H., and D. T. Rover, 1994, "Scalability of Parallel Algorithm-Machine Combinations," *Parallel and Distributed Systems*, Vol. 5, No. 6, pp. 599-613.

Weiderman, N. H., and N. I. Kamenoff, 1992, "Hartstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time Systems," *Journal of Real-Time Systems*, Vol. 4, pp. 353-382.

Zuerndoerfer, B., and G. A. Shaw, 1994, "SAR Processing for RASSP Application," *Proceedings of the 1st Annual RASSP Conference*, ARPA, pp. 253-268.

Appendix A

RT_2DFFT: Real-Time Two-Dimensional FFT Benchmark Specification

The RT_2DFFT benchmark is designed to assess the performance of a two-dimensional fast Fourier transform (FFT). The two-dimensional FFT is a kernel commonly used in synthetic aperture radar (SAR) processing, among other applications. In this benchmark, the two-dimensional FFT is treated as a compact application to illustrate the proposed real-time benchmarking methodology. The benchmark assesses the parallel processor's ability to deliver high sustained utilization on an FFT. It also assesses the performance impact of a "distributed corner turn" global communication operation (see Appendix B). This is particularly relevant as the problem size increases and more distributed processing resources are required to meet the real-time requirement (as in higher resolution and/or wider area SARs). Strict and loose latency requirements are both considered in the benchmark. The RT_2DFFT benchmark specification consists of a functional specification, a timing specification, a scalability study specification, and implementation guidelines.

Notation: In the following the complex numbers are denoted by \mathbf{C} . The set of vectors of size n with entries in a set A is denoted by A^n ; the set of two-dimensional arrays of size $m \times n$ is denoted by $A^{m \times n}$. For an array $X = \{x_{ij}\}$ in $A^{m \times n}$, the notation $x_{i,*}$ denotes the i th row of X and $x_{*,j}$ denotes the j th column of X .

Functional Specification: We assume the availability of an FFT algorithm, $y = \text{FFT}(x, \text{FFTsize})$, where x and y are complex vectors of size FFTsize and y is the discrete Fourier transform (DFT) of x . See for instance [Press, et al., 1986, page 381]. For n a positive integer, the RT_2DFFT benchmark applies the FFT to the rows of an $n \times n$ input matrix X to form the intermediate matrix Y . The output matrix Z is then formed by applying the FFT to the columns of Y . The functional specification of the RT_2DFFT benchmark is given in Figure A-1. Single precision floating point processing is assumed.

$y = \text{RT_2DFFT}(X, n)$ Input : n a positive integer Input : $X = \{x_{ij}\} \in \mathbb{C}^{n \times n}$ Auxiliary : $Y = \{y_{ij}\} \in \mathbb{C}^{n \times n}$ Output : $Z = \{z_{ij}\} \in \mathbb{C}^{n \times n}$ Algorithm : for $i \in \{0, 1, \dots, n-1\}$, $y_{i,*} = \text{FFT}(x_{i,*}, n)$ for $j \in \{0, 1, \dots, n-1\}$, $z_{*,j} = \text{FFT}(y_{*,j}, n)$
--

Figure A-1. The RT_2DFFT Benchmark Functional Specification

Timing Specification: The timing specification of the RT_2DFFT benchmark is given in terms of a periodic sequence of input matrices $X_1, X_2, X_3, \dots, X_i, \dots$. Two requirements are typical in these periodic applications: the *period* is the time interval between successive input matrices and the *latency* is the length of time required to process a single instance X_i , measured as the interval of time between when the matrix X_i leaves the data source and the corresponding results arrive at the data sink. This benchmark fixes the period at 1 second and considers two separate latency cases:

- Case 1: period = latency = 1 second
- Case 2: period = 1 second, no constraint on latency

The 1 second period for $n = 1024$ or 2048 corresponds roughly to the computational requirements of the SAR system described in [Zuerndoerfer and Shaw, 1994]. In the RT_2DFFT benchmark the timing specification is fixed for all problem sizes.

Scalability Study Specification: The scalability study for the RT_2DFFT benchmark increases the size of the $n \times n$ input matrix, while the period and latency is kept fixed. The objective is to determine the minimum machine size that meets the real-time requirement. The values of n to be considered are: 256, 512, 1024, 2048, 4096, 8192, and 16,384. Table A-1 shows the computational throughput requirements expressed in Mflop/s. We assume an FFT of length n requires $5n \log_2 n$ floating point operations, so that the total number of floating point operations for the RT_2DFFT benchmark is $10n^2 \log_2 n$. This requirement is common to both latency cases and is based on the single period of 1 second. Table A-1 also shows the memory requirements in megabytes (MB) to store one copy of the input matrix, assuming 8 bytes per element (single precision floating point complex).

**Table A-1. Processing Rate and Memory Requirements
for the RT_2DFFT Benchmark**

n	Mflop/s	MB
256	5.2	.5
512	23.6	2
1,024	104.9	8
2,048	461.4	32
4,096	2,013.3	128
8,196	8,724.2	512
16,384	37,581.0	2,048

Distributed Memory Implementation Guidelines

1. The input matrix must be stored originally on a source node(s) in memory that is not directly associated with the processors that implement the RT_2DFFT function. The matrix must be stored in row major or column major form.
2. When establishing timing performance the same matrix can be repeatedly input to the processors that implement the RT_2DFFT function (to avoid the need for disk I/O).
3. The results must be output to a sink node(s) and stored in memory not directly associated with the processors that implement the RT_2DFFT function. The result matrix must be stored in row major or column major form.
4. The source and sink nodes may be implemented on the same or different processing nodes.
5. The processing latency for a problem instance is measured as follows. A time stamp t_s is calculated at the data source right before the input data for this instance is sent from the source node. A second time stamp t_c is calculated at the data sink right after the corresponding results are received by the sink node. The processing latency for this problem instance is then $t_c - t_s$. This requires a synchronized global clock if the source and sink are on physically separated nodes. Period measurements are calculated as the difference of successive values of t_c corresponding to successive problem instances. Latency and period measurements can be calculated off-line from the time stamp data. Care must be taken that data collection does not cause any unintended disk I/O to occur during a run.
6. In the case that the input matrix (and output matrix) does not fit in the memory of a single node, then multiple source and sink nodes are necessary. The time stamp t_s of a problem instance should occur before any data is sent from the multiple sources. The processing of that instant is considered completed when all sink nodes have received all their results.

7. A benchmark run to establish valid timing performance should last for at least 15 minutes to account for any operating system dropout problems.
8. The following information and statistics should be calculated for a single benchmark run (during a post processing stage):
 - a. Histogram of period measurements. Maximum, average, and minimum period. The benchmark is considered valid only if the maximum period observed is less than or equal to the period specification: 1 second. This must be repeatable.
 - b. Histogram of latency measurements. Maximum, average, and minimum latency. The benchmark is considered valid only if the maximum latency observed is less than or equal to the latency specification. This must be repeatable.
 - c. Some small number of initial problem instances can be ignored to eliminate start-up anomalies, if present. If this is done, then the number of ignored instances should be stated.
9. For each latency case and for each feasible problem size, the smallest machine that produces a valid benchmark result should be determined. Machine size is measured in terms of the number of processing nodes used, not including processing nodes used to implement the source and sink. The only exception to this rule is that any "excess" source and sink nodes beyond the minimum number that are required to store one copy of the input and output matrices should also be counted in the machine size. Standard commercial-off-the-shelf hardware and system software configurations should be used. If a machine supports multiple configurations (for example, different amounts of memory at the processing nodes), then these different configurations must be itemized and benchmarked separately.
10. The maximum period measurement for a benchmark run is used to determine the sustained Mflop/s processing rate: $10n^2 \log_2 n / (\text{maximum period measurement})$. This value should be divided by the theoretical peak processing rate of the size machine used in the processing to determine the processing utilization percentage.
11. The following scalability plots should be generated for each latency case:
 - a. Minimum machine size as a function of problem size.
 - b. Processing utilization percentage as a function of problem size.

Appendix B

RT_CornerTurn: Real-Time Distributed Corner Turn Benchmark Specification

The RT_CornerTurn benchmark is designed to assess the performance of the “distributed corner turn” global communication operation. It is a kernel-level benchmark designed to provide useful parametric information to support subsequent real-time implementations. Corner turning is the name given for transposing a matrix in some signal processing applications. To meet stringent real-time latency constraints the rows or columns of the matrices involved are often distributed across many processing nodes.

The corner turn data redistribution operation usually consists of three phases as shown for the case of four processing nodes in Figure B-1. The rows of the matrix are assumed to be divided equally among the four processing nodes. First there are memory transposes or local “corner turns” at the nodes that take each node’s portion of the matrix stored by rows, say, and restores it by columns. This is required if the information is to be packaged into large messages for more efficient transmission. The second phase corresponds to the data distribution phase shown in step 2 of Figure B-1. In this “all-to-all” communication step each node communicates some portion of its data to every other node. Finally, a second memory copy at the processing nodes is needed to unpack the messages and to store the result by columns for subsequent processing.

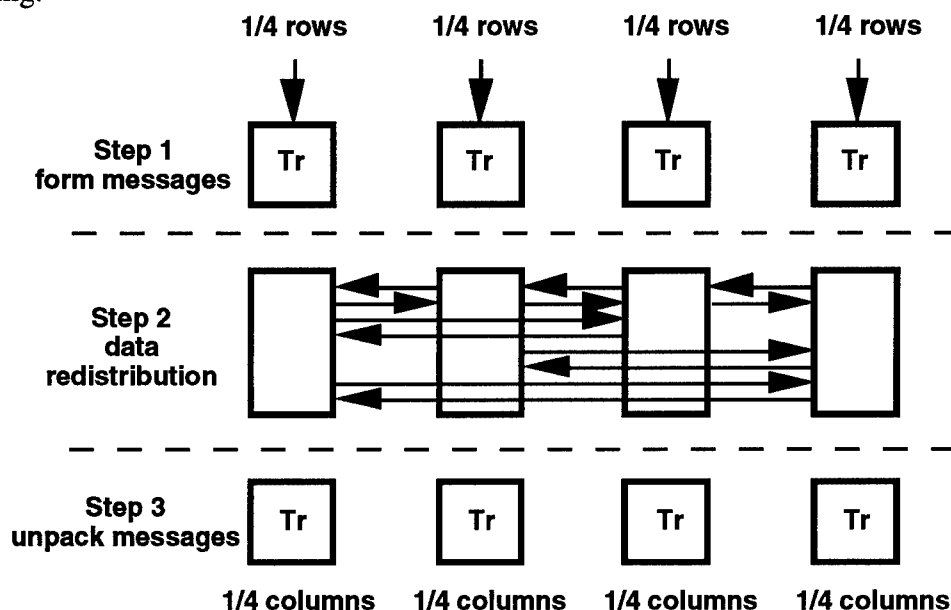


Figure B-1. Corner Turning

There are two versions of the RT_CornerTurn benchmark: “in place” and “pipelined.” In the “in place” version, steps 1, 2, and 3 take place on the same group of N processing nodes. This corresponds to the case that the processing and corner turn occur in place on a single set of nodes that process both the rows and columns. In the “pipelined” version, step 1 occurs on a group of “source” nodes, the step 2 data redistribution occurs over the network connecting the source nodes and a group of “sink” nodes, and step 3 occurs on the sink nodes. This corresponds to the case that the source nodes process the rows of the matrix and the sink nodes process the columns.

For each value of n and N , the corner turn function should be individually timed 1000 times. There are two approaches for obtaining the time for a single corner turn iteration: (1) timing at a single processing node and (2) taking the maximum of all processing node times. For simplicity we initially consider option 1. A barrier synchronization is implemented before each corner turn iteration to separate the communications involved for each iteration. Since the variability of the results are of interest for real-time applications, a histogram should be computed from these 1000 measurements along with the minimum, average, and maximum times. Some initial number of iterations can be added to account for any start up anomalies before the 1000 timings are computed.

Sample C code based on the Message Passing Interface (MPI) is given below. The variable `comm_only` can be set to a nonzero value if only the data redistribution portion is to be timed.

```
for (i = 0; i < ITER; i++)
{
    MPI_Barrier (MPI_COMM_WORLD);
    t = get_the_time ();

    if (comm_only == 0)
        step1_transpose (buffA, buffB);

    status = MPI_Alltoall (send_buff, chunk_size,
                          MPI_CHAR,recv_buff, chunk_size,
                          MPI_CHAR,MPI_COMM_WORLD);

    if (comm_only == 0)
        step3_reorder (recv_buff, send_buff);

    delta[i] = (double)((get_the_time () - t)) /
               UNITS_PER_SEC;
}
```


Scalability Study Specification

The scalability study for the `RT_CornerTurn` benchmark increases the number of processing nodes N while the $n \times n$ input matrix size is kept fixed. Separate studies are performed for individual values of n to be considered: 256, 512, 1024, 2048, 4096, 8192, and 16,384. In this case the amount of work involved in step 1 and step 3 and the message size in step 2 decrease as the machine size is increased. However the number of messages involved in step 2 increases, and the time to perform this message passing will eventually dominate.

The `RT_CornerTurn` benchmark assumes that an $n \times n$ input matrix of complex numbers is stored "by rows" as evenly as possible on N processors. The input parameter n is fixed and the number of processors N is varied over a range of values for each benchmark run. The lower bound on N is the number of nodes required to store the matrix and any buffers used. The upper bound on N is determined by the machine size (or the input matrix size). The corner turn operation remaps the data and stores the matrix "by columns" as evenly as possible. No particular approach to the corner turn is specified. In particular multi-stage procedures are allowed.

Usually group communication benchmarks of this sort assume a fixed message size and let the machine size increase or, for a fixed machine size, let the message size increase. In both cases the underlying matrix size is scaled up as part of the study. Scalability approaches that scale the problem size are not as relevant for the real-time applications under consideration. In these real-time applications data-parallel processing is usually employed across the rows and columns of a fixed matrix to meet prescribed real-time period and latency constraints. What needs to be determined is the overhead contributed by the distributed corner turn operation as a function of machine size. This information, along with the processing rates sustainable on the row and column processing can be used to size a real-time implementation. Characterizing the variability is important also to allow enough slack in the implementation so that real-time requirements are not violated.

MISSION
OF
ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.