Cluster-C*: Understanding the Performance Limits

Charles J. Turner
David Mosberger
Larry L. Peterson [1]

TR 94-09

Abstract

Data parallel languages are gaining interest as it becomes clear that they support a wider range of computation than previously believed. With improved network technology, it is now feasible to build data parallel supercomputers using traditional RISC-based workstations connected by a high-speed network. This paper presents an in-depth look at the communication behavior of nine C* programs. It also compares the performance of these programs on both a cluster of 8 HP 720 workstations and a 32 node (128 Vector Unit) CM-5. The result is that under some conditions, the cluster is faster on an absolute scale, and that on a relative, per-node scale, the cluster delivers superior performance in all cases.

August 9, 1994

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by ARPA Contract DABT63-91-C-0030 and DACA76-93-C-0026.

1 Introduction

This paper considers the question of whether data parallel programs can be effectively executed on a cluster of workstations connected by a high-speed local area network. Both aspects of this question are widely recognized as having intrinsic merit in their own right: the data parallel model offers a high potential for concurrency and is generally considered easy to program [6], and loosely-coupled workstation clusters offer an attractive cost-performance alternative to tightly-coupled multiprocessors [3, 2]. Addressing the intersection of these two topics, therefore, has a high potential for payoff.

The problem, of course, is that the fine-grained parallelism common in data parallel programs is assumed to be inappropriate for workstation clusters because of the associated high latency and low bandwidth of the network. Three factors lead us to believe that this assumption is flawed. First, recent advances in network technology have yielded nearly an order of magnitude improvement in both latency and throughput, meaning that the performance gap between loosely-coupled and tightly-coupled multiprocessors is not as dramatic as it once was. Second, analysis of a collection of data parallel programs indicates that although fine-grained, they still have well-bounded communication requirements. Third, it is possible to design an architecture that minimizes the remaining communication and control overhead by leveraging the predictable lexical and execution behavior of data parallel programs.

To test this hypothesis, we have implemented an environment for C* [4]—a data parallel language developed by Thinking Machines Corporation (TMC) for the Connection Machine family of parallel processors—on a cluster of Hewlett Packard 720 workstations connected by an FDDI network. Each workstation includes an experimental network adapter board designed for low latency, and runs a minimal OS kernel. We then experimented with a comprehensive set of test programs characteristic of operational image understanding systems. These algorithms are generally interesting because they are adaptive to diverse data inputs, span multiple processing levels (e.g., numeric, pixel-based, object-based), utilize all available C* communication operators, and are scalable to handle increasing sensor collection capabilities.

After giving an overview of C*'s communication needs and briefly describing the communication substrate that underlies our implementation of C* on a workstation cluster, this paper presents the results of a comprehensive study of the performance of the C* test suite running on both an HP workstation cluster and a more tightly-coupled multiprocessor architecture, the CM-5 [6]. From this study we are able to draw several conclusions about the appropriateness of loosely-coupled systems as execution platforms for data parallel programs.

2 Background

This section provides an overview of the C* language, a detailed description of the major communication operators provided by the language, and a discussion of the communication properties of the C* language and application programs.

2.1 C* Overview

C* adds parallelism to sequential C by extending the language to support declaration of, and operations on, parallel vector data. The fundamental unit of computation in a data parallel program is the *element*. The basic data elements of each problem serve as the building blocks for all higher-level structures: pixels of an image, nodes and arcs of a graph, and so on. Elements are organized into aggregates—homogeneous structures called *shapes*—which define the rank and dimensions of a collection of elements. Shapes may be declared either statically or dynamically, with the latter method being necessary for efficient processing of data-dependent problems.

Associated with each shape, a program may allocate *parallel variables* (*pvars*), based on standard C types. Pvars

can be allocated statically, as automatic variables, or dynamically as part of a pvar heap. Corresponding elements of pvars can be combined using parallel versions of standard C operators; given two integer pvars, a and b, of the same shape, $a + b$ computes the sum of the elements of a and b at each location in the shape. Image processing programs frequently utilize multiple shapes of varying configurations, switching between shapes as processing demands. All variables in the sequential portions of a C* programs are called *scalar* variables.

To maximize the expression of parallelism available within a problem, processing is introduced at the finest granularity possible—the element at every location of a shape. Processing is expressed using the concept of a *virtual processor* (*vp*), to abstract away any limits in the number of available physical processors. The ratio of virtual to physical processors is called the *vp ratio*. For each C* operation, each physical processor iterates sequentially over its local virtual processors. Activity of the virtual processors can be manipulated during execution with the use of the *where* statement, which *sets the context* for the duration of the block. C* retains C's single thread semantics; it is not possible for two different nodes in the cluster to execute different control flows during program execution. For example, all processors execute the same branch of each conditional.

2.2 C* Communication Operators

C* supports a wide range of communication patterns through a small set of structured operators. These operators are invoked using C* language constructs and library functions. This is in contrast to lower-level approaches requiring the user to explicitly package and transmit blocks of data. C* operators support various scalar-pvar and pvar-pvar communication operations. Both implicit and explicit addressing are supported: implicit for regular, neighborhood operations, and explicit for global or application-specified transformations. C* shapes provide a global address space for all parallel variables, allowing any element of a shape to address any other element regardless of physical location. Additionally, most operators support some form of computation in combination with the communication operation. The operations are summarized as follows:

Reduction. Computes a single scalar value from the application of a binary associative operator to the active elements of a pvar. C* supports the reductions *minimum, maximum, sum, product, bitwise logical or, logical and*, and *logical xor*. Reductions are typically used to compute global statistics or to propagate vp state up to the sequential control flow of the program.

Grid-Send. Also called NEWS send (north, east, west, south), it supports regular movement of data between neighboring elements along one axis of an N dimensional Cartesian grid. Grid communication is more efficient to implement than general communication because it uses implicit addressing and has a regularity that eliminates all congestion at the element level (i.e., no collisions). Grid-sends are used by neighborhood or window-based algorithms.

General Send. Also called the permutation operator, it utilizes the global address space of C* to provide communication between any two pvar elements within the same or different shapes. All source and destination addresses are specified on a per-element basis, allowing the permutation pattern to be constructed dynamically, by the application. Values sent to the same destination element can be combined using *add, multiply, min, max*, or *overwrite*. For example, the statement $[value]accum \mathrel{+}= 1$ is a convenient way to implement a histogram.

Scan. Also called the parallel prefix operator, it is derived from an operator of the same name first introduced in the APL language. C* supports scan for the binary associative operators *minimum, maximum, sum, copy, bitwise logical or, logical and*, and *logical xor*. When applied to a pvar, the scan operator computes all partial products of that operator, moving along one dimension of the shape, with optional segment masking.

Note that most previous work involving data parallel languages has focused on Fortran D [7]. This is significant because C* emphasizes a much richer interconnection of parallel data. This is best illustrated by the varied applications of interest to the two communities: Fortran users are primarily interested in grid-based, numerical computation, characterized by algorithms such as Jacobi and SOR, while C* is often the language of choice for computer vision programs because of the need for general sends and scans.

2.3 Communication Behavior

Whether or not a workstation cluster is appropriate for data parallel program execution is primarily dependent on the communication behavior of the programs. It is frequently assumed that fine-grained expression of parallelism necessarily implies a higher level of physical communication. We have examined a suite of data parallel programs and found that they have restricted use of communication operators in proportion to computational operations. Using static instruction frequency, the programs vary from 4 to 13% communication instructions. Dynamic instruction frequency shows an average of 8.5% communication.

Because the communication operators are part of the C* language (and libraries), it is possible to examine the source of a C* program and predict more about its dynamic communication behavior than would be possible for a similar program that uses unstructured, untyped, manual message passing. The four communication operators have different susceptibility to latency and bandwidth limitations. This is because the semantics of the operators require different amounts of local computation prior to physical communication. For example, a reduction (e.g., summation) of a float pvar, requires that each node computes the sum of all its locally allocated elements, but broadcasts only a single 32-bit result. This operation has limited bandwidth requirements, and is relatively insensitive to latency due to the many local floating point additions required.

C* operator	1K	256K	1M	32x32	512x512	1Kx1K
reduce	4	4	4	4	4	4
grid-send	8	8	8	128	2K	4K
scan	4	4	4	128	2K	4K
send	8K	2M	8M	8K	2M	8M

Table 1: Communication Operator Payload Sizes by Problem Size and Shape

Problem size is important in determining the significance of latencies caused by software and communication overheads. If the overall problem size and physical communication requirements grow at different rates, then the ratio of communication to computation will vary, making the networking component more or less significant. Sources of fixed overhead include all of the sequential portions of the C* program, the communication operation invocation and setup, and physical communication latency. If the number of virtual processors being emulated is too small, then fixed overheads can dominate overall execution time. On the other hand, when problem sizes are large, fixed overheads become less significant. If absolute latency is small then it will only be significant to small problems. The cluster architecture employs a number of optimizations to both reduce sources of latency, as well as techniques for hiding any remaining latency.

The bandwidth requirements of each communication operator also depend on program size and shape. Each communication operation requires a mixture of local and physical communication. The extent of the non-local or physical communication is determined by the shape of the computation and the way the shape is mapped onto the physical processors. Cluster-C* uses block decomposition, which allocates N/P consecutive elements to each processor. Table 1 shows the physical communication requirements for each communication operator under a variety of 1 and 2D shape sizes. One would expect that operators with small payload sizes would be more sensitive to

latency, while large payload operators would stress the bandwidth limits of the network. The ratio of local to non-local communication can be further altered by the send patterns specified by the application program. The more locality in a send operation, the less the need for raw communication bandwidth. For example, local windowing operations have lower physical communication requirements than global transformations like matrix transpose.

3 System Architecture

Cluster-C* consists of eight HP 9000/720 workstations (PA-RISC at 50 MHz). Each machine is equipped with 32 MB RAM and a Medusa FDDI controller [1]. We have measured this hardware's host-to-host throughput and latency to be 98.3Mbps and 47.8μsec, respectively. A minimal OS runs on this platform. It provides a framework for running the three key pieces of the system: the network layer, the C* run-time system (RTS), and the application programs. We use the TMC C* translator to convert C* code into C/Paris—C code with calls to our parallel computation and communication libraries.

The key observation about C* is that it exhibits a highly regular and predictable pattern of communication and synchronization. With information about shape and communication operator behavior, we have defined a model of computation and communication cost, and based on this model, have designed and implemented a highly tuned runtime system and communication protocols. Our design involves five main ideas, which we now summarize; more detail can be found in a companion paper [8].

First, all nodes in the cluster execute a copy of the same program, maintaining duplicate copies of all scalar variables and equal portions of all parallel variables in a manner similar to [5, 7]. This is in contrast to other multiprocessor-based implementations of C*—including both the CM-2 and CM-5—in which a single front-end control processor is connected by a communication tree to an array of processing elements. These systems execute all sequential code on the front-end, broadcasting an instruction stream (on the CM-2) or a stream of instruction block identifiers (on the CM-5) to the processing array. In other words, on the cluster-based implementation, all processors in the cluster act as both the front-end and a processing array member; the only form of synchronization used between nodes is for one to block waiting to receive a data message from another. No explicit synchronization primitives (e.g., barriers) are used between the nodes of the cluster. While this introduces redundant sequential computation, the cost is less than the alternative communication costs. In a cluster environment, any non-uniform identification of such a front-end processor introduces load imbalances and added communication overheads.

Second, we have structured the C* communication operators to support the overlap of communication and computation through message pipelining [7]. Each communication operator is decomposed into two distinct phases: a sending phase, during which local results are computed and sent to the other cluster nodes, and a receiving phase, during which the local processor receives intermediate results or messages from the other cluster nodes. Because C* control flow is identical to that of sequential C, and all references and updates to parallel variables are performed in a well behaved manner, it is possible to perform data-flow analysis on the parallel variables and determine when the variables are *defined*, and where subsequent *use* occurs. The extent to which these two points are separated within the program defines the maximum possible separation of send and receive phases, and therefore the maximum possible overlap of communication and computation. This allows the send phase to be placed immediately after the variable is defined, and the receive phase to be placed immediately before the first possible use of the variable. After a sender initiates a transfer, it can return to computation while the network layer transmits the data, delaying its receive phase until the time when the data is actually required for computation. By separating send and receive portions of the C* communication operations, it is possible to reduce the impact of the network latency.

Third, our system uses a highly optimized (specialized) protocol to transmit messages between physical processors. All C* communication operators are based on asynchronous broadcast and point-to-point send functions. A key observation is that while the number and type of messages sent during the execution of a particular C* program cannot

be determined *a priori*, the C* language does guarantee that messages sent during a single execution of the program will be identical for all nodes in the cluster. We exploit this regularity to define a message indexing scheme that is highly efficient for both the network interface and the language runtime system. This is in contrast to using a general-purpose communication substrate like PVM [2]. We use message vectorization [7] to aggregate multiple virtual processor messages destined for the same physical processor. The network protocols also implement implicit and explicit ACKs and eager retransmission.

Fourth, we use several low-level optimizations to minimize the latency of the communication operations. One optimization is similar to active messages [9], in that as much work as possible is done at interrupt time, thereby minimizing context switch overhead. Interactions between the interrupt handler and the main C* thread are all accomplished without the aid of heavy-weight semaphores. A second optimization is to use a sophisticated buffer management scheme that minimizes how often data has to be copied from one buffer to another.

Finally, the many virtual processors that run on each physical processor are implemented by a single thread; we do not associate a thread with each vp. Thus, the only context switches involved are between the main thread running the application and RTS code, and the interrupt handler that places each incoming message in the appropriate memory location.

4 Test Suite

Our tests come primarily from the disciplines of image processing and computer vision. This area is appealing because it has a wide range of frequently used algorithms ranging from grid-based, numeric algorithms to dynamic, irregular graph-based algorithms. Such diversity requires flexible shape creation and usage, and significant transformations between shapes. The test programs were selected for their representative nature; together they cover all of the C* communication operators in different mixtures. The nine programs are summarized as follows:

amp. Locates all pixels in a 2D input image that are significantly brighter than the average of a surrounding 5×5 neighborhood. All neighborhood summations are computed with 2D horizontal and vertical 32-bit floating point plus-scans and grid-sends. The number of pixels which pass the threshold test are computed with a plus-reduction.

col. Simple mathematical algorithm that demonstrates convergence to zero of a 1D vector of randomly generated initial values. This algorithm provides an example of minimal computation within a reduction-controlled *while* loop, performing a small number of math operations and a plus-reduction within each iteration of the loop.

dens. Computes the local density of a set of selected points in a 2D plane. The initial point locations, specified as a vector of (x, y) pairs, are projected to 1D, where a parallel range sweep algorithm is used to compute all necessary distances. During each iteration, points adjacent in 1D realign themselves for the next comparison using a 1D grid-send. The algorithm terminates when no more points have any neighbors remaining within the maximum distance.

fft. Computes the discrete fast fourier transform of a 1D complex input vector. Different spectral components are computed in $logN$ iterations, each iteration employing a different general send pattern.

hist. Histograms an 8-bit 2D input image into 256 accumulator bins using a general send with an additive combination at the destination. Each vp in the source image sends the constant 1, using its image value as the index into the accumulator array. Values in the input image are uniformly distributed to assure an even distribution of message destinations. A histogram's single communication operation generates $P - 1$ multi-fragment messages sent from all processors, resulting in $P(P - 1)$ total messages on the network.

hyp. Uses the *kmeans* algorithm to perform unsupervised classification of multi- or hyper-spectral imagery. Using B 2D images (where each of the B images contains a different 8-bit spectral band), the algorithm classifies the input image into K classes. Beginning with K initial class centroids, each image pixel computes a B-dimensional Euclidean distance from itself to each of the current K means, assuming the class of the closest centroid. During each iteration, new centroids are computed. Since the K class centroids are maintained as sequential structures, they are computed in each iteration using integer and floating point plus-reductions.

jac. The second non-image-oriented algorithm, provided for comparison purposes, Jacobi is an iterative algorithm to find an approximate solution to Laplace's heat equation, finding the steady-state temperature on a surface. During each iteration, every vp in a 2D grid computes the average of its four immediate neighbors. Jacobi uses only horizontal and vertical 2D grid-sends, and integer addition and division.

mat. Matches a set of input points to a library of templates, attempting to recognize structured objects within a 2D image. Input points are represented as a 1D vector of (x, y) pairs. The Legrange multiplier method is used to obtain the best match of the templates to each subset of candidate image points. Processing requires significant trigonometric and floating point calculation, but absolutely no communication.

obj. Extracts a set of detected objects from a 2D input image, transforming them into a more compact 1D representation, where feature and summary data is computed. Objects are moved from the sparse 2D shape using a general send of all fields of an object structure. Once in 1D, several reduction operators are used to compute global statistics of objects attributes.

5 Experimental Results

Each of the test programs described in the previous section was originally developed for the CM-2; the code was moved from the CM-2 to the cluster and CM-5 with no subsequent optimizations performed for either platform. All floating point computations were performed in 32 bits on both platforms. The cluster programs were compiled with gcc2 using the optimizer. The C* code was compiled with TMC's 7.1 C* compiler [1] that supports optimization of sequential code only. All execution times were measured with either the PA-RISC 20 nanosecond cycle counter on the cluster, or the CM timer facility on the CM-5.

The CM-5 architecture [6] consists of a front-end SPARC2 that runs all sequential portions of a C* program, and controls a back-end processing array consisting of 32 SPARC2 processors, each equipped with four vector units (VUs), for a total of 128 VU's. The array of SPARC2/4-VU groups are connected by two high-speed communication networks—a fat-tree data network for point-to-point and grid-based communication, and a control network used to synchronize computation and support global reductions, broadcasts, and scans.

5.1 Absolute Execution Speed

Figure 1 is a summary graph comparing the results of running each of the nine benchmarks on Cluster-C* and the CM-5. The graph shows the ratio of execution speeds, measured in elements (virtual processors) per second, for the cluster and CM-5 for three different sizes per program. For the purposes of this comparison, a 1K element problem size was selected to represent *small*, resulting in a vp ratio of 8 for the 128 VU CM-5 and 128 for the cluster; *large* was the largest configuration capable of running on the 8-node cluster (depending on the problem, from 32K to 1M elements), and *medium* was the problem size mid-way between these two for each application. The data is plotted on

[1] TMC notice: These results are based upon a beta version of the software and, consequently, is [sic] not necessarily representative of the performance of the full version of this software.

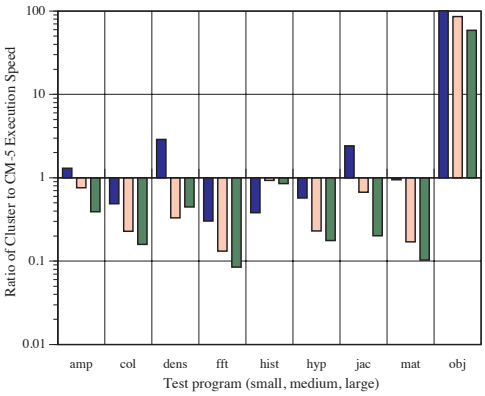

Figure 1: Ratio of Execution Speed (elements/sec) of 8-node Cluster-C* to 128 VU CM-5. Test programs compared for three problem sizes: small, medium, and large.

a log_{10} scale so that cases where the cluster are faster appear above the horizontal center line at 1.0 and cases where the CM-5 is faster appear below the line.

Where the ratio is greater than one, the 8-node cluster was faster than the 128 VU CM-5 in absolute performance; this occurs in 6 out of 27 test cases (22%). The average ratio of all test programs across the three problem sizes was 0.63, indicating that the 128 VU CM-5 was less than twice as fast as an 8-node cluster on average. Only one of the 27 tests showed a case where the cluster was worse than 1/10th the speed of the CM-5 (11.76).

Overall, the cluster performed best in comparison to the CM-5 for small problems and problems that contain a proportionally higher percentage of communication. **Amp, dens**, and **jac** all use grid-oriented communication (either 1 or 2D) in the form of grid-sends and scans, and are faster on the cluster for small sizes. As the problem size grows, however, the ratio of computation to communication increases and the CM-5's vector units become more effective, with the CM-5 becoming 1.6 to 5.0 times faster. For those applications with a limited amount of communication relative to computation, the CM-5's 128 vector units were faster than the eight HP720s.

Applications where communication dominates tend to have better relative performance on the cluster throughout the range of problem sizes. For **hist**, which performs one general send with additive combination of all colliding messages, the CM-5 is only 1.25 times faster than the cluster for all problem sizes. **Obj**, which extracts sparse information for transmission and performs a series of sends followed by a series of global reductions, is faster on the cluster in all cases.

Figure 2 provides the full data summarized by the ratios of Figure 1. It gives the execution speed, measured in elements (virtual processors) per second, obtained by the full configuration on both platforms. Execution speed takes into account CPU performance, communication speed, and software overheads. Plotted as a function of problem size, execution speed provides better insight into performance over a wider range of sizes than does elapsed time.

Figure 2 reconfirms the superior performance of the cluster for **jac, dens**, and **amp** on small to medium problem sizes, as well as **obj** for all sizes. The cluster gets relatively good performance on **hist** over a range of problem sizes. The CM-5 can be seen to delivers superior performance on **col, fft, mat**, and **hyp**—the floating point-intensive problems.

Looking in more detail, we observe that the cluster reaches its peak performance at a much smaller problem size than the CM-5, indicating a lower amount of absolute overhead in control and communication. The cluster also has less absolute variation than the CM-5, indicating a smaller absolute overhead. Finally, the sparse sends of **obj** show no signs of saturating the FDDI ring during the general sends.

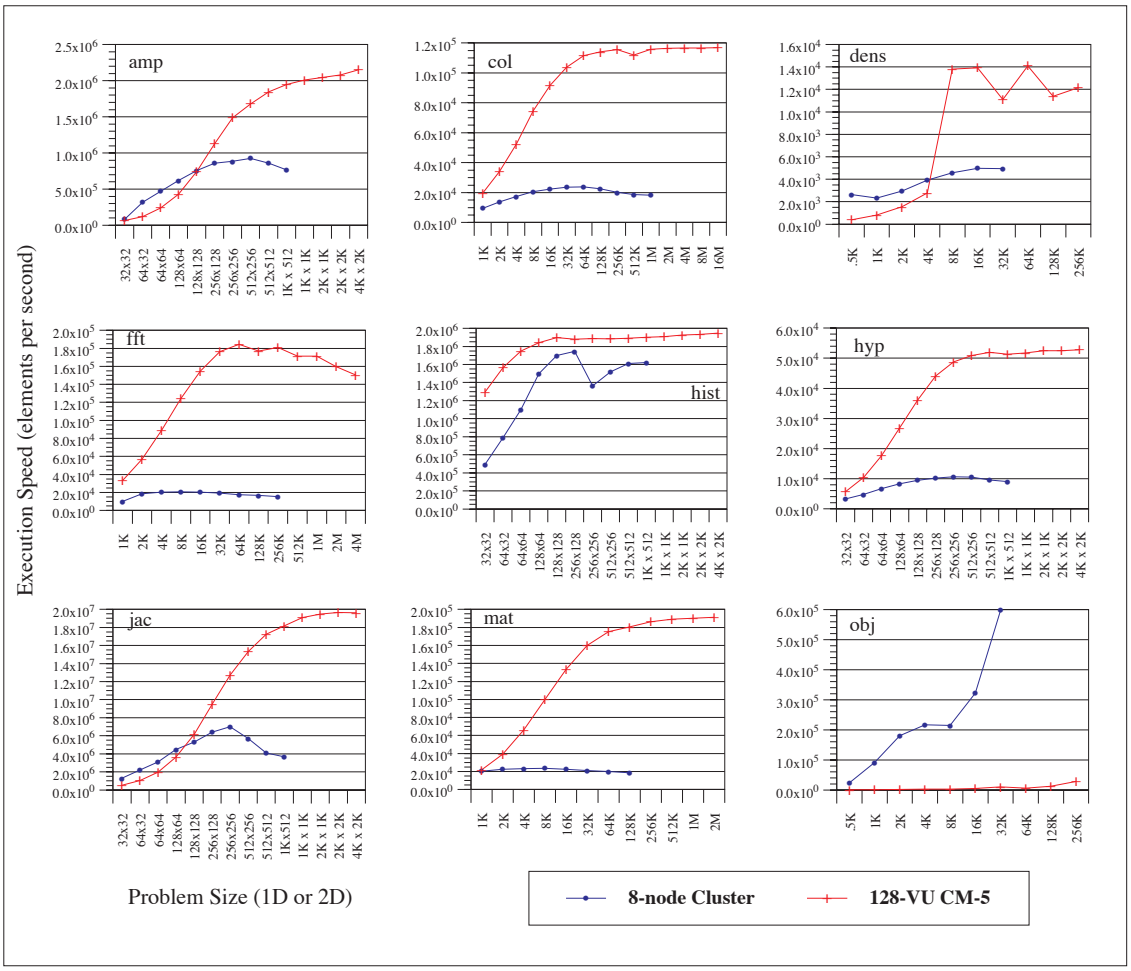

Figure 2: Absolute Execution Speed (elements/sec) graphs for all nine application programs, comparing an 8-node cluster of HP720s, to a 32-node, 128 vector unit CM-5.

5.2 Relative Execution Speed

Figure 3 illustrates the execution speed of the two platforms at a per-processor level for the nine application programs over a range of problem sizes. This compares two different absolute sizes, but the same number of virtual processors allocated either to one of the HP720s or to one of the CM-5 vector units. The main conclusion we draw is that at the per-node level, Cluster-C* is capable of delivering superior performance for all test programs, under all problem sizes, regardless of what amount and mixture of communication is used. Moreover, the rate of growth of the CM-5's execution speed is much slower than that of the cluster, indicating higher fixed overheads and poorer individual vector unit performance. For our problem set, a standard RISC processor with a good floating point unit performs better than a custom designed multi-vp vector unit.

We make three additional observations about Figure 3. First, the steep initial curves of the cluster indicate a well-behaved communication system where the overheads grow more slowly than the computational components of the tests. After reaching peak performance, a noticeable falloff in speed is apparent in several of the applications—**amp, col, hyp, jac**, and **mat**. This falloff occurs at the very top end of the problem sizes supported by an 8-node cluster, when all 32 Mbytes of physical memory are in use. At this operating point, these applications have significantly poorer cache behavior than do smaller applications. Looking at the changes in execution speed for **jac**, for example, it appears

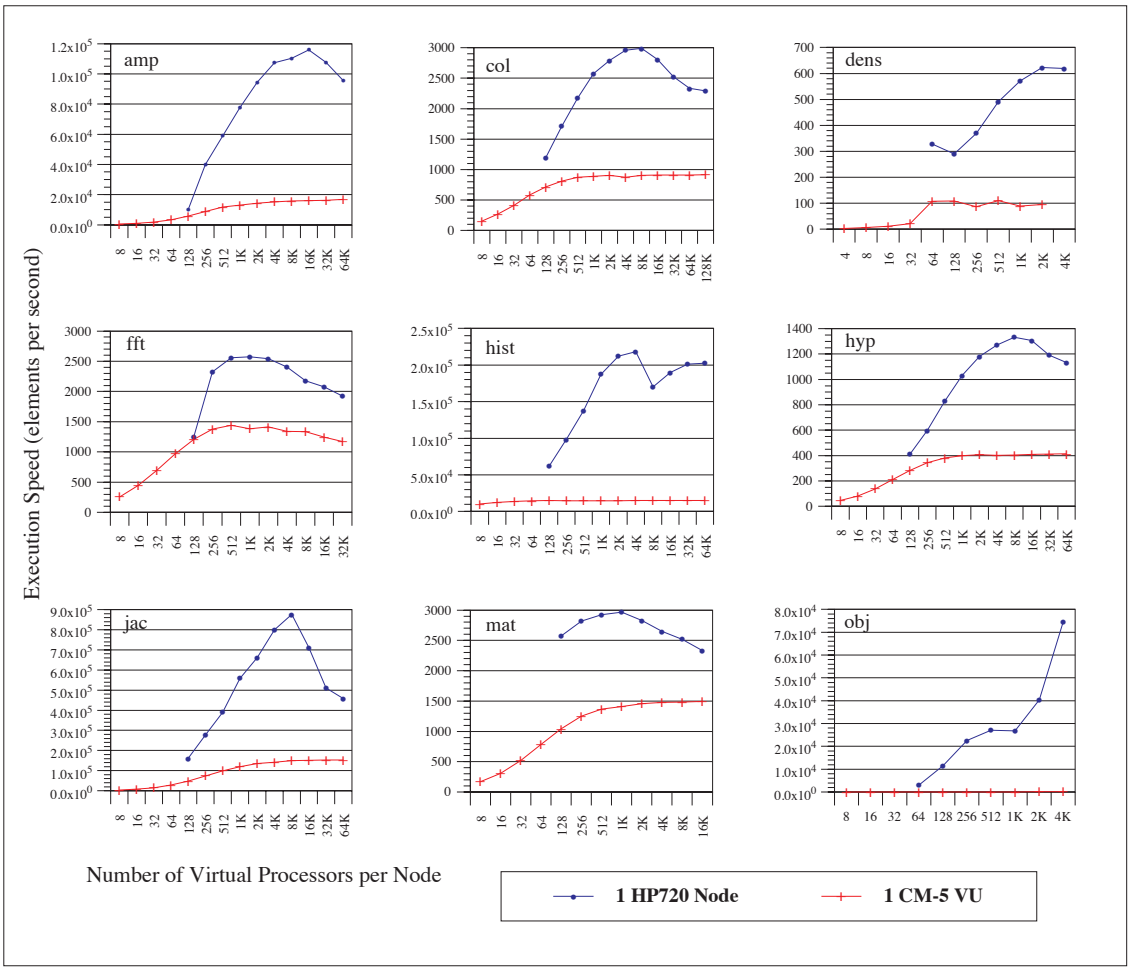

Figure 3: Relative Execution Speed (elements/sec) graphs for all nine application programs, comparing 1 node of an 8-node cluster of HP720s, to 1 vector unit of a 32-node, 128 VU CM-5.

that networking overhead is comparable to the overhead incurred by going to main memory for problem elements that would otherwise remain in the local cache.

Second, the cluster version of the **mat** test program has no communication. This is reflected in the almost level execution speed curve, with less of the typical initial increase. The CM-5 still has the characteristic increase of fixed overhead of communication and control. This is a reflection of the CM-5's use of a front-end processor and on-board SPARC2s.

Third, at a per-node level, the CM-5 has near-constant performance across all problems sizes for those applications that rely primarily on general sends—**hist** and **obj**. No benefits are observed by moving to larger problem sizes. This could indicate a limitation in the CM-5's general send capability.

In summary, the graphs for the Cluster-C* (Figure 3), demonstrate an optimal sub-problem size that would produce the best overall cluster performance. As problems increase size, there is a rapid increase of performance as computation costs outweigh either fixed or slower growing communication costs. At some point the problems become large enough that bad cache behavior begins to dominate, and the performance curves begin to fall. This is not caused by either new overhead or increasing communication costs.

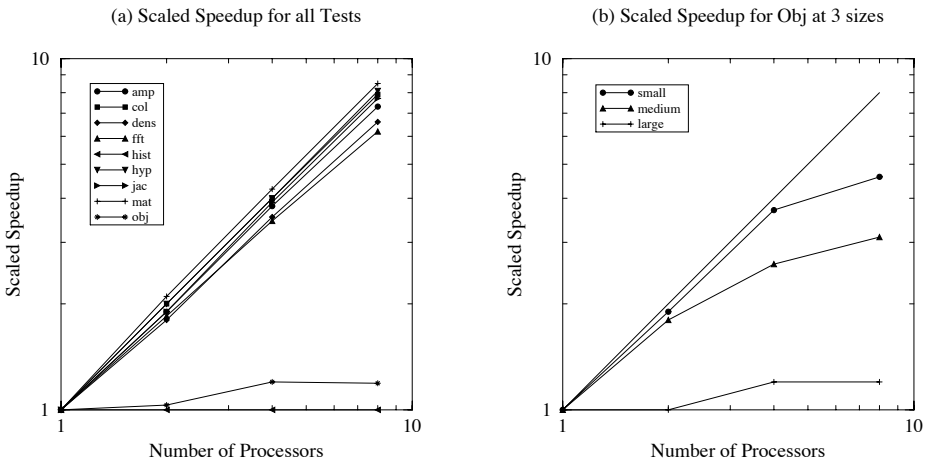

Figure 4: Speedup curves for Cluster-C* from one to 8 nodes. The left graph shows speedup for 9 test programs run with a large input size. The right graph shows speedup for one application, **obj**, for three problem sizes.

5.3 Cluster-C* Speedup

Figure 4a shows the scaled speedup of all nine test programs for a large problem size, running on 1, 2, 4, and 8 nodes. Scaled speedup is an appropriate measure for the algorithms and problem sizes characteristic of image understanding systems. In scaling all test inputs, we fixed the input for the single processor instance, and then increased the data in proportion to the number of nodes, so that the problem size per node remained constant across the full range of tests. For each program, the single node case was obtained from a one processor version of the C* program with all physical communication calls removed.

Computation-oriented programs can be expected to perform well under such conditions. The tests with little or no communication (**mat, hyp**, and **col**) have almost perfect speedup. Tests that use grid-sends or scans (**amp, dens**, and **jac**) acheive near-linear speedup. Communication-centric applications behave differently under processor scaling. While the local sub-problem size remains fixed, it represents a smaller fraction of the total problem and therefore a smaller portion of the global address space. For uniformly distributed destination addresses, the fraction of information sent off-processor by each node is $1 - 1/P$, for a P node cluster. Given that off-node sends average 12 times the cost of a memory-to-memory move, it is not surprising to see poor speedup for **hist** and **obj**, both of which make significant use of general sends. **Hist** actually has no speedup because the program consists of a single send operation. Figure 4b further illustrates the effects of heavy communication within the **obj** program, where speedup gets worse with increasing problem size.

Figure 4a shows, at least through eight nodes, that the cluster of workstations scales well for most problems. We can make the following three observations. First, applications that rely heavily on general sends will have difficulty scaling on a workstation cluster. Second, applications that use grid-based communication, either send or scan, or reductions, scale well. Third, all applications that perform a substantial amount of computation scale well.

Figure 5 reinforces the speedup results of Figure 4, showing the percentage of time each test program spent performing physical communication. Physical communication includes all time spent by the network substrate during send, receive, and interrupt handling, but not the time spent performing computation or local memory moves during the C* communication instructions. The graph includes error bars at one standard deviation around the mean across the

C* operator	1K	256K	1M	32x32	512x512	1Kx1K
reduce	763	70.1	17.6	17.5K	170.5	17.5
grid-send	340	26.1	6.5	18.3K	34.9K	17.5K
scan	1.7K	8.1	3.0	26.2K	3.3K	1.2K
send	1.5M	2.9M	2.9M	2.9M	2.9M	2.9M

Table 2: Communication Operator Execution Speed by Problem Size and Shape (bytes/sec)

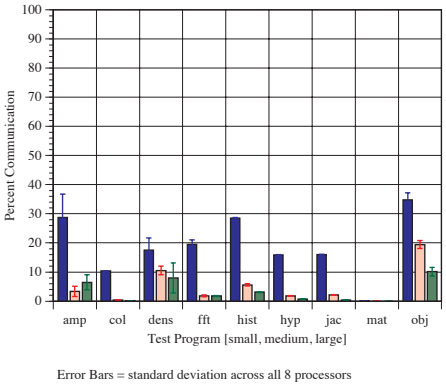

Figure 5: Percent physical communication for 8-node Cluster-C*

eight nodes in the cluster; variation in runtimes is possible under our loosely-coupled execution model. Test programs with good speedup have small amounts of physical communication (e.g. **col, hyp, mat**) for medium and large problems sizes. Grid-based applications like **amp, dens** and **jac** have slightly higher percentages of communication and also fall off slightly from linear in the speedup curves. The test with the highest percentage of communication—**obj**, at 32, 20 and 10%—had one of the worst speedups.

The average percent communication across the nine test programs was 19% for small problems, 5% for medium sized problems and 3.4% for large problems. Overall, communication costs grew sub-linearly relative to the linear growth in problem sizes. The fact that the amount of time spent on communication decreased with problem size leads us to be optimistic about running larger problems on larger cluster configurations.

Finally, we offer a micro benchmark that further supports our belief that many C* programs will scale well to larger cluster configurations before network saturation becomes a problem. Table 2 contains the maximum transmission speeds for a single HP720 across the C*-to-network interface for each C* communication operator. The maximum speed at which C* is capable of transmitting network data is 2.9 MB/sec for general send—approximately one quarter the bandwidth obtainable by the network substrate. All other C* operators perform more computation or local data movement than the general send with correspondingly lower effective transmission rates.

6 Related Work

There has been considerable research in recent years focusing on the implementation of parallel programs for distributed-memory multiprocessors. We broadly classify the most relevant of this work into two general categories.

First, the compiler community has focused attention on the problem of implementing data parallel programs on tightly-coupled, distributed-memory multiprocessors [**?**, 5, **?**, **?**, **?**]. Our work differs from this in several respects. For example, we are considering loosely-coupled workstation clusters, which means we have to look even harder at the assumptions underlying the latency-related implementation strategies. Also, we are working at the communication

subsystem level rather than at the compiler level. Finally, most of this work has focused on Fortran D rather than C*. This is significant because C* permits a much richer interconnection of parallel data than does Fortran D. This is best illustrated by the varied applications of interest to the two communities: Fortran users are primarily interested in grid-based, numerical computation, characterized by algorithms such as Jacobi and SOR, while C* is often the language of choice for image processing programs because of the need for general sends and scans.

Second, the systems community has recently started to consider the problem of implementing control parallel programs on loosely-coupled workstation clusters. Most of this work has centered around the idea of provided distributed shared memory [3, 2]. This work is similar to ours in that the focus is on the underlying consistency protocols required by the parallel program, the obvious difference being that we consider data parallel rather than control parallel programs. Another way to say this is that we attempting to define C*-specific, rather than general-purpose, protocols for distributed-memory architectures.

7 Concluding Remarks

This paper describes an architecture for implementing the C* programming language on a cluster of workstations, and shows for a suite of image understanding programs, how the cluster compares to a CM-5. The data shows that for certain small and medium problem sizes, an 8-node cluster out-performs a 128-VU CM-5 on an absolute scale, and that on a relative scale, the cluster delivers superior per-node performance in all cases. Further analysis of the communication demands of this test suite indicate that the cluster should be able to scale to much larger configurations. We are currently working on a model that should predict more accurately the extent to which Cluster-C* scales.

References

[1] D. Banks and M. Prudence. A high-performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications*, 11(2):191–202, February 1993.

[2] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A users' guide to PVM—Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA, 1991.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Symposium on Operating System Principles*, pages 152–164, 1991.

[4] J. L. Frankel. A reference description of the C* language. Technical Report TR-253, Thinking Machines Corporation, Cambridge, MA, USA, May 1991.

[5] P. J. Hatcher, M. J. Quinn, A. J. Lapadula, B. K. Seevers, R. J. Anderson, and R. R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, 1991.

[6] W. D. Hillis and L. W. Tucker. The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM*, 36(11), 1993.

[7] S. Hiranandani, K. Kennedy, and C. W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8), 1992.

[8] D. Mosberger, L. Peterson, and C. Turner. Exploiting highly reliable networks with careful protocols. Technical Report TR94-14, University of Arizona, Department of Computer Science, Tucson, AZ, March 1994.

[9] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.