

A C* Development Environment for the Meiko CS-2 Multicomputer

A research paper in partial fulfillment of the requirements
for the degree of Master of Science

Chun Zhang
Department of Computer Science
Oregon State University
Corvallis, OR 97331

zhangch@research.cs.orst.edu

Major Professor: Dr. Michael J. Quinn
Minor Professor: Dr. Vikram A. Saletore
Committee member: Dr. Prasad Tadepalli

July, 1995

Acknowledgments

I would like to express my deepest gratitude to my advisor, Dr. Michael J. Quinn, for his advice and insight. His knowledge was invaluable to the successful completion of this project.

I am most grateful to my committee members, Dr. Michael J. Quinn, Dr. Vikram A. Saletore and Dr. Prasad Tadepalli, for their support and insightful feedback of this document.

My colleague Jason A. Moore deserves special thanks for his seemingly endless patience and help. I am also indebted to my other colleagues, Jonathan King and Santhosh Kumaran, for discussions that shed light on the project. It is amazing how one's whole work can benefit from being surrounded by researchers of such calibre.

And finally, I am grateful to my parents and my brother, whose spiritual and financial support are so important to my study in the US.

Contents

1	Introduction	1
1.1	Data-Parallel Programming	2
1.2	The C* Language	3
1.3	Goals of this project	6
2	Architecture of the Meiko CS-2 Multicomputer	7
2.1	CS-2 Processors and Communications Network	7
2.2	The Elan Widget Library	9
3	The UNH C* Compiler and Run-Time Library	10
4	Porting and Optimizing the UNH C* Run-Time Library	11
4.1	Data Distribution	11
4.2	Broadcast	12
4.3	Reduce	14
4.4	Scatter and Gather	16
4.5	Grid-read and Grid-write	19
5	Profiling the Execution of C* Programs	26
5.1	Profiling Data Structure	26
5.2	Profiling Calls	28
5.3	Files Output from the Profiler	29
6	A Graphical User Interface for Performance Analysis	30
6.1	A Tour through <i>CSDE</i>	30
6.2	Major Implementation Issues	40
6.2.1	Performance Window	40
6.2.2	Calculation for bars	44

6.2.3	Compiling and Running a C* program	46
6.2.4	The Working Versions	48
6.2.5	Locating Error and Vector Messages	49
7	Comparing Performance of Run-time Libraries	52
7.1	Designing Benchmarks	52
7.2	Performance Comparison	53
8	Future Work	63
8.1	Improve Scatter and Gather	63
8.2	Unifying Bar Percentages of Copy Windows	63
8.3	Selecting Colors and Fonts	63
8.4	Integrate <i>CSDE</i> into an Interactive Debugging Tool	64
A	Benchmarks for Performance Comparisons	66

List of Figures

1	The data-parallel model.	3
2	C* examples for a 2x3 shape. (a) $z = 2*x + y$, (b) $[1][2]y += z$	5
3	The CS-2 multi-stage communication network. Although the switch component is an 8x8 crosspoint, the effective radix is 4 due to the bidirectional links.	8
4	16 node Meiko CS-2 network as a fat-tree.	8
5	A 8x5 parallel variable is distributed on 5 physical processors only according to its first dimension. The second dimension is allocated locally.	12
6	A broadcast of value 5 on a array of 6 processors.	14
7	A reduction on a linear array of 6 processors.	15
8	$[pv3]pv1 = pv2$. The use of scatter operation to send elements of $pv2$ to $pv1$, using $pv3$ as an index. For example, since the 1st element of parallel index $pv3$ is 3, the 1st element of $pv2$ is sent to the 3rd position of $pv1$. . .	17
9	$pv1 = [pv3]pv2$. The use of gather operation to get elements of $pv1$ from $pv2$, using $pv3$ as an index. For example, since the 1st element of parallel index $pv3$ is 3, the 1st position of $pv1$ gets its value from the 3rd position of $pv2$	17
10	Scattering packets on a linear array of 6 processors. The numbers inside the circles indicate the destination nodes of the packets. The destination nodes and offsets of the source values are calculated before the actual scattering occurs.	18
11	C* examples for (a) grid-read, and (b) grid-write.	20
12	An UNH grid-write example.	22
13	An OSU grid-write example.	24
14	A grid-read example.	25
15	A snapshot of <i>CSDE</i>	32

16	The FileSelectionDialog popped up from “Open”.	33
17	The dialog for choosing Compile options.	34
18	The dialog for choosing Run options.	34
19	The <i>compiling dialog</i> , popping up when Build-Compile is invoked.	35
20	The <i>running dialog</i> , popping up when Build-Compile is invoked.	35
21	The <i>compiled dialog</i> , popping up when compiling is finished.	35
22	The <i>run dialog</i> , popping up when a program has completed execution.	36
23	The error referred by message: “gaussian.cs:47: error...” is located by placing a small triangle next to line 47 in the source code.	37
24	Locating the line corresponding to the message: “393: Inner loop not vectorized...” generated by pgcc.	39
25	Broadcast	54
26	Reduction	55
27	Scatter	56
28	Gather	57
29	Grid-read	58
30	Grid-write	59
31	Bandwidth and latency for one-to-all broadcast on 16 processors.	60

Abstract

With sequential computing technology reaching its speed limits, parallel processing is emerging as the key to very-high-speed computation. However, developing a parallel program is by no means a simple task; neither is analyzing the performance of parallel programs.

C*¹ is a high-level data-parallel language that hides explicit message passing and provides an easy-to-understand virtual view of parallel computation. C* is a portable language for which a retargetable compiler has been implemented for mesh-connected MIMD multicomputers. Together with the compiler, a C* run-time library has been implemented using Intel NXlib. This project is aimed at developing a Cstar Development Environment—*CSDE*—for the Meiko CS-2 multicomputer. It includes three tasks. One is building a Meiko specific version of the C* run-time library, another is comparing the performance of the Meiko version with the performance of the original one, and the last is developing a graphical tool for C* programming and performance analysis. This paper introduces the instruments in *CSDE* and discusses in some detail the major implementation issues.

¹C* is a registered trademark of Thinking Machines Corporation.

1 Introduction

C* (*see-star*) is a data-parallel extension of C. Since 1987, C* has undergone many changes and has been implemented on a variety of machines such as the CM machines (CM-200, CM-5), Intel machines (iPSC/2, iPSC/860, Delta, Paragon, iWarp), nCUBE machines (nCUBE 3200, nCUBE 6400) and a cluster of UNIX workstations. The major goal of the University New Hampshire (UNH) C* project is to provide users with a high-level data-parallel language independent of specific machine architectures.

Parallel applications can be programmed using either low-level calls to parallel libraries (explicit message passing) or high-level languages. When programming at a lower level, users usually get better performance, because they can fully exploit the underlying hardware, and they can choose solutions and optimizations best suited for their specific application. However, the advantages of using a higher level language are significant, too. The most important benefits are portability, understandability and maintenance. Freed from dealing with the idiosyncrasies of the machine, programmers can produce code that is cleaner, easier to read, easier to understand, and hence easy to maintain. Because the code is machine independent, it can be ported to other systems with little change.

In the UNH C* project, the goal of portability is achieved mainly in two ways: the compiler produces C code instead of object code; and it relegates the machine specific functions such as inter-processor communication to the run-time library. When porting the C* language to different parallel systems, only the run-time library needs to be changed. A C* run-time library has been developed together with the UNH C* compiler. It is built using NXlib, a message-passing interface compatible with the Paragon system [8]. Since the NXlib is supported on the Meiko CS-2, the UNH C* run-time library can be used on the Meiko as well. However, this library assumes mesh-connected processors and employs a set of hypercube message passing algorithms, which do not always fit well with the architecture of

the Meiko CS-2. Moreover, it introduces some overhead by putting an extra layer of function calls between the C* run-time library and the lower level Meiko CS-2 message passing library.

This project is aimed at developing a Cstar Development Environment (*CSDE*) for the Meiko CS-2 multicomputer. It includes three tasks. One is building a Meiko CS-2 specific version of the run-time library, another is comparing the performance of the specific version with the performance of the original mesh-oriented one, and the last is developing a graphical tool for C* programming and performance analysis. For the sake of convenience, we call the Meiko-specific version of the C* run-time library the OSU version, and the library implemented by the University of New Hampshire the UNH version.

In this section, we briefly describe the data-parallel programming model. We then introduce some of the major features of C*. The section concludes with a more detailed description of our goals for this project.

1.1 Data-Parallel Programming

C* is a data-parallel language. *Data parallelism* is the use of multiple functional units to apply the same operation simultaneously to elements of a data set [7]. The data-parallel model is a SIMD machine with a front end and a back end. The front end is a uniprocessor, and the back end is logically comprised of a sufficiently large number of virtual processors (VPs) (Figure 1). The front end stores the program and the sequential variables. It executes the sequential portion of the program, issues parallel instructions to the back-end VPs and controls their execution. Theoretically, the back-end VPs work in a *lock-step* fashion, which means that no processor can proceed until all other processors finish the same operation. Since they execute identical operations simultaneously on different sets of data, parallelism and speed are achieved. In the SIMD machine, each back-end processor has a small amount of memory that can be used to store parallel variables. All the VP memories, combined with

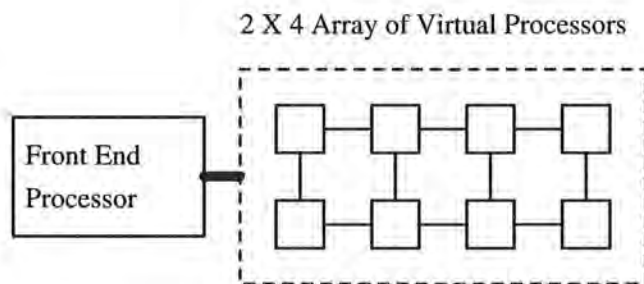


Figure 1: The data-parallel model.

the front-end memory, form a global name space whereby the VPs can communicate with each other.

As strict and simple as it is, the SIMD model can be applied to many real life problems—in general, numerically intensive operations, in particular discrete simulation methods such as the method of finite differences that “grid” up space. Moreover, the SIMD programming style can be implemented on a variety of non-SIMD machines. The UNH C* compiler implements C* in a loosely synchronous SPMD (Single Program Multiple Data) fashion using MIMD computers. Unlike the master-slave relationship of the front-end and back-end processors in the SIMD model, all processors in a MIMD machine are treated equally. Particularly in our project, every processor of the Meiko CS-2 stores a copy of the program, and parallel data are distributed among the processors. The inter processor communication is via explicit message-passing instead of through shared memory.

1.2 The C* Language

As a brief introduction to C*, this section only covers the rudimentary and relevant features, which are certainly far from embracing the full richness of this language. Interested readers are referred to [9] for a detailed description.

All data in C* are divided into two kinds, *scalar* and *parallel*. *Scalar* variables behave like variables in a sequential program. *Parallel* variables are the source of data parallelism. C*

programming allows the user to express the processing of large amounts of parallel data with the illusion that there is an unbounded number of processors onto which the data can be mapped. The template of parallel variables is called *shape*, which is a multidimensional array representing the mesh of VPs. Each element of a shape is called a *position*. To define a parallel variable or to instantiate a shape can be imagined as to fill the *positions* with *elements* of a certain type. Below is a concrete example illustrating this concept:

```
shape [128][256][2]shapeA;
int: shapeA x, y, z;
```

Here, a shape called shapeA is declared. It is a three dimensional mesh, with 128, 256 and 2 positions along the 0th, 1st, and 2nd dimensions respectively. Three parallel variables, *x*, *y*, and *z*, are declared of shapeA. Each element of the three parallel variables is an integer, and each element resides on a VP.

Parallel operations can be performed in parallel on parallel variables. C* inherits almost all data types and operators in C, while adding a new data type **bool** and new operators such as <? (minimum) and >? (maximum). The usual C operators are given new meanings when performing on parallel variables. For example, given the declarations above, we can write:

```
x = 3;
z = 2 * x + y;
[3][5][0]y += z;
```

The first statement sets each element of *x* to 3. The second statement first multiplies each element of *x* with 2, then adds them onto the elements of *y*, and finally assigns the results to *z*. Figure 2(a) shows the operation similar to the one in this statement on variables of a 2 by 3 shape. The third statement adds up all the elements of *z*, together with the element

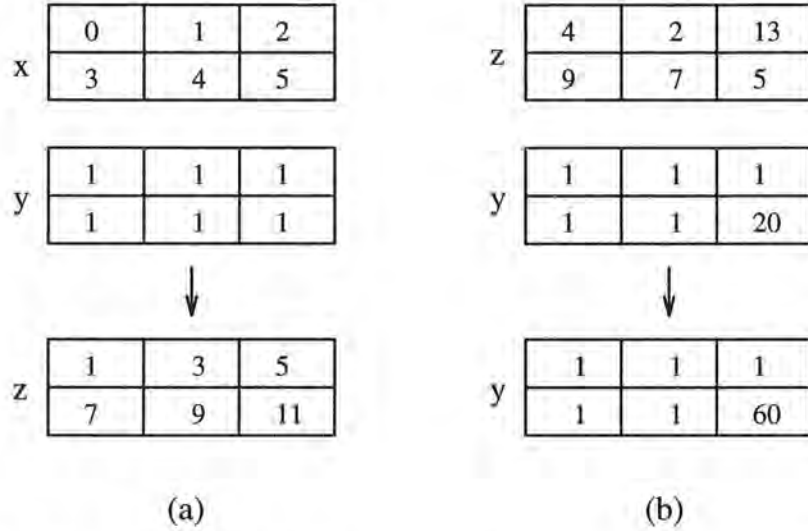


Figure 2: C* examples for a 2x3 shape. (a) $z = 2*x + y$, (b) $[1][2]y += z$.

of y residing on VP (3,5,0), and assigns the result back to that position of y . Figure 2(b) gives an example of similar operation on a 2 by 3 shape.

In C*, the programmer must specify the context of the parallel operations. This is done using **with** statements. For example:

```
with (shapeA) {
    /* operations on pvars of shape shapeA */
}
```

In many cases, we would like only a subset of the VPs to perform operations in parallel. In these situations we can use a *where* statement, which is analogous to the *if* statement in C. The condition of a *where* statement masks off the elements of parallel variables that are not to take part in the operations, while the rest of the elements execute the code segment inside the *where* block. Also like the C *if* statement, a *where* statement can have an **else** clause. The elements masked off execute the code segment in the **else** clause.

C* also has some built-in functions that programmers might find useful. One of them is `pcoord()`. This function takes on a parameter specifying the dimension number and returns

a parallel variable with each of elements being the position number along that dimension.

1.3 Goals of this project

This project started off as implementing a specific version of the C* run-time library for the Meiko CS-2 multicomputer in the Computer Science Department, Oregon State University. Our primary concern was to make the C* library as efficient as possible.

After the code for the routing library was written, we began to test and analyze the performance of the C* system. The process turned out to be tedious without a good method for retrieving large amounts of valid data, especially for problems with very large size and run on many processors. As a result, we decided to develop a profiler as part of the C* run-time library, and the profiler became our second goal in this project. The profiler is responsible for collecting timings during the execution of C* programs. It does minimum analysis of the data and "dumps" data in a reasonable manner.

The creation of the C* profiler also led to our third objective in this project: devising a graphical development environment which automatically handles the timing data and displays them visually. Furthermore, the tool integrates all the tasks of editing, programming, compiling and running a C* program.

Porting and optimizing the C* run-time library is discussed first in this paper, then the profiler and the graphical tool are introduced and some of their implementation issues are described in detail.

2 Architecture of the Meiko CS-2 Multicomputer

The Meiko CS-2 is a distributed memory MIMD computer. Each node can be considered a stand-alone workstation, because it is built from standard SPARC microprocessors (with optional vector processors for vectorizable applications), and runs standard Solaris 2.x kernels [1]. All the nodes are connected together by a high-speed interconnection network. To program the message passing among the nodes through the interconnection network, CS-2 provides some libraries. The lowest level library is the Elan Widget library (EWlib). In this section, we give brief introductions to the CS-2 processors, the communications network, and the EWlib.

2.1 CS-2 Processors and Communications Network

The Meiko CS-2 at Oregon State University currently has 17 processing nodes, each with its own computing processor and memory. The computing processors are built from SPARC microprocessors. All the processing nodes are fully connected by a multi-stage switch network, with the Elite Network Switches (ENS) being the crosspoints. Figure 3 gives the topology of the CS-2 network [5].

Notice that in Figure 3, all the higher stage switches have identical connections to the lower stage. This enables us to combine the four switches on the top and degenerate the multistage network to a “fat-tree” (figure 4).

Besides reducing the number of switches at higher stages, the fat-tree network exploits the principle of “reference of locality”. When a processor in a fat-tree network only references local data, the routing does not go through the higher stage of the network, whereas it does in a multistage switch network. Hence in a fat-tree network the bandwidth at higher stages is reduced. This property turns out to be extremely beneficial since parallel applications

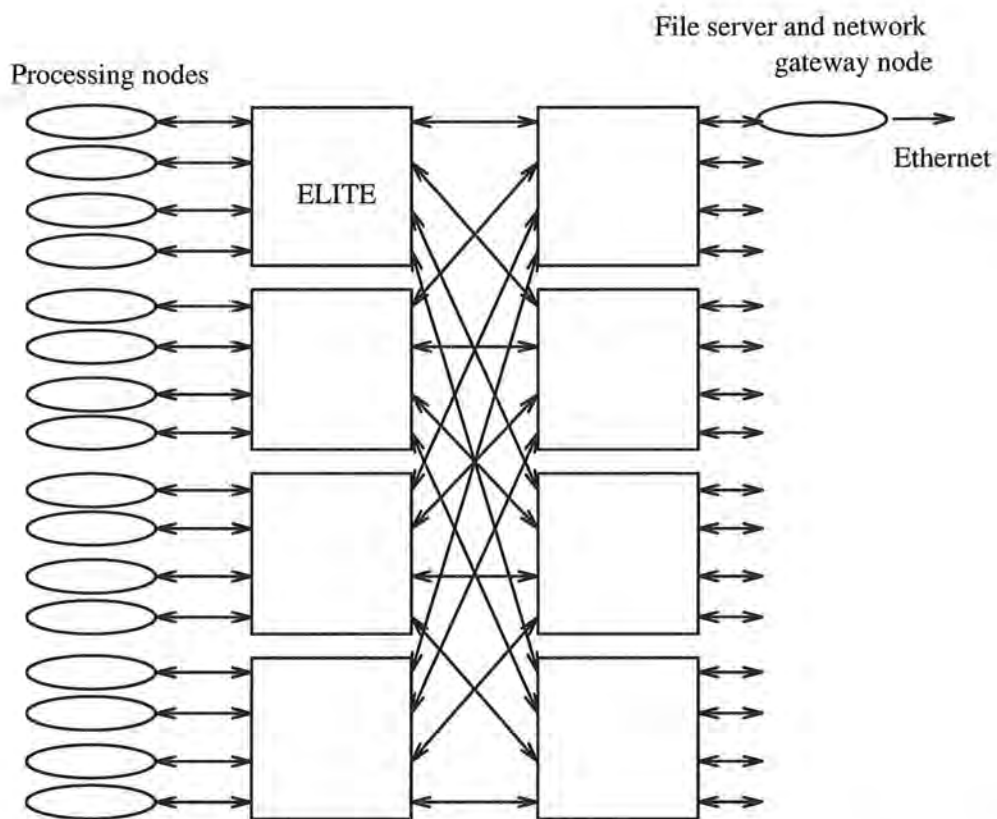


Figure 3: The CS-2 multi-stage communication network. Although the switch component is an 8x8 crosspoint, the effective radix is 4 due to the bidirectional links.

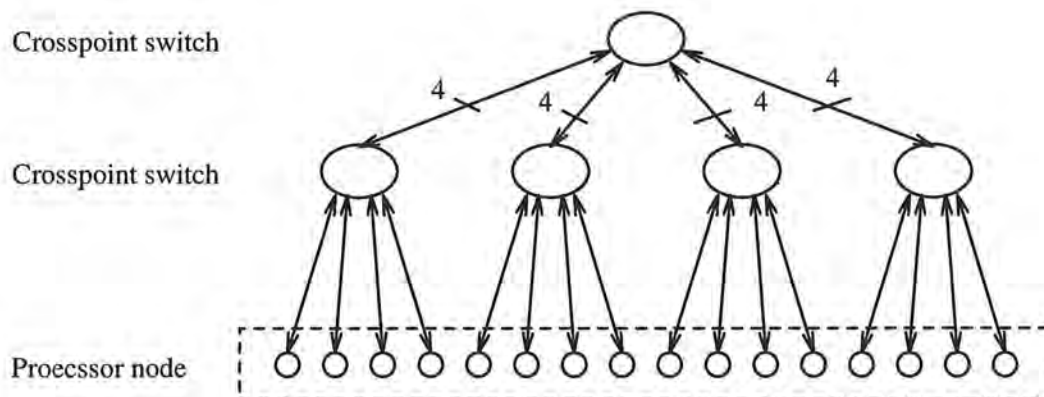


Figure 4: 16 node Meiko CS-2 network as a fat-tree.

usually show a high degree of local referencing.

Since the network is nontrivial, programming it seems to be even more complicated. However, it is not necessary for us to get into that much detail. The actual routing through the fat-tree is hidden by a set of higher level libraries provided by the CS-2. One of the lowest level library is the EWlib.

2.2 The Elan Widget Library

The Elan Widget Library (EWlib) provides a set of building blocks with which higher message passing libraries can be constructed. Functions are provided to support the following types of operations which are used in the implementation of the C* run-time library [1].

- *DMA operations.* These functions allow the user to read/write a continuous chunk of data directly from/to a remote processor, without requiring the cooperation of the remote process. They transfer data with the lowest latency.
- *Channel communications.* These functions provide untagged blocking and non-blocking point-to-point message passing. Unlike network DMAs, channel communications require the cooperation of both sending and receiving processes. They provide the simplest and lowest latency message passing with handshaking.
- *Collective communications.* These functions provide operations that perform collective communications on groups, such as broadcast, reduction, and global exchange.

3 The UNH C* Compiler and Run-Time Library

One major goal of the UNH C* compiler is portability. This is accomplished in two ways: its object code is C instead of machine or assembly language; and it relegates all the machine-dependent jobs such as communication among the processors to the routing library. The C* compiler translates C* code into C code plus calls to a communication library at points where data transfer among processors is required. The C* run-time library forms an efficient interface between the C* compiler and the underlying architecture, and it facilitates the portability of the compiler—only a new library need to be installed when porting C* to a different system. The C code output from the compiler can be made available if desired by the user, thereby providing those who are very familiar with C* a way to debug or to improve their programs.

The responsibilities of the C* run-time library can be grouped into two categories: memory related tasks and communication related tasks. The memory related tasks include the memory allocation for parallel variables, the distribution of parallel variables, and most importantly, the mapping from the VPs to the physical processors. The communication related tasks mainly consist of the six operations that we shall discuss in the next section: *broadcast*, *reduce*, *scatter*, *gather*, *grid-read* and *grid-write*.

4 Porting and Optimizing the UNH C* Run-Time Library

The construction of the Meiko C* run-time library (the OSU version) is based on the UNH version of the library. The algorithms in the UNH version can be found in [4]. The UNH version assumes processors are mesh-connected. In both the UNH version and the OSU version, the run-time support required by the C* compiler is the same. In this section, we describe in detail the optimization made by the OSU version.

4.1 Data Distribution

By data distribution, we mean the distribution of parallel variables across the physical processors. Because the parallel variables correspond to VPs, data distribution is actually a problem of mapping from virtual processors to physical processors. In section 2.1, we have described the topology of the Meiko CS-2 processors. They can also be imagined as forming a linear array, while at the same time being fully connected to each other. With this view, the mapping is indeed natural and simple, since a linear array is a one-dimensional mesh. In the OSU version, the parallel variables are laid out only according to their first dimensions, and the rest of the dimensions are just allocated locally. Figure 5 shows this scheme.

When developing a software, there are always some extreme cases to be considered. One of such cases is that the number of elements along the first dimension could be fewer than the number of physical processors the program executed on. The result would still be correct, but some efficiency is lost, because only part of the processors participate the computation. We did not try to solve this problem however, because this situation is very rare, and the effort of taking special care of it would not balance the speed we lose for the majority cases.

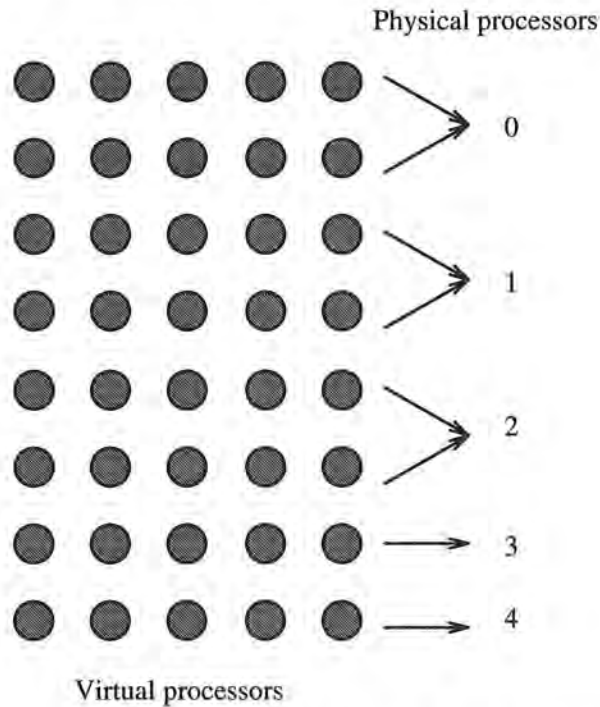


Figure 5: A 8x5 parallel variable is distributed on 5 physical processors only according to its first dimension. The second dimension is allocated locally.

4.2 Broadcast

In UNH C*, the scalar variables are duplicated on all processors. This strategy is adopted because the UNH C* is targeted at MIMD computers, so there is no real front-end processor to hold scalar variables for VPs to reference. Moreover, the duplication increases the locality of data. The problem incurred from this handling, however, is that all copies of scalar values need to be kept coherent among all processors. Therefore whenever one copy is changed on some processor, that processor needs to immediately broadcast the value to all others.

A *broadcast* distributes a value residing on one processor to all other processors. An example of C* code that causes a broadcast call to be emitted by the compiler is the following:

```
scalar_x = [5]pvar;
```

This statement assigns the 5th element of parallel variable `pvar` to a scalar variable `scalar_x`.

Here we assume that the shape of `pvar` is a one-dimensional mesh. The actions caused by this statement are as follows: the processor which owns the `[5]pvar` is found out, and then it broadcasts the value to a compiler-generated scalar temporary variable on every processor, at which point each processor does the local assignment to its copy of scalar `x`.

The Old Implementation

The broadcast algorithm in the UNH version executes on a linear array of processors in logarithmic time, assuming multi-hop messages take constant time. Broadcasts are performed on meshes of arbitrary dimension by applying this algorithm to each dimension in turn.

The algorithm assumes that each node has an id, with the source node's id being zero (This can always be achieved by rotating the source node's id to zero and adjusting the others accordingly. For instance, suppose there are n processors. If a node has id Id_{old} , and the id of the source node is Id_{source} , then after rotation, the node has id: $Id_{new} = (Id_{old} - Id_{source} + n) \% n$). In each step of the algorithm, a jump or span is calculated with which the processors are to be divided. The algorithm proceeds by letting the source node send to its partner located in the "middle" of the array, and then in the following steps, the span is divided in half, and each new subdivision then participates in the communication. At each step, twice as many nodes are involved in the communication, and the span length is halved. At the last step the span is equal to one, and any node that has not yet received a message finally receives the value. Figure 6 shows this broadcasting process on six nodes, the broadcasting value being 5.

The New Implementation

The broadcast algorithm in the OSU version is simplified to a great extent. The EWLlib provides a collective communication function called `ew_bcast()`, which does the one-to-all broadcast over a group of processors, and the data are transferred through DMA. This is called "hardware broadcasting". Not only is the hardware broadcasting much faster than

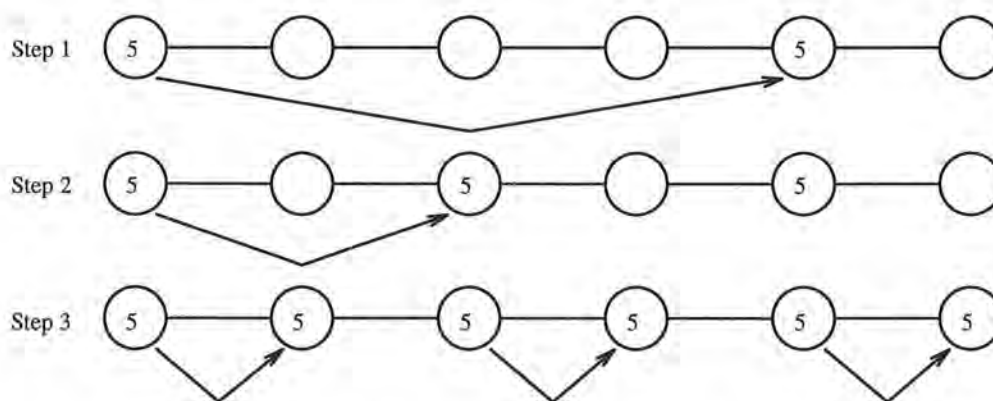


Figure 6: A broadcast of value 5 on a array of 6 processors.

the software broadcasting algorithm (see Figure 25), the resulting routine is reduced to only a few lines of code.

4.3 Reduce

Reduce is called when each active position of a parallel variable must contribute its value to some scalar expression. The code

```
scalar_x += pvar;
```

combines the sum of the active elements in `pvar` with `scalar_x`.

The Old Implementation

The algorithm begins by calculating the largest power-of-two that “fits” within the actual processor array size. This marks a dividing point between processors on the “full” half and those on the “partial” half of the linear array. On the first iteration, all processors on the partial half send their values to those on the full half. These full-half nodes then perform a logarithmic time reduction [4], resulting in the final reduced value residing on each of those nodes. For the final step, the full-half processors relay the final value to their partners on the partial half. When performing on meshes of arbitrary dimension, this algorithm is applied to

each dimension in turn. Figure 7 provides a visual representation of the algorithm executing on a linear array of six processors.

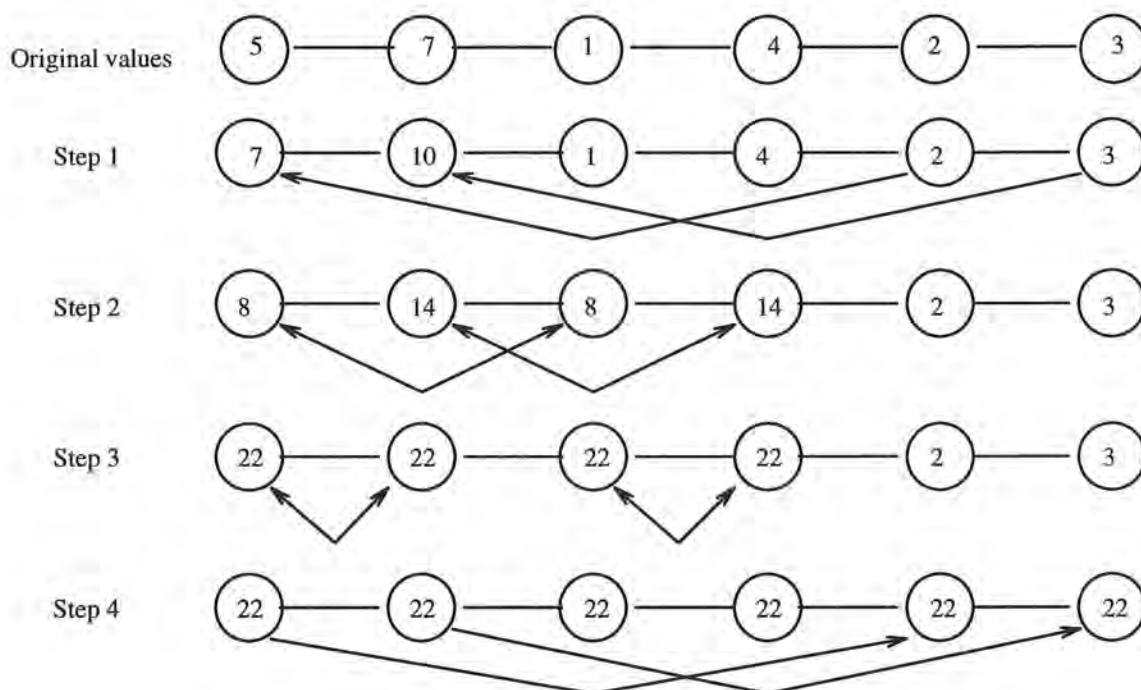


Figure 7: A reduction on a linear array of 6 processors.

The New Implementation

As with the broadcast, EWlib has a collective communication function called “`ew_reduce()`”. Once the data type and reduction operation are specified (we shall explain this a little bit later), the final result can be achieved with just one function call. We replaced the old reduction algorithm with `ew_reduce`. The software overhead introduced by the sophisticated algorithm is eliminated and the performance is dramatically improved as shown in Figure 26.

Although the “hardware reduction” can greatly reduce the amount of code and speed up the reduction process, there is one thing that took our special treatment. The working of `ew_reduce()` relies on the supply of a user-defined reduction function, which is an argument

of `ew_reduce()`. It needs to use the actual data type and reduction operation (add, multiply and maximum reduction, etc). The problem is that the run-time library cannot know this information until a program is actually compiled, yet the user-defined reduction function has to be supplied to `ew_reduce()` in programming time. As a solution, we set up a two dimensional table in which each entry is a pointer to a function, and each function represents a user-defined function of an ad hoc type and operation. Therefore, a function with operation *op* and type *type* can be obtained via `table[op][type]`. With this arrangement, it is left to `ew_reduce()` to find the correct user-defined function as its argument.

4.4 Scatter and Gather

Scatter and *Gather* are called when parallel values are to be distributed in an unpredictable pattern across the nodes. The following example demonstrates the C* code requiring calls to scatter and gather:

```
[pv3]pv1 = pv2;  /* Scatter */
pv1 = [pv3]pv2;  /* Gather  */
```

The first statement means that elements of parallel variable `pv2` are sent to `pv1`, using elements in `pv3` as indexes. The second statement means that each position of `pv1` gets an element from `pv2` using `pv3` as the index. Figures 8 and 9 give examples of these two operations.

The Algorithm

The algorithm for scatter is similar to the hypercube reduction algorithm, except in this case packets of data are sent instead of a single value. The processors are divided into two halves and the initial send is performed from the partial half to the full half. Once all packets are communicated to the full half, the scatter is performed in log step within the full-half.

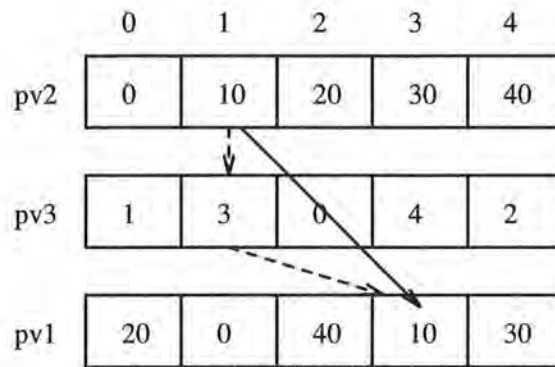


Figure 8: $[pv3]pv1 = pv2$. The use of scatter operation to send elements of $pv2$ to $pv1$, using $pv3$ as an index. For example, since the 1st element of parallel index $pv3$ is 3, the 1st element of $pv2$ is sent to the 3rd position of $pv1$.

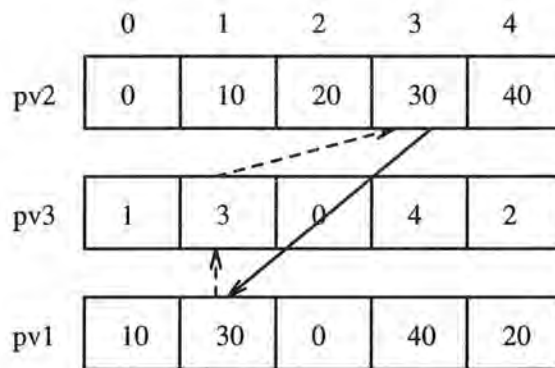


Figure 9: $pv1 = [pv3]pv2$. The use of gather operation to get elements of $pv1$ from $pv2$, using $pv3$ as an index. For example, since the 1st element of parallel index $pv3$ is 3, the 1st position of $pv1$ gets its value from the 3rd position of $pv2$.

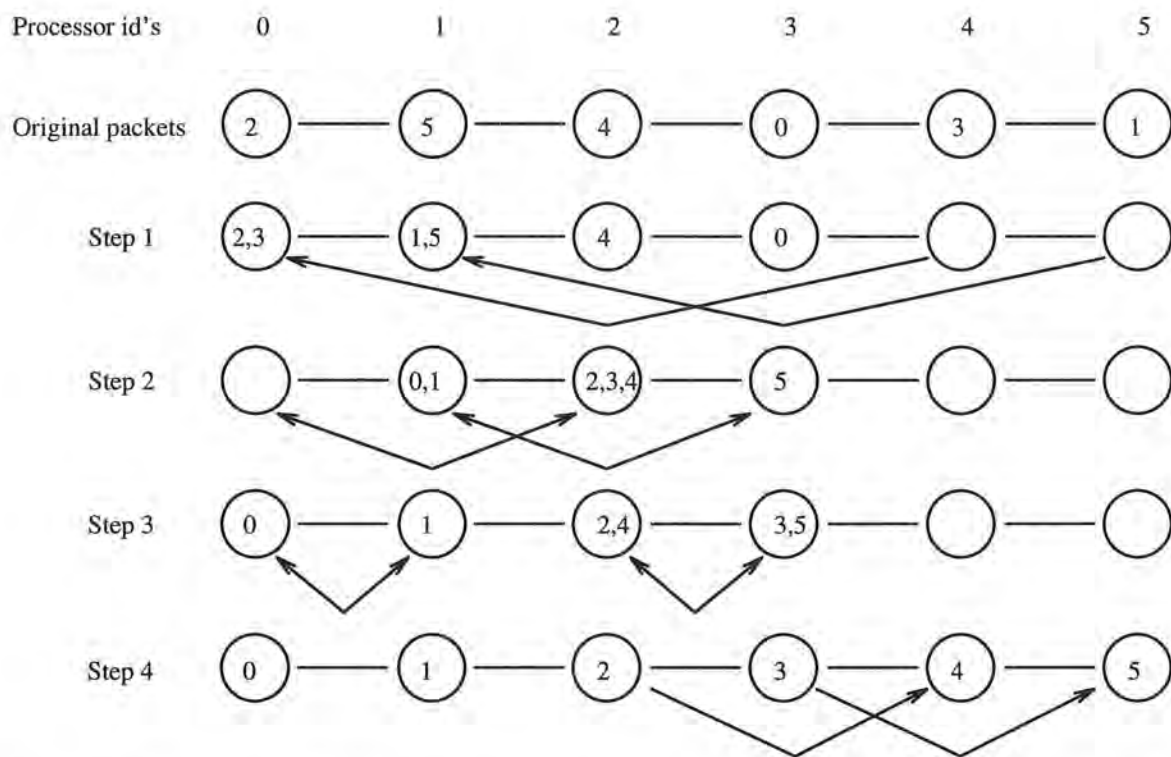


Figure 10: Scattering packets on a linear array of 6 processors. The numbers inside the circles indicate the destination nodes of the packets. The destination nodes and offsets of the source values are calculated before the actual scattering occurs.

Finally, any packets due to the partial half are communicated on the final step.

The gather algorithm requires two scatter operations. Initially, requests are scattered out to the nodes where the values reside, and then the actual values are scatter back.

We do not describe the algorithm in a great detail here, as the description would be tedious and give no insight to the problem. Figure 10 is a good presentation of the scattering process on a linear array of 6 processors.

The new implementation

The scatter and gather algorithms of the UNH version was kept and we tried to improve the effectiveness of the message passing. The EWlib's channel communication was adopted. A network of fully-connected channels are set up when a C* program starts running: processor p 's channel i is connected to processor i 's channel p . At the point where message passing is required, each processor uses the appropriate channels to transmit packets of data.

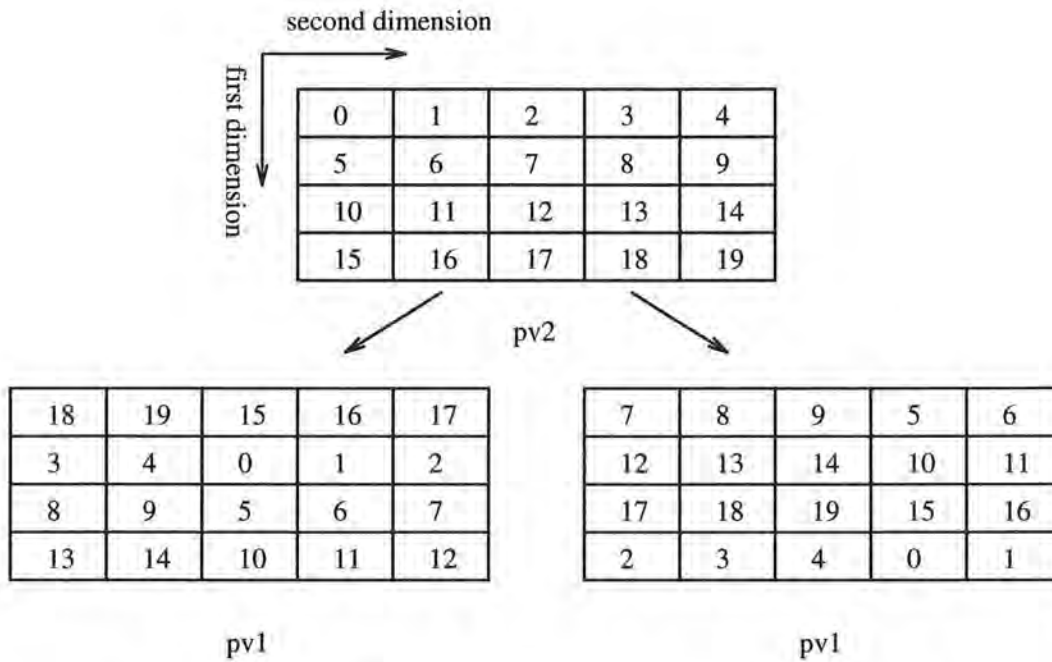
Unfortunately, the resulting performance is not what we had expected—the new implementation for scatter is a little slower than the old one (Figure 27) and the new gather is only marginally better (Figure 28).

4.5 Grid-read and Grid-write

Grid-read and *Grid-write* are generalized neighbor communication functions. The grid operations are invoked when expressions containing **pcoord** are used as a left index into a parallel expression. For example,

```
pv1 = [.+3][.-2]pv2;    /* Grid-read */  
[.+3][.-2]pv1 = pv2;    /* Grid-write */
```

The first statement shifts the elements of **pv2** down 3 positions along the first dimension and



(a) $pv1 = [+.3][.-2]pv2;$

(b) $[+.3][.-2]pv1 = pv2;$

Figure 11: C* examples for (a) grid-read, and (b) grid-write.

up 2 positions along the second dimension, then assigns the shifted elements to **pv1**. The second statement has the inverse meaning: it shifts all elements of **pv2** up 3 positions along the first dimension and down 2 positions along the second dimension, and then puts those shifted values into **pv1**. Figure 11 is an example.

The UNH *Grid-Write* Algorithm

The UNH *grid-write* algorithm is summarized below:

1. Calculate how many messages to receive/send, and where from/to.
The results are put into variables `n_to_read`, `n_to_write`, `srcs[]`,
and `dests[]`;
2. Scan through the values on this node...
if the value is active:
 put the local packets into `CS__read_buffer`,
 and put the outgoing packets into `CS__keep_buffer`;
3. For (`i = 0; i < n_to_write; i++`)
 scan through the keep buffer,
 separate the packets for `dests[i]` into `CS__write_buffer`;
 send the packets in `CS__write_buffer` to `dests[i]`;
4. Do operations on the data received.

Figure 12 is an example using this algorithm.

The OSU *Grid-Write* Algorithm

To optimize the *grid-write* operation, we introduced a `CS__whole_buffer` into the C* buffer pool. `CS__whole_buffer` is an array of buffers:

```
char *CS__whole_buffer[CS__MAXP];
```

Where `CS__MAXP` is defined to be the maximum number of processors on Meiko.

Using `CS__whole_buffer`, we changed the *grid-write* algorithm into the following one:

$$[.+7] \text{ pvd} = \text{pvs}$$

Physical processors		0	1	2
	pvd	7 8 9 10 11	12 13 0 1 2	3 4 5 6
	pvs	0 1 2 3 4	5 6 7 8 9	10 11 12 13
Step 1	Dest processors	1, 2	0, 2	0, 1
Step 2	CS__read_buffer	—	—	—
	CS__keep_buffer	0 1 2 3 4	5 6 7 8 9	10 11 12 13
Step 3	CS__write_buffer	0 1 2	5 6	10 11
	send	to P1	to P0	to P0
	CS__write_buffer	3 4	7 8 9	12 13
	send	to P2	to P2	to P1

Figure 12: An UNH grid-write example.

1. The same as before;
2. Scan through the values on this node,
 if the value is active:
 put the local packets into CS__read_buffer,
 and put the packets to nodex to CS__whole_buffer[nodex];
3. For (i = 0; i < n_to_write; i++)
 send CS__whole_buffer[dests[i]] to
 dests[i]'s CS__whole_buffer[CS__nodenum];
4. The same as before.

Moreover, the sendings in step 3 are implemented using DMA transmission.

Figure 13 shows the same operation using the optimized OSU algorithm.

The *Grid-Read* Algorithm

For *grid-read* operations on parallel variables with fewer than two ranks, both the UNH version and the OSU version adopt an optimized algorithm, in which the data buffering and copying costs are saved:

1. vpi = 0;
2. Calculate the VP that vpi will receive from, say vpj;
3. Figure out on which physical processor vpj is on, along with vpj's offset;
4. Calculate the number of elements vpi is receiving from vpj:
 n = min (ilocal-vpi, jlocal-joffset)
 where "ilocal" is the number of elements on the physical node that vpi
 resides on;
5. Get the elements;
6. vpi += n, go to step2. Loop ends when vpi > ilocal.

$$[+,7] \text{ pvd} = \text{pvs}$$

Physical processors		0	1	2
	pvd	7 8 9 10 11	12 13 0 1 2	3 4 5 6
	pvs	0 1 2 3 4	5 6 7 8 9	10 11 12 13
Step 1	Dest processors	1, 2	0, 2	0, 1
Step 2	CS__read_buffer	—	—	—
		CS__whole_buffer[1]: 0 1 2	CS__whole_buffer[0]: 5 6	CS__whole_buffer[0]: 10 11
		CS__whole_buffer[2]: 3 4	CS__whole_buffer[2]: 7 8 9	CS__whole_buffer[1]: 12 13
Step 3		Send CS__whole_buffer[1]	Send CS__whole_buffer[0]	Send CS__whole_buffer[0]
		Send CS__whole_buffer[2]	Send CS__whole_buffer[2]	Send CS__whole_buffer[1]

Figure 13: An OSU grid-write example.

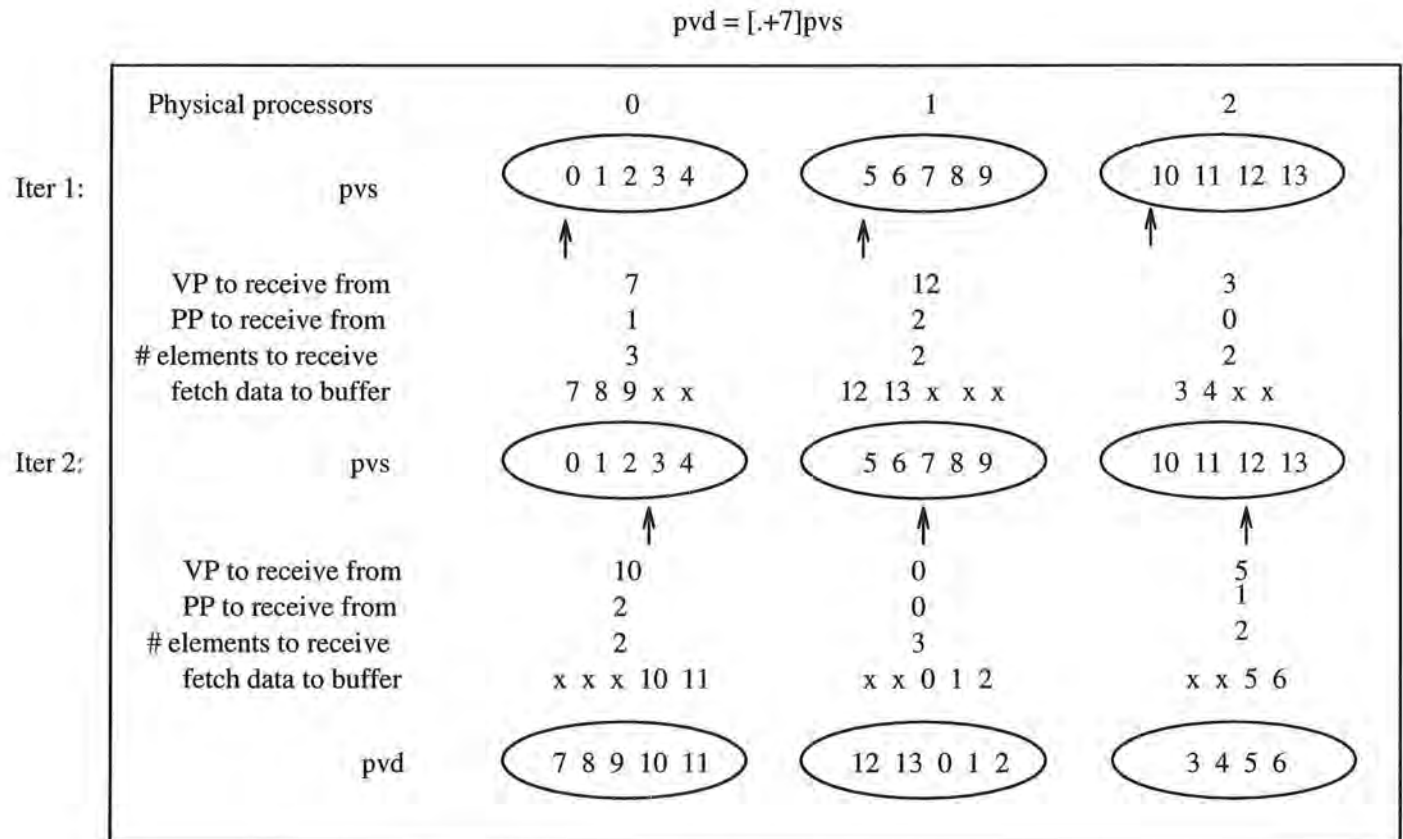


Figure 14: A grid-read example.

In Figure 14, we use the same operation as in *grid-write* to show how the *grid-read* algorithm works.

DMA transmission is used in the OSU version in replace of the NXlib transmission used in the UNH version.

The performance for *grid-read* and *grid-write* are given in Figures 29 and 30, respectively.

5 Profiling the Execution of C* Programs

When the OSU version of the run-time library was coded, we started to test the C* system. Very soon we realized that we needed a profiler to collect and analyze performance data. For instance, we would like to know how the timings are distributed among the different parts of a program, where most of the execution time is spent on, what is the cause of the inefficiency of a program, how efficient is every operation, and how each processor performs differently towards different operations, etc. Without a profiler, most of these questions are impossible or at least very difficult to answer.

The C* profiler is supported by the run-time library. It not only provides the compiler with profiling capabilities, but also can perform profiling on itself. While the compiler emits calls to begin and end profiling at a high level, the library is equipped to provide detailed information as to where the time is being spent inside the library.

5.1 Profiling Data Structure

The information for a profiling point is collected in a data structure called `CS_profile`. This structure includes such information as the profiling type, the profiling name, the name of the source file, the line number in the source code that this profiling occurs, and the timing data, etc. The structures for all the profiling points in a C* program form a linked list. The fields in `CS_profile` are shown below:

```

typedef struct CS__p {
    unsigned long CS__pro_count; /* number of times "activated." */
    int          CS__pro_type;   /* the structure's "type" */
    char         *CS__pro_name;  /* the structure's "name" */
    char         *CS__pro_file;  /* C* file assoc. with the struct */
    int          CS__pro_line;   /* C* line number assoc. w/struct */
    CS__ProType  CS__pro_acc;    /* timing "accumulator" */
    CS__ProType  CS__pro_local;   /* time spent sorting / doing local work */
    CS__ProType  CS__pro_send;    /* time spent in MsgWrite */
    CS__ProType  CS__pro_rcv;     /* time spent in MsgRead */
    CS__ProType  CS__pro_wait;    /* time spent waiting for msg to arrive */
    CS__ProType  CS__pro_extend; /* time spent reallocating buffers */
    int          CS__pro_max_readbuf_len; /* final sizes of the buffers */
    int          CS__pro_max_writebuf_len; /* Note: if the size is 0, */
    int          CS__pro_max_keepbuf_len; /* it's actually the original */
}              CS__profile;

```

For self-timing, the C* run-time library is primarily concerned with four fields in the profiling structure: `CS__pro_local`, `CS__pro_send`, `CS__pro_rcv`, and `CS__pro_wait`. `CS__pro_local` is used to accumulate the time it takes for each communication to perform all associated local work. `CS__pro_send` indicates the time required to perform the send operation and its counterpart `CS__pro_rcv` provides the time required to perform the receive operation. The last one, `CS__pro_wait` contains the time spent waiting for messages to arrive while no other work is going on.

5.2 Profiling Calls

C* profiling is provided as an option to the users. If you want to collect some data, you only need to add the compile-time flag “-profile”. With the profile switch turned on, the compiler inserts the function call `CS_ProfileStart ()` before a profiling point and the function call `CS_ProfileStop ()` after the point. The code below shows a simple example of the C code emitted by the compiler to profile a unary-reduce:

```
static CS__profile CS__temp_8 = { 0, 'C', "Unary-Reduce", "pi.cs", 24
/* Rest are zero'd by type */ };

/* Begin profiling */
CS__ProfileStart (&CS__temp_8);

/* Communication code */

/* End profiling */
CS__ProfileStop (&CS__temp_8);
```

In this example, a temporary variable called `CS__temp_8` is created by the compiler to hold the profiling data. Its fields `CS__pro_count`, `CS__pro_type`, `CS__pro_name`, `CS__pro_file` and `CS__pro_line` are initialized by the compiler, while the rest of the fields are to be filled in by the run-time library. If the program runs to a normal completion, then all the profiling information accumulated in the linked list are dumped out in a reasonable fashion. This is done by the function call `CS_ProfileDump` that is also emitted by the compiler.

5.3 Files Output from the Profiler

If a C* program is compiled with `-profile` option, running the program will produce two profiling files: a `.perf` file and a `.pix` file. Both files contain the information dumped from the profiler, but in different formats. The `.perf` file has all the raw data that the profiler can get. It is actually the print-out of the profiling linked list in a nice way. No analysis of any sort is performed on the data. This file is intended to be used by the C* programmer. The `.pix` file, on the other hand, is more terse than the `.perf` file, yet the information inside is the result of some simple processing. For instance, the `.pix` file records the average time spent on a certain profiling point, whereas the `.perf` file has all the timings such as `CS_pro_local` and `CS_pro_send` for all the physical processors that this program was executed on. The `.pix` file is intended to be used by the performance analysis tool, which is discussed in the next section.

6 A Graphical User Interface for Performance Analysis

The .perf file produced from the profiler can be used to help analyze the performance of a C* program, but it has some drawbacks. Usually a .perf file is at least twice as long as the corresponding .cs file, and its size increases superlinearly relative to the growth of the .cs file. This makes searching, comparing, and processing profiling data more difficult for longer programs. Consider a situation in which you want to find out the time spent inside a C* program's communication routines. If you were using X windows, you would open up two windows, one displaying the .cs file and one displaying the .perf file. You would scan through the .perf file, stop at each communication point, get its line number, go to that line in the other window and see what the code is, at the same time perhaps recording the timing data somewhere else. When the two files are long, and there are many communications, this process could be very tedious. We have developed a graphical tool that simplifies this process. The tool is called *CSDE*.

CSDE not only can provide performance data in graphical form, but also is capable of managing the other principle program development tasks: compiling, running and editing. This section shows what the tool can do and how it does them.

6.1 A Tour through *CSDE*

A snapshot of *CSDE*

CSDE has five primary components: the menu bar, the performance window, the edit window, the build window (or compile-run window), and the message label. Figure 15 is a snapshot of *CSDE* with the five components layed out in order from top to bottom, and left to right. The label at the bottom indicates we are viewing a program named gaussian.cs in the edit window. In the performance window, some bars with different colors and sizes

are lined up with the code of `gaussian.cs`. One of the bars is being pressed down, causing the pop-up of a dialog box, which shows the profiling type, name, and cost that this bar represents. The message in the build window, generated while running the program, shows the total execution time of the C* program.

What follows is a more detailed description of each field in the root window and its functionality.

Invoking *CSDE*

CSDE can be invoked in two ways. By just typing `csde` on the command line, the tool is started with empty windows. The second way is to add a C* program name as a command line argument. By typing `csde <filename>`, the tool is invoked with the specified file initialized in the edit window. Users can also specify some X resources on the command line via flags, for example, `csde -fg black -bg wheat`.

The menu bar

Once you are in the tool, you can use the File menu to open a file. There are three menus on the menu bar: “File”, “Build” and “Options”, each being a pull-down menu.

The File menu contains five items: “Open”, “New”, “Copy”, “Save”, and “Quit”. Clicking on “Open”, a `FileSelectionDialog` will pop up (Figure 16), which allows you to select a file for displaying in the edit window. “Save” is similar to “Open” in that a `FileSelectionDialog` will also pop up when invoked, but a file is chosen for saving instead of displaying. “New” clears all windows, allowing you to freshly start a new program; and “Quit” causes *CSDE* to terminate. Among all these menu items, “Copy” is the most interesting one. It is useful when the user wants to create multiple versions of the same program and compare their performance. The behavior of “Copy” is creating another root window of *CSDE*, we call it *Copy Window*, with the C* program being copied over to the Copy window. Then the user

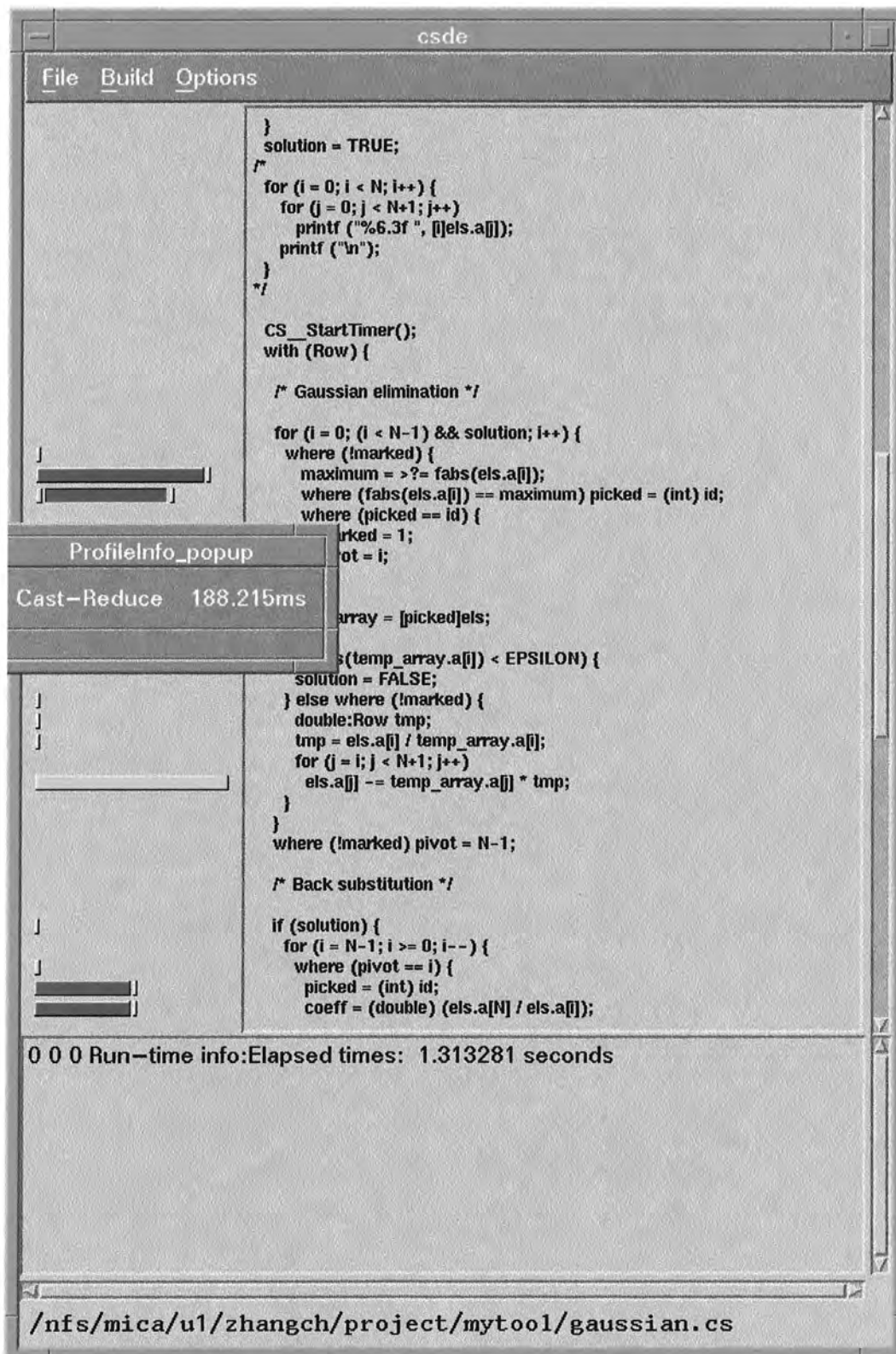


Figure 15: A snapshot of CSDE.

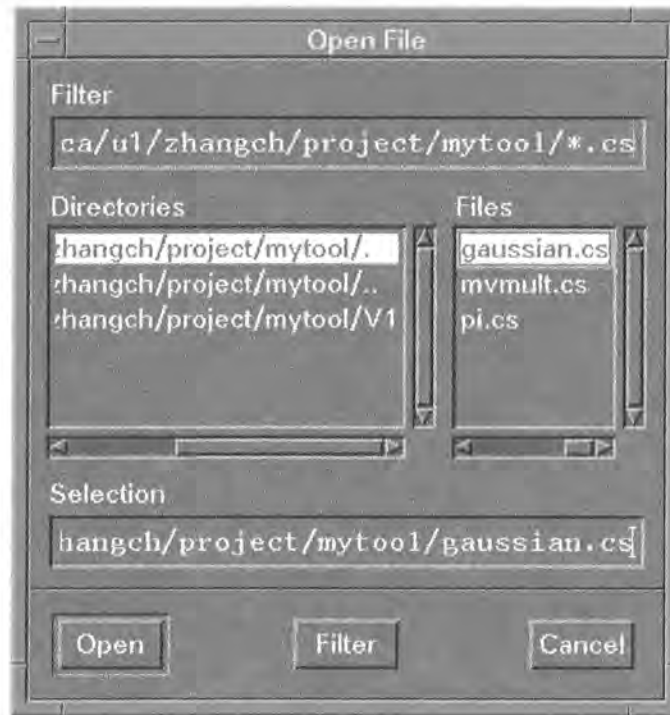


Figure 16: The FileSelectionDialog popped up from “Open”.

can modify the copied code and use the Copy window to work on the new version of the program. All the Copy windows and the original root window existing during the life time of *CSDE* are equal and independent. A new Copy window can be created from any of the old ones or the original root window.

The Build menu is for “building”—compiling and running a program. Two items are included under this menu: “Compile” and “Run”.

The Options menu is an auxiliary for Build. It also contains the two items “Compile” and “Run”, but they have totally different behaviors. When “Compile” is invoked, the dialog shown in Figure 17 pops up, and when “Run” is invoked, the dialog shown in Figure 18 pops up. These two dialogs allow you to set up the options for compiling and running a program.

Building a C* program

If a program has already been loaded in the edit window, choosing Build-“Compile” causes a

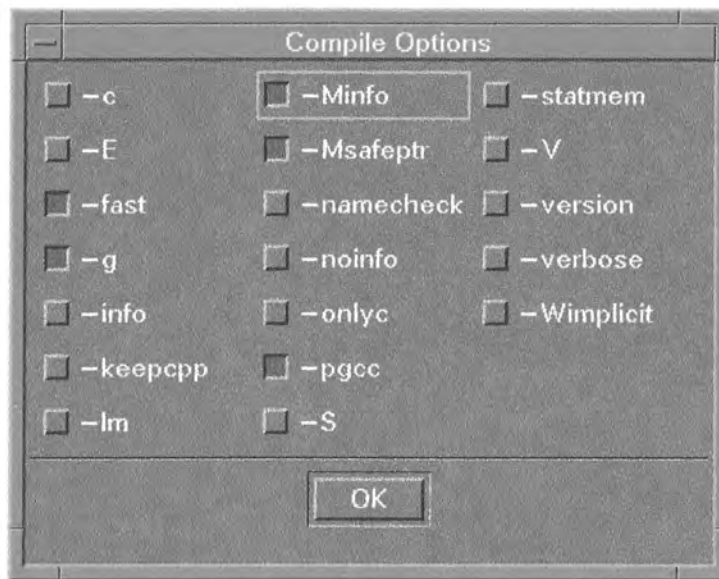


Figure 17: The dialog for choosing Compile options.

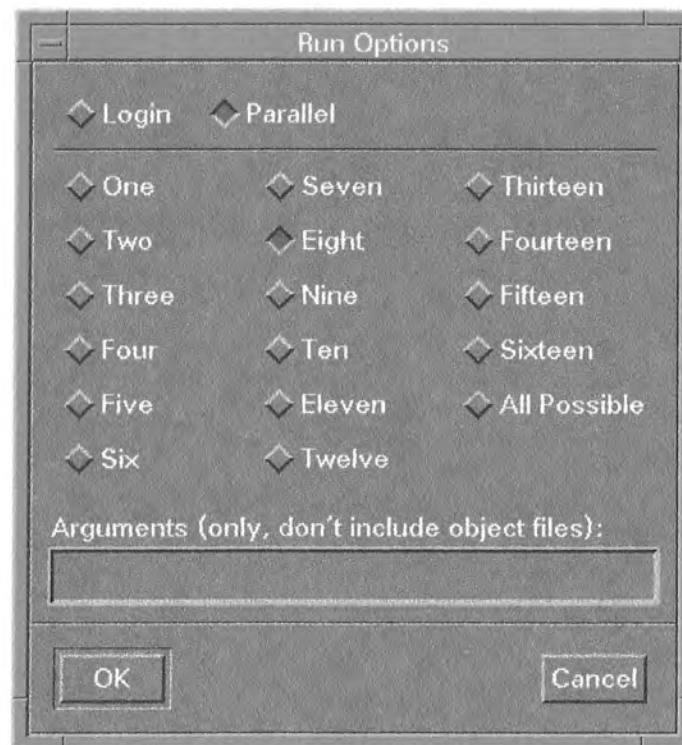


Figure 18: The dialog for choosing Run options.



Figure 19: The *compiling dialog*, popping up when Build-Compile is invoked.



Figure 20: The *running dialog*, popping up when Build-Compile is invoked.

compiling dialog to pop up (Figure 19). If the program has already been compiled, choosing Build-“Run” causes a *running dialog* to pop up (Figure 20). The termination of compiling or running is indicated when the two dialogs change to *compiled dialog* and *run dialog*, respectively (Figure 21 and Figure 22).

CSDE is designed so that when a program is being built, the user cannot interact with other parts of the tool except the dialogs. Clicking outside the dialogs causes a warning beep to be emitted.

Both *compiling dialog* and *running dialog* contain a *Cancel* button, which enables the user to kill the building process if anything appears wrong. This is extremely useful when the



Figure 21: The *compiled dialog*, popping up when compiling is finished.



Figure 22: The *run dialog*, popping up when a program has completed execution.

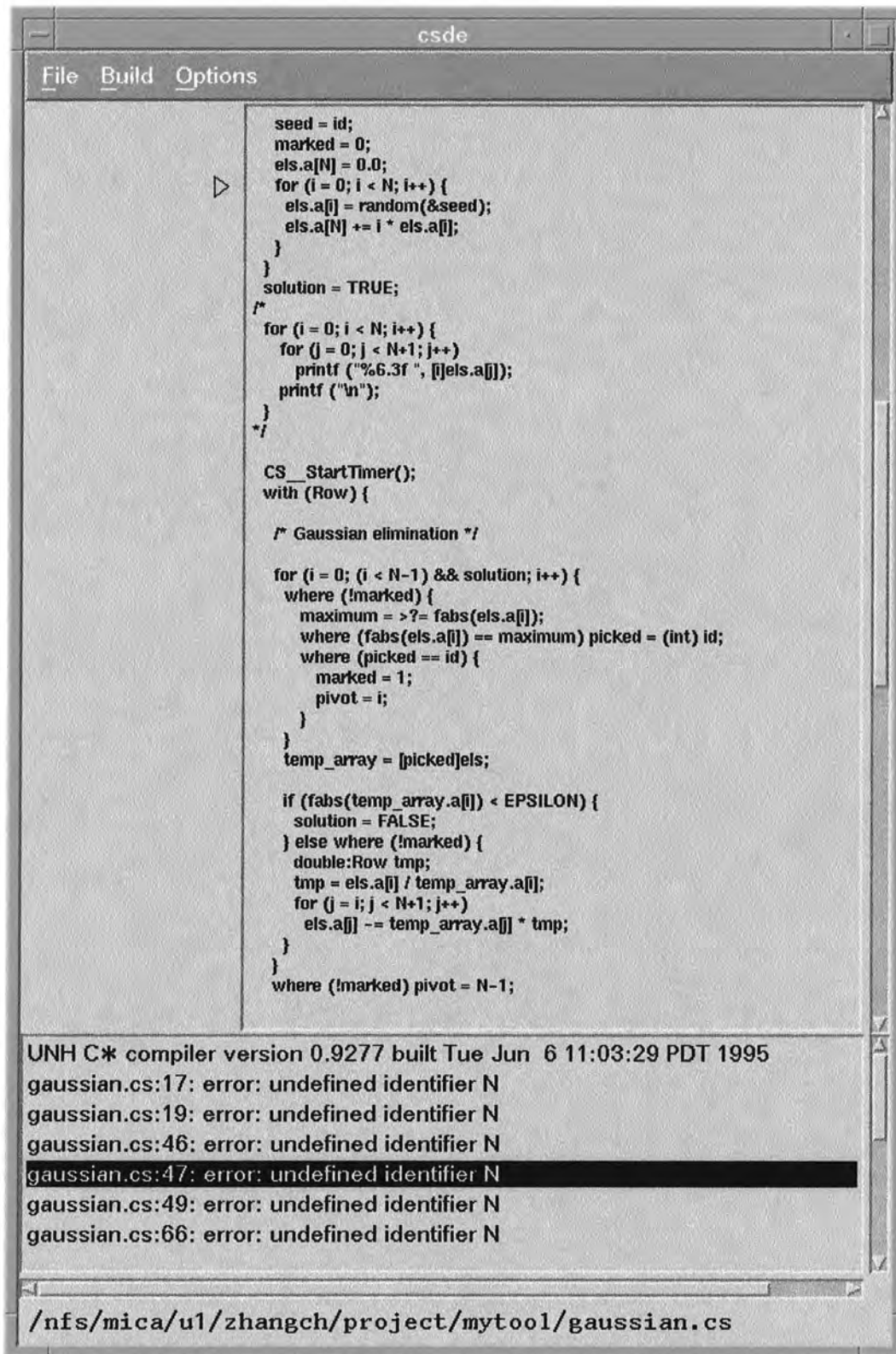
running job is queued on Meiko. In such a case, *CSDE* is also suspended, and the user would be kept waiting. Getting more and more impatient, the user would try to click on something and see what happened (this is why we disallow the interaction of other parts while building to prevent any abnormal situations or confusion). The *Cancel* button provides the user a way to take the job out of the queue.

After the successful termination of the running process, the performance bars are automatically displayed in the performance window.

The build window

When the compiling or running is over, the compile-time and run-time messages will go to the build window, including the warnings and errors. The user can click on the error messages in the build window, and *CSDE* has the ability to locate the error in the source code. Figure 23 gives an example. In the program `gaussian.cs`, we delete the line “`#define N 256`”, resulting in a list of errors produced in the build window after compilation. We then click on the error message “`gaussian.cs:47: error...`” in the build window, and a small triangle appears next to line 47 of the source code.

A more important use of this locating ability is to identify the code to which a message from the vector compiler `pgcc` refers. When running a C* program using `pgcc`, the line numbers in the compiler messages correspond to the C program emitted by the C* compiler, rather than the C* source code. For example, in Figure 24, the highlighted message in the build



The screenshot shows a window titled 'csde' with a menu bar containing 'File', 'Build', and 'Options'. The main area displays the source code for 'gaussian.cs'. A small triangle points to line 47, which is highlighted in black. The code is as follows:

```
seed = id;
marked = 0;
els.a[N] = 0.0;
for (i = 0; i < N; i++) {
    els.a[i] = random(&seed);
    els.a[N] += i * els.a[i];
}
}
solution = TRUE;
/*
for (i = 0; i < N; i++) {
    for (j = 0; j < N+1; j++)
        printf ("%6.3f ", [j]els.a[j]);
    printf ("\n");
}
*/

CS_StartTimer();
with (Row) {

    /* Gaussian elimination */

    for (i = 0; (i < N-1) && solution; i++) {
        where (!marked) {
            maximum = >?= fabs(els.a[i]);
            where (fabs(els.a[i]) == maximum) picked = (int) id;
            where (picked == id) {
                marked = 1;
                pivot = i;
            }
        }
        temp_array = [picked]els;

        if (fabs(temp_array.a[i]) < EPSILON) {
            solution = FALSE;
        } else where (!marked) {
            double:Row tmp;
            tmp = els.a[i] / temp_array.a[i];
            for (j = i; j < N+1; j++)
                els.a[j] -= temp_array.a[j] * tmp;
        }
    }
    where (!marked) pivot = N-1;
}
```

Below the code, the compiler output shows several error messages:

```
UNH C* compiler version 0.9277 built Tue Jun 6 11:03:29 PDT 1995
gaussian.cs:17: error: undefined identifier N
gaussian.cs:19: error: undefined identifier N
gaussian.cs:46: error: undefined identifier N
gaussian.cs:47: error: undefined identifier N
gaussian.cs:49: error: undefined identifier N
gaussian.cs:66: error: undefined identifier N
```

The file path at the bottom is: /nfs/mica/u1/zhangch/project/mytool/gaussian.cs

Figure 23: The error referred by message: “gaussian.cs:47: error...” is located by placing a small triangle next to line 47 in the source code.

window is for line 393, yet the file `gaussian.cs` is no more than 200 lines long. The picture shows that by clicking on the message, a small triangle takes us to the corresponding source code.

The edit modes

The program in the edit window can be in one of two modes: editable or non-editable. Only in editable mode can the program be modified. The program in the edit window begins in the edit mode, but once the program has been run and the performance bars have been displayed, the program in the edit window is no longer editable. At this point, the only way to change the code is to create another Copy window. The bars will not be copied, and hence the program in the new Copy window will be editable.

Performance bars

Performance analysis is the major feature of *CSDE*. The performance data are represented in the form of bars of different colors and sizes. A bar can have any of three different colors, representing three of the profiling types. Red represents type 'C' (communication, which has several subtypes), yellow represents type 'V' (vp-loop), and white represents type 'A' (pvar-allocation). Other profiling types such as 'F' (function) are not manipulated by this tool. The size of a bar is proportional to the cost of the profiling point. The longer the bar is, the higher the cost. Bars are drawn next to the code where the profiling occurs, so if a line has several profiling points, there would be several bars drawn next to it. The performance window is tied to the edit window via the scroll bar, thus both windows scroll simultaneously.

A performance bar contains more information than its color and size. Other information can be unveiled by clicking on the bars. This is already shown in Figure 15, in which a bar is being pressed and a small info-dialog has popped up. The dialog stays as long as the bar is being pressed, and disappears once the bar is released.

The screenshot shows a window titled "csde" with a menu bar containing "File", "Build", and "Options". The main area displays C code for a Gaussian elimination program. A vertical line on the left side of the code area indicates the current line being viewed, which is line 393. The code includes initialization, a first loop for random number generation, and a main loop for Gaussian elimination. The output pane at the bottom shows the compiler's feedback, including the message "393, Inner loop not vectorized - unsupported form of assignment stmt." and other optimization statistics. The file path at the bottom is "/nfs/mica/u1/zhangch/project/mytool/gaussian.cs".

```
id = ID;
seed = id;
marked = 0;
els.a[N] = 0.0;
for (i = 0; i < N; i++) {
    els.a[i] = random(&seed);
    els.a[N] += i * els.a[i];
}
}
solution = TRUE;
/*
for (i = 0; i < N; i++) {
    for (j = 0; j < N+1; j++)
        printf ("%6.3f ", [j]els.a[j]);
    printf ("\n");
}
*/

CS_StartTimer();
with (Row) {

    /* Gaussian elimination */

    for (i = 0; (i < N-1) && solution; i++) {
        where (!marked) {
            maximum = >?= fabs(els.a[i]);
            where (fabs(els.a[i]) == maximum) picked = (int) id;
            where (picked == id) {
                marked = 1;
                pivot = i;
            }
        }
        temp_array = [picked]els;

        if (fabs(temp_array.a[i]) < EPSILON) {
            solution = FALSE;
        } else where (!marked) {
            double:Row tmp;
            tmp = els.a[i] / temp_array.a[i];
            for (j = i; j < N+1; j++)
                els.a[j] -= temp_array.a[j] * tmp;
        }
    }
    where (!marked) pivot = N-1;
}
```

UNH C* compiler version 0.9277 built Tue Jun 6 11:03:29 PDT 1995

393, Inner loop not vectorized - unsupported form of assignment stmt.
634, Inner loop not vectorized - unsupported control flow
742, Inner loop not vectorized - unsupported form of assignment stmt.
779, Inner loop parallelized, blocked iteration allocation
Inner loop vectorized for counts >= 10, register size = 128 elements
815, Inner loop parallelized, blocked iteration allocation

/nfs/mica/u1/zhangch/project/mytool/gaussian.cs

Figure 24: Locating the line corresponding to the message: "393: Inner loop not vectorized..." generated by pgcc.

6.2 Major Implementation Issues

CSDE is constructed mostly in Motif, with just a small portion of the code making use of low-level Xlib functions. Many features in *CSDE* required much discussion and experimentation before we had a good implementation. In this section we focus on some of the more interesting and important issues.

6.2.1 Performance Window

Performance analysis is at the heart of this tool. Our initial goal was to represent the profiling data with bars aligned with the source code, so that it would be easy to determine how much time is spent in various portions of the program. Each bar would have a length, representing the cost of the represented profiling point; a color, representing type of activity; and a label, which would give more detailed information.

In Motif, there is a *ScrolledText* widget, which is a perfect candidate for displaying and editing C* programs, but this widget cannot display bars. This led us to the decision to use another widget to draw the bars, and the further decision to use the *DrawingArea* widget for that purpose. We were left with two problems: how to display the bars, particularly the portion corresponding to the part of the program in the edit window; and how to make the two widgets, the *DrawingArea* widget and the *ScrolledText* widget, scroll together.

One idea we had for solving the first problem was to draw all the bars on a pixmap, then load only the right portion of the pixmap to the *DrawingArea*. When the text was scrolled, the *DrawingArea* would need to be reloaded. This design had three problems. First, the memory required by the pixmap was too large to be handled by X. Second, when the performance window was resized, the bars were not. Third, the size of the pixmap had to be the maximum possible size of the performance window. Otherwise, when the performance window got larger

than the size of the pixmap, an ugly gap would appear.

Since the DrawingArea widget is drawable, we decided to discard the pixmap idea. Instead, we decided to draw bars directly on the DrawingArea widget. Each time the text is scrolled, the performance window is flushed and bars are redrawn. The concern about this method was the speed. Since each time we have to draw a windowful of bars one by one, if there are a lot of bars to draw, can the drawing speed catch up with the scrolling speed when the text goes very fast and flashes by? It turned out that the drawing speed is high enough to keep up with fast scrolling. There are advantages to this method. We do not need any buffering for the bars. When the performance window gets resized, the bars are also resized nicely. Moreover, the code is simpler.

In *CSDE*, each performance bar represents a profiling point. Hence each bar contains several pieces of information associated with that point. We collect the information in a data structure called BarStruct. All the bars for a C* program are put together in a barList:

```
typedef struct ProStruct
{   char        type;           /* profiling type */
    char        *typeName;      /* name of this type */
    double       time;           /* execution time */
    int          percentage;     /* percentage used for drawing the bars */
    int          barWidth;
    int          barHeight;
    struct ProStruct *next;
} BarStruct;

BarStruct      **barList;
```

The most important field in BarStruct is the *percentage*, which is the ratio of the cost of this profiling point to the cost of the most expensive point in the program. Variable

percentage is used to calculate the bar length, which is of the same percentage of the maximum drawable width of the *DrawingArea*, after some margin is left out. The source of *barList* is the .pix file, it has the following format:

```
<file name> <profiling type> <type name> <line number> <time>
```

Variable *barList* is allocated to be an array of linked lists. The size of the array is one greater than the length of the C* file. Linked list *barList[i]* groups together all the profiling points for line *i* of the C* program. After *barList* is loaded, *CSDE* draws bars in the performance window by invoking the function *redraw()*, using the following algorithm:

1. Get the width and height of the *DrawingArea* widget;
2. Clear the *DrawingArea* widget;
3. Calculate the start line and the end line of the part of the program that is displayed in the edit window;
4. For each linked list between *barList[start_line]* and *barList[end_line]*:
 For each bar in this list:
 Calculate its coordinates, width and height;
 Draw the bar (with 3D effects).

To scroll the *DrawingArea* widget and the *ScrolledText* widget together—the second problem, we also tried more than one approach. One was to use a *ScrolledWindow* to manage the two widgets and to scroll them together. But this is difficult to implement and the complexity turned out to be unnecessary. We chose to let the *DrawingArea* widget respond to the scrollbar of the *ScrolledText* widget. Or to put it in another way, the scrollbar of the *ScrolledText* controls both the *DrawingArea* and the *ScrolledText*. This may sound complicated, but the actual code is very simple:

```

/* get the vertical scroll bar of the textWidget */
XtVaGetValues (XtParent (textWidget), XmNverticalScrollBar, &vsb, NULL);

/* add callbacks to the vertical scrollbar: the purpose here is to
   attach the scrollbar to both the text and the drawing area */
XtAddCallback (vsb, XmNvalueChangedCallback, scrollActionCb, NULL);
XtAddCallback (vsb, XmNdragCallback, scrollActionCb, NULL);
XtAddCallback (vsb, XmNincrementCallback, scrollActionCb, NULL);
XtAddCallback (vsb, XmNdecrementCallback, scrollActionCb, NULL);
XtAddCallback (vsb, XmNpageIncrementCallback, scrollActionCb, NULL);
XtAddCallback (vsb, XmNpageDecrementCallback, scrollActionCb, NULL);
XtAddCallback (vsb, XmNtoTopCallback, scrollActionCb, NULL);
XtAddCallback (vsb, XmNtoBottomCallback, scrollActionCb, NULL);

```

The scrollActionCb is just a few lines long:

```

void scrollActionCb (
Widget scrollbar, XtPointer client_data, XtPointer call_data)
{
    XmScrollBarCallbackStruct *cbs =
        (XmScrollBarCallbackStruct *) call_data;
    lineOffset = cbs->value;
    redraw ();
}

```

Therefore, whenever the scrollbar moves, scrollActionCb is activated. It readjusts the DrawingArea or, more precisely, changes the values of the first_line and last_line in redraw() through the global variable lineOffset, and the bars are redrawn. This explains why the redrawing is fast enough to keep up with scrolling: the variable keeps track of the

offset of the text that is displayed in the edit window, or the number of lines above the portion of text in the edit window. Its value is obtained from the `CallbackStruct` of the scrollbar. Even if the text scrolls very fast, `lineOffset` is changed immediately and the bars are redrawn properly.

6.2.2 Calculation for bars

In this section we explain how the calculation in steps 3 and 4 of `redraw()` is done. We shall see that the calculations for bars are somewhat meticulous. Nevertheless, none of the details are ignorable. Even if we miscalculate by one pixel, the bars would turn out to be malpositioned, and they might appear to be migrating up and down while the windows are scrolled.

As shown in Figure 15, the top of the *DrawingArea* and the top of the *ScrolledText* are aligned under the menu bar. However, although the *DrawingArea* is drawable starting from right under the menu bar, the text is displayed some distance from the top of the edit window. Hence we also need to leave out the same amount of distance in the performance window. We refer to the distance as `margin`, which can be obtained from the function `XmTextGetBaseline()`. This function returns the y coordinate of the baseline of the first line of text in the specified text widget. Its returned value is relative to the top of the widget and it accounts for the margin height, shadow thickness, highlight thickness, and font ascent. Therefore, we have

```
margin = XmTextGetBaseline(textWidget) - appFont->ascent
```

where `appFont` is a global variable for the font of the text.

Another important parameter for our calculation of bar positions is the height of a line in the text. For now we assume that the height is known as `HEIGHT`, then the `start_line` and

the end_line in step 3 of redraw() are given by:

```
/* view_lines is the possible number of lines in the edit window */
view_lines = (drawingarea_height - 2*margin)/HEIGHT;
start_line = lineOffset + 1;
end_line = (lineOffset + view_lines) > workFileLines ?
           workFileLines : lineOffset + view_lines;
```

Suppose a bar is to be drawn for the *i*th line (its barStruct is in the linked list barList[i]), then its x,y coordinates, as well as its width and height are:

```
x = margin + 1;
/* The selection of x has no other reason than trying to make the bars
   look nice */
y = margin + (i-1-lineOffset)*HEIGHT + BAR_DESCENT;
/* BAR_DESCENT is the tiny distance between the top of the bar and the
   top of the corresponding text line, also is the distance between the
   bottom of the bar and the bottom of the corresponding text line.
   This constant is defined so that there is some distance between the
   vertical adjacent bars and they will not merge together */
bar_width = ptr->percentage * (drawingarea_width - 2*margin) / 100;
bar_height = HEIGHT-2*BAR_DESCENT;
```

Now we look back at the HEIGHT. We claimed that this is the height of a line in the text, but we have not found a good way to get its value. Note that HEIGHT is not the height of the font; it is greater than the font size. However, based on our experience we have a hypothesis: HEIGHT is the cursor height in the edit window. Unfortunately, we were not able to prove it because we could not get the size of the cursor. We experimented with several different

fonts, each time a hypothesized cursor height is used for HEIGHT, the results were correct. Nevertheless, in order to solely solve the problem, we need to find a general formula to get the cursor size as a function of the font, so that whatever font the user chooses to use, the bars would be positioned correctly. Since the formula has not been found, the font in *CSDE* is hard coded and the HEIGHT is fixed.

As for the mouse-clicking events in the DrawingArea widget, it is handled in the callback `drawInputCbk`, which simply gets the coordinates of the mouse and calculates whether it is positioned on a bar. If it is, we reverse the bar's shadow to make it appear being pressed, and pop up an information dialog box to display the timing data, profiling type, etc.

6.2.3 Compiling and Running a C* program

In *CSDE*, compiling and running a C* program is done in callbacks `menuCompileCbk` and `menuRunCbk`. The behavior of the two callbacks is designed as follows. A dialog box pops up when the program starts to compile/run. The box informs the user that the compilation/running is in progress. It stays until the process is finished, forbidding the user to interact with other parts of the tool, and it also provides the user an alternative to cancel the process. When the compilation/running is completed, the dialog box changes its message to inform the user of the job's completion, and it allows the user to resume interacting with other components of *CSDE*.

To forbid invoking other parts of *CSDE* while the tool is compiling/running, we only need to set the `XmNdialogStyle` resource of the dialog box to `XmDIALOG_FULL_APPLICATION_MODAL`. But to pop up the dialog box simultaneously with the starting of compilation/running is far more difficult than we expected.

A naïve approach was to pop up the dialog box first, then immediately make a system call to do the compilation/running. However, the result was not right. The frame of the dialog

would appear, but the message inside the dialog box would not appear until the system call was over. By that time the dialog was already obsolete. It took us some time to figure out the reason: the dialog message will not show up until the event loop is completed, but since the event loop is caught by the system call, the X server is not able to get back to the dialog box until the system call is over.

To attack the problem, we spawned a child process, which took over the responsibility to do the system call while the parent pops up the dialog box. We thought that by spawning a child process, the parent could get out of the event loop immediately to display the dialog box. But interestingly enough, the result was still as wrong as before.

Finally, we hit upon the right idea. We replaced the system call in the child process with an `exec` call. Two programs *APP_compile* and *APP_run* are set up for compiling and running C* programs. They invoke the C* compiler and the `prun` via `rsh` or `remsh`, depending on the machine architecture. The callbacks `menuCompileCb` and `menuRunCb` spawns a child process, which does an `exec` call to *APP_compile* or *APP_run*, and the parent process goes ahead handling the dialog box and other interactive work.

But then how does the parent know whether the child has finished so that it can change the dialog box to inform the user? Ordinarily the answer would be to let the parent *wait* for the child. However, this answer turned out to be incorrect in this particular case, because a `wait` also caused the abnormal behavior of the interface. The right solution is to use signals. Before the `menuCompileCb` spawns a child process, a signal handler `compileReset` is set up:

```
signal (SIGUSR1, compileReset);
```

and before the `menuRunCb` spawns a child process, a signal handler `runReset` is set up:

```
signal (SIGUSR2, runReset);
```


Signals SIGUSR1 and SIGUSR2 are sent out by APP_compile and APP_run respectively when the compiling or running process terminates. Once the parent process catches the two signals (through the signal handler), it knows that the child terminates and replaces the dialog box.

The implementation of “Copy” is under the similar vein. A new process is spawned and it does an exec call to *CSDE* itself.

6.2.4 The Working Versions

When a C* program is loaded into *CSDE*, the tool creates a “working copy” of it in directory /tmp. The copy is renamed with the id of the creating process attached. For instance, if the source program is called `foo.cs`, and the id of the process loading the program is 893, then the working copy is named `foo__893.cs`. Since the process id is unique, the working copy is also unique. Later all the work such as compile, run, etc, is done on the working copy instead of the original program. The changes made to the working copy can be saved and copied back to the source program via “Save”.

When a new Copy window is created by “Copy”, another working copy of the program is made for it. For example if the process controlling the new Copy window has id 910, then the new working copy is `foo__910.cs`.

After the program is compiled or run, some auxiliary files are created under /tmp. File `*.compile` contains compile-time messages that otherwise go to the stdout and stderr, and file `*.run` contains the run-time messages output from prun. The messages in these files are then parsed and displayed in the build window, which is implemented as a *ScrolledList* widget, so that users can click on the messages. A drawback of our handling of the compile-time and run-time messages is that they do not appear simultaneously with the compilation or running process. Instead they go to the build window when the process is done. This is because we could map `*.compile` and `*.run` to the *ScrolledList* only after the messages are

completely redirected from `stderr` and `stdout`.

6.2.5 Locating Error and Vector Messages

With the technique of drawing bars being established, drawing a small triangle next to a certain line was easy. The hard part is parsing the error message clicked by the user in the build window to get the line number and scrolling that line into the edit window.

Theoretically the error message can be an arbitrary string. As long as the whole sentence makes sense, a line number (if there is any) can be anywhere in the message without showing any pattern. Fortunately, the error messages are generated by a computer routine instead of by a person, so they always look the same. For example, an error containing a line number generated by the C* compiler looks like this:

```
<filename>.cs:<linenumber>: error: <reason>.
```

How to parse a message like this is quite obvious.

To scroll the error line into the edit window, we again make use of the `start_line` and `end_line` as described in `redraw()`. The algorithm is embedded in the code below:

```

/* Get the initial start_line and end_line */
view_lines = (drawingarea_height - 2*margin)/HEIGHT;
start_line = lineOffset + 1;
end_line = (lineOffset + view_lines) > workFileLines ?
            workFileLines : (lineOffset + view_lines);

/* Scroll backward */
while (line_number < start_line)
{
    lineOffset -= view_lines - 1;
    /* Scroll backward one page */
    XmTextScroll (textWidget, 1-view_lines);
    start_line = lineOffset + 1;
    end_line = lineOffset + view_lines;
}

/* Scroll forward */
while (line_number > end_line)
{
    lineOffset += view_lines-1;
    /* Scroll forward one page */
    XmTextScroll (textWidget, view_lines - 1);
    start_line = lineOffset + 1;
    end_line = lineOffset + view_lines;
}

```

Locating the line numbers associated with messages from the pgcc compiler is slightly more complex, because these line numbers refer to the intermediate C program, not the user's C* program. Given the pgcc line number, we have to find the line number in the C* program. The method is based on the observation that the C* line number is the fifth field in the CS_profile temporary variable. For instance,

```
/* A declaration of CS__temp_12 in .c file produced by the C* compiler */  
static CS__profile CS__temp_12 = { 0, 'V', "vp-loop", "pi2.cs", 26  
                                   /* Rest are zero'd by type */ };
```

This vp-loop happens in line 26 of `pi2.cs`. Suppose the pgcc line number is n . We work through the .c file produced by the compiler, counting n newline characters, then back up to match the string "static CS__Profile CS__temp_", and proceed to search for the fifth field in the structure.

In order to enable the vector location ability, *CSDE* always produces a .c file by default when compiling a program.

7 Comparing Performance of Run-time Libraries

In this section we compare the performance of the two versions of the C* run-time libraries. One utilizes the NXlib and is implemented by the University of New Hampshire—the UNH version, and the other is our OSU version implemented using the EWlib.

7.1 Designing Benchmarks

Before setting out to get the performance data, we need to decide how to write benchmark programs. There are several issues involved. First, what operations should the benchmarks contain? Second, how should we time the operations? Third, what data types should we use? Fourth, what problem sizes should we use?

Since our goal here is to compare the speed of the six communication operations instead of testing the correctness of the library (of course, we should guarantee the correctness at first), we prefer “pure” benchmarks that only contain the particular operations to compare. Therefore, to compare the performance of the broadcast of two versions, we use a program that does nothing but broadcasts.

To time the communications, we put the C* run-time library functions `CS_StartTimer()` and `CS_StopTimer()` around the compared operations, because it is the most likely way that a C* user would time his/her programs. One might ask why not use *CSDE* to measure the performance? First, *CSDE* provides a very good environment for comparing the performance among different communications in the same program, and among different versions of the same program. It is not convenient for comparing performance of different run-time libraries. Second, because the running of a program is through remote shell, the timing data are not precisely accurate: they tend to be larger than they actually are. Third, the measurement using *CSDE* would take a tremendous amount of time. By using *CSDE*, although you can

compile/run C* programs anywhere you can use Motif, the compilation and running takes much longer than on the Meiko. Furthermore, the performance data have to be retrieved by clicking on the bars and have to be recorded manually, whereas if running on Meiko and using the timers, the data can be retrieved, recorded, and processed automatically.

Since the principle domain of C* applications is scientific programming, and most scientific programs deal with doubles, we use `double` as the data type for our benchmarks.

Generally people would eschew “toy programs” and favor benchmarks of decent sizes to test their software, but this is a different scenario. For most of the communication functions, we used the same or similar algorithms as in the UNH version, but we used different message passing methods. All algorithms spend most of the time calculating source and destination processors and buffering data. The actual message transmission only takes a small percentage of the time, and the larger the data size, the smaller the percentage. Since we aim at comparing the difference of the two run-time libraries, we choose such benchmark programs that can minimize the overhead of the algorithms and focus on the difference. Therefore, we use relatively small but non-trivial problem sizes for the benchmarks.

7.2 Performance Comparison

The benchmarks we have used for testing are attached in Appendix A. For each of the compared operations, we do one thousand iterations. The actual cost is the result averaged over one thousand. The ten iterations precede the one thousand iterations are to get rid of the first big timings that are due to the setup of the CS-2 network. In all the benchmarks the correctness of the operations are also tested to ensure the validness of the timing data. The results of the comparisons are given in Figures 25 through 30. 95% confidence intervals are drawn on all curves.

Below is a summary of the comparisons.

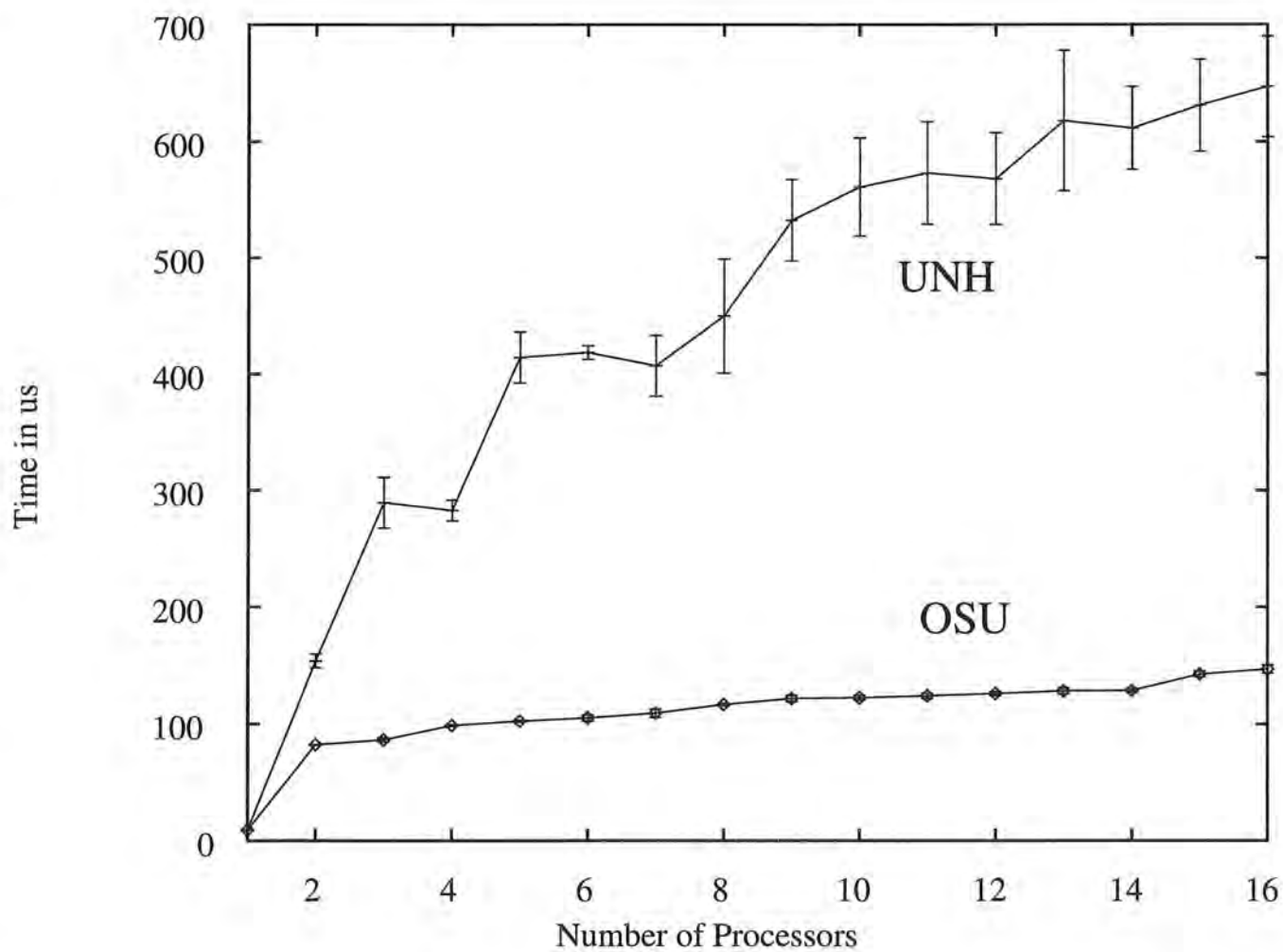


Figure 25: Broadcast

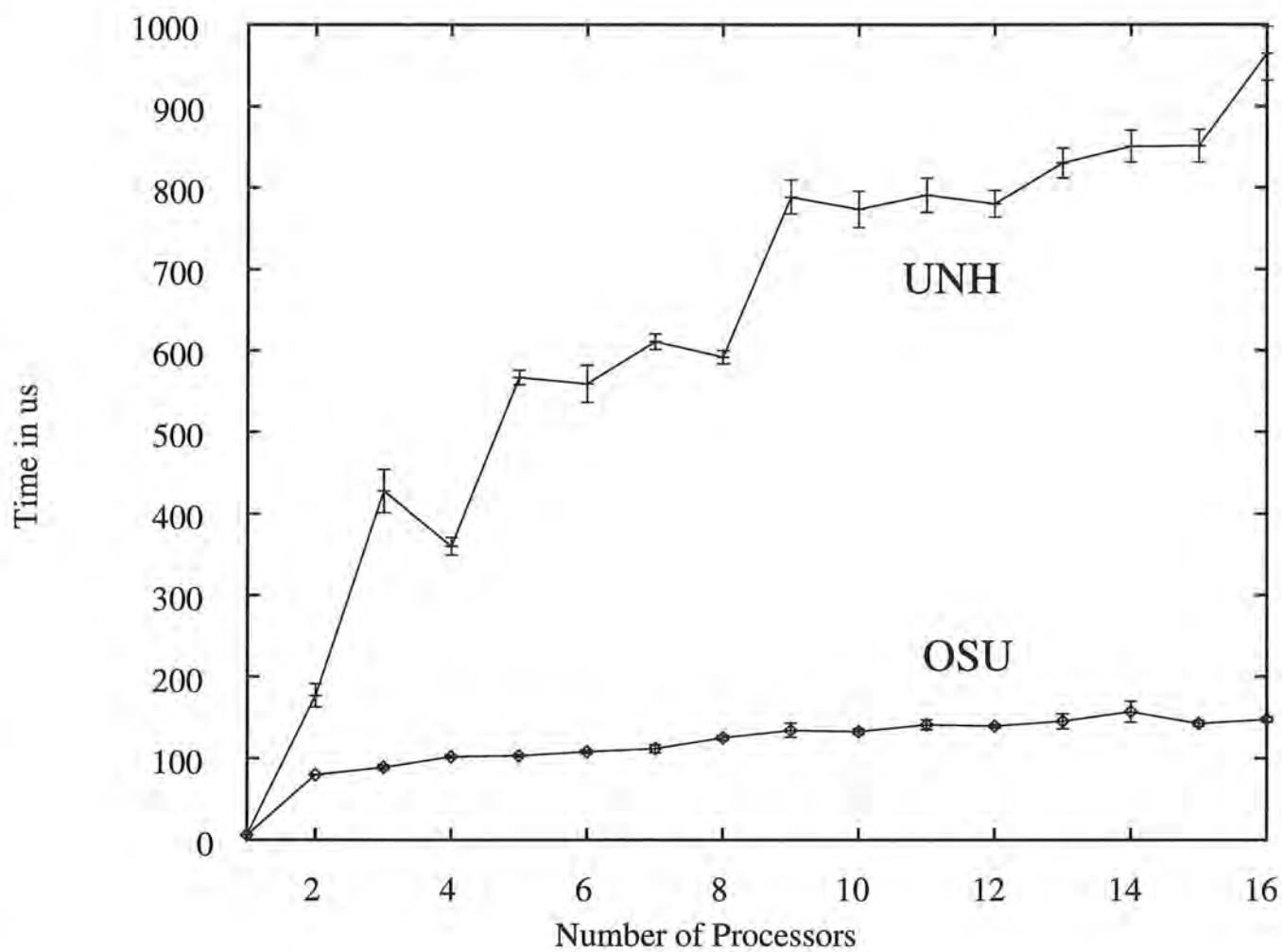


Figure 26: Reduction

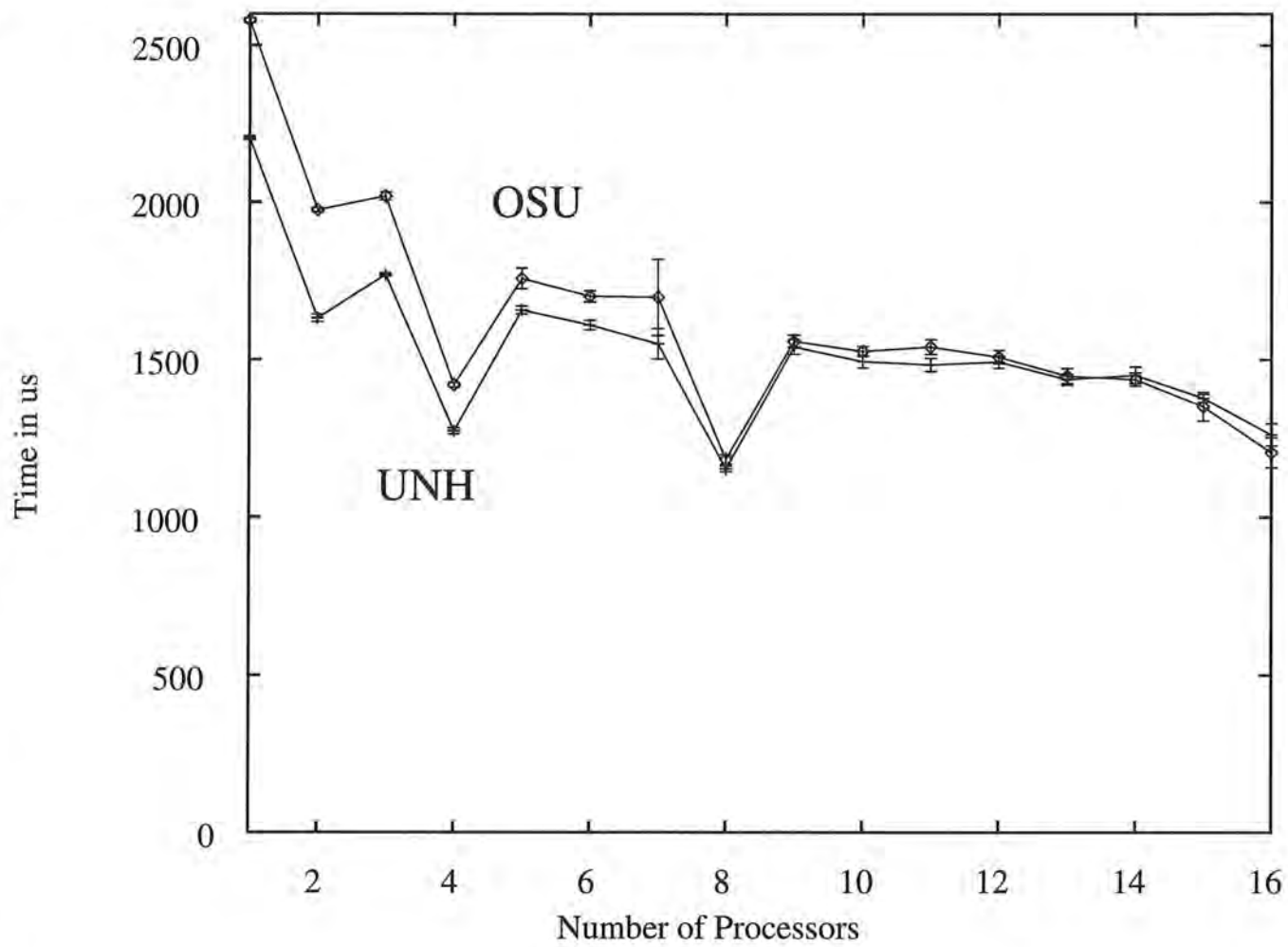


Figure 27: Scatter

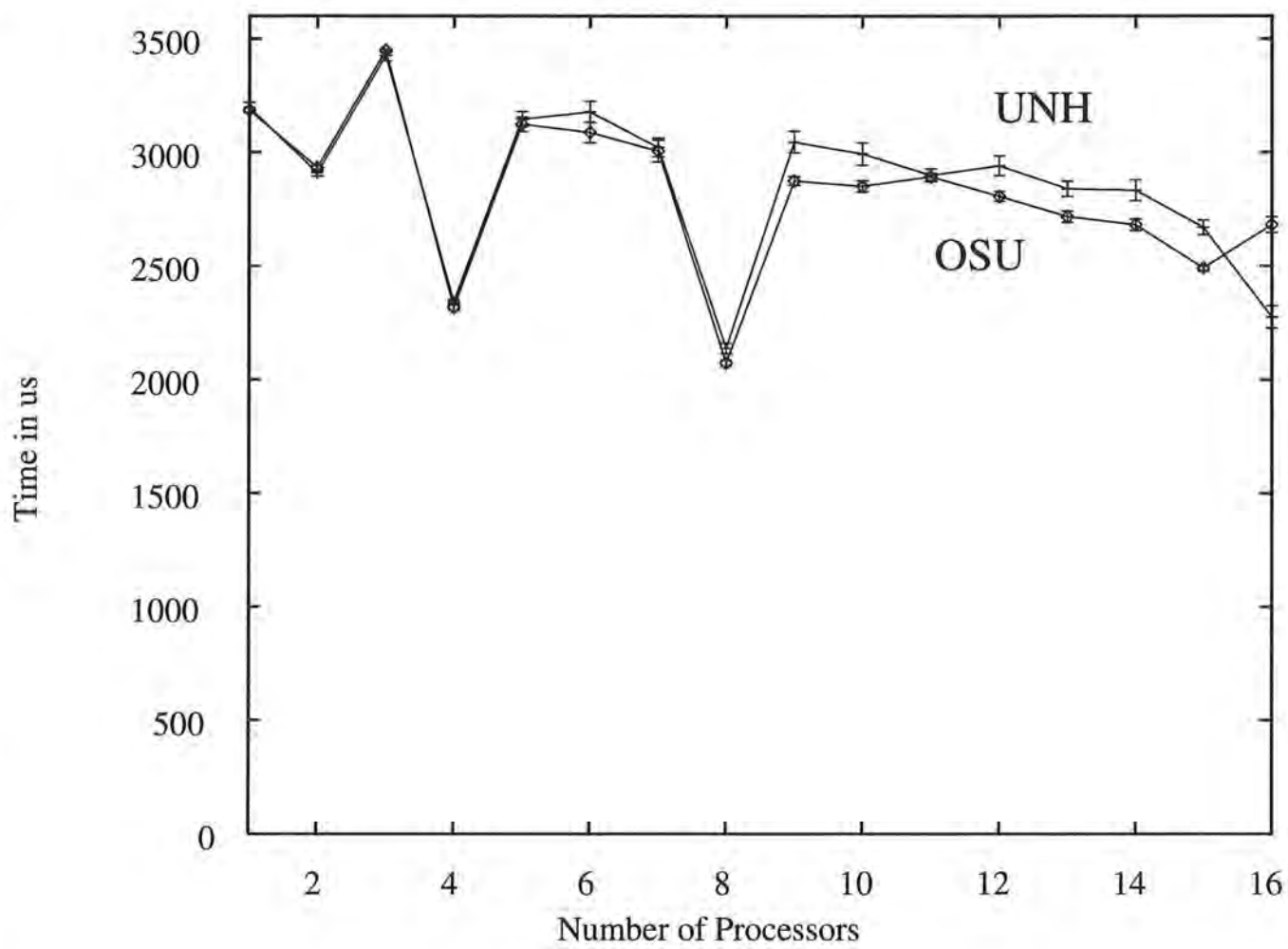


Figure 28: Gather

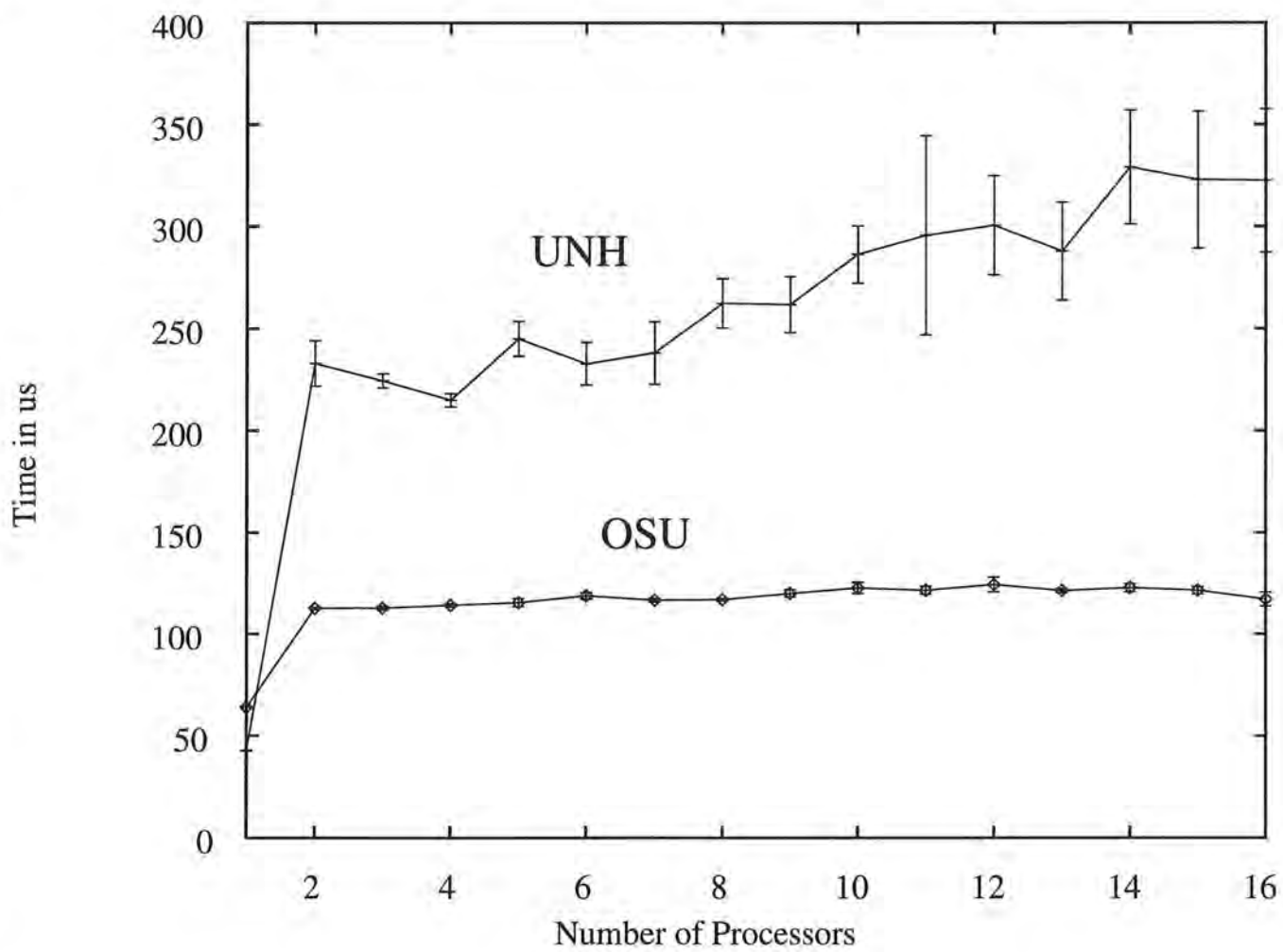


Figure 29: Grid-read

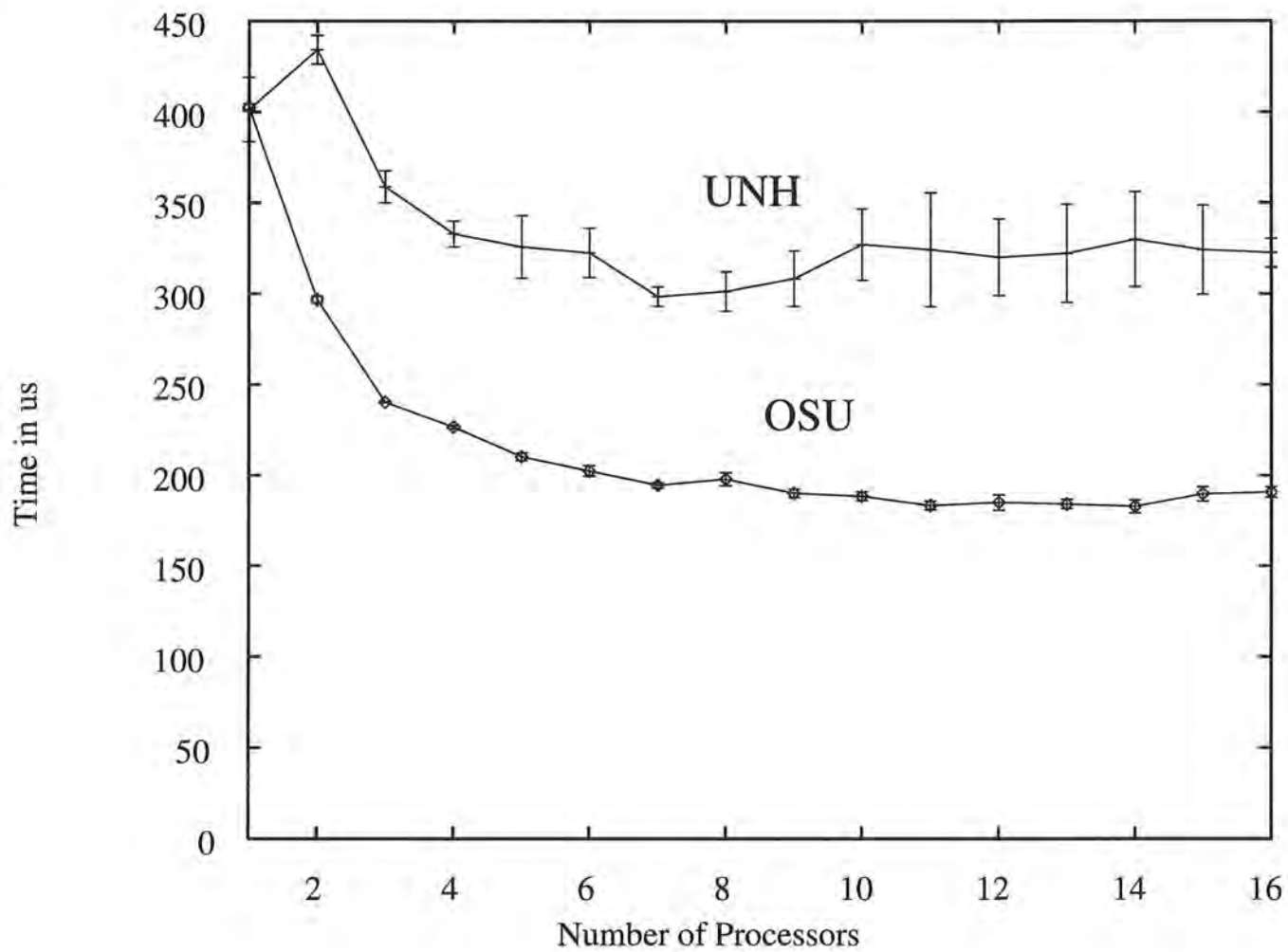


Figure 30: Grid-write

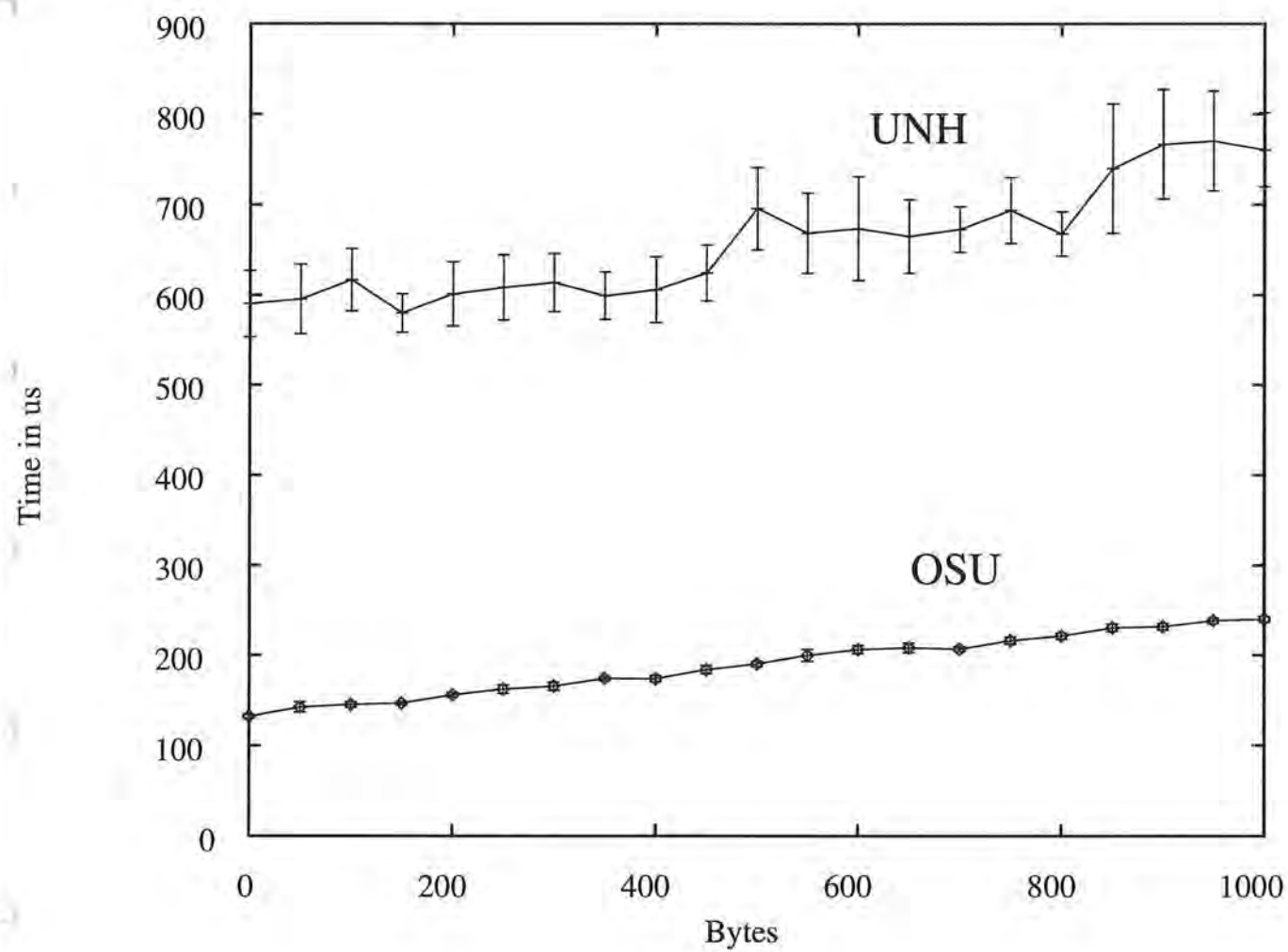


Figure 31: Bandwidth and latency for one-to-all broadcast on 16 processors.

Broadcast (Figure 25)

- A different value is broadcast in each iteration.
- The result shows that the performance of the OSU version is dramatically improved over the UNH version.
- Since the UNH version adopts a logarithmic algorithm, the curve labeled UNH reflects the execution time being proportionate to $\lceil \log p \rceil$, where p is the number of processors.

reduce (Figure 26)

- The result shows that the performance of the OSU version is dramatically improved over the UNH version.
- Like broadcast, the UNH curve also shows that the execution time is proportionate to $\lceil \log p \rceil$.

scatter (Figure 27)

- The benchmark is designed such that each processor sends a value to every processor. Therefore the communication pattern is balanced.
- The result shows that the OSU scatter is not as good as the UNH one for small number of processors. But as the processor number increases, the OSU version starts to gain speed.

gather (Figure 28)

- The benchmark is designed such that each processor gathers a value from every other processor.

- The result shows that the OSU gather is marginly better than the UNH version. Also with the increasing of the number of processors, the OSU version starts to gain more speed.

grid-read (Figure 29)

- The result shows that the OSU grid-read is better than the UNH grid-read.

grid-write (Figure 30)

- The result shows that the OSU version is better.

Bandwidth and latency for broadcast (Figure 31)

- This is to test the bandwidth and latency for the broadcast. The test is done on 16 processors.
- Approximate latency (to send one message) for the OSU version: 32.5 microsecond, for the UNH version: 147.5 microsecond.
- Approximate bandwidth for the OSU version: 9 Mb/sec, for the UNH version: 6 Mb/sec.

Conclusions

Generally speaking, the OSU routing functions execute faster than the UNH functions. Also from the comparisons broadcast and reduction operations, the performance of the library can be dramatically improved by implementing the C* run-time library using the lower level, Meiko-specific function calls. Therefore, the decision to write a Meiko-specific version of the C* run-time library was worthwhile.

8 Future Work

There are a number of optimizations that have been provided for in the design of the tool, but are beyond the scope of this paper. These are outlined below.

8.1 Improve Scatter and Gather

In last section, we have seen that the performance of OSU scatter and gather operations are not satisfactory. We should study ways to improve them.

8.2 Unifying Bar Percentages of Copy Windows

Since the Copy windows and the original root window of *CSDE* are independent, the base cost of the bars—the value against which the percentage of bars are calculated—are not forced to be uniform across all windows. That is, the bar lengths of different windows are not proportional to the absolute times. This means that even if a profiling point of one version in one window takes less time than the same point of another version in another window, its bar may be longer. This situation may cause the user to draw false conclusions. As a problem to solve in the future, we will figure out how to maintain the same base value among the Copy windows and make the bar lengths consistent.

8.3 Selecting Colors and Fonts

A major goal of GUI tool development is to provide an as-pleasant-as-possible environment. If the user hates the font or color of the tool, he/she may not be happy to use it. Therefore, a list of fonts and colors will be provided for selection in the future.

8.4 Integrate *CSDE* into an Interactive Debugging Tool

CSDE is actually a primitive interactive debugging tool. For example, for those who love to use `printf`'s, it is very convenient for them to add `printf` statements into the program via the edit window, and watch the results through the performance and the build window. But certainly a real debugging tool has much more capabilities than that. For instance, our future GUI debugger may have a source-code instrumentor, which allows users to insert break points or monitoring routines to the source code. However, this is not a small project, as it involves many compiler modifications. Also, we need to further improve the performance analysis ability. For example, we could allow the user to specify the kind of data he/she needs such as mean, standard deviation, confidence intervals, etc. of some profiling cost, and then the tool would provide those data automatically.

References

- [1] E. Barton, J. Cownie, and M. McLaren. Message Passing on the Meiko CS-2. *Parallel Computing*, 20:497–507, 1994.
- [2] CONVEX Computer Corporation. *CXtrace User's Guide*, 1994.
- [3] D. Heller and P. M. Ferguson. *Motif Programming Manual*. O'Reilly & Associates, Inc., 1994.
- [4] K. P. Herold. A retargetable C* Communication and Run-time Library for Mesh-Connected MIMD Multicomputers. Master's thesis, University of New Hampshire, 1992.
- [5] Meiko World Incorporated. *CS-2 Communications Network*, 1993.
- [6] Meiko World Incorporated. *CS-2 Elan Communication Processors*, 1993.
- [7] M. J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, Inc, 1994.
- [8] G. Stellner, S. Lamberts, and T. Ludwig. *NXLIB User's Guide*. Institut für Informatik, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Technische Universität München, 1993.
- [9] Thinking Machines Corporation. *C* Programming Guide*, 1993.

A. Benchmarks for Performance Comparisons

Broadcast

```
/* File:      broadcast.cs
 * Purpose:   Benchmark program for comparing the performance of the broadcast
 *           operation in the UNH and OSU versions of the C* routing library.
 */
#include <stdio.h>
#define N 32
#define REPEATS 1000

main( )
{
    shape [N]sA;
    double: sA pv;
    double sv[REPEATS];
    int i;

    with (sA)
    { /* Initialize pv */
        for (i = 0; i < N; i++)
            [i]pv = i*0.14;

        /* First 10 iterations to get rid of the big timings
         * due to the set-up of the CS-2 network
         */
        for (i = 0; i < 10; i++)
            sv[i] = [i%N]pv;

        /* Real iterations come here. In each iteration, a
         * different value is broadcast.
         */
        CS__StartTimer();
        for (i = 0; i < REPEATS; i++)
            sv[i] = [i%N]pv;
        CS__StopTimer();
    }

    /* Check correctness */
    for (i = 0; i < REPEATS; i++)
    { if (sv[i] != (i%N)*0.14)
      { fprintf(stderr, "Error: the value is not right!\n");
        break;
      }
    }
}
```

Reduce

```
/* File:      reduce.cs
 * Purpose:   Benchmark program for comparing the performance of the
 *           reduction operation in the UNH and OSU versions of the
 *           C* routing library.
 */
#include <stdio.h>
#include <math.h>
#define N 32
#define REPEATS 1000

main( )
{
    shape [N]sA;
    double: sA pv;
    double sv;
    int i;

    with (sA)
    { /* Initialize pv */
        for (i = 0; i < N; i++)
            [i]pv = i*0.519;

        /* First 10 iterations to get rid of the big timings
         * due to the set-up of the CS-2 network
         */
        for (i = 0; i < 10; i++)
            sv = +=pv;

        /* Real iterations come here. */
        CS__StartTimer();
        for (i = 0; i < REPEATS; i++)
            sv = +=pv;
        CS__StopTimer();
    }

    /* Test correctness of the reduction */
    if (abs(sv - (N-1)*N/2*0.519) > 0.000001)
        fprintf (stderr, "Error: the value is not right!\n");
}
```

Scatter

```
/* File:      scatter.cs
 * Purpose:   Benchmark program for comparing the performance of the
 *            scatter operation in the UNH and the OSU versions of the
 *            C* routing library.
 */
#include <stdio.h>
#define MAXP 16
#define N MAXP*MAXP
#define REPEATS 1000

main()
{
    shape [N]sA;
    double:sA pvs, pvd;
    int:sA pvi;
    int i;

    with (sA)
    { /* Initialize pvs, pvi */
        for (i = 0; i < N; i++)
        { [i]pvs = i*0.1;
          [i]pvi = (i/MAXP) + MAXP*(i%MAXP);
        }

        /* First 10 iterations to get rid of the big timings
         * due to the set-up of the CS-2 network
         */
        for (i = 0; i < 10; i++)
            [pvi]pvd = pvs;

        /* Real iterations come here */
        CS__StartTimer();
        for (i = 0; i < REPEATS; i++)
            [pvi]pvd = pvs;
        CS__StopTimer();
    }

    /* Check correctness */
    for (i = 0; i < N; i++)
    { if ([i]pvd != ((i/MAXP) + MAXP*(i%MAXP))*0.1)
      { fprintf (stderr, "Error: the value is not right!\n");
        break;
      }
    }
}
```


Gather

```
/* File:      gather.cs
 * Purpose:   Benchmark program for comparing the performance of the
 *            gather operation in the UNH and the OSU versions of the
 *            C* routing library.
 */
#include <stdio.h>
#define MAXP 16
#define N MAXP*MAXP
#define REPEATS 1000

main()
{
    shape [N]sA;
    double:sA pvs, pvd;
    int:sA pvi;
    int i;

    with (sA)
    { /* Initialization */
        for (i = 0; i < N; i++)
        { [i]pvs = i*0.1;
          [i]pvi = (i/MAXP) + MAXP*(i%MAXP);
        }

        /* First 10 iterations to get rid of the big timings
         * due to the set-up of the CS-2 network
         */
        for (i = 0; i < 10; i++)
            pvd = [pvi]pvs;

        /* Real iterations come here */
        CS__StartTimer();
        for (i = 0; i < REPEATS; i++)
            pvd = [pvi]pvs;
        CS__StopTimer();
    }

    /* Check correctness */
    for (i = 0; i < N; i++)
    { if ([i]pvd != ((i/MAXP) + MAXP*(i%MAXP))*0.1)
      { fprintf (stderr, "Error: the value is not right!\n");
        break;
      }
    }
}
```

Grid-read

```
/* File: gridread.cs
 * Purpose: Benchmark program for comparing the performance of the
 *          grid-read operation in the UNH and the OSU versions of
 *          the C* routing library.
 */
#include <stdio.h>
#define N 32
#define REPEATS 1000

main()
{
    shape [N]sA;
    double:sA pvs, pvd;
    int i;

    with (sA)
    { /* Initialization */
        for (i = 0; i < N; i++)
            [i]pvs = i*0.01;

        /* First 10 iterations to get rid of the big timings
         * due to the set-up of the CS-2 network
         */
        for (i = 0; i < 10; i++)
            pvd = [.+2]pvs;

        /* Real iterations come here */
        CS__StartTimer();
        for (i = 0; i < REPEATS; i++)
            pvd = [.+2]pvs;
        CS__StopTimer();
    }

    /* Check correctness */
    for (i = 0; i < N; i++)
    { if ([i]pvd != ((i+2)%N)*0.01)
        { fprintf (stderr, "Error: the value is not right!\n");
          break;
        }
    }
}
```

Grid-write

```
/* File: gridwrite.cs
 * Purpose: Benchmark program for comparing the performance of the
 *          grid-write operation in the UNH and the OSU versions of
 *          the C* routing library.
 */
#include <stdio.h>
#define N      32
#define REPEATS 1000

main()
{
    shape [N]sA;
    double:sA pvs, pvd;
    int i;

    with (sA)
    { /* Initialization */
        for (i = 0; i < N; i++)
            [i]pvs = i*0.01;

        /* First 10 iterations to get rid of the big timings
         * due to the set-up of the CS-2 network
         */
        for (i = 0; i < N; i++)
            [i+2]pvd = pvs;

        /* Real iterations come here */
        CS__StartTimer();
        for (i = 0; i < REPEATS; i++)
            [i+2]pvd = pvs;
        CS__StopTimer();
    }

    /* Check correctness */
    for (i = 0; i < N; i++)
    { if ([i]pvd != ((i+N-2)%N)*0.01)
      { fprintf (stderr, "Error: the value is not right!\n");
        break;
      }
    }
}
```

Bandwidth and Latency for the Broadcast

```
/* File:      bandwidth.cs
 * Purpose:   Benchmark program for comparing the bandwidth and latency
 *           for the broadcast operation in the UNH and OSU versions
 *           of the C* routing library.
 */
#include <stdio.h>
#include <string.h>
#define N 16
#define REPEATS 1000
#define SIZE 1000

main()
{
    shape [N]sA;
    struct foo {char x[SIZE];};
    struct foo: sA pv;
    struct foo sv;
    int i, j;

    with (sA)
    { /* Initialize VP values */
        for (i = 0; i < N; i++)
        { for (j = 0; j < SIZE-1; j++)
            [i]pv.x[j] = 'A'+i;
            [i]pv.x[SIZE-1] = 0;
        }

        /* First 10 iterations to get rid of the big timings
         * due to the set-up of the CS-2 network
         */
        for (i = 0; i < 10; i++)
            sv = [i%N]pv;

        /* Real iterations come here. In each iteration, a
         * different message is broadcast.
         */
        CS__StartTimer();
        for (i = 0; i < REPEATS; i++)
            sv = [i%N]pv;
        CS__StopTimer();
    }

    /* Test correctness of the broadcasting */
    for (i = 0; i < SIZE-1; i++)
        if (sv.x[i] != 'A'+ (REPEATS-1)%N)
        { fprintf (stderr, "Error: the value is not right!\n");
          break;
        }
}
```