Porting a Vector Library: a Comparison of MPI, Paris, CMMD and PVM*

Jonathan C. Hardwick
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
jch@cs.cmu.edu

Abstract

This paper describes the design and implementation in MPI of the parallel vector library CVL, which is used as the basis for implementing nested data-parallel languages such as NESL and Proteus. We outline the features of CVL, and compare the ease of writing and debugging the portable MPI implementation with our experiences writing previous versions in CM-2 Paris, CM-5 CMMD, and PVM 3.0. We give initial performance results for MPI CVL running on the SP-1, Paragon, and CM-5, and compare them with previous versions of CVL running on the CM-2, CM-5, and Cray C90. We discuss the features of MPI that helped and hindered the effort, and make a plea for better support for certain primitives. Finally, we discuss the design limitations of CVL when implemented on current RISC-based MPP architectures, and outline our plans to overcome this by using MPI as a compiler target. CVL and associated languages are available via FTP.

1 CVL overview

CVL (C Vector Library [6]) is a library of over 220 low-level vector functions callable from C. It provides an abstract vector memory model that is independent of the underlying architecture, and was designed so that efficient implementations could be developed for a wide variety of parallel machines. Machine-specific versions currently exist for the Connection Machines CM-2 and CM-5, the Cray Y-MP and Y-MP/C90, and the MasPar MP-1 and MP-2 [10], as well as a portable serial version written in ANSI C.

CVL was developed primarily as the lowest-level component of a three-tier system that implements the portable nested data-parallel language NESL [4][1]. Figure 1 shows an overview of the current system.

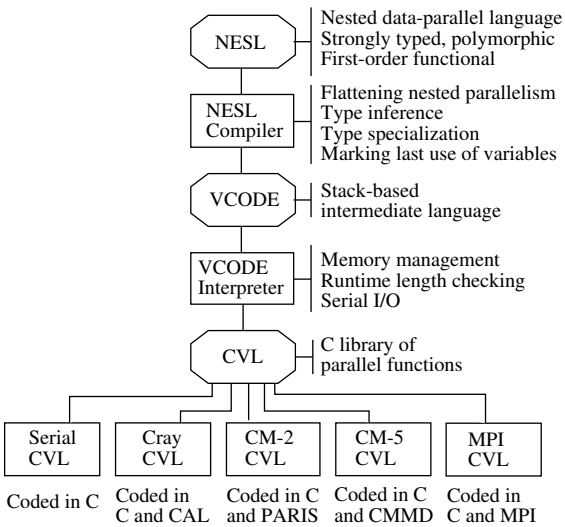

Figure 1: Overview of the NESL system

NESL is a parallel functional language with an ML-like syntax, and is designed to be used interactively. It is the first language to fully support nested data-parallelism: that is, parallelism both within nested data structures and across nested data-parallel function calls. This enables it to combine the advantages of data-parallel and control-parallel languages. NESL's basic data type is a vector, and nested parallelism is achieved through the ability to apply functions in parallel to each element of a vector. For example, a sparse array can be represented as a vector of rows, with each row represented by a subvector that

*This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. IBM SP-1 time was provided by the Argonne National Laboratory. Connection Machine CM-5 time was provided by the National Center for Supercomputer Applications (Grant TRA930102N).

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

[1] It is also used as a stand-alone C vector library, and as a back end by the Proteus language [15].

contains the nonzero elements (and corresponding indices) in that row. A parallel function that sums the elements of a vector can then be applied in parallel to sum each row of this sparse matrix. This ability to operate efficiently on irregular data structures is one of NESL's main strengths.

NESL is compiled into VCODE, a stack-based intermediate language [5]. NESL's nested data structures are flattened out into segmented vectors [3], allowing a single function call to operate on the entire data structure at once. The resulting VCODE is then interpreted, with the interpreter using the CVL library to achieve portability and efficiency. Since VCODE functions operate on vectors, the interpretive overhead of each instruction is amortized over the length of its vector operands [7].

To support high-level languages, CVL supplies a rich variety of vector operations, including elementwise function application, global operations such as scans and permutations, and ancillary functions such as timing and memory allocation. Most functions are supplied in both unsegmented and segmented versions, with the segmented versions being used to implement nested data parallelism. The functions whose implementations are likely to vary the most between different machine architectures can be broken down into two categories:

Scans and Reductions These apply an associative combining operator such as addition or maximum across a vector, returning either a single value (reduction), or a vector containing the "running total" (scan, or parallel prefix [2]). Their implementation on a parallel machine is normally via a binary tree combining network, either in hardware (CM-2, CM-5) or in software (MPI).

Permutations CVL has an extensive set of functions which permute the elements of a vector into a new vector. They are specialized by type, segmentation, direction, mapping, and default behavior. Their implementation is complicated by the fact that the resulting communication pattern is data-dependent and may not be balanced.

The rest of the paper is organized as follows. In Section 2, we outline the decisions to be made when porting CVL to a new platform. In Section 3 we describe the CM-2, CM-5, PVM, and MPI implementations in terms of how these decisions were made for the different platforms. In Section 4 we offer some comparative performance results, both for low-level CVL primitives and for NESL programs. Finally in Section 5, we offer some conclusions and recommendations for future work based on these results.

2 Porting CVL: the choices

There are six main choices to be made when porting CVL to a new platform:

Language & Library The choice of language (and possibly communication library) for a CVL implementation will affect its ease of development and final performance. This decision will also have an impact on most of the other decisions listed below.

Data Distribution All CVL data is stored in a block of "vector memory", which can only be accessed and modified through CVL routines. A method of distributing this memory across the machine must be chosen. Historically, all parallel implementations of CVL have used a block distribution.

Pointer Representation To enforce the independence of vector memory, CVL defines the C type `vec_p` as an abstract handle to a vector. A representation of this must be chosen, with the restriction that vectors may be reused to store data of different types, and hence a `vec_p` must be maximally aligned on the target architecture.

Segment Representation CVL defines a segmented vector as an unsegmented vector that contains the actual data, plus a "segment descriptor" that describes how to partition the unsegmented vector into subvectors (the separation of data from its structure enables elementwise functions to be oblivious to the structure of their operands, and allows sharing of both data vectors and segment descriptors). The contents of this segment descriptor must be decided upon. At its simplest, a segment descriptor can be a vector of segment lengths. However, on machines with high communication costs, additional information may be precomputed and stored in the segment descriptor, in order to reduce the amount of communication subsequent segmented functions must perform.

Host Model Depending on the machine architecture, CVL may be implemented in either a hostless or host/node style. In the hostless style an identical copy of CVL (and the associated user program) runs on each node. This requires less coding, and has the advantage of minimizing synchronization overhead — nodes can run free on sequences of instructions which involve no communication operations. The alternative is to use a host/node style, where only the host machine runs the user program (note that the host may be either the front-end computer attached to a parallel machine, or a chosen node of the parallel machine). The CVL

library on the host is reduced to a set of function stubs that broadcast function calls and arguments to slave processes running on the nodes. This has several advantages. First, the slave process on each node is relatively small, leaving more room for user data. Second, the host and nodes can overlap computation; the user program on the host can continue execution while the nodes execute CVL instructions. Finally, this may be the only way to give the user program the single-process, Unix-style environment it expects.

Message Buffering If a message-passing system is used, there may be some performance advantage to be gained by buffering messages in user space before sending, rather than relying entirely on system buffering [17]. However, user buffering also introduces the problem of how nodes determine when all messages have been sent and received. It is not enough for the nodes to use a barrier to agree that all messages have been sent, since messages may still be in transit. There are at least two obvious solutions: requiring each node to send at least one message to every other node, and repeatedly performing a global summation of the number of messages sent by each node minus the number it has received, terminating when the total falls to zero.

3 Existing CVL implementations

In this section we describe the machine-specific CVL implementations for the CM-2 and CM-5, the prototype PVM implementation, and the portable MPI implementation. For all four implementations we discuss the particular solutions chosen to the CVL porting decisions outlined in Section 2, and their impact on development time and final performance.

3.1 CM-2 CVL

The CM-2 implementation of CVL is written in C and Paris [19], a parallel instruction set for the CM-2's SIMD processing array. The main alternative was CM Fortran [21], but it could not be used because at the time it did not have the ability to alias arrays (for example, storing an array of floating-point numbers where an array of integers used to be), and therefore did not meet CVL's vector reuse requirements.

CM-2 CVL vector memory maps naturally to the Paris concept of fields, and a `vec_p` is defined as a Paris field ID. Paris has direct support for segmented operations, so the segment descriptor is designed around the segmentation format that Paris expects. The CM-2 is a SIMD machine with an explicit host/node model, so the host model decision was made for us. Finally, Paris supplies its own permutation primitives, rather than explicit message-passing functions, so no buffering was needed.

Paris ease of development: Paris has direct equivalents for many of the CVL functions, easing their implementation. However, development and debugging was hampered by the three classical problems of coding on a remote shared supercomputer:

- Unfamiliar and limited debugging tools.
- Competition for editing and compilation resources with other users on the front end machine, and for machine resources on the supercomputer.
- A limited account allocation that discourages the use of run-time debugging.

Debugging therefore tended to be of the "core dump and printf" style.

Paris performance: There are several performance penalties involved in using Paris:

- Paris uses the older "fieldwise" representation of data, requiring extra transpose operations to use the CM-2's floating-point processors. The newer and more efficient "slicewise" representation used by CM Fortran is not supported by Paris. This contributes to a factor of 4 difference in speed between CM Fortran and NESL when running the floating-point intensive linefit benchmark described in Section 4.3 [7].
- Paris communication operations use one virtual processor per element, which requires making "aliases" of dynamically-sized fields before accessing them. However, it is only possible to make aliases of originally allocated fields, and not of fields constructed by pointer manipulation. Since CVL does not know whether a particular `vec_p` is original or constructed, it must copy each vector into a freshly-allocated field before making an alias of it.
- Paris fields are restricted to 64 kbits per processor in length, whereas the machine typically has 256 kbits or 1024 kbits available. This limits the maximum problem size that can be run with CM-2 NESL.

CM-2 CVL is therefore not fully exploiting the machine in terms of floating point performance, memory traffic, or memory utilization. In practice, CM Fortran wins on benchmarks that emphasize floating-point operations, whilst Paris wins on benchmarks that emphasize communication [7].

3.2 CM-5 CVL

CM-5 CVL is written in C and the CM-5 message-passing library CMMD [22]. Again, CM Fortran could not be used because at the time it lacked array aliasing capabilities.

CM-5 CVL uses a blocked data distribution, with `vec_p`'s represented as 64-bit-aligned pointers. Note that because the nodes involved in a single CM-5 program run identical code, these pointers are identical across nodes. Segment descriptors are relatively large, reflecting high communication costs on a MIMD machine. The CM-5 may be programmed in either a hostless or host/node style, and the initial CVL implementation was hostless to speed development. However, the semantics of the VCODE interpreter were later extended to include the ability to spawn subprograms (such as an X11 interface). Since the CM-5 does not run a full Unix-style operating system on each node, there was no way to support this ability using the hostless model. CM-5 CVL was therefore rewritten to use the host/node model. The CM-5's dedicated control network means that the overhead of broadcasting each instruction is relatively small compared to the cost of executing it. Benchmarks at the time showed that the host/node implementation was in fact marginally faster than the hostless implementation for large NESL problems, due to the overlap in computation between the VCODE interpreter on the host and the CVL slave processes on the nodes.

CM-5 CVL permutation functions are written using the CMAM active message implementation [23]. With one word reserved for the active message function pointer, this restricts the message payload to four words on a standard CM-5. Therefore, for nearly all permutations, no buffering is used: vector elements are sent to the new destination as soon as the correct processor and offset have been calculated. Given the CM-5's fast control network, the global-sum-until-zero method is used for determining when all messages have been received.

CMMD ease of development: CMMD supplies scan and reduction functions that use the CM-5's control network, in both unsegmented and segmented versions. This greatly eased the writing of this class of functions.

CM-5 CVL tracked the development of CMMD through three major versions, necessitating constant rewriting. Active messages were introduced as a third-party extension to the first version, were incompatible with the second version, and were finally adopted by TMC in the third revision.

With limited account resources available, debugging was again of the "core dump and printf" style, with very little interactive use of the debugger on running programs. However, the ability to recover vector data from node core dumps was a big improvement over the CM-2's opaque memory model.

CMMD performance: The choice of C and CMMD involves performance penalties similar to those encountered with C and Paris on the CM-2. When the CM-5's vector units were introduced, offering memory-to-memory performance an order of magnitude better than that achievable with the SPARC processor on each node, only CM Fortran was upgraded with direct support for them. Using the vector units from C later became possible via a set of assembler macros [20]. With limited resources available, updating CM-5 CVL to use the vector units has not been attempted. Note that a CVL implementation using the new aliasing capabilities of CM Fortran would be able to fully realize the floating-point performance of both the CM-2 and the CM-5.

3.3 PVM CVL

A proof-of-concept implementation of CVL was written in C and PVM 3.0 [11], to explore the possibilities of a more portable CVL. A subset of the CVL functions were written in order to get an idea of the relative speed and usability of PVM for our application.

For speed of development, the design of PVM CVL was mostly copied from that of CM-5 CVL. Since the nodes in a PVM process are not guaranteed to be identical, the `vec_p` was represented as a 64-bit-aligned offset to be added to a per-node memory base. The host/node style of CM-5 CVL was also adopted, although this was probably a mistake. The intent was that a user would run the host process on their own workstation, with the node processes running on a cluster or supercomputer. However, since PVM's multicast operation was actually a linear series of point-to-point sends, the overhead to broadcast each new instruction was considerable.

PVM CVL's permutation functions relied entirely on system buffering, with each processor sending exactly one PVM buffer to every other processor. This requires at least as much system buffer space as the sum of the messages being sent. An additional problem is that the interconnection network is not used efficiently: there is no traffic for most of the function whilst the individual nodes pack messages into buffers, then a sudden burst as every node tries to send buffers to every other node.

PVM ease of development: In terms of usability, PVM is a great improvement over both Paris and CMMD. Since it runs on workstations, development can be done on the researcher's own machine, eliminating the problems of competition for resources of a shared supercomputer. Further-

more, all the familiar debugging tools are available to a programmer, and can be used to step through a program in real time without worrying about wasting account resources.

Unfortunately, manufacturer's extensions to PVM to improve its performance (such as `pvm_fastsend` on the Cray T3D [9]), are not supported by other PVM implementations. A portable PVM version of CVL would therefore either have to sacrifice performance on such platforms, or require constant updating as new machine-specific PVM implementations appear.

PVM performance: There were sufficient performance problems with PVM CVL to cause its development to be abandoned, although a rewrite with the latest PVM may solve some of them:

- Since PVM provided no combining operations (support for reductions has since been been added), they were written using the standard binary-tree message-passing algorithm, where the number of message delays is logarithmic in the number of processors. The heavyweight PVM messages meant that each scan or reduction operation carried a large overhead in terms of execution time.
- Using PVM functions to pack elements into a system buffer before sending does not work well for CVL permutations, since we do not know in advance if successive elements in a vector are being sent to the same processor. PVM packs therefore tended to be made on single elements, requiring an extra function call for every element transferred. A possible solution to this would be to implement additional user buffering and use PVM's new `pvm_psend` function to pack and send a buffer in a single operation.
- Even in the latest version, PVM's support for asynchronous communication to overlap computation and communication and to reduce system buffering remains poor [17].

3.4 MPI CVL: portability and performance?

Development of MPI CVL was started in the hopes that it would combine the advantages of PVM (portability, ease of development), with performance approaching that of vendor communication libraries. The design incorporates lessons learned from the CM-5 and PVM implementations. In particular, data distribution and pointer representation are identical to those of PVM CVL, the segment descriptor format is similar to that of CM-5 CVL, and most of the function implementations were based on those for the CM-5.

Since broadcast operations will probably be relatively expensive on many of the machines on which MPI CVL will run, the hostless model was chosen. This limits the capabilities of programs using MPI CVL on machines without a full Unix-style operating system on each node, and in particular means that the VCODE interpreter cannot spawn subprocesses on such machines. This was accepted as the price to be paid for a portable library that achieves reasonable performance.

Permutation functions are implemented using MPI's nonblocking asynchronous sends and receives, with buffering in user space to aggregate messages being sent to the same processor. The asynchronous functionality enables communication and computation to be overlapped on machines that support the offloading of communication responsibilities from the main processor. Thus, every node has four buffers for every other node; one being sent, one being filled, one being received, and one being unpacked. A global summation operation using MPI's built-in reduction function is currently used to decide when all messages have been sent and received.

MPI ease of development: MPI inherits all of PVM's ease-of-use advantages that stem from the ability to develop code on a local machine. All MPI CVL development was done using ANL/MSU MPI [12] on a workstation, with final ports to the SP-1, Paragon and CM-5 taking less than a day each (the bulk of the time taken for each port was spent iteratively refining the code to pass the idiosyncrasies of each machine's C compiler). This compares with the several months of effort taken to write the machine-specific CM-2 and CM-5 CVL implementations.

In terms of features, MPI's support for nonblocking asynchronous sends and receives is very welcome, as is the provision of scans and reductions and their extensibility with user-defined combining functions. However, the definition of MPI's scans as inclusive rather than exclusive is annoying, necessitating extra communication to generate exclusive scans for operators with no inverse (for example, a maximum-scan). Direct MPI support for segmented scans would also improve MPI CVL's performance; they are currently implemented with user-defined combining functions that access state variables describing the vector's segmentation.

MPI performance: Full performance figures are given in Section 4. Currently, performance tuning for a particular MPI/machine combination is limited to choosing a size for the message buffers. A large buffer typically allows greater bandwidth, but consumes memory that could otherwise be used for user data. The right tradeoff depends on the amount of memory on individual nodes, the number of nodes in

the machine, and the shape of the message-size/bandwidth curve. Results for 16 nodes of a Paragon are shown in Figure 2, with 4 kbytes being chosen as a suitable buffer size.

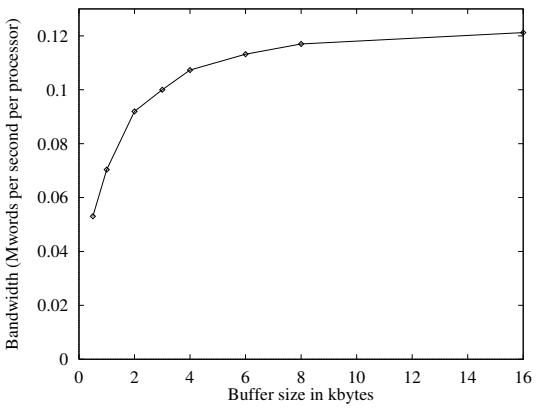

Figure 2: Effect of buffer size on asymptotic bandwidth for a random permutation in Paragon MPI CVL (16 processors).

Since the buffer space required per node is proportional to the number of processors, this scheme is not scalable to thousands of processors. However, it is reasonable given the machine sizes and node memories in normal use today. For example, when using 64 nodes of a Paragon the buffers account for 1 Mbyte on each node, out of a total memory of 16-64 Mbytes per node.

The buffer size for SP-1 MPI CVL also defaults to 4 kbytes; initial experiences with an SP-2 switch suggests that the newer machine could profit from larger buffers. The buffer size for CM-5 MPI CVL has been limited to 1 kbyte for the initial release of ANL/MSU CM-5 MPI, since asynchronous transmission and reception of larger buffers is unreliable when many messages are sent.

4 Results

In this section we compare the scalability and efficiency of MPI CVL running on a TMC CM-5, an Intel Paragon, and an IBM SP-1, with the machine-specific CM-5 CVL implementation (lack of time precluded full communication benchmarking on the SP-1). We also provide brief comparisons with the performance of the CM-2 and C90 CVL implementations (the latter was implemented in Cray assembler). The ANL/MSU portable MPI implementation of July 22 was used for the MPI benchmarks. This is implemented on top of an abstract device interface, easing the task of porting it to new machines, but adding some overhead since it is currently layered on top of the manufacturer's own message system. The communication performance of the ANL/MSU MPI implementation on all these platforms is expected to improve.

4.1 Scalability of MPI CVL

First, we consider the question of how well MPI CVL scales as the number of processors is increased. Scalability of elementwise CVL functions is clearly perfect, given a blocked data distribution. Assuming a binary tree algorithm is used by an MPI implementation for the collective communication operations, their fixed overhead scales with the logarithm of the number of processors, whilst the per-element cost remains perfectly scalable. The only remaining CVL functions that might not scale are the permutations. An example to show scalability of a CVL permutation function on the machines tested is shown in Figure 3. This plots the number of processors against the asymptotic out-of-cache bandwidth of the CVL default permute function performing a random permutation on an unsegmented vector of 64-bit words.

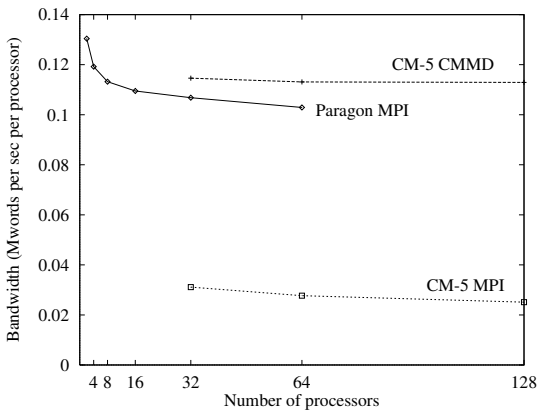

Figure 3: Scalability of CVL's asymptotic communication bandwidth for random permutation (horizontal = perfect).

It can be seen that CM-5 MPI CVL is currently well below the bandwidth achieved by CM-5 CMMD CVL, due partly to the small usable buffer size mentioned in Section 3.4. Also, note that we are not even approaching the bandwidth limits of the underlying hardware, due partly to the extensive memory traffic (fetching elements and indices, possibly writing them to buffers before sending, and then unpacking elements from buffers on arrival), and partly to the extra computation necessary to calculate which processor indices map to. Thus, this graph should not be taken as an indication of the scalability of the underlying hardware platform or vendor message system.

Table 1 compares the per-node communication performance shown in Figure 3 with the per-node out-of-cache

performance of CVL's 64-bit floating-point addition on the different machines, and adds results for SP-1 MPI CVL, CM-2 CVL and C90 CVL. CM-2 figures are per floating point processor.

CVL Implementation	MFLOPS	Mwords/s	Ratio
SP-1 MPI	7.9	0.28	28
Paragon MPI	2.6	0.11	24
CM-5 MPI	1.0	0.03	33
CM-5 CMMD	1.0	0.11	9
CM-2 Paris	0.34	0.015	22
C90 CAL	220	104	2.1

Table 1: Asymptotic out-of-cache floating-point and communication performance per processor.

There are several interesting aspects to this table. First, the different platforms running MPI have similar ratios of communication-to-computation performance. A low ratio is desirable for communication-intensive applications such as NESL. Second, given identical elementwise performance, the ratio of CM-5 MPI CVL is much worse than that for the CM-5 CMMD CVL. Finally, none of the MPP platforms come close to the ratio achieved on one processor of the C90. This suggests that there is a long way to go before the dominance of fast vector machines can be seriously challenged.

4.2 CVL instruction overhead

Apart from asymptotic performance, we must also consider the fixed overhead of CVL instructions when comparing CVL implementations. This affects the vector length $n_{1/2}$ at which half of the peak performance is achieved. In MPI CVL, the most significant sources of this are the fixed overhead of MPI's scan and reduction routines. These are used in CVL's scan and reduction functions, and in permutation functions to compare message counts and determine when all messages have arrived. There is also additional overhead involved in sending and receiving messages. The overhead of an integer plus-reduce (summation) function for each platform is given in Table 2. The number of CVL out-of-cache 64-bit floating-point additions this represents is also given, to compensate for different clock rates.

The vast difference between CM-5 MPI and CM-5 CMMD can be explained by the fact that there is currently no support in ANL/MSU MPI for using machine-specific collective operations. Thus, the CM-5's control network is ignored, and the reduction is performed using point-to-point messages instead.

Since a plus-reduce operation is always performed at least once by every permutation function, we would expect these overheads, when combined with the increased overhead due to MPI's layering of communication functions on the underlying vendor system, to result in very high $n_{1/2}$ values for the permutation operations. That is, the asymptotic communication speeds shown in Table 1 should only be achievable with a high number of elements per processor. Figure 4 shows the bandwidth achieved at various ratios for two of the three MPI platforms, as well as that achieved by CM-5 CMMD CVL. Again, a random permutation of 64-bit words is used. Note that the horizontal axis has a log scale, in order to more clearly show all the points of interest.

CVL Implementation	Overhead (uS)	FLOPS
SP-1 MPI	520	4100
Paragon MPI	700	1820
CM-5 MPI	2000	1940
CM-5 CMMD	75	70

Table 2: Overhead of CVL plus-reduce function and the number of CVL out-of-cache 64-bit floating point additions this represents (32 processors).

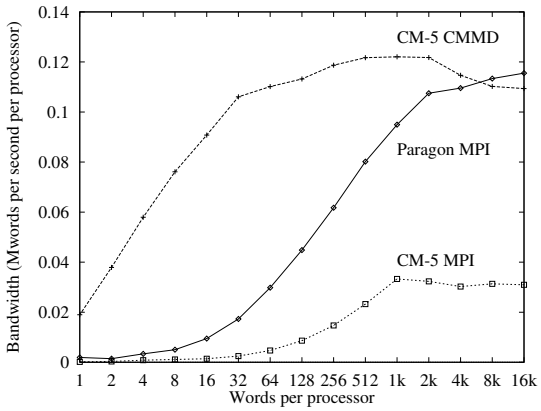

Figure 4: Effect of number of 64-bit words sent on CVL bandwidth for random permutation (32 processors).

As can be seen, CM-5 CMMD CVL achieves 50% of its peak bandwidth when there are only 4 or 5 words on each processor. By comparison, the CM-5 and Paragon MPI CVL implementations do not reach this figure until there are approximately 250 words on each processor.

4.3 Comparison of NESL performance

the relative performance of MPI CVL in real use is approximately what would be expected based on the benchmarks of its component functions. To illustrate this, we show the performance of Paragon MPI CVL, CM-5 MPI CVL, and CM-5 CMMD CVL running two NESL bench-

marks (the code for the benchmarks is given in the Appendix).

The first benchmark fits a line to a vector of coordinates, using the algorithm described in Press *et al* [16, section 14.2]. The only communication that takes place is five plus-reductions. Thus, this is an "embarrassingly parallel" benchmark. The results are shown in Figure 5.

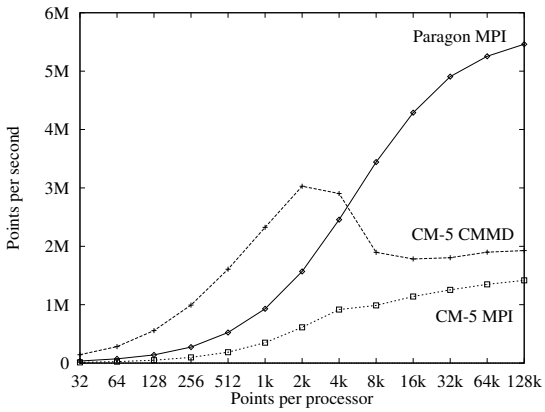

Figure 5: System performance on NESL linefitting benchmark (32 processors).

Note the high vector half-lengths for the MPI implementations, due to the overhead of the plus-reductions. Also, we can see the performance of CM-5 CMMD reaching a peak as node caches fill, and then falling off as the vector length increases.

The second benchmark finds the median element in a vector of keys, using the recursive quickselect method [14]. This algorithm partitions the data and then calls itself recursively on the partition containing the result, requiring dynamic memory allocation and data redistribution to achieve load balancing. The benchmark therefore tests the communication performance of the underlying CVL implementation. The results are shown in Figure 6. Here MPI's high communication overhead is even more apparent, with very high resultant values of $n_{1/2}$.

5 Summary and conclusions

MPI CVL has fulfilled its promise of portability and ease of development. However, it does not yet come close enough to the performance of machine-specific CVL implementations (such as that written in CMMD for the CM-5) to allow the Scandal project to abandon support for those implementations. We separate our comments on MPI into those specific to a particular implementation, and those aimed at the standard as a whole.

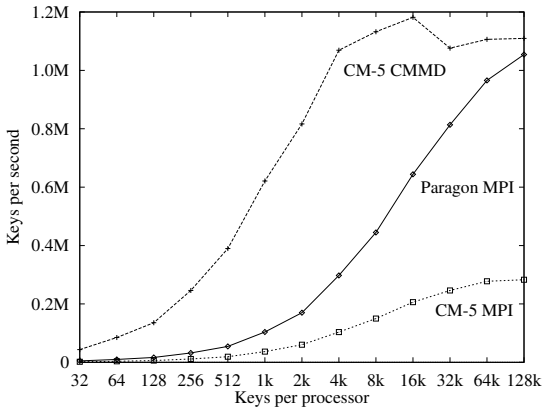

Figure 6: System performance on NESL median benchmark (32 processors).

To achieve high peak performance as well as low $n_{1/2}$ vector lengths for MPI CVL, we would want the following from an ideal MPI implementation:

- Full support for the capabilities of the underlying machine (for example, mapping MPI's scan and reduce primitives onto the CM-5's control network, and supporting asynchronous communication on the Paragon via its message coprocessor).
- Bandwidth approaching the maximum achievable on the machine (the ANL/MSU implementation promises this, although it has not yet been achieved for the Paragon or CM-5).
- Low fixed overhead for MPI communication functions.

In terms of future revisions to the MPI standard, our main request is support for exclusive scans, as explained in Section 3.4. Segmented scans and reductions would also be a benefit, and would bring MPI into line with the functionality provided by HPF [13].

It is hoped that a future version of this paper will be able to report results using a range of MPI implementations, and that some or all of our wishes will be granted.

5.1 Future work

As should be obvious from the performance figures in Section 4, CVL's elementwise and communication operations do not approach the peak performance of the current RISC-based MPPs on which it runs. Communication optimizations for VCODE have been proposed [18] which will reduce the number of calls made to CVL permutation

functions. However, the poor performance of CVL on elementwise operations is due to a fundamental mismatch between the structure of the library and current ratio between processor speeds and main memory bandwidth. Since elementwise CVL operations are essentially single loops over the data, they get no benefit from the cache for large problem sizes, and become limited by main-memory bandwidth. This problem becomes progressively worse as RISC CPU speeds continue to outrun DRAM bandwidth [1].

One obvious solution is to abandon CVL and the VCODE interpreter, and instead use a compiler that can perform loop fusion and other optimizations [8]. This can be combined with the communication optimizations mentioned above, and with new models for the control and partitioning of nested data parallel programs. We are now actively working on a system that will achieve this. To achieve portability, and to reduce our dependence on any one machine or manufacturer, it will use MPI as the communications substrate.

Acknowledgements

NESL, VCODE and CVL are creations of the Scandal project, and I'd like to thank all past and present members of the project for their advice and guidance. In particular, Guy Blelloch's NESL compiler and Jay Sipelstein's VCODE interpreter helped to uncover many bugs, and Marco Zagha provided the initial design of CM-2 CVL and a performance target to aim at in his C90 CVL implementation. I would also like to thank William Gropp and Rusty Lusk for their rapid and helpful responses to MPI bug reports and questions.

CVL implementations and Scandal papers are available on the WWW, at `http://www.cs.cmu.edu:8001/Web/Groups/scandal/home.html`, and via FTP from `nesl.scandal.cs.cmu.edu`.

References

[1] David H. Bailey. RISC microprocessors and scientific computing. In *Proceedings of Supercomputing '93*, pages 645–654, November 1993.

[2] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[3] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[4] Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[5] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings of Frontiers of Massively Parallel Computation*, pages 471–480, October 1990.

[6] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, February 1993.

[7] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[8] Siddhartha Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, October 1991.

[9] Cray Research, Inc. *PVM and HeNCE Programmer's Manual*, May 1993. SR-2501 2.0.

[10] Rickard E. Faith, Doug L. Hoffman, and David G. Stahl. UnCvl: The University of North Carolina C vector library. Version 1.1, May 1993.

[11] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3.0 User's Guide and Reference Manual*, February 1993.

[12] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Technical Report PREPRINT MCS-P342-1193, Argonne National Laboratory, April 1994.

[13] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

[14] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4(7):321–322, 1961.

[15] Peter Mills, Lars Nyland, Jan Prins, John Reif, and Robert Wagner. Prototyping parallel and distributed programs in Proteus. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 10–19, Dallas, Texas, December 1991. IEEE.

[16] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes*. Cambridge University Press, Cambridge, 1986.

[17] William Saphir. A comparison of communication libraries: NX, CMMD, MPI, PVM. NASA Ames User Seminar, `http://lovelace.nas.nasa.gov/Parallel/People/wcs/talks/message_passing_comparison.ps`, November 1993.

[18] Jay Sipelstein. *Data Representation Optimizations for Collection-Oriented Languages*. PhD thesis, School of Computer Science, Carnegie Mellon University. To appear.

[19] Thinking Machines Corporation, Cambridge, MA. *Paris Reference Manual*, February 1991. Version 6.0.

[20] Thinking Machines Corporation, Cambridge, MA. *CDPEAC: Using GCC to program in DPEAC*, December 1992.

[21] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, December 1992. Version 2.0.

[22] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual*, May 1993. Version 3.0.

[23] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

A Appendix

A.1 NESL linefit benchmark

This is the NESL code for fitting a line using a least-square fit. The function takes sequences of `x` and `y` coordinates and returns the intercept (`a`) and slope (`b` and their respective probable uncertainties (`siga` and `sigb`).

```
function linefit(x, y) =
let
  n    = float(#x);
  xa   = sum(x)/n;
  ya   = sum(y)/n;
  Stt  = sum({(x - xa)^2: x});
  b    = sum({(x - xa)*y: x; y})/Stt;
  a    = ya - xa*b;
  chi2 = sum({(y-a-b*x)^2: x; y});
  siga = sqrt((1.0/n + xa^2/Stt)*chi2/n);
  sigb = sqrt((1.0/Stt)*chi2/n)
in
  (a, b, siga, sigb);
```

A.2 NESL Median benchmark

This is the NESL code for median finding. The function `select_kth` uses the quickselect algorithm to return the `k`th smallest element of `s`. This is used by `median` to find the median element.

```
function select_kth(s, k) =
let pivot = s[#s/2];
    les = {e in s | e < pivot}
in
  if (k < #les) then
    select_kth(les, k)
  else
    let grt = {e in s | e > pivot}
    in if (k >= #s - #grt) then
         select_kth(grt, k - (#s - #grt))
       else pivot;

function median(s) = select_kth(s, #s/2);
```