Integrating Message-Passing with Vector Architectures

Celso L. Mendes*

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
E-mail: mendes@cs.uiuc.edu

Abstract

Vector architectures provide excellent computational throughput, while successfully tolerating memory latency by pipelining memory accesses. In this paper, we propose a generalisation of vector architectures to message-passing multicomputers, which combines the efficiency of vector computation with the scalablity of distributed-memory systems.

In our proposed architecture, each node is a conventional vector processor (with chaining capability and pipelined functional units) augmented by native instructions to send and receive messages through vector registers. In this scheme, inter-node communication can be performed via vector-send/receive instructions, gaining the benefits of communication pipelining, reduced memory copies (memory-to-register-to-register instead of memory-to-memory-to-cache), and lower communication latency (due to tight processor-communication coupling). We show that this strong integration between functional and communication units can lead to substantial performance improvement over conventional message-passing multicomputers.

We model pipelined computation-communication systems both analytically and with a detailed instruction-level simulation, and compare this simulation data with empirical results from an Intel Paragon. Preliminary data from a matrix multiplication example indicates our proposed vector-parallel architecture offers significant scalability benefits over existing message-passing systems.

1 Introduction

High performance has been the major motivation for parallel processing. Vector architectures provide excellent computation thoughput and have been a natural choice for sicentific applications in the past. However, these architectures are optimized for single node performance, and thus do not scale appropriately with an increasing number of processors; their overall performance is constrained to within a limited range, determined by the individual processor speeds.

On the other hand, multicomputers, consisting of a large collection of autonomously processing nodes that communicate by passing messages across a high-speed interconnection network, have demonstrated their potential to achieve the highest levels of performance

*Supported by CNPq/Brasil, process 280005/94-6(NV).

among current machines. Nevertheless, their widespread adoption has been hindered by some factors. One of these is their performance variability. As problem sizes or system configurations grow, some applications yield only a small fraction of peak system performance, whereas others approach the system's theoretical peak. Such relatively lower performance is often associated to a poor matching between the application's communication model and the machine's interconnection mechanisms. It is generally accepted that the performance of multicomputers is strongly dependent on their communication infrastructure.

We propose a new parallel architecture, which combines the high computation performance of vector architectures with the scalability of multicomputers, and overcomes the communication bottleneck by strongly integrating the computation and communication mechanisms. This architecture implements message-passing as native processor operations, and changes the memory-to-memory communication paradigm of current systems into a new form, with data flowing directly between the vector registers of communicating processors. These communication operations can be fully chained to regular computation instructions, in a pipelined fashion. Our goal is to show that this integrated architecture provides much better scalability than conventional systems for a large variety of scientific applications.

To implement our proposed architecture, we need to extend a vector processor with certain resources in its datapath and control unit, and also add an appropriate interconnection network, with corresponding interface modules. We will limit our discussion to the required datapath additions. Because the communication bottleneck of current multicomputers is usually caused by the network acess mechanisms inside the node, *not* by network contention, we assume a simple network model, without contention, where message transfer time follows a cost model that is a linear function of message length.

We model the behavior of our system analytically, and validate our model with a detailed instruction-level simulation. The simulation infrastructure also enables us to compare the behavior of our architecture with currently existing systems. Our preliminary results show that the proposed architecture can achieve both better single processor performance (due to vector processing) and, most importantly, better scalability (due to lower communication overhead) than a conventional multicomputer.

The rest of this paper is organized as follows. In §2 we analyze the behavior of current message-passing systems, observing some of their problems. Then we present our proposed design in §3, describe our simulation environment in §4, and show performance results from a matrix multiplication example in §5. We review related work in §6, and conclude in §7, pointing to our next planned steps in this ongoing study.

2 Motivation

Our previous research on performance prediction on multicomputers [9] showed that increasing the number of processors in the system may often produce diminishing performance returns. This is due to a combination of several factors. First, there is Amdhal's law, which states that the serial component of a program will become a bottleneck when more processors are used. Also, with more processors, the new balance between communication and computation may become poor, because of the communication structure in the underlying machine. We now explicitly show how this second factor can degrade performance on current systems. Next, we suggest an alternative approach, using the concept of vector based message-passing, from which we will derive our proposed architecture.

Number of Segments	Time (msec)	
	iPSC/860	Paragon
1	470.2	102.9
2	422.9	97.9
4	399.9	96.2
8	389.5	95.6
16	386.8	95.8
32	390.2	97.2
64	401.2	100.0

Table 1: Execution times for optimized *receive/daxpy* fragment.

2.1 Problems with Current Systems

We present an example with a very simple code fragment that illustrates some of the problems with message-passing on current systems. In this code fragment, a processor waits for a message and then uses the incoming data to compute a *daxpy* operation, typical of many scientific applications. The major part of the code for the receiving processor is as follows:

```
double  y[N], x[N], a ;
receive(msg_type,x,N*sizeof(double)) ;
for (i=0; i<N; i++) y[i] += a * x[i] ;
```

A common optimization to this code consists of decomposing the original message and loop sections into a programmable number of smaller segments, so that computation on one segment overlaps with communication of data for the next segment. This is possible with the use of non-blocking message-passing calls, and the insertion of appropriate synchronization between segments. Table 1 shows the resulting execution times for the optimized fragment on an Intel iPSC/860 and on an Intel Paragon, with a data set size of $N=2^{17}$ and a varying number of segments. The first row in that table (one segment case) corresponds to the non-optimized message-passing scheme. On the iPSC/860, using sixteen segments improves performance by as much as 18 percent, while on the Paragon the best improvement is on the order of 7 percent, for eight segments.

In general, we can determine an optimal number of segments by using a model that represents the communication and computation times for a data set size s as follows

$$\begin{cases} T_{commun}(s) = a + bs \\ T_{comp}(s) = cs \end{cases}$$

The cost for execution of the optimized version of our fragment with K segments is

$$T_{fragment} = T_{commun}\left(\frac{N}{K}\right) + (K-1)max\left\{T_{commun}\left(\frac{N}{K}\right), T_{comp}\left(\frac{N}{K}\right)\right\} + T_{comp}\left(\frac{N}{K}\right)$$

and assuming that the communication time is greater than the computation time, it becomes

$$T_{fragment} = \left(a + b\frac{N}{K}\right) + (K-1)\left(a + b\frac{N}{K}\right) + c\frac{N}{K} = Ka + bN + \frac{cN}{K}.$$

System	a (msec)	b (msec/byte)	c (msec/byte)
iPSC/860	0.4	3.6×10^{-4}	9.1×10^{-5}
Paragon	0.4	2.0×10^{-5}	7.8×10^{-5}

Table 2: Parameters for computation and communication models.

The optimal value for K is the one such that $T_{fragment}$ is minimum, or

$$\frac{dT_{fragment}}{dK} = 0 \Rightarrow a - \frac{cN}{K_{opt}^2} = 0 \Rightarrow K_{opt} = \sqrt{\frac{cN}{a}}$$

By executing the non-optimized *daxpy* fragment with the same data set size of $N=2^{17}$, we observed the following values:

iPSC/860: $T_{commun}(N) = 374.8$ msec, $T_{comp}(N) = 95.4$ msec
Paragon: $T_{commun}(N) = 21.4$ msec, $T_{comp}(N) = 81.5$ msec

We can obtain the value of c directly from $T_{comp}(N)$. To determine a and b, however, we repeated the non-optimized execution with a different data set size ($N'=2^{16}$), obtaining $T_{commun}(N')=187.6$ msec on the iPSC/860 and $T_{commun}(N')=10.9$ msec on the Paragon. The resulting parameters are in Table 2. Taking these values into our iPSC/860 model yields

$$K_{opt} = \sqrt{\frac{cN}{a}} = 15.44$$

and thus $K=16$ is the closest option, confirming our measurements on the iPSC/860.

When the computation time is greater than the communication time, as in the Paragon, there is less potential gain from the optimized approach. To understand why it is so, we consider the case of the optimized *daxpy* fragment on the Paragon with four segments. Table 3 shows the detailed timing of the receiving processor across each phase of the execution (the synchronization time is accounted as communication). The duration of the last computation segment, which does not overlap with communication, is about one forth the duration of the computation interval $T_{comp}(N)$ in the non-optimized execution, as expected. However, the first three computation segments are extended, simply because there is simultaneous communication (reception of the incoming data for the next segment), and so there exists contention for the local memory. [1] That causes an overhead of up to 17 percent in the computation intervals. Such overhead hinders the gains obtained from the overlapping between computation and communication, and is the reason for the relatively lower improvement achieved on the Paragon with the optimized *daxpy* example.

Our experimental results from Table 1 reveal performance degradation when we increase the number of segments above a certain threshold. This is due, in part, to the fixed latency cost that is present on each communication operation; when we use more than a given number

[1] To verify this in practice, we conducted additional tests, varying the length of the incoming message, and observed that the amount of perturbation was proportional to the message length

Segment Number	Time (msec)	
	Communication	Computation
1	5.4	23.6
2	0.08	23.5
3	0.08	23.3
4	0.03	20.2

Table 3: Timing of optimized *daxpy* execution with four segments on the Paragon.

of segments, the total cost of the latencies for all segments becomes so high that it offsets the gains achieved by the overlapping between computation and communication. Thus, we can conclude that any reduction in latency will bring an improvement in performance for the optimized approach.

Another point to mention is the influence of memory-based communication. We showed that the use of memory buffers for holding message data degraded performance in this example. Computation times were extended because there was contention for the local memory. The model of communication based on memory buffers may be convenient for the sending processor, but it is usually not necessary in the receiving side: the incoming message is often consumed by ongoing computation, and does not need to be stored in the receiver's memory after that. This transient traffic of incoming data through the local memory of the receiver can be potentially damaging to its performance, as demonstrated by the results from Table 3.

2.2 Advantages of a Vector Approach

Scientific applications have been effectively using vector architectures for quite a long time. Such architectures have proven very convenient to manipulate the regular data structures, like arrays or matrices, that are common in numeric algorithms. Also, compilation techniques for vector machines are now at a reasonably mature stage, and compilers can often create efficient vector code for a given application/system pair.

Most vector architectures provide some form of *chaining*, where the intermediary results from a functional unit are immediately used by another functional unit executing the next vector instruction. This feature of synchronized, simultaneous operations, implicit in vector architectures, is a key factor in allowing us to overcome the communication bottleneck in a parallel machine. If we consider that communication and computation are handled by separate "functional units," then we can also chain those two tasks: data arriving in a message can be immediately consumed by computation, in a pipelined fashion.

In this scenario, it seems natural to consider communication simply as another vector operation; it is a special type of operation in the sense that it is not executed by any functional unit inside the processor, but in the network that interconnects the processing nodes. Assuming a vector processor on each node, message-passing can occur between vector registers of the communicating processors. Like regular vector operations, communication operations can stall on data or functional hazards. A vector-receive, for example, could stall until the corresponding message arrives at the node, while a vector-add followed by a chained vector-send would cause the send to stall until the first operand of the addition is produced.

This scheme can bring several benefits, as compared to current message-passing multicomputers. First of all, message latency is clearly lower, because message operations become native instructions of the processor, and it is no longer necessary to invoke the operating sys-

tem or to spend extra processor cycles managing the network interface. Most importantly, message data would reside inside the processor, *not* in memory. That might reduce the time to access such data, and make this access time more deterministic as well. With the current model of memory-based messages, access time is strongly dependent on factors like memory speed, memory bus traffic, cache organization, etc.

3 Parallel Vector Architecture

Our proposed architecture consists of a collection of nodes, with each node comprising a vector processor and some number of local memory banks. The node also has an interface to an interconnection network, that it uses to exchange messages with other nodes. In addition to the usual vector operations that are present on most vector processors, we also have vector instructions for handling communication. This is, in fact, one of the unique features of our design: a high degree of integration between the communication and computation structures.

3.1 Node Architecture

It is a well known fact that vector processors must contain fast scalar functional units, so that the serial program components do not impose a severe bottleneck during execution. For this reason, we chose a RISC architecture as the basic building block in our design. We selected the DLX architecture, as presented in the book by Hennessy and Patterson [5], and extended it to a vector architecture, DLXV, following the suggestions in that same book.

DLXV is a vector processor that has both scalar and vector functional units. The non-pipelined scalar units were already present in DLX, whereas the vector functional units, exclusive of DLXV, are fully pipelined and allow chaining between vector operations.

In addition to the same register set of DLX (general-purpose registers R0-R31 and floating-point registers F0-F31), DLXV has a vector register file composed of a group of vector registers. Each vector register has sixty-four 64-bit elements. There are also two special registers, VLR (Vector-Length Register) and VMR (Vector-Mask Register). The contents of VLR may vary between 0 and 64, defining the length of any vector operation; VMR is a 64-bit register, which can be used to disable operations on particular elements of a vector (by storing the value 0 in the corresponding bit of VMR). There are 64-bit pipelined buses between the vector register file and memory, in both directions. Each vector register has one write port and a configurable number of read ports, so that more than one vector functional unit may receive data from the same vector register simultaneously, using the independently addressable read ports. We assume that each vector functional unit can be connected to any vector register, by means of crossbar switches [7].

3.2 Message-Passing Structure

In our architecture, nodes exchange data by message-passing. Message data originate in a vector register of the sending processor, and eventually reach a vector register of the receiving processor. The length of a message is defined by the value in VLR of the sending processor at the moment that the message-passing instruction is issued. Thus, the maximum message length is given by the maximum vector length, which is the size of a vector register. Every message is tagged by an integer number representing the message's *type*. We added to DLXV three instructions supporting message-passing:

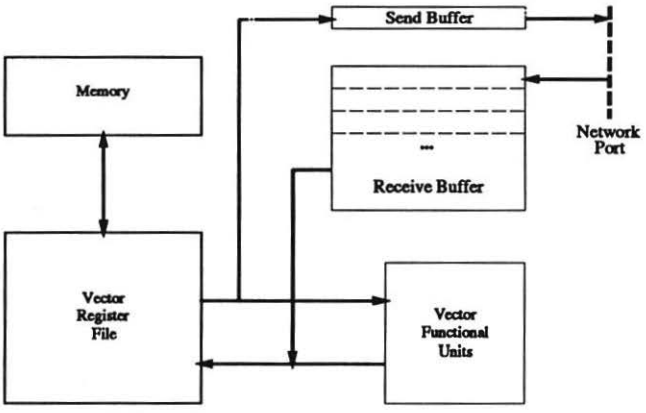


Figure 1: Organization of the *send* and *receive* buffers in the network interface.

- *sendv* R_a, R_b, V_c: Send a message with type given by the contents of register R_b to the destination node indicated by register R_a; message data will come from vector register V_c;
- *recvv* V_d, R_e: Receive into vector register V_d a message of type given by register R_e;
- *smsg* $R_f, R_g, value$: Set register R_f if a message with type given by $(R_g) + value$ has arrived in the node and is ready to be processed by a *recvv* instruction.

Our network interface design includes a *send buffer* and a *receive buffer*. The send buffer can store one message, and its main goal is to prevent the sending processor from stalling when data is temporarily blocked from flowing into the network. A vector-send operation normally causes data in the underlying vector register to be transferred to the send buffer, and from there to the network. If the send buffer is full, the vector-send stalls the processor.

The receive buffer can contain a given number of incoming messages. Its main function is to store messages that arrive at the node before the corresponding *recvv* instruction has been issued by the local processor. When the *recvv* instruction is issued, and the message has already arrived in the node, the message contents are transferred from the receive buffer to the designated vector register. If the message has not yet arrived, the *recvv* will stall the processor. Thus, both *sendv* and *recvv* are blocking operations. Figure 1 shows the send and receive buffers with their connection to the vector register file. In practical terms, these buffers simply work as a "communication functional unit." When no stalls occur, the *sendv* and *recvv* operations can be chained to regular vector instructions.

3.3 Flow Control

We implement flow control for data exchange between nodes by using two special types of system-level control messages, named *probe* and *acknowledgement* messages. When a node issues a vector-send instruction, and the send buffer is free, besides starting to transfer the data to the send-buffer, the node also sends a *probe* message to the destination, to check if there is space for the data in the remote receive buffer. Upon receiving such probe, the network interface in the destination will try to allocate a free buffer entry in its receive buffer

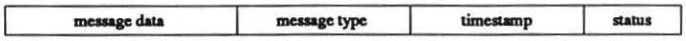


Figure 2: Structure of an entry in the receive buffer.

and, upon doing so, send back to the source node an *acknowledgement* message. When the acknowledgement returns, the source node starts sending the data in its send buffer.

We assume that control messages have higher priority than regular data messages. With this protocol, data messages are not transmitted if we cannot ensure that the destination has space for them. The *sendv* instruction stalls if the send buffer is already full, while the *recvv* instruction stalls when no message with the given type is available in the receive buffer. The *smsg* can be used to check for message arrival, avoiding the blocking caused by a *recvv* stall.

Because the receive buffer may contain several messages at a given moment, when the *recvv* instruction is issued we need to do type matching between the type designated in the *recvv* and the types of messages in the buffer. Among those entries with the appropriate type, ideally we would select the oldest message. For this reason, each entry in the receive buffer is timestamped with the time of arrival. Figure 2 shows the various fields for each entry in the receive buffer. There is also a *status* field that indicates when the entry contains valid data, or when the entry is not yet valid but allocated for an expected message.

The specific number of entries in the receive buffer is a design parameter. As the number of entries increases, more messages can be accepted before a *recvv* instruction is executed, thus requiring looser coordination between sends and receives, but the hardware costs for type checking and timestamp comparison also increase. As stated in [4], any message-passing scheme has to assume that the user program is "well-behaved," to some extent, in its buffering requirements. The selection of the number of entries is a tradeoff between the coordination flexibility for the programs and the associated hardware costs.

4 Simulation Environment

In order to allow an evaluation of our design, we implemented a simulator of its datapath, using as a starting point the DLXsim simulator [6] available for the original DLX architecture. First, we extended DLXsim to simulate the uniprocessor vector architecture (DLXV), as we reported in [8]. This first extension implements all the DLXV instructions. It also reproduces possible conflicts for memory bank accesses.

We then added to our DLXV simulator the three message-passing instructions (*sendv*, *recvv*, and *smsg*), the structures corresponding to the send and receive buffers, and the rest of the infrastucture for communication. By replicating this enhanced simulator, so that each node was simulated by a different process, we achieved simulation of our complete vector architecture with integrated message-passing, and we named this final simulator as DLXVMPsim. Currently, DLXVMPsim is running in a distributed form on an Intel Paragon, where each physical node executes a copy of the enhanced DLXV simulator, and thus simulates one node of our proposed architecture. Although this scheme, in principle, would limit the number of processors in the virtual system to the number of nodes in the real machine where simulation is running, we can use our simulator on top of other simulation packages that provide an environment with more logical nodes than physically available [2].

As with any parallel simulation, we must enforce some form of synchronization between the several processes, so that the simulation accurately reproduces the behavior of the cor-

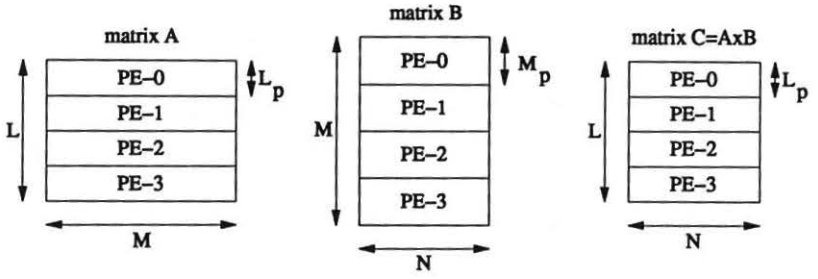


Figure 3: Data distribution for matrix multiplication example with four processors.

responding target system. We follow the same conservative synchronization mechanism used in the WWT project [11], where simulation on each processor is allowed to proceeed for a fixed number of cycles, known as *quantum*, in simulated time. The quantum must be less than or equal to the latency of the target system, so that all events originating on a remote node that can affect a node in a given quantum are known at the quantum beginning.

5 Application Example

We illustrate the use of our architecture with a matrix multiplication example. As Figure 3 shows, we use the same network topology independent algorithm from [3], changing only the data distribution to blocks of rows, instead of columns. Each processor computes a region of the product matrix using its local rows of matrix A and either local or remote rows from matrix B. Thus, data from matrix B is communicated among processors. The original algorithm is such that each message would contain exactly one row of matrix B. In our vector implementation, however, the messages contain segments of a row, with a segment length of 64, corresponding to the size of a vector register. Initially, we derive a model for the execution time in our architecture, to get a first estimate of the achievable performance in this particular case. Then we compare the expected behavior predicted by the model with simulation results obtained with different matrix sizes. Finally, we compare our results to the performance observed on a real system, an Intel Paragon.

5.1 Modeling of Expected Performance

The original matrix multiplication algorithm, executed by each node in an SPMD fashion, can be represented in a condensed form as shown in Figure 4. In this original scheme, the sending of a given row is executed nearly at the same time as the receive, and we only start the computation in the inner loop after the remote row arrives. There is a trivial optimization for this algorithm, which consists of sending the rows in advance, on a previous iteration of the outer loop. Thus, instead of sending row $k\%M_p$, we send row $(k+1)\%M_p$, and the message containing data for iteration $k+1$ is overlapped with computation of the inner loop on iteration k. In our vector architecture, we must process the whole body of the outer loop by segments of length 64, the size of our vector registers.

With the optimization described above, and defining $K_n = N/64$, we can express the operations in terms of vector instructions as indicated in Figure 5. We derive the expected

```
for k = all N rows of matrix B
{
     /* get access to a row of matrix B */
     if (row k not local)
     {
          send local row k%Np of matrix B
          receive remote row B[k,1:N]
     }
     /* update local rows of matrix C */
     for i = all Lp local rows of matrix C
     {
          C[i,1:N] += a(i,k) x B[k,1:N]
     }
}
```

Figure 4: Original matrix multiplication algorithm.

execution time (in number of cycles) for this optimized version as

$$T_{total} = MK_n(T_{load} + T_{inner})$$

where T_{load} is the time to load vector register $V0$, either with local or remote data from matrix B, and T_{inner} is the time of the computation loop.

Assuming that all the vector operations are chained, and that there is perfect overlap between computation and communication (which is true when the time for the inner loop is greater than the time for transmission of a message, so that the vector-receive does not stall), T_{load} will represent only the startup costs for the operations, and is given by

$$T_{load} \cong T_{vload-startup} + T_{send-startup} + T_{recv-startup}.$$

Assuming $T_{vload-startup} = 12$, $T_{send-startup} = 1$ and $T_{recv-startup} = 1$ (no stalls), we have

$$T_{load} = 14.$$

With chaining between all vector operations in the the inner loop, we estimate its execution time as

$$T_{inner} = T_{loop-setup} + T_{fu-startup} + 64L_p$$

where $T_{loop-setup}$ is the overhead to set up the loop, and $T_{fu-startup}$ is the startup cost for all the involved functional units, given by

$$T_{fu-startup} = T_{scalar-load} + max\{T_{vmult} - 1, T_{vload}\} + T_{vadd} + T_{vstore} = 2 + max\{7 - 1, 12\} + 6 + 1 = 21.$$

Assuming $T_{loop-setup} = 10$, we have

$$T_{inner} = 31 + 64L_p$$

and the total execution time becomes

$$T_{total} = MK_n(14 + 31 + 64L_p) = \frac{MN}{64}\left(45 + \frac{64L}{P}\right).$$

```
for k = all N rows of matrix B
{
     for s = all Kn segments
     {
          base = s * 64
          if (row k not local)
          {
               load-vector row B[(k+1)%Np,base:base+63] into V0
               send-vector V0
               receive-vector remote row B[k,base:base+63] into V0
          }
          else
          {
               load-vector row B[k,base:base+63] into V0
          }
          for i = all Lp local rows of matrix C
          {
               load-scalar value a(i,k) into F0
               vector-multiply scalar x vector: V1 <-- F0 x V0
               load-vector C[i,base:base+63] into V2
               add-vector: V3 <-- V1 + V2
               store-vector V3 into C[i,base:base+63]
          }
     }
}
```

Figure 5: Optimized matrix multiplication algorithm with vector operations.

Thus, the expression for the execution time of the optimized matrix multiplication program is

$$T_{total} = MN\left(0.7 + \frac{L}{P}\right) cycles$$

This expression shows that the problem has a cost complexity of $O(LMN/P)$, as expected when communication is not a bottleneck.

5.2 Simulation Experiments

We started our experiments by simulating a non-optimized version of this algorithm in our architecture, with matrices of size 64x64. We assume in our network model a message latency of 20 processor cycles, and a network bandwidth of one cycle per byte; these would correspond, on a system with a 50MHz clock, to a 400 ηsec latency and a 50 MB/sec bandwidth. Table 4, in its first column, shows the simulation results for a varying number of processors. Using data provided by the simulator, we can confirm the non-optimal behavior of this version of the program: Figure 6 shows some of the output information for a particular execution (four processors), indicating that a significant number of cycles were lost due to the wait for message arrival (Vrecv Stalls).

As a next step, we simulated the optimized version of the matrix multiplication algorithm, with two different matrix sizes, obtaining the results in the last two columns of Table 4. Figure 7 compares the observed simulation results and the expected values derived from our model, showing that the model indeed captures the behavior of the optimized program.

Finally, we executed the same optimized program on an Intel Paragon and in our simulator, with a matrix size of 256x256. On the Paragon, we implemented the *daxpy* function

Number of Processors	Execution Time (cycles)		
	Non-Opt.64x64	Optim.64x64	Optim.128x128
1	267533	268685	2119181
2	157725	141837	1087501
4	102808	78413	571661
8	75369	46701	313741
16	61643	37801	184781

Table 4: Simulation results for matrix multiplication program.

with a commercial library routine, to maximize performance, and used the nonblocking *isend* calls. Figure 8 shows the speedup obtained on the two architectures, where one can see that our system clearly achieves better scaling.

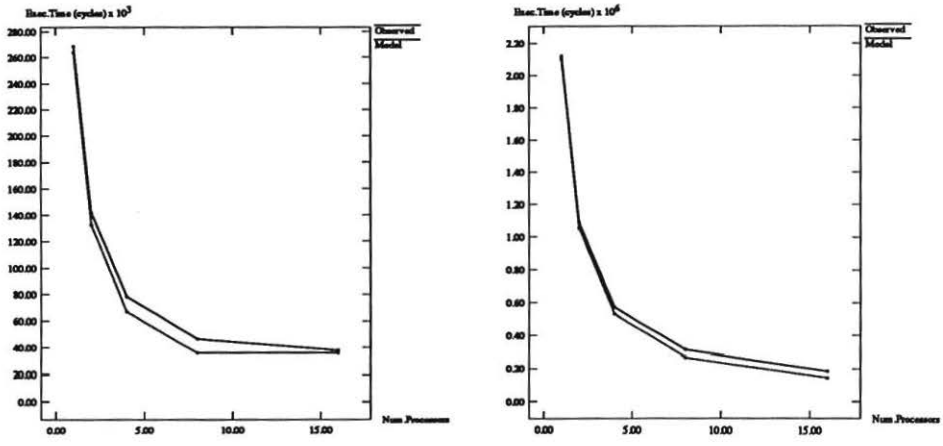
Part of the reason for the better performance on the vector architecture comes from the numbers on Table 5, showing the time, in cycles, for the uniprocessor execution on both systems. For a 256x256 matrix multiplication, there are 2×256^3 or 33.6×10^6 floating-point operations. The vector architecture, with independent multiply and add functional units, achieves nearly two results per cycle. The Paragon, however, despite having independent multiply and add units, takes five times longer, probably because of memory access delays for non-cached data.

For the parallel execution on the Paragon, as the number of processors increases, the operating system overhead involved in the message-passing calls becomes significant, as compared to the computation for each node. Hence there is severe performance degradation. On the vector architecture, however, there is no operating system overhead, and communication remains, for the most part, "hidden" inside the computation intervals. Even with sixteen processors, we observed no stalls due to communication. In this case, the deviation from ideal speedup is due simply to the fact that the startup costs for the vector operations are no longer negligible in comparison to the reduced execution time.

The superior single node performance (obtained with the pipelining of memory accesses), associated to the effective chaining between computation and communication (represented by the absence of communication stalls), allowed our vector architecture to achieve much better scalability than the Paragon.

```
Load Stalls = 5936
Floating Point Stalls = 2976
Vsend Stalls (Send-Buffer full): 0
Vrecv Stalls (msg not in Recv-Buffer): 26088
Total Vector Stalls = 38744
Total Stalls = 47656
Total integer operations = 50656
Total floating point operations = 240
Total trivial vector operations = 0
Total full vector operations = 4256
Vector elementwise operations = 272293
Total instructions = 55152
Total operations = 323189
Total cycles = 102808
```

Figure 6: Simulation results (PE-0) for non-optimized matrix multiplication, four processors.



(a) 64x64 matrices

(b) 128x128 matrices

Figure 7: Comparison between simulation and prediction results on matrix multiplcation.

6 Related Work

There have been recent debates on the effectiveness of vector and massively parallel architectures on scientific applications [10] [12]. Indeed, vector processing capability exists on some current parallel systems. On the CM-5 [13], there are separate scalar and vector processors on each node, whereas the Fujitsu VPP500 [14] uses a traditional vector processor with scalar and vector functional units. To our knowledge, however, none of the existing systems provides communication support as a native processor feature. They all implement interprocessor communication of vector operands by moving data across memories of the underlying nodes. The J-Machine [1] has processors with native communication instructions, but the system is not targeted at scientific applications; there is no floating-point hardware support.

Shlomo [15] compared vector and superscalar architectures, in terms of resource utilization for execution of a vectorizable code fragment. His main conclusion was that current superscalar architectures are inferior because of their limited memory bandwidth, and because of the implicit prefetching of memory data into vector registers occurring in vector load instructions of vector architectures. That study did not address the use of such architectures in parallel systems.

System	Time (cycles)
Paragon	84725000
Vector Archit.	16857101

Table 5: Times for uniprocessor execution on 256x256 matrix multiplication.

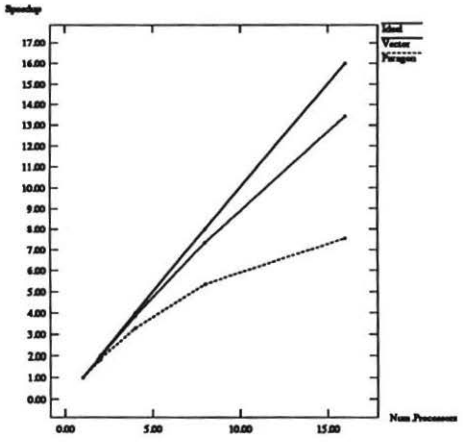


Figure 8: Speedup comparison on 256x256 matrix multiplication.

7 Conclusions and Future Work

We have shown, with concrete examples, some of the problems of the memory-to-memory communication paradigm used on existing multicomputers. Our approach, based on a tight integration of the computation and communication mechanisms of a vector architecture, avoids such problems by passing data directly between registers of the underlying processors, and by chaining such message-passing operations with regular computation in the processors. Our preliminary simulation results on a matrix multiplication example show that this approach can achieve better scalability than existing systems, and can be a viable alternative to conventional architectures in the cases where the communication behavior represents a performance bottleneck.

Our current research effort is focused on two major directions. The first is to extend the tests of our architecture with more applications, and conduct detailed measurements of resource utilization and degree of parallelism. We are particularly interested in observing the performance implications resulting from variations in the hardware costs, like the capacity of the receive buffer and the number of vector registers. Also, we will assess the degree of overlapping between computation and communication, represented by the number of communications stalls, for this larger application suite. The second goal is to contrast, in detail, our design to a superscalar based multicomputer, specially in terms of memory behavior under the presence of communication. We suspect that, on a superscalar based system, message-passing calls to the operating system introduce significant cache pollution, causing severe performance degradation on subsequent computation sections.

Acknowledgments

We would like to thank Prof. Daniel Reed (University of Illinois) and Dr. Brian Totty (Silicon Graphics Inc.) for their helpful suggestions and comments about this work.

References

[1] DALLY, W. J., FISKE, J. S., KEEN, J. S., LETHIN, R. A., NOAKES, M. D., NUTH, P. R., DAVISON, R. E., AND FYLER, G. A. The Message-Driven Processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro 12*, 2 (April 1992), 23–39.

[2] DICKENS, P. M., HEIDELBERGER, P., AND NICOL, D. M. *Parallel Direct Execution Simulation of Message-Passing Parallel Programs.* ICASE/NASA Langley Research Center, June 1994.

[3] EICKEN, T. V., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture* (Gold Coast, Australia, May 1992), pp. 256–266.

[4] FRANKE, H., HOCHSCHILD, P., PATTNAIK, P., PROST, J.-P., AND SNIR, M. *MPI on IBM SP1/SP2: Current Status and Future Directions.* IBM T. J. Watson Research Center, 1994.

[5] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[6] HOSTETLER, L. B., AND MIRTICH, B. *DLXsim — A Simulator for DLX.* University of California, 1990.

[7] LEE, C. G., AND SMITH, J. E. A study of partitioned vector register files. In *Proceedings of Supercomputing'92* (Minneapolis, November 1992), pp. 94–103.

[8] MENDES, C. L. Extending DLXsim for parallel architectures. In *Proceedings of the 6th Brazilian Symposium on Computer Architecture* (Caxambu/MG, August 1994).

[9] MENDES, C. L., AND REED, D. A. Performance stability and prediction. In *Proceedings of the IEEE/USP Workshop on High Performance Computing - WHPC'94* (São Paulo, March 1994), pp. 1–15.

[10] MONTRY, G. Panel: Massively parallel vs. parallel vector supercomputers: A user's perspective. In *Proceedings of Supercomputing'93* (Portland, November 1993), pp. 918–920.

[11] REINHARDT, S. K., HILL, M. D., LARUS, J. R., LEBECK, A. R., LEWIS, J. C., AND WOOD, D. A. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the ACM Conference on Measurement & Modeling of Computer Systems - SIGMETRICS'93* (Santa Clara, May 1993), pp. 48–60.

[12] TAKAMURA, M., AND UTSUMI, T. Why vector parallel? In *Proceedings of the High Performance Computing Conference'94* (Singapore, September 1994), pp. 394–398.

[13] THINKING MACHINES CORPORATION. *CM5 Technical Summary*, October 1991.

[14] UTSUMI, T., IKEDA, M., AND TAKAMURA, M. Architecture of the VPP500 parallel supercomputer. In *Proceedings of Supercomputing'94* (Washington, November 1994), pp. 478–487.

[15] WEISS, S. Optimizing a superscalar machine to run vector code. *IEEE Parallel & Distributed Technology 1*, 2 (May 1993), 73–83.