Fast Message Assembly Using Compact Address Relations

Peter A. Dinda David R. O'Hallaron
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
(pdinda,droh)@cs.cmu.edu

Abstract

Message assembly and disassembly represent a significant fraction of total communication time in many parallel systems. We introduce a run–time approach for fast message assembly and disassembly. The approach is based on generating addresses by decoding a precomputed and compactly stored address relation that describes the mapping of addresses on the source node to addresses on the destination node. The main result is that relations induced by redistributions of regular block–cyclic distributed arrays can be encoded in an extremely compact form that facilitates high throughput message assembly and disassembly. We measure the throughput of decoding–based message assembly and disassembly on several systems and find performance on par with copy throughput.

1 Introduction

Many parallel programs perform operations on distributed data structures by using a message passing system to transfer data items from noncontiguous addresses in the memory of a source node to noncontiguous addresses in the memory of a destination node. These programs typically amortize the fixed overhead of transferring a message by packing a collection of data items into a contiguous message buffer, which the message passing system then transfers over the network as a single message. When the message arrives at the destination, the message is copied from the network to a buffer, which the user program then unpacks to the appropriate memory locations. We refer to the process of packing the data as *message assembly* and to unpacking as *message disassembly*.

On machines with fast communication systems, the performance of the message assembly and disassembly steps can have a significant impact on the overall communication performance of a program. In such systems, the memory system tends to be the bottleneck rather than the communication system [25]. The range of these systems is increasing with the advent of high speed networks [21]. Further, even on machines with slow communication systems, overly slow message assembly and disassembly can lead to disappointing overall performance.

The message assembly and disassembly operations needed to implement distributed data structures such as distributed arrays are nontrivial and the running times of different algorithms can vary significantly. For example, the running time of the assembly step for a High Performance Fortran (HPF) [10] block–cyclic assignment statement can vary by several orders of magnitude, depending on the nesting of the assembly loops [9]. As another example, because of asymmetry in the Cray T3D memory system, a transpose operation implemented with contiguous reads and strided writes runs twice as fast as the equivalent operation implemented with strided reads and contiguous writes [25].

This paper describes a purely run–time, inspector/executor [18, 1] approach for message assembly/disassembly and provides an initial characterization of its performance on real systems. In our approach, when a data transfer is first encountered, the inspector precomputes

Supported in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152, in part by the National Science Foundation under Grant ASC-9318163, and in part by a grant from the Intel Corporation.

SIGMETRICS 96-5/96 Philadelphia, PA, USA

the mapping of source node addresses to destination node addresses and stores this *address relation* in some encoding. The executor actually performs the transfer by reading and decoding the stored address relation to generate the addresses needed to assemble the message on the source node and disassemble it on the destination node. The intuition is that reading and decoding the address relation is cheaper than recomputing it on the fly and that by repeatedly using a stored relation, we can amortize the initial cost of computing and encoding it.

Our innovation is to encode the address relation in new, compact ways that facilitate fast message assembly/disassembly executors — they are "fast" in the sense that data transfer rates to and from the message buffer approach the copy throughput of the memory system for the reference pattern. In previous work [6] and in the extended version of this paper [5], we derive and validate a model that predicts, for a simple index array encoding, that the regime where the inspector/executor approach is faster than recomputing address relations on the fly is quite large, and even extends to cases where computing a member of the relation requires only a few instructions. Here, we concentrate on the executor and develop encodings that exploit the attributes of relations induced by redistributions of HPF regular, block–cyclicly distributed arrays to achieve low storage costs and fast message assembly and disassembly.

We begin with an overview of address relations and approaches to generating them in Section 2. The next section, Section 3, characterizes the address relations induced by redistributions of block–cyclic arrays, points out the regularities these relations exhibit, and describes four representative redistributions. Section 4 introduces address relation encodings that can encode arbitrary relations, but exploit the regularities we describe. Next, in Section 5, we measure the performance of message assembly/disassembly routines based on the different compact encodings of address relations. The performance on each of the four representative redistributions of Section 2 is measured. Results are given for several common systems: DEC Alpha [8], Intel Paragon [12], IBM RS/6000 [16], IBM SP/2 [19], Gateway Pentium 90 [17], and Micron Pentium 133. The encouraging conclusion is that in most cases, assembly and disassembly routines based on a highly compact encoding can run at or near the copy throughput of the memory system for the memory reference pattern of the relation. Finally, Section 6 discusses other implications of decoding precomputed address relations as well as some open questions that still need to be addressed before the ultimate usefulness of our approach can be understood.

2 Generating address relations

The transfer of data items from the memory of a *source* node to the memory of a *destination* node can be characterized with a binary relation R, which is called an *address relation*. The relation R is a set of *tuples* $(s_i, d_i) \in R$, where s_i is an address on the source, called a *source address*, and d_i is an address on the destination, called a *destination address*.

There are two constraints on the elements of an address relation: (1) source and destination addresses point to valid memory locations, and (2) destination addresses are unique.[1] Thus, address relations can describe arbitrary permutations such as transpose operations, as well as replication of source data items on the destination, but as defined here, they cannot describe operations like reductions or scans.

For a particular data transfer, the message assembly and disassembly routines must somehow generate the appropriate address relation. Where the addresses are computed is determined by the message passing model. With a *send/receive* model, the source node computes the source addresses and the destination node computes the destination addresses. With a *put* model, the source node computes the entire address relation and passes the destination addresses along with the data. With a *get* model, the destination node computes the entire address relation and sends the source addresses to the source, which replies with the data.

One approach for generating the address relation is to recompute the addresses on the fly in the assembly routine. For example, at a high level an assembly routine to transfer n data items using a *send/receive* model has the following form:

```
loop i = 1 to n
    recompute source address s(i)
    buffer[i] = memory[s(i)]
endloop
```

Even for fairly regular access patterns, such as those induced by HPF whole array assignment statements, recomputing the addresses each time can be complex. Parallelizing compilers can help a great deal by exploiting compile–time knowledge to generate specialized assembly loops for each data transfer [23, 24]. However, if the data access pattern is irregular (and thus unknown to the compiler), or if a parallelizing

[1] In general, it would be possible to relax this constraint by associating a binary associative combining operation with each address relation to handle non–unique destination addresses. However, we will assume unique destination addresses in this paper.

compiler is not available, then a general–purpose run–time library assembly routine is necessary, and there is a limit to the number of special–case optimizations that such a library routine can exploit. Of course, a parallelizing compiler is also limited to some number of optimizations, but it is practical to support a far larger number because optimizations can be selected at compile–time instead of run–time.

We are exploring an alternative approach, based on inspector/executor [18, 1], that allows a general–purpose message assembly routine in a run–time library to run efficiently without any special–case run–time or compile–time optimizations. The idea is that instead of recomputing the address relation from scratch during message assembly, the relation is precomputed and *encoded* (i.e., stored in memory in some form), either at startup time or the first time a particular data transfer is encountered. This inspector phase takes the following form:

```
loop i = 1 to n
    compute source address s(i)
      and receiver address r(i)
    encode s(i), r(i) into relation
endloop
```

The encoded address relation is then *decoded* inside the message assembly loop to provide addresses. The message assembly routine (the executor) to transfer n data items using a *send/receive* model would have the following form:

```
loop i = 1 to n
    decode relation to generate
      source address s(i)
    buffer[i] = memory[s(i)]
endloop
```

Depending on the message passing model (send/receive, put, or get), the assembly and disassembly loops will have a different form, but a library can support all three forms by simply providing the facility to compute an address relation.

Notice that the two assembly loops above have a similar form. The intuition behind the decoding approach is that it should be faster to decode a precomputed form of the address relation than to recompute it on the fly each time. The decoding time depends on the relation itself and its encoding in memory.

3 Representative address relations

In general, the elements of an address relation are arbitrary. However, in many cases the address relations induced by operations on distributed data structures have considerable regularity. For example, consider the redistribution of a distributed array in an HPF program. Each dimension of a regular block–cyclic distributed array has some size N and is distributed over some number of nodes P with some fixed block size M. Conceptually, the dimension is split into $\lceil N/M \rceil$ blocks of size M ($N - M(\lceil N/M \rceil - 1) \leq M$ for the final block) which are then distributed round robin over the P nodes. The most common regular block–cyclic distributions are BLOCK, where there are only $\lceil N/M \rceil = P$ blocks and CYCLIC, where $M = 1$. Each dimension of an array can be independently distributed.

The idea of array redistribution is to change the location (node and address) of elements of the array so as to alter its distribution. Because the redistribution may be onto a disjoint group of nodes, we refer to *source* and *destination* distributions and nodes. Figure 1 shows a source distribution of a 2–dimensional array with a (BLOCK,*) distribution (over two processors.) The shaded array elements are those that node 0 owns. The "*" indicates a non–distributed dimension. The same figure also shows a destination distribution of (CYCLIC,*) (also over two processors.) To redistribute the array, for each pair of source and destination nodes, we compute the intersection of the elements the nodes own (which is highlighted in the figure) and map the source node address of each element in the intersection to its address in the destination node. Data is gathered from the source node addresses to a contiguous buffer, communicated, and scattered to the destination node addresses.

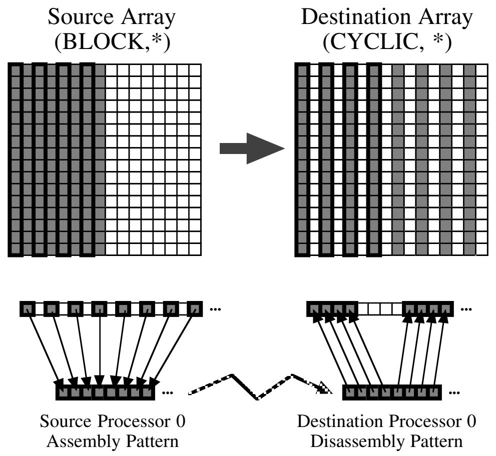

Figure 1: Redistributing regular block–cyclic arrays

For 1–dimensional BLOCK and CYCLIC arrays, it can clearly be seen that only simple stride–j to stride–k mappings can occur in an address relation induced by redistribution. BLOCK to BLOCK redistributions result in unit strides on both the source and destination nodes while the combination of CYCLIC and BLOCK results in a unit stride on either the source or destination node and a non–unit stride on the other. CYCLIC to CYCLIC redistributions result in non–unit strides on both nodes, as do certain forms of transpose operations. With more than one dimension, the access pattern is the Cartesian product of the access patterns generated for each matching dimension [22]. This can also be seen in Figure 1. The basic mapping is stride–2 to stride–1 repeated four times. This pattern is then repeated stride–8 to stride–8 16 times.

We place array redistributions of BLOCK and CYCLIC distributed arrays into four equivalence classes based on whether their basic source and destination reference patterns have unit or non–unit strides. The four two–dimensional array redistributions (of a 1024×1024 array on four processors) of Figure 2 are representatives of the four classes. Each distribution is over four nodes. We include a data transpose operation instead of a CYCLIC to CYCLIC redistribution to demonstrate that our technique extends to transposes as well.

4 Encoding address relations

In this section, we use the four representative address relations from Figure 2 to drive the development of compact address relation encodings that can exploit the regularity of relations induced by redistributions of regular block–cyclic arrays, while retaining the expressive power to encode arbitrary relations.

We begin with the AAPAIR encoding, which is a simple index array that exploits no regularity in the relation, and then introduce a series of encodings (AABLK, DMRLE, and DMRLEC) that incrementally exploit more regularity in the relations without loss of expressive power. For each encoding, we place a total order $\leq$ on the tuples of the address relation:

$$(s_i, d_i) \leq (s_j, d_j) \Leftrightarrow (s_i < s_j) \vee ((s_i = s_j) \wedge (d_i \leq d_j)).$$

In other words, the tuples are ordered by source node address first, then by destination node addresses. The encodings we describe are also readily adaptable to ordering by destination addresses first. Because on decoding the relation, we generate addresses according to this order, the destination–first order may be preferable on machines which perform strided stores badly. Each encoding is designed so that it can be used efficiently to assemble and disassemble messages. The idea is that we repeatedly load some *key* from a stored relation, expand it to a *symbol* according to the encoding, and then use the symbol to gather or scatter some number of data elements. We do not consider encodings that are wholly expanded before use. The innovation here is not in developing simple data compression techniques, but in applying these techniques to address relations and using them during message assembly and disassembly.

Figure 3 shows the total space required for address relations on source node 0 for each of the representative relations of Figure 2. The sizes are in bytes and measured on the DEC 3000/400, which is a 64–bit platform. Relative sizes are the same on 32–bit platforms, but the absolute sizes are halved. Note that the chart has a log scale. The difference between the AAPAIR and DMRLEC encodings is over three orders of magnitude. Further, the absolute size of the DMRLEC–encoded relations is roughly 1Kbytes, which is quite small.

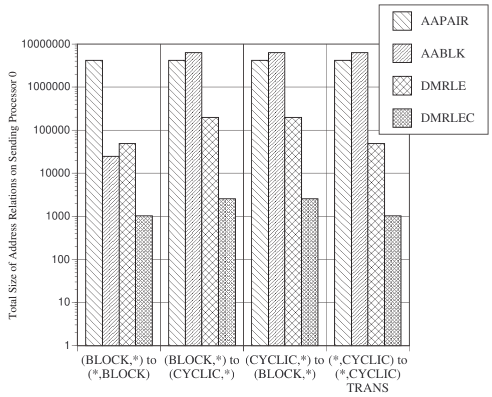

Figure 3: Total size of all address relations on sending node zero in each encoding

The AAPAIR encoding (Address, Address PAIR) is simply an array of source-destination address pairs. It takes advantage of no memory access patterns and always requires storage space proportional to the number of data items transferred. The keys and symbols for AAPAIR are the tuples themselves.

The AABLK encoding (Address, Address BLocK) run-length encodes [14] blocks of stride–1 to stride–1 mappings. The symbols generated are of the form (s, d, l), which means that addresses s through $s + l - 1$ on the source map to address d through $d + l - 1$ on the destination. The keys are simply the symbols. Notice that this is sufficient for the first representative redistribution in Figure 2, but not the others.

The DMRLE encoding (Difference Map, Run-Length Encoded) generalizes AABLK to map blocks of addresses with a stride–j source to a stride-k destination. To encode the

Redistribution		Assembly		Disassembly		Tuples	
Source	Destination	Read	Write	Read	Write	(0 to all)	(0 to 0)
(BLOCK,*)	(*,BLOCK)	1	1	1	1	262144	65536
(BLOCK,*)	(CYCLIC,*)	4	1	1	1	262144	65536
(CYCLIC,*)	(BLOCK,*)	1	1	1	4	262144	65536
(*,CYCLIC)	(CYCLIC,*)^T	4	1	1	1024	262144	65536

Figure 2: Representative redistributions of 1024 $\times$ 1024 distributed array (four processors), basic memory access strides, and number of tuples. ("T" represents a transpose)

relation in DMRLE, we form the difference map sequence $\langle (ds_i, dd_i) \rangle$ from the ordered sequence of tuples $\langle (s_i, d_i) \rangle$:

$$(ds_i, dd_i) = \begin{cases} (s_0, d_0) & i = 0 \\ (s_i - s_{i-1}, d_i - d_{i-1}) & i > 0 \end{cases}$$

The difference map is then run-length encoded to produce symbols of the form (ds, dd, l) which are stored as keys. Notice that this encoding significantly reduces space requirements for the three redistributions where AABLK fails, and performs only marginally worse for the fourth.

The DMRLE encoding efficiently encodes the basic stride–j to stride–k mapping, but for multidimensional arrays this will be repeated many times. Clearly, because the symbols for inner dimensions will occur more frequently than for outer dimensions, a Huffman-encoding [11] of the symbols would most readily reduce their storage cost. However, Huffman codes are of variable length, so keys could span word boundaries and be expensive to decode during message assembly. Instead, we adopt a dictionary substitution encoding [14] of the symbols as fixed length keys. The key length is the minimum power of two bits necessary to represent all the unique symbols. This scheme is efficient for message assembly because keys cannot span word boundaries and there is always the same number of keys per word.

To encode in DMRLEC (Difference Map, Run-Length Encoded, Compressed) the relation is first encoded in DMRLE and all unique (ds, dd, l) symbols are found via hashing. These symbols are stored in a table of size 2^k for the minimum k possible. Each symbol is encoded as a 2^k-bit key representing its offset in the table. As can be seen from Figure 3 this technique reduces the storage cost of the relations induced by the encoding relations by two orders of magnitude over DMRLE. Overall, DMRLEC is three to four orders of magnitude more space efficient than AAPAIR.

5 Measured performance

We have developed analytical models for predicting the performance of the AAPAIR encoding [6, 5]. These models show that the inspector/executor approach using AAPAIR outperforms the recomputing on the fly approach in a surprising number of instances.

Performance for the more compact encodings is difficult to model analytically since it depends heavily on the access patterns in the relation, unlike with the AAPAIR encoding. Instead, we directly measured the performance of message assembly and disassembly on six different machines. For each machine, each encoding (AAPAIR, AABLK, DMRLE, and DMRLEC) was measured on each of the four representative redistributions of Figure 2. The performance measure in all cases is the absolute copy throughput (in MBytes/sec) of the assembly and disassembly routines. By copy throughput, we mean the number of MBytes that are packed into (unpacked out of) the message buffer each second during message assembly (disassembly).

The measurements are taken from **ART** — **A**ddress **R**elation **T**oolbox — a portable library, entirely written in C and designed to be used with a conventional message passing system based on the *send/receive* model. The address relations used are those induced between source node zero and destination node zero for each of the four representative block–cyclic redistributions of Figure 2. The distributed array contains 1024 $\times$ 1024 8-byte doubles, so the zero to zero relation assembly and disassembly steps copy 512 KBytes. A larger 2048 $\times$ 2048 array (2048 KBytes assembled/disassembled) was also tested with only minor changes in measured copy throughput.

The identical C code was compiled and measured on the DEC 3000/400 [8], the Intel Paragon [12], the IBM RS/6000 [16] 250, the IBM SP-2 [19], and two Pentium [17, 2] systems: a Gateway Pentium 90, and a Micron Pentium 133. The details of each testing environment are

Machine	Node	Compiler	Operating System
DEC 3000/400	Alpha (21064 133 MHz)	GNU C 2.6.3, DEC CC 3.11	OSF/1 2.0
Intel Paragon	i860 (i860XP 50 MHz)	Intel C R4.5	OSF/1 1.0.4
IBM RS/6000 250	PowerPC (601 80 MHz)	IBM XLC 1.3.0.4	AIX 3.2
IBM SP-2	POWER2 (66 MHz)	IBM XLC 1.3.0.33	AIX 3
Gateway Pentium 90	i386 (Pentium 90 MHz)	GNU C 2.4.5	NetBSD 1.0A
Micron Pentium 133	i386 (Pentium 133 MHz)	GNU C 2.4.5	NetBSD 1.0A

Figure 4: Testing environments

given in Figure 4. Figure 5 shows the results for the first four machines, while Figure 6 shows the results for the Pentium–based machines. Each row shows the results for a single machine. Message assembly throughput is shown on the left and disassembly throughput on the right. Superimposed on each graph are annotated lines that represent the throughput of the library `memcpy()` routine and the throughput of C-coded strided copies of doubles. A line annotated "stride–k" on an assembly graph indicates the throughput of a copy loop that copies from a stride of k doubles to a stride of 1, while on a disassembly graph it indicates the throughput of a copy loop that copies from a stride of 1 to a stride of k doubles. Each chart measures the absolute throughput on the same scale, so that results can be compared across machines. Except for the Pentium machines (Figure 6), only the throughput of the AAPAIR (as a baseline) and DMRLEC encodings are shown. This is because the story is the same for these machines: DMRLEC is always the fastest compact encoding; DMRLE has similar performance, and AABLK is the slowest for all but the first redistribution. The situation is more complex on the Pentium machines, so the throughputs of all four encodings are shown.

The principle observation is that, for most of the machines, message assembly and disassembly using the DMRLEC encoding operates at rates that are at or close to the copy throughput possible for their memory reference patterns. The remainder of this section analyzes each machine in turn.

On the DEC 3000/400, DMRLEC gives ideal assembly and disassembly throughputs for all the test cases. DMRLEC benefits from the separate load/store pipeline in the Alpha 21064 processor [15, 7]. Even on a load stall, integer instructions associated with DMRLEC can continue to issue.

On a Paragon node (i860 [13] processor), DMRLEC gives near ideal assembly throughput for all the test cases, as shown in Figure 5. The disassembly results are somewhat unexpected, however. Although the first test case (stride–1 writes) operates with ideal throughput, the third test case (stride–4 writes) actually does slightly better than a stride–4 copy loop. We explain this due to different code generation on the part of the compiler for the disassembly loop and the stride–4 copy loop. As can be seen by the factor of two difference between a compiler generated stride–1 copy loop and the library `memcpy()` routine, there is considerable room for improving on the stride–k copy loops generated by the Paragon node compiler.

The RS/6000 250 (PowerPC 601 [4]) also achieves near ideal assembly and disassembly throughput with DMRLEC. The IBM XLC compiler has software-pipelined each of the assembly, disassembly, and strided copy loops.

The throughput measurements of message assembly and disassembly using DMRLEC on the IBM SP-2 are also close to ideal, tracking the throughput of the strided copy loops closely. The IBM XLC compiler has software–pipelined assembly, disassembly, and each of the strided copy loops. Because the POWER2 [27, 3] processor, like the Alpha, can continue to issue integer instructions while a load miss is being handled, instructions associated with DMRLEC are free.

The performance of the Pentium systems is more complex than that of the other machines, as can be seen in Figure 6. On both machines, the assembly and disassembly throughput of DMRLEC is lower than DMRLE. On the faster Micron machine, only the first test case achieves copy throughput for message assembly, and then only in the simple AABLK encoding. Message assembly and disassembly appear to be instruction issue limited for DMRLEC on both machines. Because the ratio of memory copy bandwidth (as measured by `memcpy()`) to instruction issue rate is higher on the Micron, it is not surprising that it performs comparatively worse in message assembly and disassembly using DMRLEC.

We believe that three factors contribute to this behavior. First, the GNU C compiler does not support Pentium optimizations, so clearly, no scheduling is done to improve

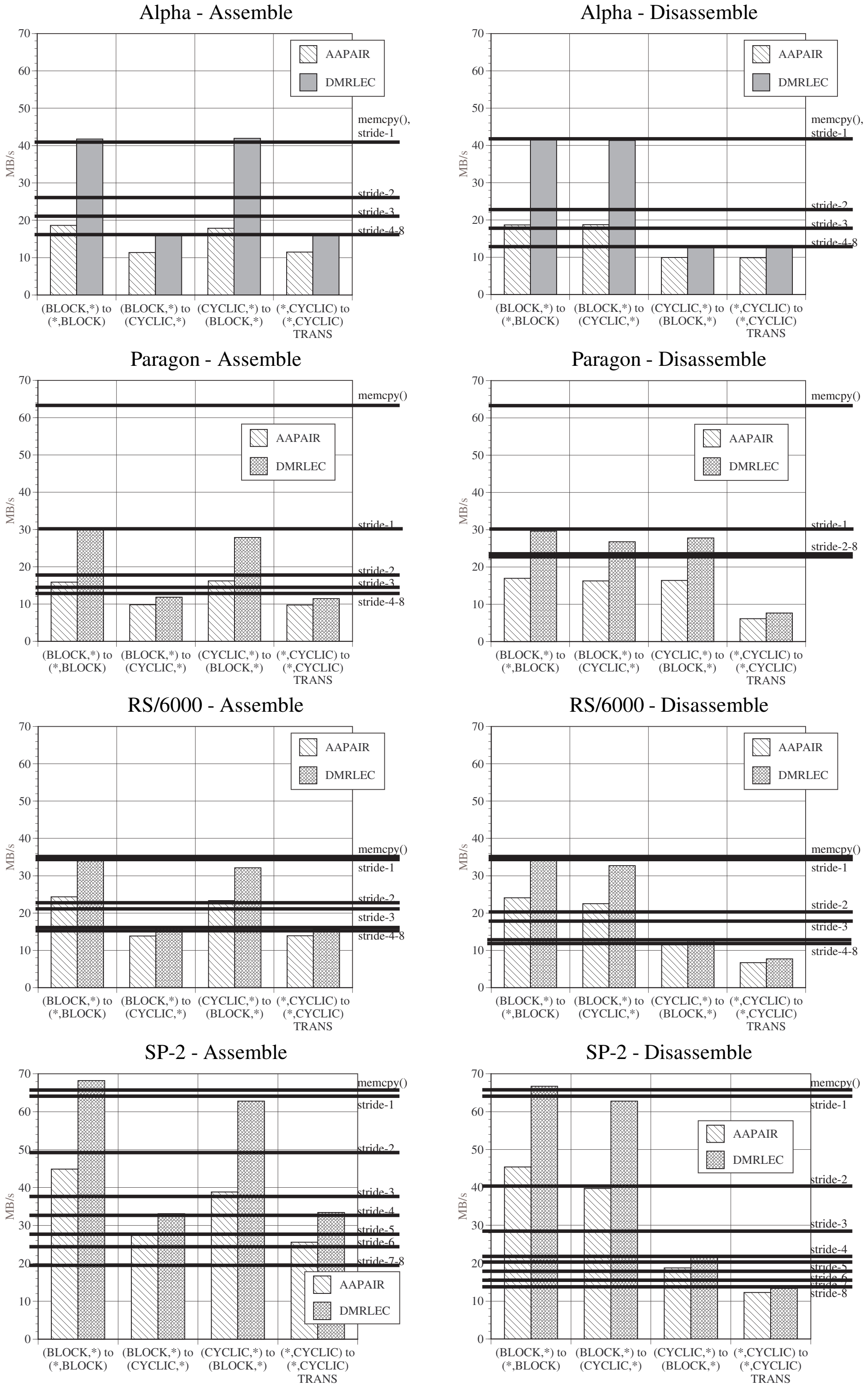

Figure 5: Throughput (MBytes/sec) of message assembly and message disassembly using AAPAIR and DMRLEC on DEC Alpha 3000/400, Intel Paragon, IBM RS/6000, and IBM SP-2

the multiple issue rate. Second, the Pentium is both register starved and non–orthogonal. There are a significant number of instructions in the message assembly loop that do nothing but move values associated with the compact encodings to and from the appropriate registers for certain operations. In other cases, memory operands are used where registers would appear with a larger register set. Finally, although the latency of memory operands is shortened by the on–chip cache, the cache is not actually dual ported, but banked, so cache references can become serialized.[2] Note that even with this complexity, DMRLEC always performs better than AAPAIR on the Pentium.

6 Discussion

By taking address computation out of the critical path of data transfer, the decoding approach makes message assembly throughput depend on the address relation and the memory access pattern it encodes instead of how the address relation is computed. One advantage of this is that it becomes easier to achieve reasonable communication performance without a large number of special–case optimizations in the compiler and run–time library. Instead of optimizing each address computation algorithm for each architecture, only message assembly for each address relation encoding needs to be optimized for each architecture in order for all data transfers to benefit.

Another interesting property of the decoding approach is that the entire relation is available to be optimized at a new level of the software hierarchy. For example, by ordering the relation by destination node addresses first, we can optimize for reference locality on the destination node instead of on the source node. As shown in [25], the choice is architecture–dependent and can have a significant impact on performance.

Stored encoded address relations also appear to be a good interface from user programs to the communication system because they are implementation neutral. This permits communication systems to be highly specialized for the available hardware while still presenting the same interface to the programmer or compiler. However, to best utilize the hardware, it may be necessary to change the encoding of the address relation before using it. For example, the relation may be transformed into a DMA gather map on the source to make best use of a network adapter such as the one described in [20]. It is unclear whether this is the optimal approach, since the main processor could presumably support better encodings than the network adaptor. However, it is an intriguing possibility.

Address relations could also be a powerful abstraction in the interface *between* the layers of a communication system. Consider a TCP/IP implementation where each layer merely added tuples necessary to prepend its header to the packet instead of recopying it. The data link layer could then simply gather the contents of the packet directly to the wire. With appropriate user-level semantics, such a communication system could avoid all data copies.

Even with the compact address relation encodings described in this paper, it is still useful to be able to bound the amount of memory used to store address relations. Address relations can still have huge space requirements if they do not have regularity to exploit. This leads naturally to a caching approach. When the allotted amount of memory is in use and a new relation needs to be stored, a victim relation must be chosen and flushed from memory to make room. User or compiler-supplied hints would be helpful to minimize cache thrashing. For example, if the run–time system were told that a group of relations were associated with the same collective communication operation, then it might make sense to flush and restore these relations as a group. Another possibility is to publicize the characteristics of the address relation cache so that compilers and programmers can make reasonable choices, as with a memory cache.

Since address relations can be decoded at near copy throughputs on most systems, it might be interesting to investigate using precomputed address relations as the basis for a generalized form of `memcopy()` on uniprocessors. An example application for such a library routine could be bit block transfers (bit–blit) and transfers of regions in graphical interfaces and games.

7 Conclusions and open issues

This paper introduced the notion of an address relation for describing data transfers between the nodes of a parallel system, and it described how compact encoded forms of these relations can serve as the basis for fast message assembly and disassembly. The main result is that message assembly and disassembly based on decoding compact address relations typically perform at near copy throughput for the relations induced by redistributions of regular block–cyclic distributed

[2] We also compiled the library with Microsoft's Visual C++ 2.2, which includes Pentium-specific optimizations, but saw no difference in performance during informal tests. This leads us to believe that the architecture or memory system is the overriding factor in the lack-luster performance of the Pentiums.

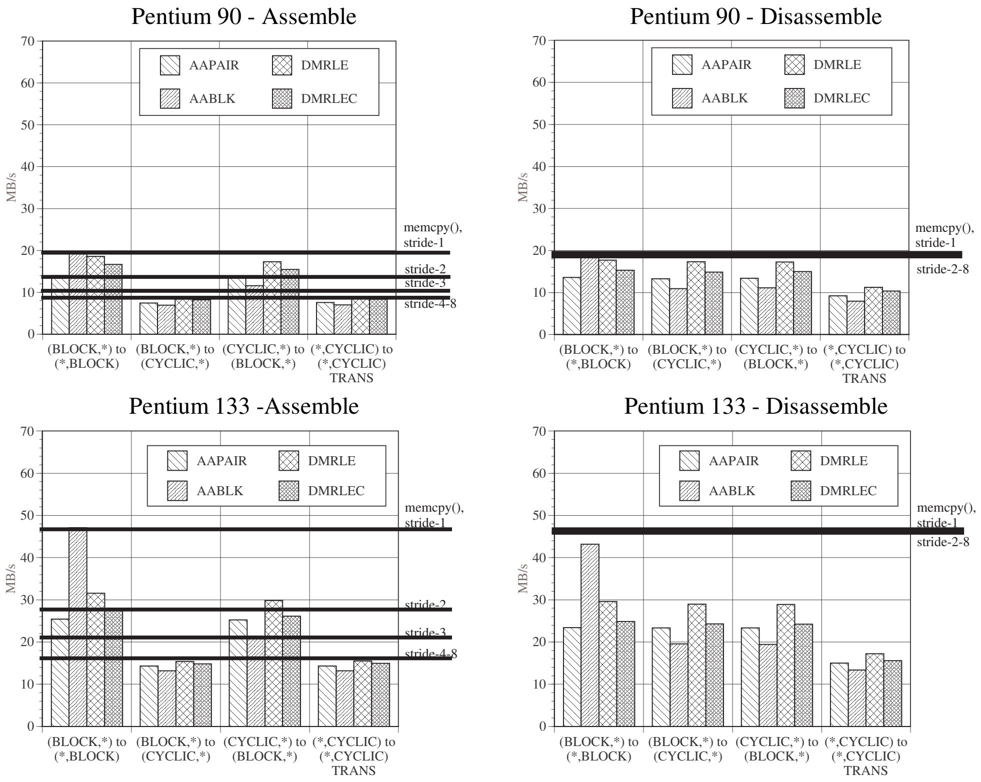

Figure 6: Throughput (MBytes/sec) of message assembly and message disassembly using AAPAIR, AABLK, DMRLE, and DMRLEC on Gateway Pentium 90 and Micron Pentium 133 systems

arrays. A side effect is that the storage requirements for these relations is drastically reduced to a practical level.

While the paper shows encouraging results for a carefully constrained (but very useful) set of relations, it does not address the fundamental question of how to characterize machines and address relations so that we can decide whether to decode or recompute, and if we decide to decode, what encoding of the relation to use. This issue will become increasingly important for relations with irregular memory reference patterns, since the potential benefit of decoding would seem to be greatest for the irregular cases. It will be necessary to understand how to encode and decode relations that are more irregular than the relations described in this paper, as well as to understand the performance impact of these encodings. One motivating application for these irregular relations is sparse matrix–vector multiplication over irregular finite–element meshes.

Another avenue to explore is the design of more aggressive encoding schemes such as the Huffman–based [11] DMRLE encoding mentioned in Section 4. Another possibility is to use LZW [28, 26] or a variant thereof for further compression. [3]

We are also interested in how much the performance of message assembly and disassembly depends on single–node compilers. Many parallelizing compilers generate sequential source code and rely on the single–node compiler to produce good results. However, as can be seen from the factor of two difference between a compiler–generated copy loop and the library `memcpy()` routine on the Paragon (Figure 5), this reliance may be unwarranted. If so, then the decoding approach becomes more attractive since the assembly routine for each encoding could be low–level coded for each architecture. Such low–level coding would be probably be impractical for the recomputing approach due to the many variants of assembly routines. Instead, a parallelizing compiler could precompute address relations at compile–time, and rely on low–level coded library of message assembly routines to achieve near copy throughput. In general though,

[3] Early experiments with the `gzip` LZW–based compressor show dramatic compression, but we do not yet know if LZW–based decoding can be made to run at copy throughput speeds.

the appropriate interaction between a parallelizing compiler and a run–time message assembly library based on decoding precomputed address relations is not yet understood.

References

[1] AGRAWAL, G., SUSSMAN, A., AND SALZ, J. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transaction on Parallel and Distributed Systems 6*, 7 (July 1995), 747–754.

[2] ALPERT, D., AND AVNON, D. Architecture of the Pentium processor. *IEEE Micro 13*, 3 (June 1993), 1–21.

[3] BARREH, J., DHAWAN, S., HICKS, T., AND SHIPPY, D. The POWER2 processor. In *Proceedings of COMPCON '94* (San Francisco, CA, March 1994), IEEE, pp. 389–398.

[4] BECKER, M. C., ALLEN, M. S., MOORE, C. R., MUHICH, J. S., AND TUTTLE, D. P. The PowerPC 601 processor. *IEEE Micro 13*, 5 (October 1993), 54–68.

[5] DINDA, P. A., AND O'HALLARON, D. R. High throughput message assembly using compact address relations. Tech. rep., Carnegie Mellon University, 1996. To Appear.

[6] DINDA, P. A., AND O'HALLARON, D. R. The performance impact of address relation caching. In *Proceedings of the Third International Workshop on Languages, Compilers and Run–time Systems for Scalable Computers* (1996), B. K. Szymanski and B. Sinharoy, Eds., Kluwer Academic Publishers, pp. 213–226.

[7] DOBBERPUHL, D. I. A. A 200-MHz 64-bit dual-issue CMOS microprocessor. *Digital Technical Journal 4*, 4 (1992), 35–50. ftp://ftp.digital.com/pub/Digital/info/DTJ/axp-cmos.txt.

[8] DUTTON, T. A., EIREF, D., KURTH, H. R., REISERT, J. J., AND STEWART, R. L. The design of the DEC 3000 AXP systems, two high performance workstations. *Digital Technical Journal 4*, 4 (1992), 66–81. ftp://ftp.digital.com/pub/Digital/info/DTJ/axp-dec-3000.txt.

[9] GUPTA, S., KAUSHIK, S., HUANG, C., AND SADAYAPPAN, P. On compiling array expressions for efficient execution on distributed-memory machines. Tech. Rep. OSU-CISRC-4/94-TR19, Computer and Information Science Research Center, Ohio State University, Apr. 1994.

[10] HIGH PERFORMANCE FORTRAN FORUM. High Performance Fortran language specification, version 1.0. Tech. Rep. CRPC-TR92225, Center for Research on Parallel Computation, Rice University, May 1993.

[11] HUFFMAN, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (September 1952), 1098–1101.

[12] INTEL CORP. *Paragon X/PS Product Overview*, March 1991.

[13] INTEL CORP. *i860 Microprocessor Family Programmer's Reference Manual*, 1992.

[14] LELEWER, D. A., AND HIRSCHBERG, D. S. Data compression. Tech. Rep. 87-10, University of California – Irvine, 1987.

[15] MCLELLAN, E. The Alpha AXP architecture and 21064 processor. *IEEE Micro 13*, 3 (June 1993), 36–47.

[16] OEHLER, R., AND GROVES, R. IBM RISC system/6000 processor architecture. *IBM Journal of Research and Development 34*, 1 (Jan 1990), 23–36.

[17] SAINI, A. An overview of the Intel Pentium processor. In *COMPCON Spring 1993 Digest of Papers* (San Francisco, CA, 1993), pp. 60–62.

[18] SALTZ, J., , PETITON, S., BERRYMAN, H., AND RIFKIN, A. Performance effects of irregular communication patterns on massively parallel multiprocessors. *Journal of Parallel and Distributed Computing 13* (1991), 202–212.

[19] SNIR, M. Scalable Parallel Computing - The IBM 9076 Scalable POWERParallel-1. In *ACM Symposium on Parallel Algorithms and Architectures* (June 1993), ACM, p. 42.

[20] STEENKISTE, P., ZILL, B., KUNG, H., SCHLICK, S., HUGHES, J., KOWALSKI, B., AND MULLANEY, J. A host interface architecture for high-speed networks. In *Proceedings of the 4th IFIP Conference on High Performance Networks* (Liege, Belgium, December 1992), IFIP, Elsevier, pp. A3 1–16.

[21] STEENKISTE, P. A. A systematic approach to host interface design for high-speed networks. *Computer 27*, 3 (March 1994), 47–57.

[22] STICHNOTH, J. Efficient compilation of array statements for private memory multicomputers. Tech. Rep. CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, Feb. 1993.

[23] STICHNOTH, J., O'HALLARON, D., AND GROSS, T. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing 21*, 1 (Apr. 1994), 150–159.

[24] STICHNOTH, J., AND T., G. A communication backend for parallel language compilers. Proc. Fifth Workshop on Compilers for Parallel Languages, Malaga, Spain (Tech Report UMA-DAC-95/09), pages 65–77, 1995.

[25] STRICKER, T., AND GROSS, T. Optimizing memory system performance for communication in parallel computers. In *Proc. 22nd Intl. Symp. on Computer Architecture* (Portofino, Italy, June 1995), ACM/IEEE, pp. 308–319.

[26] WELCH, J. A technique for high performance data compression. *IEEE Computer 17*, 6 (June 1984), 8–19.

[27] WHITE, S. W. Power2: Architecture and performance. In *Proceedings of COMPCON '94* (San Francisco, CA, March 1994), IEEE, pp. 384–388.

[28] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (May 1977), 337–343.