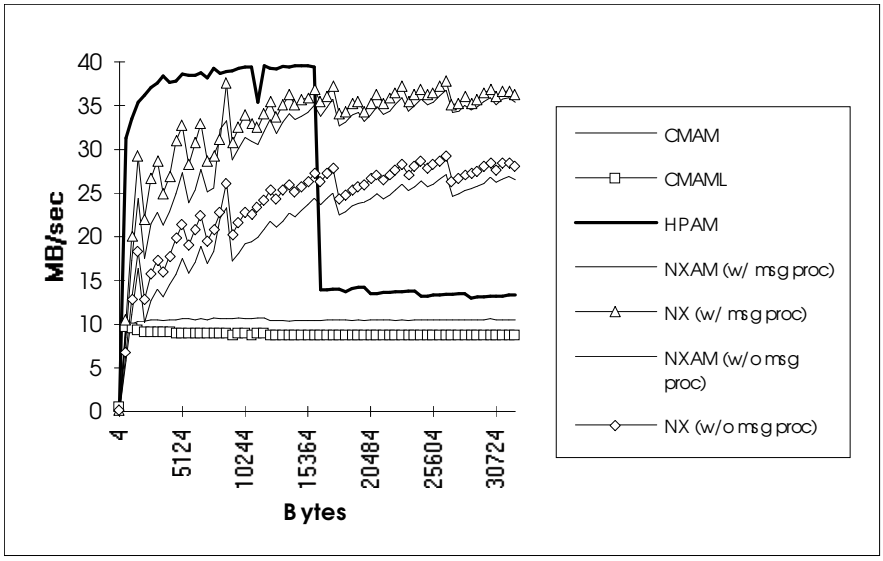

FIGURE 10. The bandwidth of four active message layers and NX.

Figures 11 through 15 show the bandwidth of the five Libsplit-C bulk transfer operations.

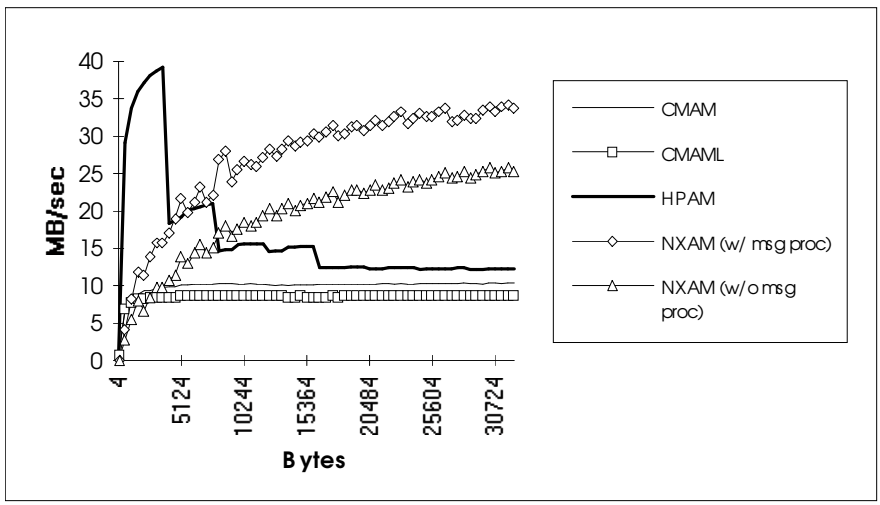

FIGURE 11. Bandwidth of bulk store.

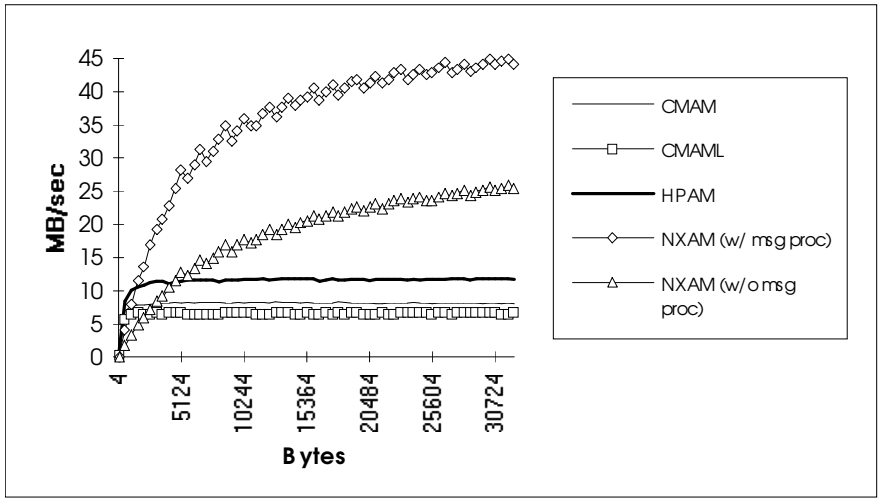

FIGURE 12. Bandwidth of bulk get.

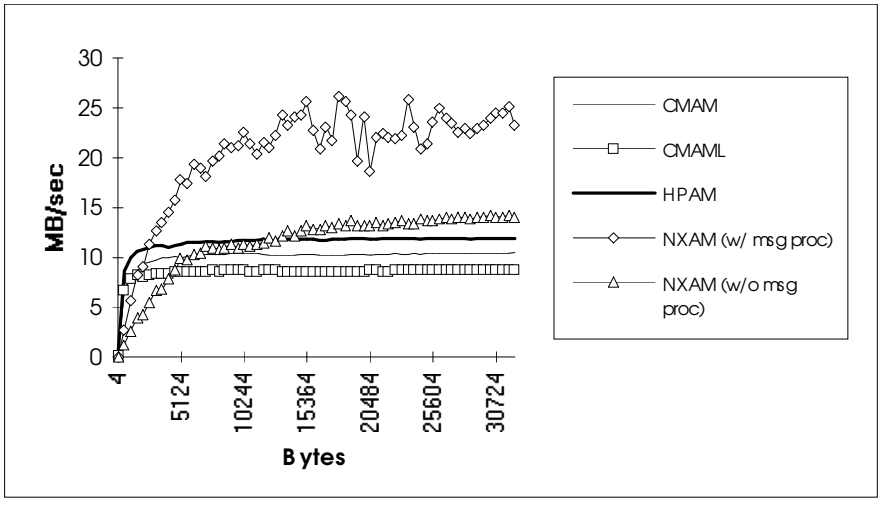

FIGURE 13. Bandwidth of bulk put.

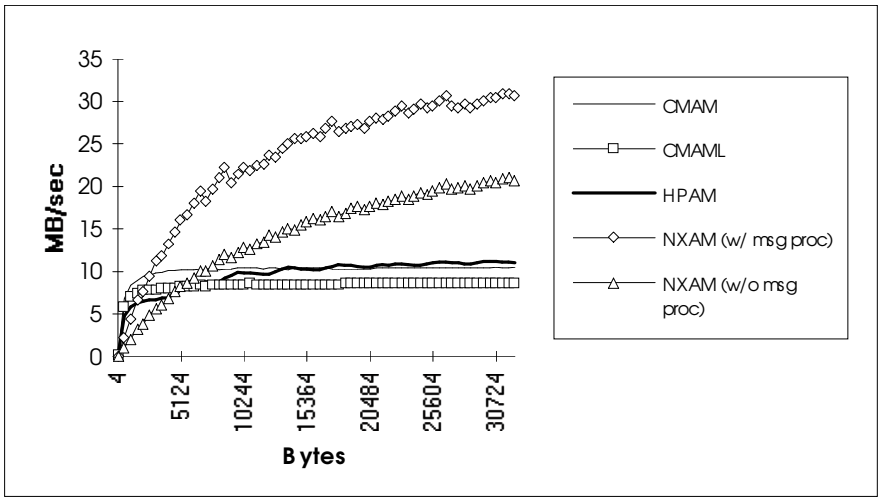

FIGURE 14. Bandwidth of bulk read.

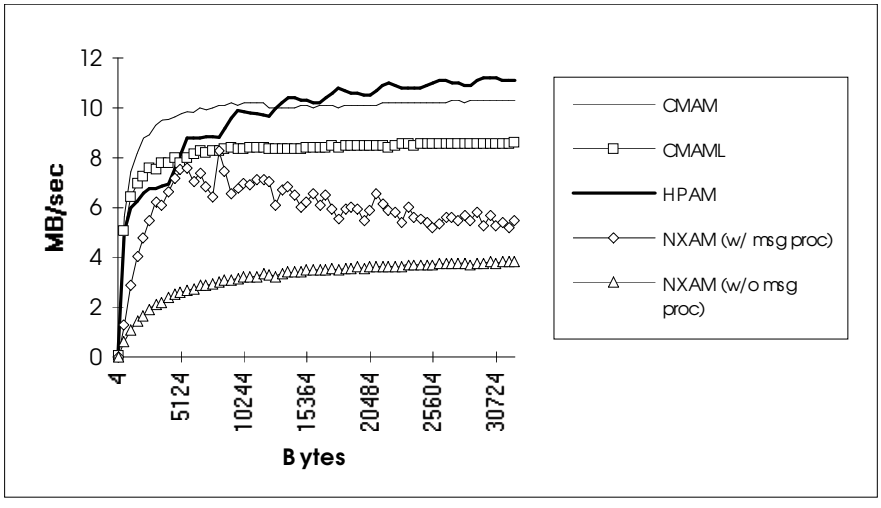

FIGURE 15. Bandwidth of write.

5.3 Global Communication

The cost of global communication is a function of the number of processors. Because we only had four HP workstations with medusa cards, we were only able to measure their performance for four processors. CM5 performance was measured on 4 nodes to compare with the HPs, and on 64 nodes because a 64 node CM5 was available. The performance of the Paragon differs by an order of magnitude, so it is displayed separately.

5.3.1 Global Communication on the CM5 and the HPs

For global communication functions, CMAML is the fastest because it uses the CM5's control network. Barrier on CMAM also uses this network. CMAM for other operations and HPAM are slower because they must perform each of these functions by sending messages. The times for both the CM5 and the HPs are for four processors.

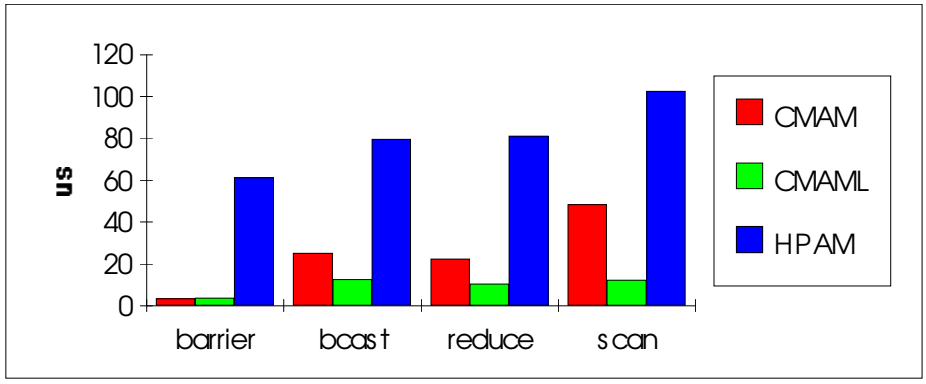

FIGURE 16. Cost of global communication for four processors on CMAM, CMAML, and HPAM.

5.3.2 Global Communication on the CM5

The times for broadcast, reduce, and scan are higher for CMAM on a 64 node machine. The times for CMAML remain about the same because CMAML uses the control network.

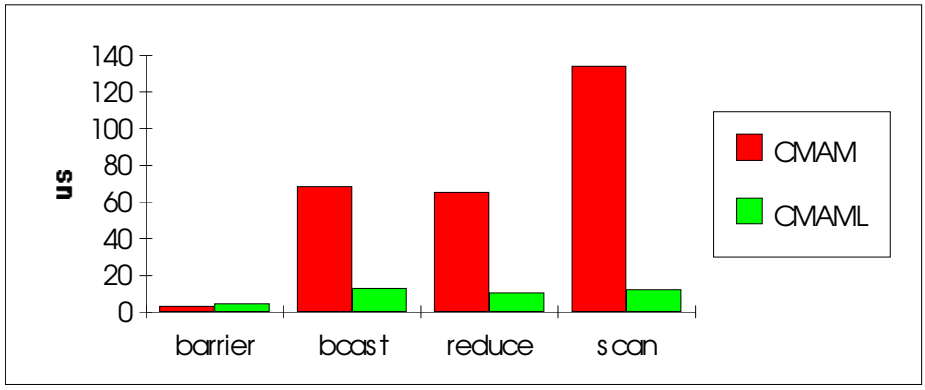

FIGURE 17. Cost of global communication for 64 processors on CMAM and CMAML.

5.3.3 Global Communication on the Paragon

Figure 18 shows two sets of times for the Paragon. Barrier is the cost of global communication using the same implementation that was used on the CM5. For this version, flow control is handled with barriers. The dataflow version uses static dataflow techniques. It

avoids the cost of the barriers and it can be pipelined. Barrier could not be optimized so its cost is the same for both versions.

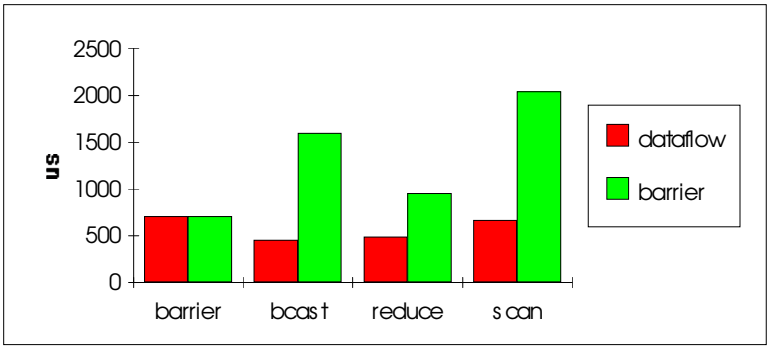

FIGURE 18. Cost of global communication for eight processors on the Paragon.

5.4 Overhead of Split-C Pointer Operations

The previous sections have shown the overhead of using Libsplit-C operations versus active messages or message passing. This section shows the overhead of several Split-C pointer operations.

Sptr[i] is the cost of indexing a spread array. For_my_1d is a macro that iterates through all the local elements of a one dimensional spread array. The cost of one iteration of the macro is shown. For_my_2D iterates through all the local elements of a two dimensional spread array, and the cost is for one iteration. Sptr++ is the cost of incrementing a spread pointer.

The cost of the Split-C pointer operations is a rough guide of the relative scalar performance of the processors on the different machines. The CM5 is built with 32 MHz Sparc2 processors. The HPs are built with 99 MHz PA RISC processors. The Paragon is built with 50 MHz i860 processors.

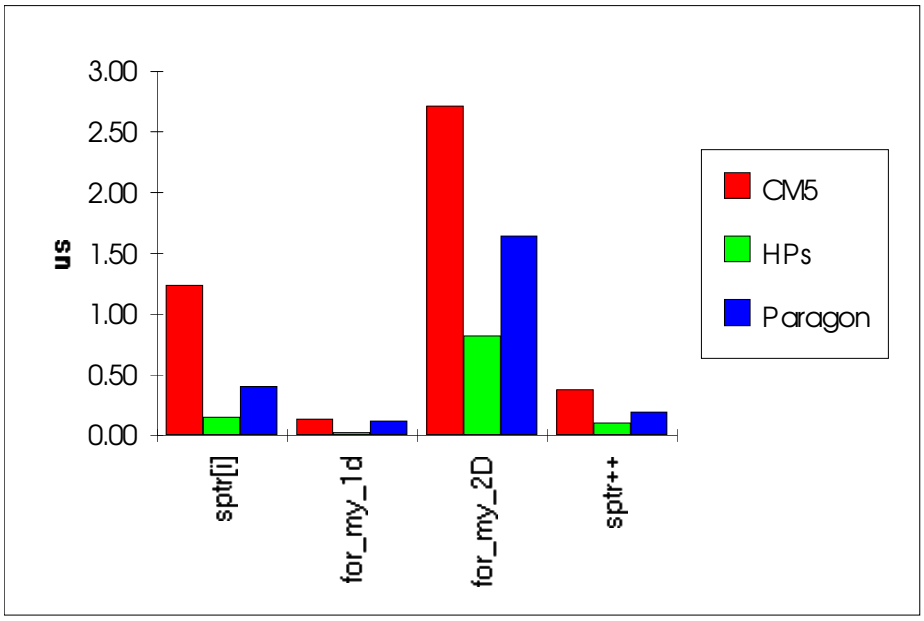

FIGURE 19. Overhead of Libsplit-C array, loop, and pointer operations.

6.0 Performance Comparison of Applications on Three Platforms

This section continues to investigate the efficiency of Libsplit-C by describing the performance of applications on the different implementations. The applications, radix sort and matrix multiply, were measured for different problem sizes on four HPs, four nodes of the CM5, and four nodes of the Paragon. For the CM5 and the Paragon, scaled speedup curves were also generated. For the HPs, we were unable to generate speedup curves because we only had four machines available. Also, because our Paragon is only 8 nodes, the SDSC Paragon was used to generate the speedup curves. The SDSC Paragon is running NX (version 1.2) and is slower than ours. The performance of global memory and bulk transfer on the SDSC Paragon is described in Appendix A. Our Paragon is running NX (version 1.1).

6.1 Matrix Multiply

Figure 20 shows the performance of matrix multiply for different problem sizes. Two square matrices were multiplied. 64x64 means that each processor contained a 64x64 element square of each matrix. The performance on the HPs drops from 87 to 29 MFLOPS when going from 128x128 to 256x256. This happens because the HPs have a 256KB cache. For small problem sizes, the matrices fit in cache.

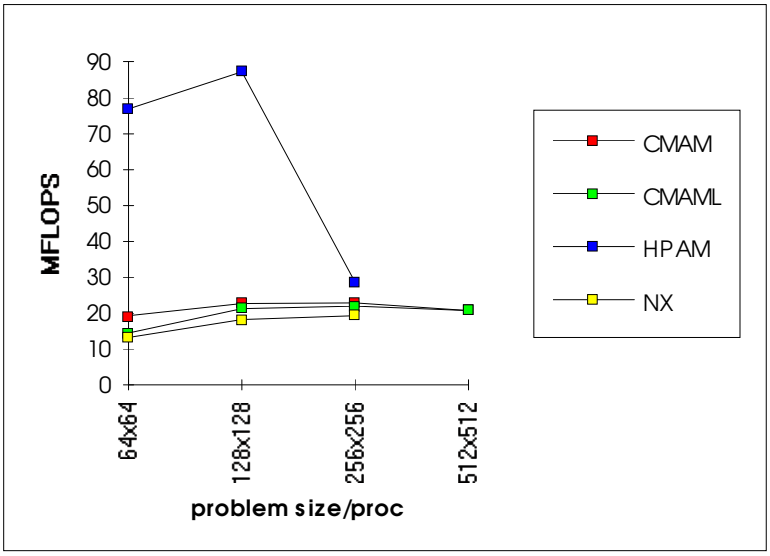

FIGURE 20. The performance of matrix multiply for different problem sizes on four processors.

TABLE 1. The performance of matrix multiply for different problem sizes (in MFLOPS).

	CMAM	CMAML	HPAM	NX
64x64	19	14	77	13
128x128	23	21	87	18
256x256	23	22	29	19
512x512	21	21		

Figure 21 shows the scaled speedup of matrix multiply on the CM5 and on the SDSC Paragon. For this problem, two square matrices were multiplied, and each processor contained a 128x128 element square of each matrix.

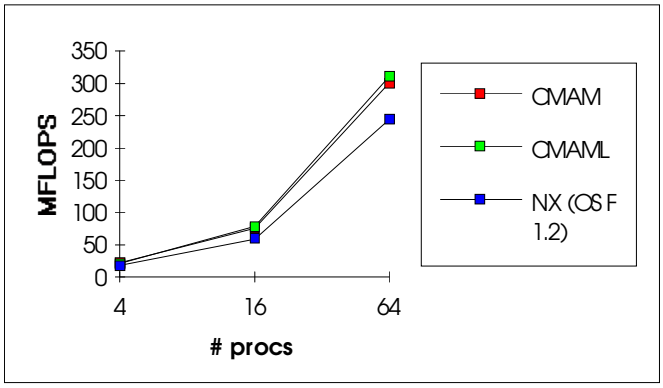

FIGURE 21. Scaled speedup of matrix multiply.

TABLE 2. Scaled speedup of matrix multiply (in MFLOPS).

# PROCS	CMAM	CMAML	NX (1.2)
4	23	21	18
16	76	79	59
64	301	311	245

6.2 Radix Sort

The radix sort operates on 32 bit integers. It has three stages: fill, scan, coalesce. Fill histograms the keys locally. Scan computes the global histograms using the global communication function, all_scan_add. Coalesce moves the keys to their correct positions. The fill time is always small. The scan time is a function of the radix and is constant with the number of keys per processor. Coalesce time is a function of the number of keys per processor and the number of passes that have to be performed.

For the CM5 and the HPs, global communication is fast enough that a 16 bit radix could be used. For a 16 bit radix, 65,536 scans must be performed for each pass, and two passes are required. Because global communication on the Paragon is much more expensive, an eight bit radix was used. For an eight bit radix, 256 scans must be performed for each pass, and four passes are required. The two extra passes increase the coalesce time on the Paragon, but this is advantageous because the global communication time is so slow.

Figures 22 through 25 show the performance of radix sort on CMAM, CMAML, HPAM, and NX. Each of these figures is for times on four processors.

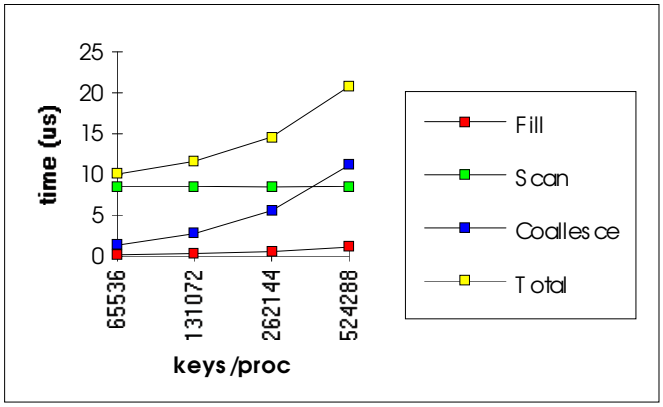

FIGURE 22. Performance of radix sort on CMAM for different problem sizes.

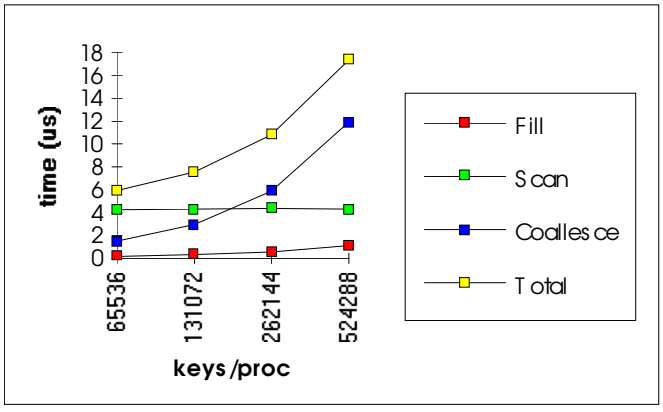

FIGURE 23. Performance of radix sort on CMAML for different problem sizes.

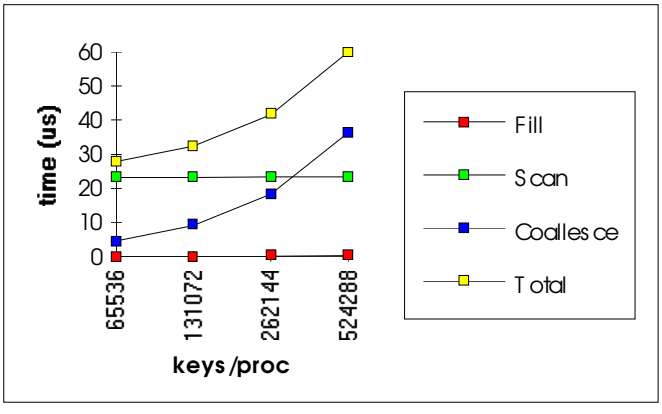

FIGURE 24. Performance of radix sort on HPAM for different problem sizes.

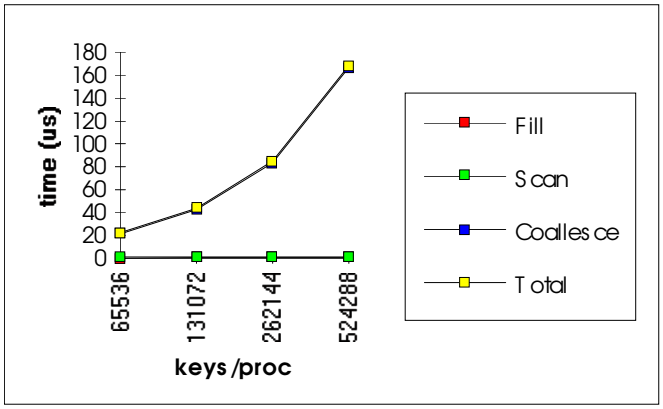

FIGURE 25. Performance of radix sort on NX for different problem sizes.

Figure 26 shows the scaled speedup of radix sort. For the times in this figure, 524,288 keys per processor were used. On the CM5, a radix of size 16 was used, and on the Paragon, a radix of size eight was used. The times for the Paragon on this figure differ from those in figure 25 because these times are for the Paragon at SDSC running NX version 1.2. For the performance of the SDSC Paragon on global memory and global communication, see appendix A.

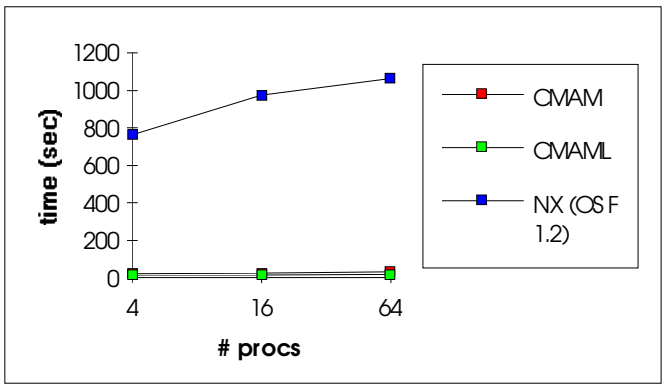

FIGURE 26. Scaled speedup of radix sort.

TABLE 3. Scaled speedup of radix sort. (Times in seconds.)

	4	**16**	**64**
CMAM	21	16	762
CMAML	26	17	977
NX (1.2)	32	17	1063

7.0 Conclusion

The experience of porting Libsplit-C to a different platforms has helped us determine how portable are global memory, bulk transfer, global communication, the global heap, and atomic procedures. It has also demonstrated that Libsplit-C can be implemented on a variety of architectures with little overhead to the message layer. Finally, Split-C can be used to write programs that are portable across distributed memory machines.

Global memory and atomic procedures were implemented easily on top of active messages. The main issue was to use the calling convention of the processor so that Libsplit-C always passes arguments in registers instead of passing them through memory. The easiest way to implement global memory on message passing was to build active messages on message passing.

For bulk transfer, a network that is both unordered and has a small maximum packet size forces the implementation to have segments. Segments cost a round trip latency for a store or a put. A network with a large maximum packet size, or an ordered and reliable network, does not need segments. In order to achieve maximum bandwidth on message passing, an additional send is required to post a receive in advance on the remote node.

For global communication functions, if the hardware doesn't provide support for them, the functions must be built from point to point messages. Implementations that are built using messages must control flow. Architectures that support barrier in hardware can control flow using barriers. Architectures that must build barrier from messages can use techniques from static dataflow. In architectures that do not provide support for a fast barrier, using static dataflow instead of barriers increases performance by 2 to 3.5 times.

Architectural support for a global heap is not required. All_spread_malloc can be implemented on multiple platforms using sbrk. However, spread_malloc is much more difficult to implement because it requires autonomous global communication.

On the CM5, Libsplit-C adds less than 25% overhead and latency to active messages. On the HPs, it adds less than 25%. On the Paragon, it adds less than 6% to the active message layer. The active message layer adds 13 to 35% overhead to the message passing layer.

The experience with matrix multiply and radix has shown us that an application that is written in Split-C can be run efficiently on multiple machines with a minimum of tuning.

8.0 Acknowledgments

I would like to thank David Culler for his guidance and advice, Kathy Yelick for her helpful comments and suggestions, and Arvind Krishnamurthy for his help with the Split-C compiler. I would like to thank Rich Martin for his help in understanding the HPAM layer, and all the others who provided me with ideas and insight.

I would like to thank the organizations that provided us with the NSF Presidential Faculty Fellowship CCR-9253705, the NSF Infrastructure Grant CDA-8727788, and the Eugene

C. Gee and Mona Fay Gee Scholarship. I would also like to thank Intel for its donation of an Intel Paragon, and IBM for its donation of an SP1. I would like to thank LANL for time on its CM5, and SDSC for time on its Paragon.

Appendix A: Performance of SDSC Paragon

Figure 27 shows the overhead and latency of NX, NXAM, and the five global memory operations on the SDSC Paragon running OSF 1.2. The methods used to gather these numbers are the same as those described in section 5.1.

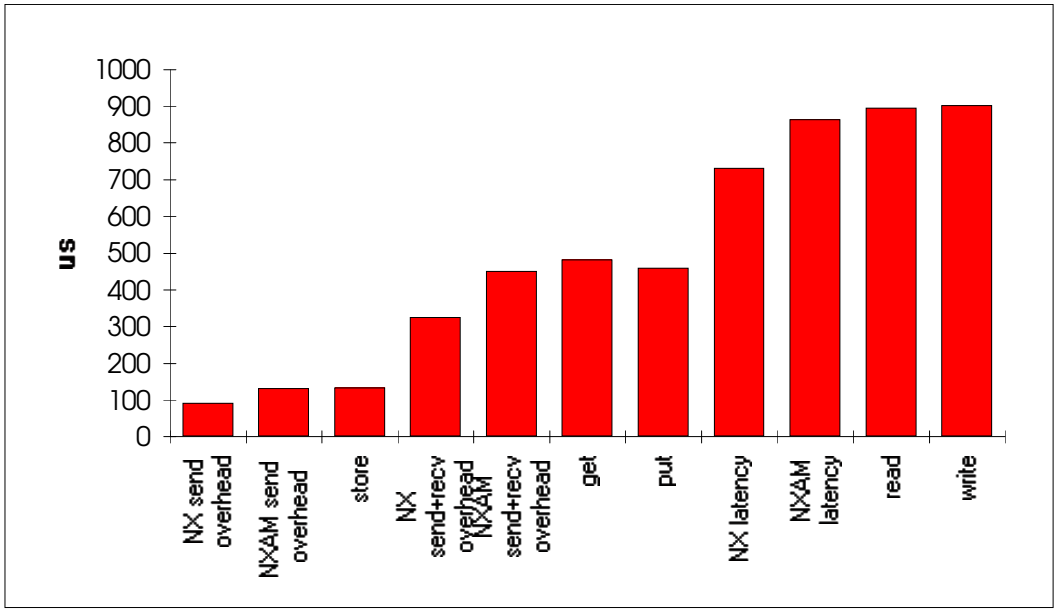

FIGURE 27. Overhead and latency of NX, NXAM, and global memory on Paragon running OSF 1.2.

Figure 28 shows the bandwidth of NX, NXAM, and the five bulk transfer operations. The methods used to gather these numbers are the same as those described in section 5.2.

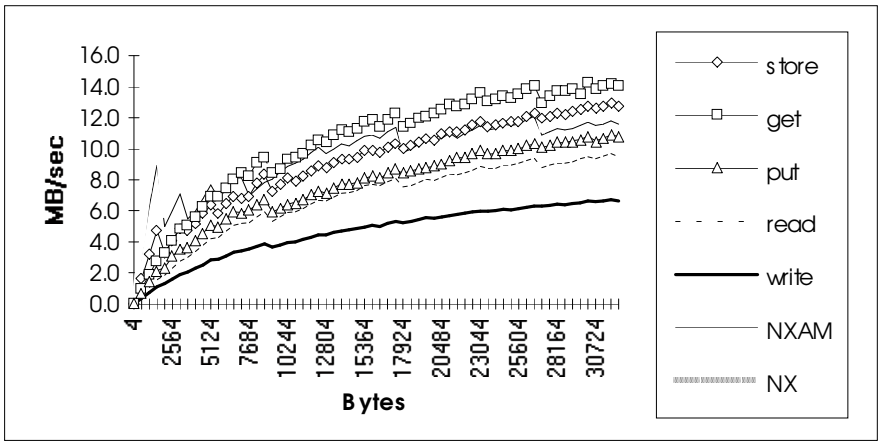

FIGURE 28. Bandwidth of NX, NXAM, and global memory on Paragon running OSF 1.2.

Appendix B: Performance of IBM SP-1

After this thesis was written, Libsplit-C was ported to the IBM SP-1. This implementation of Libsplit-C was built on the message passing layer, MPLp, and is almost identical to the implementation on NX. This port took about one day.

Here are the performance numbers for overhead and latency, bandwidth, global communication, and the applications. Latency and overhead are shown for the message passing library, MPLp, the active message layer that is built on the message library, and the five Libsplit-C global memory operations. The overhead and latency is about twice the overhead and latency for the HPs.

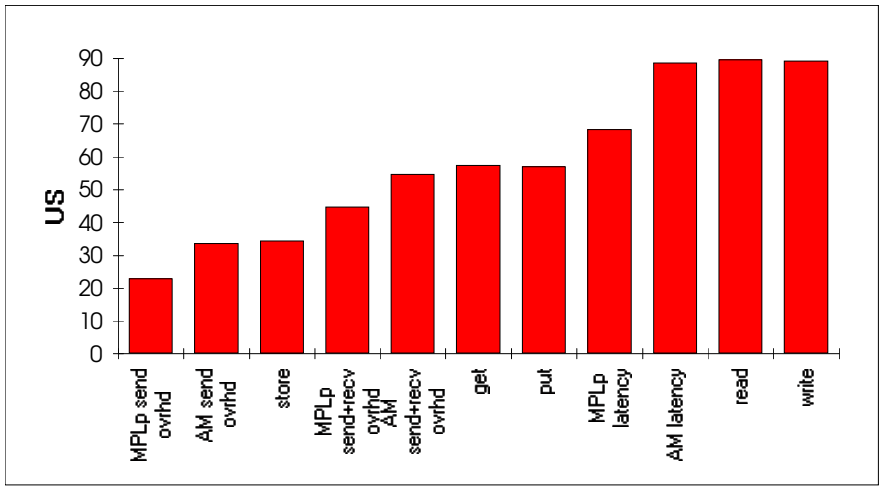

FIGURE 29. Overhead and latency on the SP-1.

Bandwidth is shown for the message library, the active message layer, and the five Libsplit-C bulk transfer operations. Two graphs are shown. The first one shows message sizes of up to 4KB. The second shows message sizes of up to 16KB. The maximum bandwidth is slightly less than the peak bandwidth achieved on the CM5.

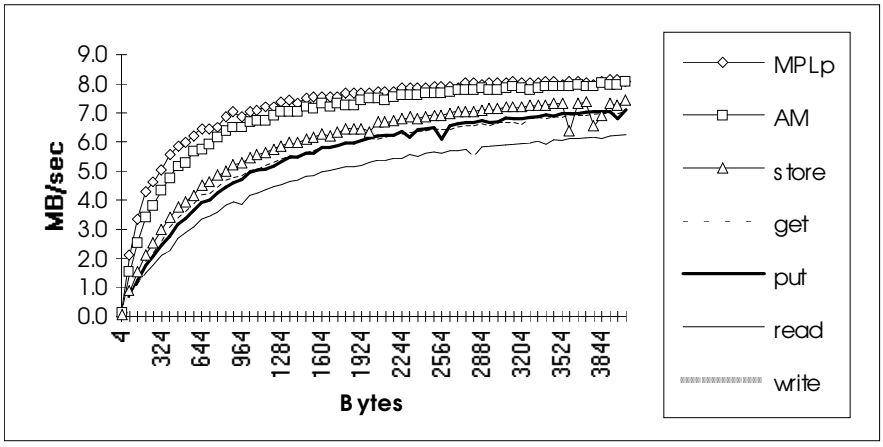

FIGURE 30. Bandwidth for message sizes up to 4KB.

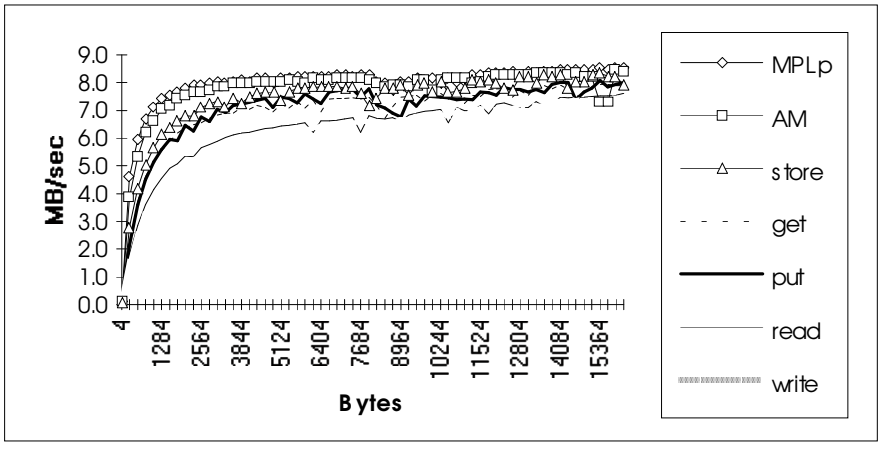

FIGURE 31. Bandwidth for message sizes up to 16KB.

Figure 32 shows the performance of global communication.

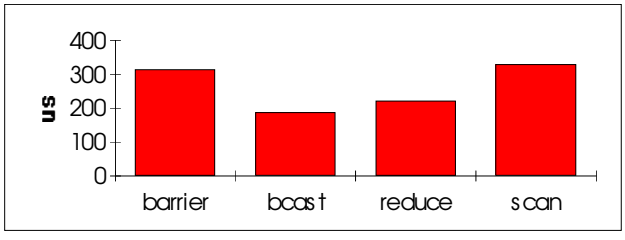

FIGURE 32. Performance of global communication.

Figures 33 and 34 show the performance of matrix multiply and radix sort for different problem sizes. For radix sort, a 16 bit radix was used. Because only an eight node SP-1 was available, speedup curves could not be generated for the SP-1.

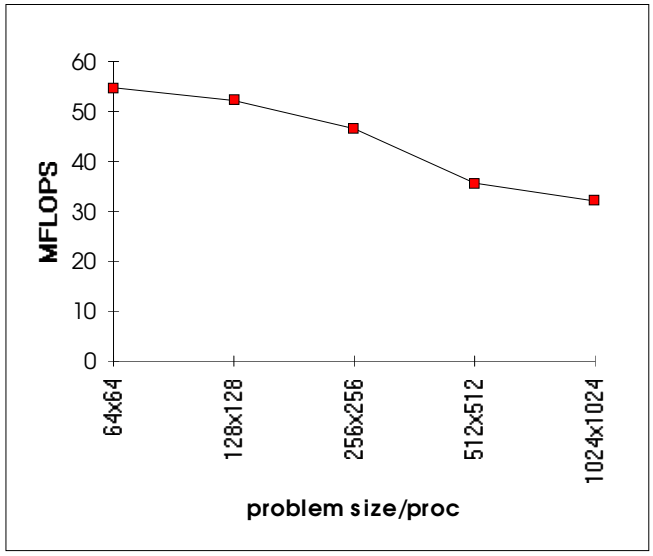

FIGURE 33. Performance of matrix multiply for different problem sizes.

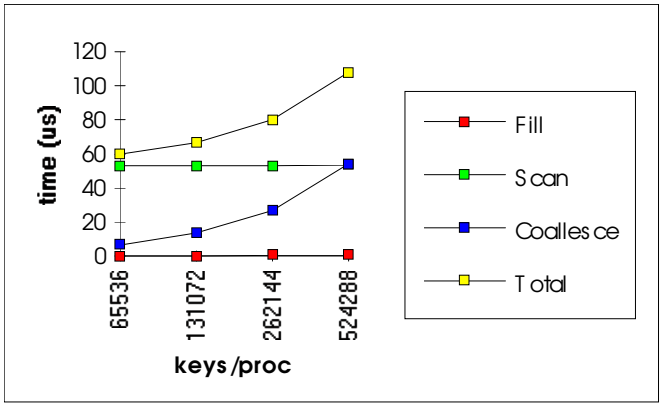

FIGURE 34. Performance of radix sort for different problem sizes.

Appendix C: Charts in Numerical Form

This appendix has tables with the numerical values of the graphs that appear in this paper.

TABLE 4. Overhead and latency of Libsplit-C on CM5 and HPs.

	CMAM	**CMAML**	**HPAM**
AM send overhead	1.6	2.1	3.2
store	2.1	2.7	3.4
AM send + recv overhead	3.4	4.3	16.5
get	4.4	5.2	16.9
put	4.1	6.8	16.9
AM latency	13.1	14.5	35.2
read	14.0	17.8	35.9
write	13.4	14.9	35.5

TABLE 5. Overhead and latency of Libsplit-C on Paragon.

	w/ msg. proc.	**w/o msg. proc.**
NX send overhead	35	61
NXAM send overhead	52	84
store	53	87
NX send + recv overhead	91	182
NXAM send + recv overhead	107	292
get	114	286
put	109	290
NX latency	164	321
NXAM latency	189	458

TABLE 5. Overhead and latency of Libsplit-C on Paragon.

	w/ msg. proc.	**w/o msg. proc.**
read	198	460
write	198	464

TABLE 6. Overhead and latency on the SP-1.

Op.	**Time (us)**
MPLp send overhead	23
AM send overhead	33.7
store	34.5
MPLp send + recv overhead	44.8
AM send + recv overhead	54.6
get	57.5
put	57
MPLp latency	68.3
AM latency	88.6
read	89.6
write	89.2

TABLE 7. Overhead of Libsplit-C pointer operations.

	CM5	**HPs**	**Paragon**
sptr[i]	1.24	0.15	0.41
for_my_1d	0.13	0.03	0.12
for_my_2D	2.27	0.82	1.65
sptr++	0.38	0.10	0.19

TABLE 8. Performance of global communication for four nodes on CM5 and HPs.

	CMAM	CMAML	HPAM
barrier	3.4	3.6	61.3
broadcast	25.2	12.5	79.5
reduce	22.3	10.5	81.1
scan	48.1	12.1	102.4

TABLE 9. Performance of global communication on 64 nodes of CM5.

	CMAM	CMAML
barrier	3.2	4.4
broadcast	68.3	13.0
reduce	65.3	10.4
scan	134.1	12.2

TABLE 10. Performance of global communication on eight nodes of Paragon.

	dataflow	barrier
barrier	705	705
broadcast	453	1597
reduce	486	951
scan	661	2039

TABLE 11. Performance of global communication on eight nodes of SP-1.

Op.	Time (us)
barrier	314
broadcast	187
reduce	221
scan	329

Appendix D: Instructions for Porting Libsplit-C to a New Message Layer

There are two steps to implementing Libsplit-C on a new message layer. First, you must determine how to implement the different components of Libsplit-C efficiently on the new message layer. Then, you must add a directory, and add and change certain files in Libsplit-C. The details of what files to change and add are described in the file PORTING.

For global memory, an active message layer is the most portable interface. The one thing to consider when implementing a new message layer on the active message interface is that this interface should match the calling convention of the processor. Floats should be passed in registers instead of going through memory. If you are implementing Libsplit-C on message passing, there is an active message layer, nxam, for the Paragon that is built on message passing. Another one, sp1am, exists for the SP-1.

For bulk transfer, you may or may not have to set up a segment before you send a message. If packet sizes are large, or if the network is ordered and reliable, the message layer will not need segments. Code from the CMAM implementation can be used as a starting point if segments are needed. Code from the HPAM implementation can be used as a starting point if segments are not needed. If you are implementing Libsplit-C on message passing, you will want to send a message to the remote node which posts a receive before doing a put or a store. This will save the message layer a copy. See the implementation on the Paragon or on the SP1 for an example.

For global communication, you should use hardware implementations if they exist. If barrier is the only function that exists, the others must be constructed using point to point messages. But, the barrier can be used for flow control. See the CMAM implementation for an example. If barrier is not supported in hardware, all global communication functions must be constructed using point to point links, and flow control must be down using techniques from static dataflow. See the implementation on HPAM for an example.

The global heap should be portable to any implementation that supports the Unix system call `sbrk()`. However, a problem will arise if the message layer calls `malloc()`. `All_spread_malloc()` performs a reduction across the processors in order to determine the maximum heap value across the processors. If the message layer calls malloc while this reduction is in progress, one of the nodes may call `sbrk()` in which case the value for the top of the heap may be incorrect. The implementation that is built on PVM fixes this problem by ensuring that the free pool will always have enough extra space just in case malloc is called from within `all_spread_malloc()`.

Atomic procedures are supported easily once an active message interface exists. Because the easiest way to implement global memory is to use active messages, atomic procedures should come for free.

References

[1] J. Bennett, J. B. Carter, W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. Proceedings of the Second ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming. March 1990.

[2] D. E. Culler et. al. Introduction to Split-C.

[3] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. Proceedings of the 19th International Symposium on Computer Architecture, May 1992.

[4] R. P. Martin. HPAM: An Active Messages Layer for a Network of HP Workstations.

[5] D. J. Scales, M. S. Lam. A Flexible Shared Memory System for Distributed Memory Machines.