Shared Memory versus Message Passing for Iterative Solution of Sparse, Irregular Problems

Frederic T. Chong*
ftchong@lcs.mit.edu

MIT Laboratory for Computer Science

The benefits of hardware support for shared memory versus those for message passing are difficult to evaluate without an in-depth study of real applications on a common platform. We evaluate the communication mechanisms of the MIT Alewife machine, a multiprocessor which provides integrated cache-coherent shared memory, message passing, and DMA. We perform this evaluation with "best-effort" implementations which solve several sparse, irregular benchmark problems with a preconditioned conjugate gradient sparse matrix solver (ICCG).

We find that machines with fast global memory operations do not need message passing or bulk transfer to support our irregular problems. This is primarily due to three reasons. First, a 5-to-1 ratio between global and local cache misses makes memory copies in bulk communication expensive relative to communication via shared memory. Second, although message passing has synchronization semantics superior to shared memory for data-driven computation, efficient shared memory can overcome this handicap by using global read-modify-writes to change from the traditional owner-computes model to a *producer-computes* model. Third, bulk transfers can result in high processor idle times in irregular applications.

Unlike previous studies [CSBS95], we have extensively studied the entire ICCG algorithm to ensure that our results are consistent with a complete, best-effort implementation. We obtain supporting data by instrumenting two state-of-the-art scientific packages: Chaco [HL95], a graph partitioner and mapper from Sandia National Laboratories; and BlockSolve [JP94], a conjugate gradient linear systems solver from Argonne National Laboratory.

1. Data Mapping (Spectral Bisection)
2. Matrix Reordering (Multicolor)
3. Incomplete Cholesky Factorization
4. Iterative Solution
   Triangular Solves (2)
   Inner Products (2)
   Vector Updates (3)
   Matrix-Vector Products (2)

Table 1: Computations for Parallel ICCG

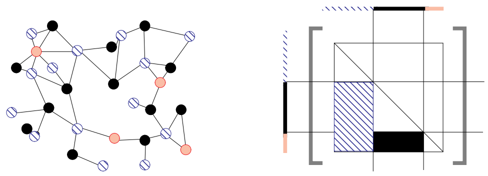

Figure 1: Graph coloring and resulting matrix structure using multicolor reordering and incomplete Cholesky factorization.

Parallel implementation of ICCG involves four computation phases, shown in Table 1. First, an input matrix is mapped onto a target architecture, usually via a graph partitioner. Second, the matrix is renumbered via a reordering algorithm to provide better convergence and parallelism properties during factorization and solution. Third, a preconditioner is computed by calculating the incomplete Cholesky factors. Fourth, using an initial guess, we iteratively arrive at a solution.

Multicolor reorderings are similar to red-black relaxation. Nodes are colored to ensure independence and then the computation works on the

*This work is advised by Anant Agarwal and is funded in part by ARPA contract N00014-94-1-0985 and in part by NSF grant MIP-9504399. Address: Room 613, 545 Technology Square, Cambridge MA 02139.

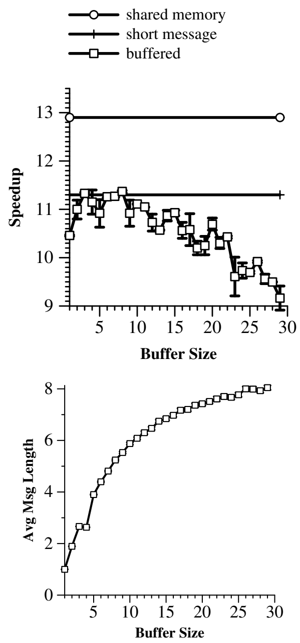

Figure 2: **(Left)** BCSSTK32 with larger buffers on 32 processors. Buffering is in terms of non-local edges of the computation. Buffering of 10 indicates that up to 160 bytes of data are sent in a message. **(Right)** Average number of edges sent per message at each buffering level for this benchmark on 32 processors.

nodes one color at a time. Specifically, the nodes of the graph are colored such that no edge connects nodes of the same color, as shown on the left side of Figure 1. Then the matrix A is renumbered such that the row and column numbers are sorted by color. After incomplete factorization, this results in triangular matrices with nonzero structure as on the right side of Figure 1. Each block of nonzeros represents a portion of computation which can proceed in parallel, dependent only upon blocks earlier in the sorted ordering of colors (blocks to the left in the lower triangular matrix).

Figure 2 shows performance for a large benchmark matrix, BCSSTK32, a finite-element mesh with 2-million edges. Communication is buffered for aggregation into large messages. The top of Figure 2 shows speedup for 32 processors as message buffering increases. For reference, speedups for shared memory and short message implementations are also plotted as horizontal lines.

The bottom of Figure 2 shows the key limitation to message buffering. As buffering increases, the average length of messages quickly reaches a limit caused by dependencies in the program. Recall that, in order to avoid deadlock and satisfy data dependencies, a message may need to be sent before filling a buffer. The amount of independent data available for aggregation is divided by p^2, where p is the number of processors. BCSSTK32 is limited to an average message length of about 8 edges (128 bytes), and consequently receives the most benefit from buffer size 8. OCEAN is similarly limited. Our smaller benchmarks have message lengths which are too small receive no benefit from aggregation.

Acknowledgements: Thanks to Fredrik Dahlgren, Matt Frank, Steve Keckler, Kathy Knobe, John Kubiatowicz, Joel Saltz, Rob Schreiber, Shamik Sharma and Deborah Wallach.

The full paper and related documents are available from http://www.ai.mit.edu/people/ftchong/

References

[CSBS95] Frederic T. Chong, Shamik D. Sharma, Eric A. Brewer, and Joel Saltz. Multiprocessor runtime support for irregular DAGs. *Parallel Processing Letters: Special Issue on Partitioning and Scheduling for Parallel and Distributed Systems*, pages 671–683, December 1995.

[HL95] Bruce Hendrickson and Robert Leland. The Chaco user's guide. Technical Report SAND94-2692, Sandia National Laboratories, July 1995.

[JP94] Mark T. Jones and Paul E. Plassman. BlockSolve v2.0: Scalable library software for the parallel solution of sparse linear systems. ANL Report (updated draft) 92-46, Argonne National Laboratory, October 1994.