From AAPC Algorithms to High Performance Permutation Routing and Sorting

Thomas M. Stricker and Jonathan C. Hardwick
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Several recent papers have proposed or analyzed optimal algorithms to route all-to-all personalized communication (AAPC) over communication networks such as meshes, hypercubes and omega switches. However, the constant factors of these algorithms are often an obscure function of system parameters such as link speed, processor clock rate, and memory access time. In this paper we investigate these architectural factors, showing the impact of the communication style, the network routing table, and most importantly, the local memory system, on AAPC performance and permutation routing on the Cray T3D.

The fast hardware barriers on the T3D permit a straightforward AAPC implementation using routing phases separated by barriers, which improve performance by controlling congestion. However, we found that a practical implementation was difficult, and the resulting AAPC performance was less than expected. After detailed analysis, several corrections were made to the AAPC algorithm and to the machine's routing table, raising the performance from 41% to 74% of the nominal bisection bandwidth of the network.

Most AAPC performance measurements are for permuting large, contiguous blocks of data (i.e., every processor has an array of P contiguous elements to be sent to every other processor). In practice, sorting and true $h-h$ permutation routing[1] require data elements to be gathered from their source location into a buffer, transferred over the network, and scattered into their final location in a destination array. We obtain an optimal T3D implementation by chaining local and remote memory operations together. We quantify the implementation's efficiency both experimentally and theoretically, using the recently-introduced copy transfer model, and present results for a counting sort based on this AAPC implementation.

1 Introduction

With the advent of new parallel machines, specialized algorithms have been developed for all-to-all personalized communication (AAPC) on all common supercomputer interconnects. There is a simple upper bound for AAPC performance since the algorithm is bisection limited, and optimal algorithms are easy to find for most network architectures. Most practical methods use *a priori* knowledge about the communication pattern, and attempt to minimize congestion and contention in the network. For further details, we refer the reader to a good description of the history of AAPC and a survey of algorithms [2].

An efficient AAPC implementation is critical, since it determines the speed of transposes and array redistributions. Furthermore, it is a significant factor in the overall performance of commonly-used algorithms such as sorting. For many modern parallel machines, the fastest sorting algorithms are based on counting algorithms (e.g., radix sorts). Again, we refer to previous surveys of sorting algorithms and implementations [3, 10, 2, 5].

In Section 2 we describe an AAPC implementation on the Cray T3D [1] that achieves performance close to nominal bandwidth. Section 3 shows how the performance of local and remote memory operations affects the end-to-end routing performance. In Section 4 we present performance results for a counting sort based on this AAPC implementation.

2 AAPC on a commercial platform

In all-to-all personalized communication every processor has a block of data to send to every other processor in the system. A simple implementation loops through a set of send and receive calls on every processor, and leaves it to the message-passing system or the network to find an adequate schedule to deliver the messages. With no synchronization, the flow control of the overloaded routing system quickly skews schedules, resulting in poor performance. To avoid this we used an optimized AAPC routing schedule for all our measurements, although we observed that if there is no synchronization between steps then there is little difference between the performance of a fixed schedule, a skewed schedule, and a randomly-permuted schedule,

On the T3D sending data from memory to the network is done by the processor, while receiving data to memory is handled by separate dedicated hardware (the deposit engine). Therefore each processor can source and sink a transfer at full speed. The T3D uses packets of up to 32 bytes sent along fixed, deterministic routes. We are measuring a balanced AAPC, so each processor sends the same amount of data to every other processor. We instrumented a simple AAPC implementation to record the performance of each individual transfer. With P processors there are P^2 of these transfers. Figure 1 shows a histogram of transfer rates over all 512×512 routes of an AAPC without congestion control on a 512-processor T3D.

In this figure we can clearly see how congestion affects the throughput (notice also the good fit to a Poisson distribution, due to the short packets and nearly random schedules). The average aggregate performance is 15,945 MByte/s, or 31.1 MByte/s per processor. This corresponds to 41% of the nominal bisection bandwidth of a

[1] Where $h = \frac{n}{P} \gg 1$, the number of elements per processor.

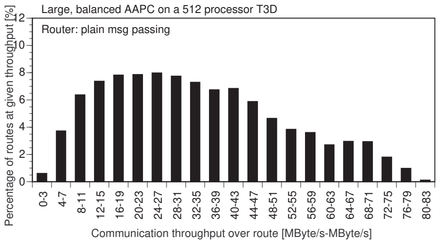

Figure 1: Histogram of transfer rates for the 512^2 routes in an AAPC algorithm without congestion control on a 512-processor T3D. Each transfer is 64 kBytes.

512-processor T3D.[2]

Figure 2 shows the performance of an improved AAPC based on congestion-free communication phases separated by global hardware barriers. We show histograms for the performance of both the machine routes and the algorithm phases. The aggregate transfer rate is 21,200 MByte/s (41.4 MByte/s per processor), or 55% of the nominal bisection bandwidth. This is very disappointing, since in theory all routes are disjoint and congestion free.

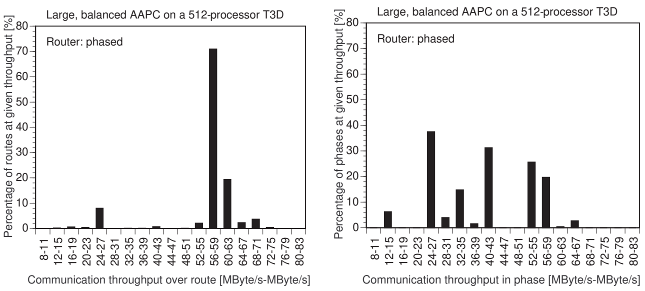

Figure 2: Histograms of transfer rates for the 512^2 routes (left), and the 512 phases (right) in a phased AAPC algorithm on a 512-processor T3D. Each transfer is 64 kBytes.

2.1 Adapting to the routing table

A closer look at Figure 2 reveals a bimodal distribution, with several routes experiencing congestion and running at about half speed. These slow routes are due to a poor interaction between the machine's routing decisions and the linear 8-node ring patterns used by the AAPC. Routing in the x and z dimension of the T3D network is done within a torus of 8 nodes, and some routes must travel exactly 4 hops. These routes can be routed either +4 on the surface or -4 through the back-loop, with bit 1 of a processor's x and z coordinates determining which way to go. The bit patterns for the particular AAPC patterns used result in the same route being chosen every time. The solution is a revision of the AAPC patterns (see [8] for details). This improves aggregate performance to 23,910 MByte/s (46.7 MByte/s per processor), or 62% of the nominal bisection bandwidth.

[2] The nominal bisection bandwidth is based on a link speed of 75 MByte/s. Experimentally, 78.9 MByte/s can be achieved for unidirectional traffic and 73.2 MByte/s for simultaneous bidirectional traffic between two nodes, each consisting of two processors.

2.2 Optimizing the routing table

Despite the modified algorithm, several routes still suffer from congestion. Specifically, routes at certain positions in the z dimension with nearby service nodes are affected (service nodes are only connected in the x and z dimensions, and are always in the "downstream" path of a regular mesh node). This is due to a flaw in the routing table: routing shortcuts should only "cut" a corner on a right turn, but the corner is also being cut on a left turn, resulting in an early corner turn in a service node rather than in a compute node. As a consequence the y link between the service node and the corner node is overloaded. This is shown in Figure 3, with a service node inserted into a 3×3 section of mesh.

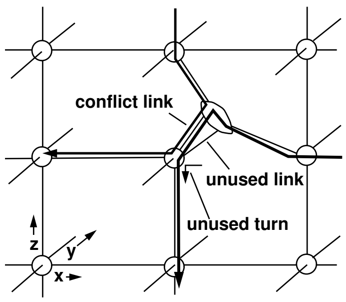

Figure 3: The effect of service nodes on corner-turning routes in the T3D's torus topology.

Eliminating the erroneous corner turn from the routing table improves aggregate performance to 27,852 MByte/s (54.4 MByte/s per processor), or 73% of the nominal bisection bandwidth (note that the modification results in a more balanced network, and that no routes get longer). However, there are still a few congested routes, due to another routing table error. Specifically, ties between surface and back-loop routes crossing service nodes are determined according to absolute distance (including the service nodes) rather than distance using regular compute nodes only. After another routing table modification we achieve close to uniform performance on all routes, as can seen in Figure 4, with 28,416 MByte/s for a 512-processor T3D (55.5 MByte/s per processor), or 74% of the nominal bisection bandwidth. Table 1 shows the evolution of performance during our optimization of the routing table and communication patterns. For smaller machine sizes we achieve a higher percentage of the bisection bandwidth, and we therefore speculate that some of the remaining 26% is due to flow-control packets.

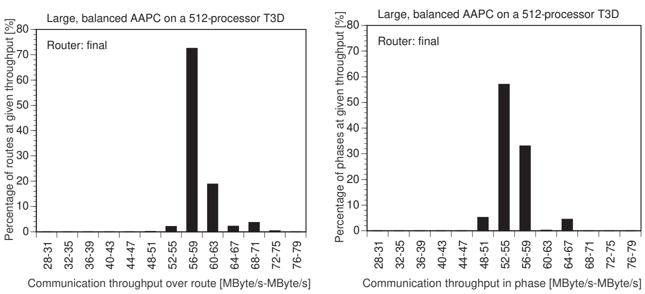

Figure 4: Histograms of transfer rates for the 512^2 routes (left), and the 512 phases (right) of a phased, machine-specific AAPC on a 512-processor T3D with a modified routing table. Each transfer is 64 kBytes.

Description	MByte/s per proc.	MByte/s aggregate	Percent of peak
No phases	31.1	15,945	41%
Phased	41.4	21,200	55%
Pattern fix	46.7	23,910	62%
Router fix	54.4	27,852	73%
Final	55.5	28,416	74%
Nominal	75.0	38,400	100%

Table 1: Performance of AAPC algorithms with different levels of optimization on a 512-processor T3D.

3 Impact of memory system performance

A true permutation routing primitive requires more than just a complete exchange of contiguous blocks of data. The data must be gathered from a source array at the sender and placed into a destination array at the receiver according to an index table specifying the layout of the permuted elements in local memory. The corresponding local memory operations are potentially very expensive.

The Cray T3D offers a choice of communication styles with which to solve this problem. We can either pack buffers locally at the sender, transfer contiguous blocks, and unpack them at the receiver, or we can use "chaining" to perform the whole operation, similar to the methods used in vector machines for memory to memory transfers.[3] A similar problem occurs for regular transposes. The tradeoffs are described in [6], which also introduces the copy-transfer model used to reason about the different internal data transfers involved in a composite communication primitive such as a full permutation AAPC.

In the copy-transfer model a permutation is defined by its memory access pattern. While the data elements are stored in a distributed array, the permutation itself is specified by a table of index pairs, where each table entry contains a source index and a destination index. Using the direct deposit model [7], synchronization and consistency are guaranteed by the use of hardware barriers, and the data transfers are performed by remote stores using the messaging system. For distributed memory systems the index relation table must be maintained in a certain order, in order to group all transfers for a given source-destination pair. As in the case of regular transposes, the correct tradeoff between packing buffers and chaining multiple gather, transfer and scatter steps together can be determined from the measured machine parameters. For the T3D we obtain two formulas, one for buffer packing and one for chained transfers, as shown below. For further details see [6].

Buffer-packing	Chained
${}_\omega\overline{Q}_\omega \;=\; {}_\omega C_1 \circ ({}_1S_0 \| N_d \|_0 D_1) \circ_1 C_\omega$	${}_\omega Q_\omega \;=\; {}_\omega S_0 \| N \|_0 D_\omega$
$\lvert{}_\omega\overline{Q}_\omega\rvert = 14.2$ MBytes/s	$\lvert{}_\omega Q_\omega\rvert = 32$ MBytes/s

The derived performance figures show that chained communication is a clear winner for random permutations (${}_\omega Q_\omega$ remote copies, where ω indicates an indexed random access to memory). Comparing transposes to permutations we find that the indexed access patterns in permutations are more expensive than the fixed strides found in transposes.[4] An interesting specialization of the general random permutations are "grouped" permutations, where the source data is pre-sorted per destination processor. Access to the source elements is now contiguous and there is no need for a gather operation. This primitive, written as ${}_1Q_\omega$ in the copy-transfer model, is a fast building block for counting sorts since the coarse local pre-sort of the data elements can be achieved with an extra store while performing the second counting step of a sorting pass. In Table 2 we compare estimated and measured performance, and calculate the fraction of nominal bisection bandwidth achieved. k is a constant stride and is assumed to be $\gg 1$.

[3] One difference between a shared-memory machine such as the Cray C90 and a distributed-memory messaging-passing machine such as the Cray T3D manifests itself in the fixed overhead to select a new communication partner. In the case of the T3D this overhead is one to two orders of magnitude higher than the time to handle a single store, and therefore a one-pass permutation algorithm ("dealing out the card deck") will not result in good performance.

[4] In transposes the indices can be computed on the fly during the communication step, while for a true random permutation both the source and destination index must be loaded from local memory.

	Model		Measured		
AAPC type	Symbol	MB/s /proc.	MB/s /proc.	Total MB/s	% of peak
Block	${}_1Q_1$	69	55.5	28,416	74 %
Transpose	${}_1Q_k$	38	29.5	15,104	39 %
Grouped	${}_1Q_\omega$	34	27	13,824	36 %
Random	${}_\omega Q_\omega$	32	22	11,264	29 %

Table 2: Measured and derived performance of different AAPC types on a 512-processor T3D.

4 Using AAPC for a T3D counting sort

To illustrate the application performance gains possible with a fast AAPC, we have developed an efficient counting sort for the T3D. This can be used as the basic step of a radix sort algorithm or can be further refined into a single-pass sample sort. Note that all the performance-critical building blocks of a counting sort are loops with non-cacheable memory operations, and that on systems with modern high performance microprocessors arithmetic operations are cheap relative to local memory and communication operations. This permits an elegant description of the counting sort algorithm in the copy transfer model in terms of local and remote memory system performance. It is instructive to use a common yardstick (i.e., MByte/s of data moved) to derive a limit on memory system performance for all components of the algorithm. We can measure these numbers with micro-benchmarks and relate them to architectural specifications. These performance limits can then be used to decide on the optimal way to compose local bucket scans, transposes, and permutations into a counting sort for a particular machine. Table 3 shows the measured and derived costs of the operations required for a counting sort on the T3D. For both k and ω access types the average stride is about half the number of buckets.

	Model			Meas.
Primitive	Work	MB/s	Symbol	MB/s
Bucket Counting	key	21	${}_\omega C_k$	14
Local Permutation	key	21	${}_\omega C_k$	8
Global Permutation	key	34	${}_1Q_\omega$	20
Bucket Transpose	bucket	38	${}_pQ_1$	20
Bucket Reduction	bucket	220	${}_1C_0$	125
Bucket Scan	bucket	90	${}_1C_1$	60
Bucket Untranspose	bucket	38	${}_1Q_p$	20

Table 3: Measured and derived costs of operations required for a T3D counting sort, assuming a uniform distribution of 16-bit keys.

The memory system behavior of the bucket counting loop consists of reading a contiguous stream of indices and accessing the bucket array. Each bucket access is a read-increment-write, resulting in a widely strided store. Since there is no additional destination index needed for the store, its access pattern is strided (k) rather than indexed (ω). The second local step performs the pre-sort necessary for an efficient grouped permutation on a distributed-memory message-passing machine. The unexpectedly poor measured performance of this local step is due to the lack of a write-back cache on the T3D. Specifically, updating the bookkeeping structures results in an extra read-increment-write to an array of size p, where p is the number of processors. Although the bookkeeping array is small enough to fit in the T3D's cache, the write-through policy of the cache forces the processor to wait for the main memory system, significantly affecting overall performance. On a machine with a write-back cache (e.g., the Intel Paragon) we would expect actual performance to be much closer to the model's prediction.

The copy-transfer model assumes that all basic steps operate on the same number of elements. This is not true for the bucket-work and key-work loops of a counting sort, and therefore the model is extended with a new parameter f, denoting the ratio between bucket work and key work. The optimal trade-off between bucket work and key work must be computed separately: an explanation of how to do this is given in [10]. For our counting sort algorithm, we assume a 512-processor T3D, one billion keys (2^{30} keys total, 2^{21} per processor), one 16-bit radix pass over 65536 buckets, and 48 bits of data attached to each key (since the DEC Alpha architecture can handle 64-bit words at the same speed as 16- or 32-bit words).

Given these parameters, f is $\frac{1}{32}$, and the bucket work is negligible. We can model the performance of counting sort as follows:

$$\frac{1}{\frac{1}{21}+\frac{1}{21}+\frac{f}{38}+\frac{f}{220}+\frac{f}{90}+\frac{f}{38}+\frac{1}{34}} = 7.88 \text{ MB/s per proc.}$$

In practice, the measured performance of a counting sort algorithm on a 512-processor T3D is 3.99 MBytes/s per processor, corresponding to 2042 MBytes/s for the full machine, or 255 million keys per second. For both the modelled and measured performance, the speed of local and remote memory accesses are the limiting factor, although the asymptotic $O(n)$ complexity of counting sort and the congestion-free AAPC routing imply linear scalability across a wide range of numbers of keys and machine sizes.

A previously reported portable implementation of radix sort written in Split-C achieves a sorting performance of 4 million 32-bit keys in just under 6 seconds on an 8-processor T3D [2], for a sorting performance of approximately 330 kBytes/s per processor. This is equivalent to two 16-bit counting sort passes, each with a memory performance of approximately 660 kBytes/s per processor. A subsequent implementation of a highly optimized sample sort, which only requires a single counting and routing pass, sorts 4 million 64-bit integers in 0.894 seconds on a 16-processor T3D, for a sorting performance of 2.24 MBytes/s per processor [4].

5 Conclusions

We have quantified the performance benefits of a global well-synchronized all-to-all personalized communication implementation, using the best-known message-passing mechanism on the Cray T3D. We have also shown the need to consider low-level engineering details, such as the choice of AAPC patterns and the particular routing table used, in order to fully utilize the network of a commercial machine. Our research raised the efficiency of AAPC from 41% to 74% of the nominal network bisection bandwidth. In the process we realized that AAPC is not limited to a complete exchange of contiguous data blocks, and that there are general permutations of elements involving expensive local memory operations. We used the copy-transfer model, a simple throughput-oriented model of memory system performance, to derive an upper performance bound for permuting and radix sorting on the Cray T3D. Analyzing the performance in terms of the architecture of the memory and communication systems, it is apparent that the bottlenecks lie in the memory system rather than in the communication network or the combinatorial aspects of the algorithm. We implemented a counting sort algorithm that approaches these performance bounds, sorting a billion words (16-bit key plus 48-bit data) in 4 seconds on a 512-processor T3D.

Acknowledgements

We would like to thank Steve Scott of Cray Research for his qualified and friendly assistance with the construction of routing tables and Ken Mortensen for installing them. Access to a 512-processor Cray T3D was provided by the Pittsburgh Supercomputing Center, whose staff gracefully dealt with all our requests.

References

[1] R.H. Arpaci, D.E. Culler, A. Krishnamurthy, S.G. Steinberg, and K. Yelick. Empirical evaluation of the Cray T3D. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 320–331, Portofino, Italy, June 1995.

[2] D. Bader, D. Helmann, and J. JaJa. Practical parallel algorithms for personalized communication and integer sorting. Technical Report TR-CS-95-3548, UMIACS, University of Maryland, College Park, MD 20742, December 1995.

[3] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.. Plaxton, S.J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. Symp. Parallel Algorithms and Architectures*, June 1991.

[4] D. Helmann, D. Bader, and J. JaJa. A parallel sorting algorithm with an experimental study. Technical Report TR-CS-95-3549, UMIACS, University of Maryland, College Park, MD 20742, December 1995.

[5] T. Stricker. Supporting the hypercube programming model on meshes; (a fast parallel sorter for iwarp). In *Proc. Symp. Parallel Algorithms and Architectures*, pages 148–157, San Diego, June 1992.

[6] T. Stricker and T. Gross. Optimizing memory system performance for communication in parallel computers. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 308–319, Portofino, Italy, June 1995.

[7] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proc. Intl. Conf. on Supercomputing*, pages 1–10, Barcelona, July 1995.

[8] Thomas M. Stricker and Jonathan C. Hardwick. From AAPC algorithms to high performance permutation routing and sorting. Technical Report CMU-CS-96-120, School of Computer Science, Carnegie Mellon University, April 1996. To appear.

[9] A. Thakur, R. Ponnusamy, A. Choudhary, and G. Fox. Complete exchange on the CM-5 and Touchstone Delta. *Journal of Supercomputing*, (8):305–328, 1995.

[10] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings Supercomputing '91*, pages 712–721, November 1991.