Unstructured Grids on SIMD Torus Machines

Petter E. Bjørstad

Institutt for Informatikk
Høteknologisenteret
University of Bergen
N-5020 Bergen
Norway

Robert Schreiber

Research Institute for
Advanced Computer Science
MS T27A-1
NASA Ames Research Center
Moffett Field, CA 94035

Abstract

Unstructured grids lead to unstructured communication on distributed memory parallel computers, a problem that has been considered difficult. Here, we consider adaptive, offline communication routing for a SIMD processor grid. Our approach is empirical. We use large data sets drawn from supercomputing applications instead of an analytic model of communication load. The chief contribution of this paper is an experimental demonstration of the effectiveness of certain routing heuristics. Our routing algorithm is adaptive, nonminimal, and is generally designed to exploit locality. We have a parallel implementation of the router, and we report on its performance.

1 Introduction

The subject of this paper is the implementation of unstructured communication on highly structured parallel machines. In particular, we investigate the implementation of distributed gather and sparse matrix-vector multiply operations on a two-dimensional, toroidal, SIMD processor grid with local memory; our testbed is the Maspar MP-2.

In a gather operation, a distributed data vector, the *destination* vector DEST is created by transferring values from a distributed *source* vector, SRC, according to distributed index vectors. For purposes of discussion, we adopt the following notation. If X is a distributed data vector, then the element of X stored at processor p and at offset k within that processor is denoted X$[p;k]$. We assume that processor p stores NSRC$[p]$ elements of SRC and NDEST$[p]$ elements of DEST, and two integer vectors of length NDEST$[p]$ — HOME and OFFSET. The value to be placed in DEST$[p;k]$ (on processor p) is obtained from SRC[HOME$[p;k]$; OFFSET$[p;k]$]. Thus, the gather operation may be written:

$$\mathrm{DEST}[p;k] \leftarrow \mathrm{SRC}[\mathrm{HOME}[p;k];\mathrm{OFFSET}[p;k]],$$

for all $(p;k)$ with $1 \leq p \leq$ NPROC and $1 \leq k \leq$ NDEST$[p]$.

The reason for our interest in this primitive and its SIMD machine implementation is that it is essential for the efficient execution of sparse matrix-vector products. The problem of multiplying a sparse, often unstructured matrix with a sequence of vectors arises when using iterative methods for the solution of sparse linear systems and also when solving the corresponding eigenvalue problem. In such methods, the matrix-vector multiplication is a key operation. The matrices are often very large; those arising from discretized partial differential equations in three space dimensions are exemplary in this respect. When the grid that leads to the sparse system is a rectilinear mesh, implementation of matrix-vector product involves simple, nearest neighbor communication on a processor grid. But when the mesh is unstructured, gather operations such as we have described are necessary.

Significant early studies of this problem used the Connection Machine CM-2 as a testbed. In their studies, Hammond [3] and Dahl [2] concluded the following.

1. Good performance can be obtained by assigning rows of the matrix to processors. Row r is stored by processor MAP$[r]$. The vectors X and AX, which store the vector and the result of the matrix-vector product, are mapped such that AX$[r]$ is stored by processor MAP$[r]$, and, in general, X$[r]$ is stored at processor MAPX$[r]$.

In the solution of systems and the computation of eigenvalues, the coefficient matrix A is ordinarily square, and then it is appropriate to insist that MAP be the same as MAPX, since the vectors X and AX undergo elementwise dot product and *axpy* operations which require that they be aligned. In the rest of this paper, we shall assume that A is square, and that MAPX is the same as MAP; the techniques we describe, however, clearly extend to the general case.

2. The mapping can be chosen a priori, as a function of mesh topology, to minimize the cost of the matrix-vector product. This mapping step has been studied extensively, and many effective heuristic methods, such a spectral recursive bisection, simulated annealing, Kernighan-Lin, and Cyclic Pairwise Exchange have been proposed [5].

3. Given such a mapping, the implementation of the matrix-vector product can be reduced to a gather operation (processor p gets X[j] from processor MAP[j] for all j such that A[i,j] is a nonzero mapped to processor p) followed by a communication-free sparse dot product operation.

4. Dahl and Saltz, *et al.* show that naive use of online message routing is inefficient for gather and sparse matrix-vector operations, especially in coarse-grained parallel applications, in which many matrix rows and vector elements are mapped to each processor. In this, case, by preprocessing, one can accomplish several important optimizations. First, redundant communication can be eliminated, and communication between processor pairs can be amalgamated (so that only one, long message is sent) [8]. Second, one can achieve communication performance better than that provided by the online machine communication layer by heuristically optimized, offline routing [2].

Sparse matrices arising in other contexts may not be amenable to this general approach. If the graph of A does not have small separators, a mapping providing the locality we require is not possible. A second approach, which has been advocated and implemented by Hendrickson, Leland, and Plimpton [4]; Lewis and van de Geijn [7]; and Ogielski and Aiello [9] is quite competitive to the approach taken here. This is also the right idea for dense matrices (complete graphs have big separators). We compare this alternative approach with ours in a later section.

Our experimental implementation is on the Maspar MP-2. The MP-2 has two separate hardware communication mechanisms: the router and the Xnet. The router is a low-bandwidth interconnect that implements arbitrary transfers of data from the processors to a permutation of the processors. The Xnet is a high-bandwidth toroidal shift network that permits all processors simultaneously to send a datum to their neighbor at a distance d in a certain direction, the same direction for all the processors. When $d = 1$, the Xnet bandwidth is 16–50 times greater than the router's. This speed disparity seems to be a feature that SIMD architectures have in common. The CM-2, in particular, had it even though it used the *same* hardware (*i.e.* wires) for both toroidal shift and general communication. For this reason, we have attempted, by using heuristic optimizations to predetermine the routing and scheduling of the data, to perform gather operations as a sequence of toroidal shifts.

The problem of packet routing in interconnection networks, including grids and tori, has been heavily studied in the theory of parallel algorithms [1, 6]. We feel this study adds to what is known, because the routing problems we study are drawn from practice, and we report on the actual performance of out proposed methods on those problems. The problems differ from those studied in that they are nonrandom, have enormous locality, are not permutations, and do not route one packet from each processor. Because we consider problems of large granularity and use offline methods, we expect to achieve better performance.

We began this effort because we felt that the notion that SIMD grid architectures were not usable for less than perfectly well structured computations was wrong, and that the key to making them useful was to use the structured, fast grid communication rather than the fully general router. We believe that this study, while it does not definitively resolve this question, does make that viewpoint more plausible.

2 The Maspar architecture

The MP-2 is comprised of three subsystems: a front-end workstation, the Array Control Unit (ACU), and the Processing Element (PE) Array. The ACU is a 32-bit, custom integer RISC that stores the program, the instruction fetching and decoding logic, and is used for scalars, loop counters and the like. It includes its own private memory.

Performance of the MP-2

Operation	Cycles	Ops/sec (16K machine)
Xnet[1]	179 Kw/s	2.9 Gw/s
Floating mult-add	144 Kops/s	2.4 Gflops/s
Router (approx)	6.3 Kw/s	103 Mw/s
Load	178 Kw/s	2.9 Gw/s
Indirect load	89 Kw/s	1.5 Gw/s

The PE array is a two-dimensional mesh of processors. Each processor may communicate directly with its eight nearest neighbors. The processors at the edges of the mesh are connected by wrap-around channels to those at the opposite edge, making the array a two-dimensional torus. Each PE in the array is a RISC processor with 64K bytes of local memory. All PEs execute the same instruction, broadcast by the ACU.

The hardware supports three communication primitives: front-end — PE array communication; nearest neighbor communications among the PEs; and communication in arbitrary patterns through a hardware global router. The nearest-neighbor connection, called the Xnet, has a bandwidth of one bit per machine clock. The bandwidth of the router is at best one-sixteenth of a bit per clock per PE (sixteen PEs share one connection into the router) and it can drop by a factor of two to three due to congestion at internal nodes of the router's network. The communication primitives are expressed in the instruction set as synchronous, register-to-register operations. This allows interprocessor communication with essentially no latency and high bandwidth.

The MP-2 uses 32-bit hardware integer arithmetic, with microcode for higher precision and floating-point operations. All operations occur on data in registers; only load and store instructions reference memory. The machine has a peak performance close to 2.4 Gflops Mflops using 64 bit IEEE arithmetic.

The Maspar is programmed in either a data parallel C called MPL, or in Maspar's subset of Fortran 90 (which includes a FORALL statement). We have implemented the preprocessor in Maspar Fortran. The code for accomplishing the communication and the matrix-vector product (*autopilot Subway*) is in MPL. Variables in Maspar Fortran can be declared to reside on the front-end, or on the processor array. The axes of array variables stored on the processor array may be mapped to either of the machine dimensions or to memory.

3 The Subway router compiler

Our software router is called *Subway*. The Subway router compiler determines a sequence of synchronous toroidal shift communications on a processor grid that implements a given gather operation. As the name hints, we use a highly structured communication system, an urban subway system, as a conceptual model of the communication tasks and the hardware resources at our disposal. In this system the data to be moved are passengers, the Xnet wires are the subway tracks and a specific Xnet instruction corresponds to a train departure. The number of Xnet operations is a very good measure of the communication time in our system. Thus, our object is to move all the data with as small a number of Xnets (train departures) as possible.

We first describe the input to and output from Subway, and indicate how that output is subsequently used to move data. The three distributed arrays, NDEST, HOME, and OFFSET are the necessary inputs to Subway. Subway determines the number, NSEND, of Xnet operations required to accomplish the gather. In the arrays DIRECT and DIST, of length NSEND, it records the direction (North, South, East, West, Northeast, Northwest, Southeast, or Southwest) that each train travels, and the distance it travels. These three variables reside on the front-end. They completely specify the sequence of Xnet operations required.

To complete the schedule, Subway also generates two arrays of PE memory addresses, both of which are distributed. LOAD_ADDRESS$[p;k]$ specifies the memory address from which data are loaded into a register prior to the Xnet operation on that register, on processor p at cycle k, for $1 \leq k \leq$ NSEND. STORE_ADDRESS$[p;k]$ specifies the memory address to which data are stored after they move, on processor p at cycle k, $1 \leq k \leq$ NSEND.

3.1 Autopilot subway

With the tables generated by Subway, a gather operation can be carried out rapidly, as follows. All elements of SRC that are actually going to be sent to other processors are moved to a single distributed array PLATFORM, which is used to hold the data as it moves through the system. The initial locations on PLATFORM are also determined by Subway and stored in INIT_ADDRESS:

```
Do k = 1 to maxval[NSRC]
  forall (p = 1 : NPROC)
    PLATFORM[p; INIT_ADDRESS[p; k]]
      = SRC[p; k];
Enddo
```

Then, the data are moved:

```
Do cycle = 1 to NSEND-1
  forall (p = 1 : NPROC)
    TRAIN[p] =
      PLATFORM[LOAD_ADDRESS[p; cycle]];
  Circular shift (Xnet) TRAIN toroidally in
    direction DIRECT[cycle],
    distance DIST[cycle];
  forall (p = 1 : NPROC)
    PLATFORM[STORE_ADDRESS[p; cycle]]
      = TRAIN[p];
Enddo
```

At the completion of this loop, all passengers have arrived at their destinations. Their locations on PLATFORM are known. For completion of the gather operation, the data now have to be gathered locally from PLATFORM. Another distributed array, FINAL_ADDRESS, tells where to find this data:

```
Do k = 1 to maxval[NDEST]
  forall (p = 1 : NPROC)
    DEST[p; k] =
      PLATFORM[p; FINAL_ADDRESS[p; k]];
Enddo
```

Thus, the routing task is to determine the scalar NSEND, the front-end arrays DIST and DIRECT, and the four distributed arrays of PE memory addresses.

3.2 The router compiler

Define a passenger as a four-tuple $< x, y, dx, dy >$ where (x, y) is the passenger's current station, and $(x + dx, y + dy)$ is the passenger's final destination.

In the discussion below, *train* denotes a particular distance/direction pair; *station* denotes a processor, identified by its coordinates (x, y), *e.g.* station $(3, 4)$. The *velocity* of a train is its direction, (u, v); its *speed* is $\mid (u, v) \mid$ A speed-one train is a *local* and a higher speed train is an *express*. A *departure* describes a particular run of a train; the *load factor* of a departure is the fraction of stations that have loaded passengers; the load factor of a train is its integrated (over all departures) load factor. The *distance* for a given passenger $< x, y, dx, dy >$ is $\mid (dx, dy) \mid$ which is the minimum number of Xnet hops to reach the destination, namely, $\max(|dx|, |dy|)$. The *offset* of a passenger is her (dx, dy) pair.

The basic data structure used by Subway is a distributed collection of stacks. For each station, there is a stack for every train. In a given stack frame, we can store a passenger's index and the offset (dx, dy) to her destination. In addition, we have a distributed array of stack pointers.

The overall outline of Subway is this.

First, the collection of passengers is determined. One ticket is created for each passenger, at the passenger's home station, giving the relative distance to her destination. Redundant communication is not generated; even if two or more matrix nonzeros from the same matrix column are mapped to a given processor, only one ticket for the corresponding vector element is written.

The distributed data needed to create tickets is not the same as the distributed data structure that holds a sparse matrix or the distributed arrays HOME and OFFSET described above. These, in fact, are stored at the destination rather than at the source processor of each passenger. In order to create the tickets for matrix-vector products, we need, at processor MAP[j], the structure of column j of the matrix. Thus, a general-purpose implementation of Subway must do a sparse matrix transpose (of the matrix structure but not the elements). Our experimental data, however, only includes matrices whose structure is symmetric. We have therefore not implemented the transpose, which would add to the preprocessing time.

Next, the router module *which-train* determines, for each passenger, the distance and direction of the first train on which that passenger will ride.

The passenger then stacks for the train (*i.e.* she is pushed onto the stack for her first train at her home station.)

Now the system is ready to run. It continues, while there are any passengers in the system who have not arrived at their destinations, with the following steps:

1. (*Next-Departure*) By examing the sizes of all the stacks (*i.e.* the distributed demands) for trains, pick the next train to run. Call this the *active* train.

2. (*Load-Train*) Pop the stack for the active train at each station. The fraction of nonempty stacks is the instantaneous *load factor* for this departure.

3. (*Run-Train*) Move the passenger data (by a toroidal shift) the characteristic distance and direction of the active train. Decrement (by the velocity of the train just run) the offset of each passenger.

4. (*Unload-Train*) Place arriving passengers (those whose offset is now zero) in the arrival hall at their current station. For transit passengers (offset not yet zero) use *which-train* to determine their next train. Stack them at their current stations.

As passengers move through the system, their movements are recorded, in order to allow Subway to generate the various routing table data structures mentioned above.

3.3 Routing heuristics

We first introduce some notation and terminology.

The most important aspect of Subway is its routing heuristic, *which-train*. When a passenger $< x, y, dx, dy >$ first arrives at station (x, y), *which-train* is used to determine the distance and direction that this passenger will travel on its next trip. For example, *which-train* may decide that passenger $< 2, 3, 4, 2 >$, currently at station $(2, 3)$ and bound for station $(6, 5)$, which is at a distance of four from $(2, 3)$, will leave station $(2, 3)$ on a Northeast bound express, speed two, and arrive next at station $(4, 5)$, becoming passenger $< 4, 5, 2, 0 >$.

3.4 How *which-train* works.

Which-train is a rule-based system, in which the heuristics of Subway are all encoded.

1. Parity.

The Maspar system is like a chessboard. Xnet communication in the diagonal directions is like the move of a bishop. Using such communication, one may move from white squares only to other white squares. Communication in the Cartesian directions allows a passenger to reach any destination. We say that the *parity* of passenger $< x, y, dx, dy >$ is even or odd depending on whether $dx + dy$ is even or odd. A passenger can reach her destination without the use of Cartesian trains only if she has even parity.

Diagonal trains are faster than Cartesian trains, however, in the following way. A given passenger of even parity can always reach her destination in the smallest possible number of hops, on diagonal trains exclusively. For example, to go East a distance eight, one may go four to the Northeast and then 4 to the Southeast. (And there are many other shortest diagonal paths). But the converse is false. To go four to the Northeast requires eight trips on Cartesian local trains.

For this reason, we only use local (distance 1) Cartesian trains. We use them only for odd-parity passengers, and odd-parity passengers must take them, thereby changing to even parity. Moreover, we only run the Cartesian trains, (in round-robin fashion) allowing no diagonal trains to run, until all demand for them is exhausted. This builds up the queues for the diagonals, resulting in greater load factors.

2. Adaptive and nonminimal routing.

Aside from parity, the primary determining factor in train choice is: "Does the train go in the direction I am headed?" There may not be a unique choice: an eastbound passenger may use a Northeast or a Southeast train. Define a train to be *direct* for a passenger if a trip on that train reduces the passenger's distance by the speed of the train. Our first criterion then is that we use direct trains.

Our strategy, however, is adaptive. First, when there is more than one direct train for a passenger, *which-train* puts the passenger onto the least crowded of the stacks. Thus, it looks at traffic information available locally, at the passenger's current station.

After trying this limited adaptive routing strategy, we observed that, late in the rush hour, trains ran with very low load factors because of one station that still had passengers trying to leave. By the use of simple visualization tools, we observed that certain stations tend to be far more crowded than average[1]. Thus, we decided that a strategy that requires direct trains is not optimal, and we experimented with more flexible, nonminimal routing.

In a distributed implementation, what is immediately available to *which-train* in order to route adaptively is the number of passengers in each local stack. The adaptive strategy we use allows passengers to take an indirect route. We choose an indirect train t_{ind} instead of a direct train t_{dir} if

$$\rho(|stack(t_{dir})| - \alpha) > |stack(t_{ind})|.$$

We experimentally determined that $\alpha = 3$ and $\rho = .65$ produced good results.

[1] Veteran users of subways will not be terribly surprised by this observation.

3. Express trains

Our first experiments with expresses (Xnet[k] operations, for $k > 1$) were disappointing. We found that including expresses in the system increased the number of Xnets. The reason is that expresses ran often, and their load factors were very low. We later decided on a further adaptive strategy. In the experiments reported in Section 4 we ran diagonal trains of speeds one, two, four, and eight. After all parity issues are resolved with speed one Cartesian trains, the diagonal trains run. A passenger takes the fastest diagonal train she can, but will not "overshoot" her final destination.

Initially only speed eight trains run. They run until their load factors drop below a threshold — ten percent in the experiments. Then speed four trains run, etc. This was the best system we have devised. The reason that fast trains are useful is that the time for a cycle is dominated by the indirect load and store, and this cost is insensitive to Xnet distance. Thus, as long as their load factors are high enough, expresses are worthwhile.

In general, *next-departure* runs trains of a given class (Cartesian local or diagonal speed four, for example) round-robin, skipping any for which there is no demand anywhere, until that train class is shut down.

4 Experimental results

Our experiments employ a collection of grids used by practitioners in finite element and finite volume method solutions of partial differential equations. One is a two-dimensional unstructured triangular mesh covering the flow field in the neighborhood of an airplane wing section. The airfoil has multiple elements, and the grid is highly nonuniform, with a mesh length ratio of several orders of magnitude. Another, *bracket* is a discretization of a machine part using tetrahedra. The last is a tetrahedral decomposition of the space around a full airplane, the Lockheed Viking.

The collection of grids was assembled by Steven Hammond in an earlier RIACS research effort. That effort [3] resulted in the development of a grid mapper, which determines the array MAP by a heuristic optimization strategy. Hammond's method is designed to load balance the vertices and, subject to that load balance constraint, to minimize a measure of communication complexity. The measure is given by

Grid	Dimensions	Vertices	Edges	$\Lambda/\lvert E\rvert$
3ELT	2	4720	27444	0.82
BRACKET	3	62631	733118	0.32
VIKING	3	156317	2118662	

Table 1: Grids employed.

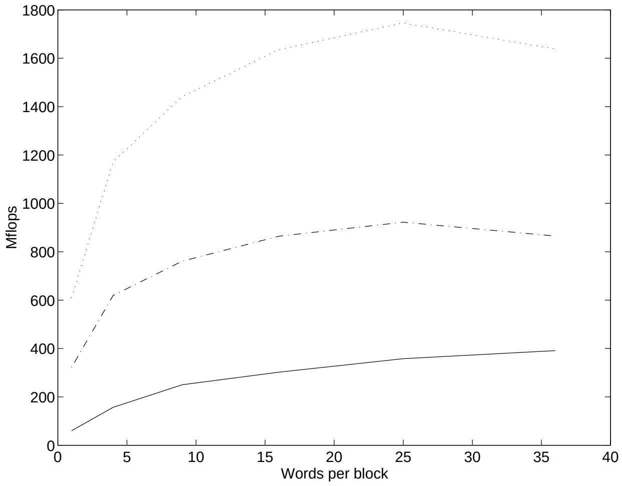

Figure 1: Performance on bracket, 16K processors

$$\Lambda(MAP) \equiv \sum_{\text{mesh edges } (i,j)} \mathrm{dist}(\mathrm{MAP}(i), \mathrm{MAP}(j)),$$

where dist(p, q) denotes the distance, in the machine, between processors p and q. Thus, Λ, whose units are bit-hops, when divided by the total machine bandwidth, is a lower bound on communication time. Hammond's experiments with the CM-2 using Dahl's communication compiler showed that Λ was a reasonably faithful predictor of communication time.

Some statistics of the grids, and their mapping to a 1K-processor machine, are given in Table 1. The last column is the average dilation per grid edge. Clearly, we have substantial locality in these mappings.

First, we give the achieved performance on matrix-vector product. We assume that the matrix structure consists of a sparse collection of small dense blocks of a given size. We tried block dimensions of from one up to five (for Viking) or six (for bracket). Figures 1 and 2 show performance in Mflops as a function of block size. (The horizontal axis is the square of the block dimension — the number of nonzeros in a block.) The lowest of the three curves gives actual performance.

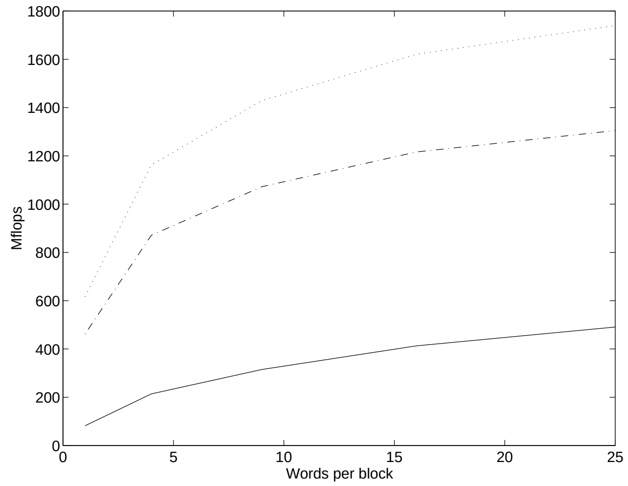

Figure 2: Performance on Viking, 16K processors

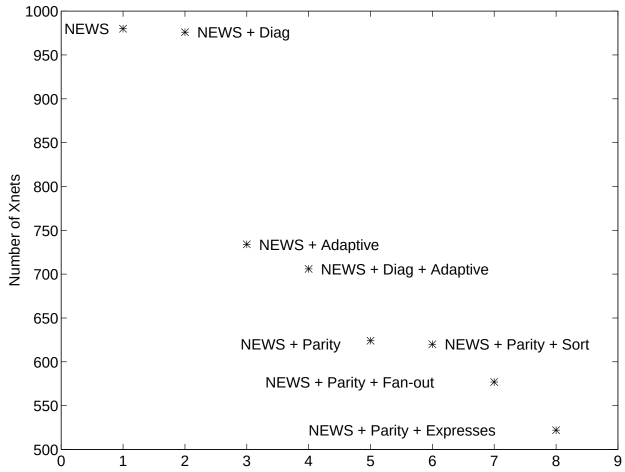

Figure 3: Xnets for different strategies; bracket on 1K processors

The middle curve gives the performance of the local dot products — the communication time has been ignored. The topmost curve is the local dot product performance rescaled to eliminate load imbalance effects. ¿From these data we see that load balance is fairly poor — the mapper should be improved to distribute the work rather than the number of matrix rows evenly. Once that is done, the local performance is a reasonably high fraction of peak. Due to sparsity, we are forced to use indirect loads in the dot product code, which slows it down. Most important, we see that for larger block sizes, we obtain 500 Mflops. Communication time is still greater than the computation time. While this performance is already interesting, we believe it can be improved, perhaps by as much as a factor of two, with further improvements to our implementation.

We next illustrate the effect (on the number of train departures) of some of the heuristics discussed above. Figure 3 gives several data points. All data points except the lowest use local trains (speed one) only. The two uppermost use either Cartesian only or Cartesian and diagonal without the parity heuristic discussed in Section 3.4, and without nonminimal adaptivity. The two points below those show the effect of the nonminimal adaptive routing. The NEWS+Parity data point shows the added effect of the parity optimization in addition to the nonminimal adaptive routing. The Sort data point shows the negligible effect of sorting the stacks initially (after passengers are stacked at their home stations) in "longest trip goes first" order. The Fan-out data point shows what happens when a single SRC value is needed by multiple remote processors, and we allow it to be delivered via a broadcast tree instead of separate messages. For this, we had to modify our data structures so that a single passenger may have multiple destinations. Finally, the best data point was obtained by using fan-out and express trains as described in Section 3.4.

We also implemented an optimized code for use of the Maspar router. It removes redundant communication as does Subway. It also makes some attempt to schedule the use of the router to avoid destination congestion. We compared Subway with this approach and with an implementation of the Ogielski/Aiello technique (which is Xnet based, like Subway). We used bracket on 1K, 4K, and 16K machines as the test. Both Xnet based techniques are considerably faster than the router. Subway is also better than the Ogielski/Aiello technique (which is meant for random sparsity). The greater the locality of the mapping, the greater its advantage. In particular, for bracket on a 16K machine, where there are only four vertices per processor on average, the two methods are close.

4.1 Preprocessing times

Our sequential Fortran implementation of Subway takes 206 (resp. 1262) seconds for bracket (resp. Viking) on a Sparc 10, simulating a 1K processor Maspar. For comparison, 100 iterations of the conjugate gradient method using autopilot Subway (with block size one) for the matrix-vector product takes 4.5 (resp. 8.3) seconds on the MP-2. Fortunately, Subway is essentially an embarrassingly parallel algorithm: it runs in 10.8 (resp. 42.8) seconds on the MP-2. Thus, preprocessing time is modest. Our Maspar Fortran implementation of Subway is straightforward and we have made no attempt at optimization.

The space needed for the router tables is considerable. It is 12 Mbytes for bracket on a 1K Maspar. For comparison, the matrix itself requires 6.4 Mbytes. We have, however, also made no effort to reduce the memory required (by using short integers, for example.) Moreover, the table size and the preprocessing time are independent of the block size. Thus, the time and space required do not appear to us to be a critical issue for nonadaptive grid applications.

5 Conclusions

Many questions remain. Charles Leiserson has pointed out that distance-2 Cartesian is to distance-1 diagonal as distance-1 diagonal is to distance-1 Cartesian. This opens a whole new area for investigation. The optimal strategy for choice of trains is far from clear.

The measure of congestion to use in adaptive routing is also not clear. That ours works is interesting, but others, less local than ours, may be better.

We have deferred work on our mapper, which clearly needs to be improved in two ways. First, load balance based on vertex count is too crude. Second, we may, on the basis of the Subway data, be able to define a more accurate predictor of communication time. The predictor used now takes no notice of heavy local traffic, which seems to be critical.

Whether offline routing is useful for adaptive grid codes remains to be seen.

Acknowledgements

This work was supported by NASA under Contract NAS 2-13721 between NASA and the Universities Space Research Association (USRA). We would like to thank Frederik Manne for giving us his highly-tuned MPL version of the Ogielski/Aiello method, and Steven Hammond for his help in mapping the grids.

References

[1] K. Bolding and L. Snyder. Mesh and torus chaotic routing. *Proc. Advanced Research in VLSI and Parallel Systems Conf.*, March, 1992.

[2] E. D. Dahl. Mapping and compiled communication on the Connection Machine system. In D. W. Walker and Q. F. Stout, editors, *Proc. Fifth Distributed Memory Computer Conference*, Charleston, S.C., April, 1990. IEEE Computer Society Press.

[3] Steven Warren Hammond. *Mapping Unstructured Grid Computations to Massively Parallel Computers.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY. 1992.

[4] Bruce Hendrickson, Robert Leland, and Steve Plimpton. An efficient parallel algorithm for matrix-vector multiplication. Sandia National Laboratory Tech. Report SAN92–2765 · UC–405. March 1993. *Intl. J. High Speed Comput.*, to appear.

[5] Bruce Hendrickson and Robert Leland. The Chaco user's guide. Sandia National Laboratory Tech. Report SAN93–2339 · UC–405. November 1993.

[6] F. Thomas Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufman, San Mateo, CA, 1992.

[7] John G. Lewis and Robert A. van de Geijn. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. *Proc. Supercomputing '93*, pp. 484–492. IEEE Computer Society Press, 1993.

[8] R. Mirchandany, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. *Proc. Second Intl. Conf. on Supercomputing,* pp. 140–152. St. Malo, France, July 1988.

[9] A. T. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. *SIAM J. Scient. and Stat. Comput.* 14 (1993), pp. 519–530.