Hypercube Algorithms on the Polymorphic Torus

Anne C. Elster
Hungwen Li

TR 89-1003
May 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This research was partially supported by IBM agreement 12060043.
Hypercube Algorithms on the Polymorphic Torus

Anne C. Elster
Cornell University

Hungwen Li
IBM Almaden Research Center

March 1989

Abstract

The Polymorphic Torus architecture is a reconfigurable, massively parallel fine-grained system, which in its two-dimensional \( (N^2) \) case has a lower wiring complexity than, say, hypercubes. However, due to the dynamic connection features at run-time, it allows several parallel structures such as trees, rings and hypercubes to be emulated efficiently.

In this paper, we consider algorithms that are especially well-suited for hypercubes, i.e. algorithms that take advantage of the relatively high connectivity of the hypercube topology, and show how these algorithms attain comparable bounds on a 2-D Polymorphic Torus. In particular, algorithms for dense matrix vector multiplication (including using 2 orthogonal trees for the matrix-vector case), sparse matrix-vector multiplication, and the FFT are discussed.

1. Introduction

As the demand for processing power increases faster than the advancements in semiconductor technology, highly parallel computers are becoming a more and more integral part of scientific computation. Significant speed-ups, in addition to increased performance due to technological advancements, may be obtained by distributing the computational load among several interacting processors.

The scheme for highly parallel systems is usually to let each processor have its own local memory. In this case, partial results need to be passed as messages among the processors in order to collect/compute the final answer(s). Since this communication time, at least with today’s technology, is significantly longer than a basic operation, say, scalar multiplication, care must be taken to minimize it as much as possible. Also here, a second level of parallelism may be introduced by introducing vector-processors at each processing unit.

Most distributed memory machines are commonly classified according to Flynn’s taxonomy [1,2] as either Multiple Instruction Multiple Data (MIMD) machines or Single Instruction Multiple Data (SIMD) machines. In an MIMD machine each processing unit is capable of executing its own program. The program usually includes synchronization operations to resolve inter-processor data dependencies. SIMD machines, on the other hand, each have a copy of the same program. Each statement may or may not be executed at an individual processor depending on its internal state. Thus the processing units may not execute the same code, but are nevertheless lock-stepped instruction-wise (processors not executing an instruction must wait for the processing units that do execute it). This taxonomy may, of course, also be applied to shared memory systems.

Maresca, Lavin, and Li [3] further distinguish between non-autonomous and autonomous systems, types of autonomy, network topologies, and data-widths as they classify several popular parallel systems. Other considerations are parallelisms introduced by pipelining [4], and hierarchical systems such as the Multicluster framework [5], an MIMD system which may have special purpose SIMD processors attached.

Although commercial parallel machines based on vector processors or SIMD processor array schemes have been available for over 10 years, it is only recently that commercial machines involving hypercube interconnection schemes have emerged. Hypercubes are of special interest since they can efficiently emulate other common parallel topologies like the mesh, ring, tree, etc [6]. This allows for efficient implementation of several types of parallel algorithms making such systems very versatile.

A hypercube interconnection network consists of \( n = 2^k \) processing elements, often with fairly large (0.5 - 1.5Mb for the Intel iPSC) local memories, but no shared address space. However, since each processing element is connected to \( k \) other processors [7], the interconnection complexity does not become very feasible to implement if one considers systems with more than that a couple of hundred processors.

In this paper, we will therefore consider a novel, most highly parallel architecture, the Polymorphic-torus [8-11] and compare it to hypercubes. The Polymorphic-torus is an SIMD system that has a toroidal mesh interconnection structure, an interconnection
network with a constant number of interconnections per processor. This allows the system to be implemented with thousands of processing elements (or possibly 1-2 million) by avoiding the logarithmic growth in interconnections for a pure hypercube. In general, fewer interconnections imply that a major penalty in communication time could be expected. However, a key feature of the Polymorphic-torus allows for idle processors to be bypassed within the order of a machine cycle at run-time in effect creating near-neighbor links across distant active processors.

Commercially available hypercubes include the Intel iPSC systems, NCUBE [12], and the Connection Machine [13], to name the most popular ones. The iPSC and NCUBE are both MIMD systems with floating-point processors at each node, whereas The Connection Machine is currently a bit-serial SIMD system with a shared floating-point chip per every 16 processing units. The Connection Machine is, however, so far, the system that offers the highest degree of parallelism (65,536 processing units versus Intel’s 128 and NCUBE’s 1024).

In this paper, it will be shown how some of the algorithms well-tailored for the hypercube also perform well on the Polymorphic-torus. A key to this performance is that the Polymorphic-torus allows one to bypass idle processors within the order of a machine cycle. First, an efficient dense matrix multiplication technique for grids when the input matrices are loaded in will be indicated. This algorithm is optimal for both hypercube and mesh structures using a standard in-order computation, and can hence not be improved for standard matrix multiplication techniques.

The case where one has a string of matrix multiplications to perform, i.e. the data is already loaded, is then discussed. How to perform such a dense matrix-multiplication algorithm efficiently on a hypercube using an orthogonal tree structure, is shown by Elster and Reeves [14]. Due to the polymorphic bypass feature, these structures can be equally efficiently implemented on the Polymorphic-torus, This implementation will be discussed in more detail. To further demonstrate the power of the Polymorphic-torus’ unique switches, it will then be shown how to implement an efficient sparse matrix-vector multiplication algorithm for the Torus. Finally, how the FFT, an optimal algorithm for the hypercube, can be reasonably efficiently implemented on the Polymorphic-torus, will be discussed.

2. Dense Matrix Multiplication

One of the most common computations in scientific computing is the multiplication of matrices and vectors. This section offers suggestions on how this computation can be done on some typical parallel systems in the dense case. Matrix-vector multiplication can be described as the multiplication of a vector with each row of a given matrix [15]:

If \( A \in \mathbb{R}^{m \times n} \), \( x \in \mathbb{R}^n \),

\[
y = A \cdot x \quad \text{where} \quad y_i = \sum_{j=1}^{n} a_{ij} \cdot x_i \quad \forall i = 1, \ldots, m. \tag{1}
\]
Whereas a matrix-matrix multiplication may be thought of as the multiplication of each row of the first matrix with each column in the second matrix:

If \( A \in \mathbf{R}^{m \times n} \), \( B \in \mathbf{R}^{n \times p} \), \( C \in \mathbf{R}^{m \times p} \),

\[
    C = A \cdot B \quad \text{where} \quad c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}.
\]  

(2)

These tasks lend themselves well to parallel processing due to their inherently independent subtasks of vector operations. We shall see later that the matrices may not only be distributed row or column-wise (vectors), but also as blocks of vectors or submatrices. How to partition the problem efficiently also depends on the interaction between the memory and the processors of the parallel machine in question.

2.1.1. Dense Matrix Multiplication on Load-in Data

Considering a mesh-connected system (both the hypercube and the Polymorphic-torus embed this structure rather straight-forwardly), dense matrix-matrix multiplication can be performed by loading in one of the matrices from one side of the processor grid and sliding it across the grid over the already stored matrix elements of the other matrix. This "wave-front" approach requires the extra start-up time of preparing the matrix to be loaded, but takes only \( n \) steps to complete for an \( n \times n \) matrix.

Since this is the lower bound for regular in-order matrix-matrix multiplication, the Polymorphic-torus and the hypercube, in general, do not have topological advantages over mesh-structured architectures. Notice, however, that this approach assumes I/O handling on all boarder processors. On an Intel cube with I/O confined to node 0, the cost of a local distribution across the first row/column would therefore also have to be added.

2.1.2. Dense Matrix-Vector Multiplication on Resident Data

The matrix-vector multiplication and transpose algorithms described by Elster and Reeves [14] are especially designed to take advantage of the full hypercube topology. By using orthogonal tree structures operating on submatrices (Figure 1) rather than the more conventional distributed algorithms [16-19], communication costs are minimized, especially for computations concerned with near-neighbor data (e.g. seismic and image processing problems). Implemented on the Intel iPSC, the algorithms were an extension to the high level library for hypercubes developed at the Chr. Michelsen Institute, Bergen, Norway [20]. The algorithms presented may also be used on a standard processor grid with some minor modifications and loss of efficiency (the hypercube has a more powerful interconnection network, i.e. more nodes are directly connected). The Polymorphic-torus, however, is able to compete with the hypercube by taking advantage of its dynamic routing feature, as shall be shown in the comming section.
A usual data mapping scheme for matrices is to use a two-dimensional mesh embedding of the hypercube processors. To optimize the use of each processor, it is desirable to incorporate all nodes (processors) into the mesh and distribute the matrix data as evenly as possible among the nodes.

A matrix may be embedded on the hypercube using all nodes by numbering the processors according to a Gray code, a binary number representation where only one bit is changed for any two given consecutive numbers [16,21].

Optimal performance is achieved by distributing the matrix as equally as possible among the processors available. A necessary condition for distribution of equally sized data blocks, would require the matrices to be of size $pc_1 \times kpc_2$ ($p = 2^i$; $k = 1$ or $2$; $c_1c_2 = \text{constant}$), given the above matrix embedding.

On the Polymorphic-torus, smaller block-sizes can be favorably used since a large number of processors assumed. One can also use the same communication strategy as the hypercube technique since the tree structure may be approximated by short-circuiting the intermediate idle processors as depicted in Figure 2.
Instead of using connections of higher dimensions as the hypercube does, direct links are formed within the order of a machine cycle for provide a direct path between the two tree nodes. This is made possible by the special programmable hardware switch which consists of fully connected 5-way links (North, East, West, South, and PE) which may "connect" in any permutation of direction (e.g. North-South).

The Polymorphic-torus hence offers to accommodate the faster "hypercube" technique of performing the global sums through tree structure (O(\log_2 n)) communication, rather than shifting the partial results across the whole processor grid (O(n)). Unlike hypercube systems, the interconnection complexity per processor for the Polymorphic-torus remains constant regardless of the system size. Extensions to matrix-matrix multiplication can be done by repeating a matrix-vector operation for each row. Other more direct alternatives should also be considered.

3. Sparse Matrix-Vector Multiplication

Sparse matrix operations are significantly important to many engineering and scientific computing applications using, for example, finite element analysis. In this section, how one can efficiently take advantage of the the Polymorphic-torus in implementing an algorithm for sparse matrix-vector multiplication will be presented. Since comparable work is not yet (to our knowledge) known to exist for the hypercube, the algorithm’s advantages over standard mesh systems will also be emphasized.

The algorithm has two phases. The first phase condenses the sparse matrix to a form suitable for the Polymorphic-torus. The second phase does the actual multiplication. The first phase is described in section 3.1 and the second in section 3.2.
3.1. Structured Condensation

This section describes a sparse matrix representation in Section 3.1.1 and a condensation algorithm in Section 3.1.2 to convert the representation into a format better suited for the Polymorphic-torus.

3.1.1. Sparse Matrix Representation

Traditionally, a sparse matrix is represented by three arrays [22, 23]: (1) an Element array, $E$, which holds the values of the nonzero entries of the matrix; (2) a Row Pointer array, $RP$, and (3) a Column Index array, $CI$, where the latter two arrays store the connectivity of the sparse matrix.

The construction of the three arrays is illustrated in Figure 3.

Row pointer/ column index representation:

Element array: $E = 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 1.0, 2.0, 3.0$;

Row Pointer array: $RP = 1, 3, 4, 6, 8, 9, 10, 11, 12, 13$;

Column Index array: $CI = 2, 5, 1, 3, 6, 2, 4, 4, 2, 7, 6, 8$;

Figure 3. Sparse Matrix Representation.
The element array $E$ is built by scanning in a row-major order but only the nonzero entries are recorded. For a matrix with $n$ rows, the RP array contains $n+1$ items in which the $i$-th item points to the position of the first nonzero entry of the $i$-th row in the $E$ array. The $n+1$-th item which is one greater than the number of nonzero entries in the matrix is regarded as a delimiter. For example, the third item in $RP$ (i.e. 4) points to the 4-th item in array $E$ (i.e. 4.0) which is the first nonzero entry of the third row. The $CI$ array has as many items as the $E$ array and each item contains the column position of the corresponding element in the array $E$. For example, the column position of 4.0 is 3.

### 3.1.2. Condensation Algorithm

The purpose of the Condensation algorithm is to convert the representation of the sparse matrix to a format better suited for the Polymorphic-torus. Specifically, the goals are:

1. to assign only the nonzero entries of the matrix to processors so that the converted format is denser and the utilization of the SIMD system can be increased;

2. to preserve the connectivity of the sparse matrix; and

3. to relate the RP and CI arrays to the virtual coordinates of the processors (i.e. [layer, pidx, pidy]) in the Polymorphic-torus.

Figure 4. shows the algorithm expressed in the C language with parallel constructs as defined in the paper by Li, Wang, and Lavin [24].

The above algorithm consists of two phases. In the first phase, the RP array is checked to determine the length of a logical row (i.e. row_size). Due to the sparsity, the number of nonzero entries in a row of the sparse matrix is usually small. Therefore, multiple matrix rows can be bound to a single processor row, to obtain a denser data structure. Furthermore, the length of a logical row is always chosen to be a power of 2 to simplify the control. Figure 5 depicts this multiple-row binding.

The second phase of the algorithm calculates the virtual processor coordinate for each element in the array $E$. An example of this calculation using the matrix in Figure 3 as input, is shown in Figure 6. For example, the first element is bound to processor [0, 0, 0] (layer=0, pidx=0, pidy=0). A "counter" is increased to check whether the next element belongs to a new matrix row. If not (i.e. the case of both 2nd and 3rd elements), "pidy" is increased while "layer" and "pidx" remain unchanged. When a new matrix row begins (i.e. 4th element), decision is made whether a new processor row needs to be started. If yes, "pidx" is increased to reflect the binding. This process continues until one layer is completely filled then "layer" is increased so that a second layer of memory is needed as a virtual processor array. For simplicity, the algorithm listing handles only the single-layer situation. Although in Figure 5, only the $E$ array is bound to a virtual processor, other arrays (e.g. $CI$) can be bound similarly.
Condensation:(RP, E, N)
int RP[N]; /* N: side size of processor array */
float E[];
{
int i=0;         /* dummy variable */
int row=0;       /* total no. of rows in the matrix */
int max=0;       /* maximum length of row in the matrix */
int row_size=1;  /* size of a logical row */
int pointer=1;   /* pointer to the beginning of next row */
int counter=1;   /* counter of elements in array E */
int layer=0, pidx=0, pidy=0; /* virtual coordinate */
struct quadruple /* data structure of mapping form */
    { float element; /* nonzero value and virtual coordinate */
      int layer, pidx, pidy; }
quadruple mapping[];
boolean start; /* flag indicating start of a logical row */

/* PHASE 1. Check RP array to count the no. of rows and
determine the max.length of row: */
while (RP[i]!=NULL){
    if ((RP[i]-RP[i-1]) > max)
        { max=RP[i]-RP[i-1]; }
    i=i+1;
}row=i;
if (max > N) exit(); /* row length > array size, exception */

/* Determine the size of a logical row: */
/* for matrix side size > processor array side size: */
/* Try to map multiple matrix row into proc. row: */
if (row > N){
    while (row_size < max) { row_size=row_size*2; }
} /* for matrix side size smaller than and equal to processor array side size */
else { row_size=N; }

/* PHASE 2. Read Element array and determine the coordinate of binding processor */
i=0;
While (E[i]!=NULL){
    /* binding element */
    mapping[i].element=E[i];
    mapping[i].layer=layer;
    mapping[i].pidx=pidx;
    mapping[i++].pidy=pidy;
    if (++counter==RP[pointer]){ /* beginning of a new logical row */
        start=TRUE; /* binding a new logical row below */
        while (start || (RP[pointer]==RP[pointer+1])){
            /* while loop also detects empty rows */
            pidy=(pidy/row_size+1)*row_size;
            if (pidy >= N) { /* jump to a new physical row */
                pidx=pidx+1;
            } /* fold to next layer while fill up array plane */
            if (pidx==N) { pidx=0; layer++; }
            pidy=0;
        start=FALSE; pointer++; }
        else{ pidy++; }
    }
}

Figure 4. Condensation Algorithm
(a) 8x8 sparse matrix with maximum row-length equal to 2 (max=2).

(b) matrix adjusted in row direction. row_size = 2 determined.

(c) matrix mapped into 4x4 array; two logical rows are bound per array row.

Figure 5. Binding of Multiple Logical Rows.
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>E[i]</td>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
<td>5.0</td>
<td>6.0</td>
<td>7.0</td>
<td>8.0</td>
<td>9.0</td>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>pointer</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>RP[pointer]</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>counter</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>layer</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pidy</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>pidx</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

N = 4; /* 4x4 array */
max = 2; /* maximum length */
row = 8; /* no. of rows */
row_size = 2; /* size of logical row */

Figure 6. Execution Snapshot of Matrix in Figure 3.

The condensation algorithm can be executed on the host as a pre-processing step or by the I/O subsystem of the Polymorphic-torus. Since the algorithm features a simple "counting", "comparing" and "incrementing" mechanism, the condensation pre-processing is fast when executed on the host. When the condensation is performed as part of the I/O subsystem, only a simple circuit is required for the enhancement. The significance of the I/O enhancement is that (1) I/O bandwidth is significantly reduced, i.e., three-array representation is transferred instead of the full matrix; and (2) a sparse matrix can be handled by an SIMD system as a high-level object, which greatly simplifies the programming.

Although simple, the condensation algorithm is powerful and delivers a denser result than the standard method used for, say, the DAP [25]. In comparison, the condensation algorithm "squeezes" the matrix in the "pidy" direction while the method used for the DAP [25] squeezes the matrix in the direction of "layer". The difference in the condensation philosophy lies in the difference in the architectural support. For standard mesh architectures, to fully utilize the row/column highway capability, the placement of a non-zero element is confined to both row and column directions. For the Polymorphic-torus, the "short-circuit" capability allows us to move the non-zero elements more freely. With more freedom in moving the non-zero elements, the condensation algorithm yields a denser matrix, which leads to a higher utilization of the SIMD system.

3.2. Condensed Matrix-Vector Multiplication Algorithm

Using the structured condensation algorithm, a sparse matrix can be mapped onto the Polymorphic-torus densely while preserving the sparse connectivity. Due to the condensation, a high percentage of processors in the Polymorphic-torus will be utilized. However, the condensation is only part of the accomplishment. The condensation algorithm also transforms the sparse matrix into a format which takes advantage of the polymorphic feature.
One advantage is that the nonzero entries in the same row of the matrix are bound into the same row of the Polymorphic-torus. By embedding a sum tree for each matrix row in the Polymorphic-torus [8-11], the summation of the product terms from row/vector multiplication can be performed in an optimal logarithmic time. The other advantage is that a global bus can be formed among all processors so that variables (e.g., the vector elements) can be broadcasted in the order of unit time. These advantages can be seen in the second-stage algorithm, the "Condensed Matrix Vector Multiplication Algorithm" which is described next.

The purpose of this algorithm is to compute $X' = AX$, where $A$ is a condensed $n \times n$ sparse matrix with nonzero entries $a_{ij}$ contained in the $E$ array and column index $j$ in the $CI$ array; $X$ is a $n \times 1$ vector containing $x_j$ and $X'$ is the updated $X$. Iterative methods are used to obtain the solution of the problem and the inputs to this algorithm are:

1. the size of the logical row,
2. initial guess of $x_j$'s with $x_j$'s located at the first locations of each logical row;
3. the element array, $E[ ]$, bound to processor [layer, pidx, pidy] according to the condensation algorithm; and
4. the column index array, $CI[ ]$, distributed in the same way as the $E$ array.

The output of the algorithm is the updated $x_j$'s for the next iteration in a complete matrix solver. Furthermore, the new $x_j$'s are in the same place as the old $x_j$'s. The algorithm is shown in Figure 7.

The complexity of the algorithm can be analyzed in accordance with the three phases of the algorithm as follows: In the first phase, the distribution phase, the variables $x_j$ in the dense vector, are broadcast in sequence to processors that hold $a_{ij}$. This phase needs $O(n)$ steps for an $n \times n$ matrix. Note that a global bus is configured for the broadcasting, and the sender to and the receivers from the bus are determined simultaneously and locally in each processor by using the Column Index (CI) bound to each processor at the condensation stage. In the second phase, a floating-point multiplication is performed locally in every processor to produce a product; this takes one floating-point multiplication step. Finally, in the third phase, new $x_j$'s are obtained by accumulating the corresponding partial products. This third phase takes $O(\log(\text{logical\_row\_size}))$ floating-point addition steps because all corresponding partial products for each $x_j$ are allocated in the same processor row by the condensation algorithm and furthermore $n$ sum trees (one for each condensed matrix row) can be embedded simultaneously in the Polymorphic-torus. For simplicity, only the single-layer case is handled by the above algorithm. When multiple layers are presented, the complexity is in proportion to the number of layers.

The algorithm takes advantage of the Polymorphic-torus' unique capability to dynamically reconfigure a global bus and many sum-trees to the size of the matrix row after the condensation. In fact, the reconfiguration required for the sparse matrix vector multiplication is dynamic and dependent heavily on the local conditions. The polymorphic feature strongly facilitates such a local and data-dependent reconfiguration. It is therefore understood that the speedup is attributed partially to the dynamic
Condensed Matrix-Vector Multiplication(n,a,ci,row_size,x)
int row_size, n; /* size of logical row & side size of matrix */
int ci; /* column index of a_{ij} distributed as described */
float a, x; /* a_{ij} and x_j */
{
  int i; /* dummy variable */
  int x_index=0, y_index=0; /* index of processor which broadcasts x_j */
  int distance=1; /* rel. dist. used in log. tree sum */
  boolean btemp=TRUE; /* TRUE while holds partial sum */
  float ftemp;

  /* Phase 1: Distribute x_j */
  for(i=1; i<=n; i++){
    SYNC { /* global synchronization */
      if (ci==i) { /* check column index to determine whether to receive x */
        /* get x from the first processor in the i-th logical row */
        ftemp=x of [x_index, y_index];
      }
    }
    /* up to beginning of next logical row */
    y_index=y_index+row_size;
    if (y_index>=N) { y_index=0; x_index=x_index+1; }
  }
  /* Phase 2: Multiplication */
  x=a*ftemp;
  /* Phase 3: Summation by tree sum */
  for (i=0; i<log row_size; i++){
    SYNC { /* global synchronization */
      /* accumulate partial sum by left sibling */
      if ( btemp && (pidy[i]==0)) { /* check the i-th bit of pidy */
        x=x+x of [pidx, pidy+distance];
      }
    } else{
      btemp=FALSE;
      distance=distance*2; /* incr. dist. for higher level */
    }
  }
}

Figure 7. Condensed Matrix-Vector Multiplication

reconfigurability and partially to the condensation. The merit of this two-stage approach is discussed further in the next section.
3.3. Discussions

The choice of condensing a sparse matrix along the row direction is driven by the polymorphic feature of the Polymorphic-torus, namely, multiple sum trees can be formed adaptively to the condensed matrix rows for a logarithmic-time summation and a global bus can be formed with a unit broadcast time. (Note that in a very large Polymorphic-torus, the aforementioned logarithmic and unit time may need an slightly elongated machine cycle. Detailed discussion on timing can be found in [10]. Contrasted with a mesh architecture such as the MPP [26-29], the polymorphic feature is consequently superior, since they would have to shift the data across the processor. Global broadcast and summation trees can be similarly implemented on hypercubes, but pose a difficulty when several trees of different sizes are needed, as in our algorithm.

The condensation process, on the other hand, converts the sparse matrix into a dense data structure whose density implies low number of idle processors and high system utilization. Furthermore, it allows very large matrices to be handled by a small processor array that otherwise may not have enough memory. The condensed data structure, however, is not always 100% full because the side size of the Polymorphic-torus may not be a multiple of the size of a logical row. Nevertheless, when it is a multiple, the condensed matrix is indeed full implying that every processor in the Polymorphic-torus is active and contributes to the matrix solving.

Compared with the layer-oriented condensation previously implemented on the DAP mesh architecture [25], our condensation method packs the matrix along pidy-direction (or row direction) and gives higher density. In fact, it is extremely difficult, if not impossible, for the layer-oriented condensation to yield a full matrix. Our condensation algorithm is also applicable to mesh and hypercube architectures. However, the efficiency for fixed mesh systems will be lower since it lacks the capability of embedding simultaneous sum trees. These tree are also hard to efficiently implement on hypercubes. In summary, the condensation results in higher density but requires a corresponding reconfigurability for the best speedup.

Compared with the worst-case arithmetic operation count $O((n/N)^3)$ of the DAP algorithm, the arithmetic operation count for the polymorphic-torus ($O(1)$ for multiplication and $O(\log(\text{logical\_row\_size}))$ for addition) produces a faster solver. However, our $O(n)$ cost in the broadcasting phase may be inferior to that of DAP ($O((n/N)^3)$) for a large $N$. Nevertheless, considering the relative low broadcasting cost to the cost of the floating-point operation (e.g. 1:9 in DAP), our algorithm is more suitable for highly parallel bit-serial SIMD machines that incur significant arithmetic cost.

It is appropriate to ask what the ideal architecture and algorithm for sparse matrix vector multiplication would be. Will a hypercube network be a good candidate? The algorithm/architecture mapping is expected to bear the following features: (1) a matrix row is mapped onto a group of processors that can be structured as a tree; (2) a matrix column is mapped onto a group of processors that can be structured as a bus; and (3) the condensed structure is full.
Such a mapping then allows for an $O(1)$ broadcast, an $O(1)$ multiplication and an $O(\log(\text{row size}))$ summation, which is the lowest complexity of the sparse matrix vector multiplication. Moreover, there will be no idle processors since the structure is full.

To our best knowledge, such an ideal mapping has not been exploited. The Mesh-Of-Tree (MOT) [30] can come close to it by mapping a matrix row to a tree group but will have difficulty in mapping a matrix column conveniently. A high-degree network such as the hypercube [13] may offer a potential solution. However, no such study has to our knowledge been reported for the hypercube, so the ideal mapping remains as an open research topic. Hypercubes also become difficult to realize for systems with more than a few hundred due to their interconnection complexity.

4. DFTs

The Fast Fourier Transform [31, 32] is perhaps considered the ultimate hypercube algorithm in that its communication paths nicely map onto the hypercube structure. Defining the Discrete Fourier Transform (DFT) of a vector $x$ of length $n$ as:

$$ y \leftarrow F_n x $$

where $F_n$ is the matrix consisting of the $n$th root of unity $\omega_n = e^{-2\pi i/n}$ giving:

$$ [F_n]_{jk} = \omega_n^{jk}.$$

When calculating these "butterflies", the communication pattern for the DFT's signal flow graph matches a collapse in a hypercube dimension for each stage. This, however, presumes one has $k$ connections going into each of the $2^k$ processor in the system. In the case of the Polymorphic-torus, one only has 4.

If a straight-forward approach utilizing the bypass feature of the Polymorphic-torus was followed, it will mean that for, say, $n = 8$, stage 1 can be performed in parallel as with the hypercube (all near-neighbors), but in stage 2, two steps must be made, while in the last stage, all butterflies across processors need to be performed separately in order to avoid collisions. In general, for an $n$-bit transform where $n = 2^k$, stage 1 is completely parallelized, where as stage $k-2$ has $k/4$ steps, and stage $k-1$ takes $n/2$ steps to be performed, the total number of steps would be $\sqrt{n}$. For a hypercube, this would be merely $k$ steps.

By utilizing toroidal connections and reordering the processors, a performance of $\sqrt{n}/2$ steps may be achieved. However, considering the elongated machine cycles that may arise, the actual speed-up may be minimal. A more reasonable approach would be to follow the parallel shift methods used for standard mesh where data from half of the row/column processors get shifted to the other half for the highest dimension. Gutierrez has implemented such an algorithm on the MPP [33]. Notice that this would require a store-and-forward mechanism in order to perform the shifts in parallel and also yield an $O(\sqrt{n})$ algorithm for both the Polymorphic-torus and the mesh. Considering that the FTT causes half the processors to be constantly busy, it is therefore reasonable that not much can be gained from the polymorphic bypass feature. This will
also be true for other algorithms in this class, such as Bitonic sorting.

5. Conclusions

We have presented a dense and a sparse matrix vector multiplication algorithm, as well discussed the DFT algorithm for the Polymorphic-torus, a reconfigurable massively parallel fine-grained SIMD architecture. We emphasized in the paper how reconfigurability, or the polymorphic feature, can help to match algorithmic structures to the architecture. We showed how this reconfigurability could help in mapping both the the orthogonal tree structure used in the dense case, and the sparse matrix techniques onto a regular network topology.

The dense matrix algorithm is able to use the same orthogonal tree-structured communication pattern as the hypercube by dynamically creating a "tree" structure across idle processors. In the sparse case, the condensation process is an algorithm that converts the sparse connectivity into a more uniform data structure suitable for the topology of the Polymorphic-torus. More specifically, many matrix rows can be packed into a processor row such that there are fewer idle processors and the system utilization is higher. The condensation algorithm can be run on the host as a preprocessing step or can be run by the SIMD subsystem as a part of the I/O and memory management. In the latter case, the sparse matrix is treated as a high-level object; this not only reduces the I/O bandwidth requirement but simplifies the programming of a SIMD system. The condensed matrix vector multiplication that follows requires the Polymorphic-torus to reconfigure itself to match the condensed data structure aforementioned. The polymorphic feature allows all multiplications be done in order unit time while the summation is done in logarithmic time, which is optimal. Algorithms that require a lot of processors to be involved at each step, such as the DFT, does however, not benefit as much from the polymorphic features.

The condensation algorithm presented is applicable to other SIMD machines such as the hypercube, the DAP, or the MPP. When applied to the MPP, the utilization of the MPP is higher due to the higher density achieved by the condensation. For its application to the DAP, there can be no improvement because the DAP lacks the required companion reconfigurability and the condensation algorithm does not effectively use the DAP's capability in row/column broadcast.

In a more general view, what has been exploited in this paper are methodologies of mapping static task graphs onto a massively parallel fine-grained SIMD architecture with regular network topology in order to achieve near-hypercube performances.

6. Acknowledgments

This research was partially supported by IBM agreement 12060043.
References


