The “Guess what? We have a hypercube” document

Alan Edelman
Thinking Machines Corporation
Internal Note

August 21, 2003

Abstract

Here I try my best to explain how to use the hypercube wires on the Connection Machine. At one time, direct use of the wires was considered beyond the reach of reasonable programming, but this is no longer true. We now have the primitives and expertise that make self routing reasonable and actually quite fun. No previous experience is assumed, all you have to do is be willing to learn about the capabilities now available in CMIS. In this document I emphasize less what algorithms can be implemented using the hypercube wires, and more the mechanics of how to use them. It might be nice to have another document that shows off how the hypercube can be used in non-trivial ways.

The CM-2's power is being terribly under-utilized. It is my hope that several people will use this document as a departure point for implementing fast communications routines. There is no doubt that fixed communications patterns can be much faster. This document is not at the level of a textbook yet, but I have tried my best to explain as much as possible. I strongly encourage people using the wires to send questions to the “wires” mailing list and work with me (while I am still around) , Mark Bromley, or Steve Heller to take advantage of the expertise that we have gained.

Just lately I have been hearing talk about implementing stencils as a program product. Another product that would run beautifully fast on the machine is an n-body package. Other primitives that can and should be written are fast power of 2 news, spread row and column simultaneously (perfect for LU), spread from diagonal accross row and column simultaneously (perfect for Jacobi eigenvalues and SVD), scans etc. There is plenty of work to go around, all of which we have the know-how to solve.

There have been many changes in the use of the wires in the past months. It went from impossible (the days of Steve Vavasis) to requiring microcode trickery and speed hacking (this is where I started working) to being more or less straightforward and convenient (Mark Bromley deserves tons of credit for this). Benchmarks based on news being fast and everything else being slow should be thrown away. AE's hearing about code being communication bound should be asking probing questions about why.

Contents

1 Introduction

Everybody talks about how slow communication is on the Connection Machine, but now you too can do something about it. Perhaps a good start is to mention a few statements that you might consider surprising:

1. The full bandwidth of the hypercube is available, though it is rarely used in current software.
2. It is a myth that only local applications can run at high speeds on this machine.
3. With the full bandwidth, the systolic part of a single precision full bandwidth real matrix multiply at VP ratio 128 was timed to run at 1.6 gigaflops on an 8k machine. That's almost 13 gigaflops scaled up, folks!
4. News is one of the **worst** communications patterns on the CM2 as it currently uses only one wire, due to our gray code embedding.
5. Butterflies that are not pipelined are also as bad.
6. Some researchers would actually enjoy thinking about "true" hypercube algorithms, i.e., algorithms that fully exploit the bandwidth of the machine.

Let me recount the history of multiwire communication as I know it. I apologize in advance for errors and omisions. Please inform me of these errors. Cube swaps probably began with Guy's memory to memory cube swap, which is used in the current implementations of scans and spreads. I am sure Alan Ruttenberg also used direct memory to memory cube swaps in the FFT. The point of departure of multiwire communications from my point of view is Steve Vavasis who implemented a spread using Johnsson-Ho trees in microcode. It is nothing less than remarkable that he did this during a summer. The first conception and experimentation of a cube swap from transposer to transposer was Alex Vasilevsky's as it became clear that this was appropriate for the FFT. I believe that I wrote the first working version of this primitive, and designed a sped up version (the "it can't be done in a four microcode instruction pipe" Brewster–Edelman beer bet). Mark Bromley got this faster version working and into CMIS, and created many clever primitives which make the actual task of communications easy for the first time. Mark and I had at first worked independently realizing that a cube swap from transposer to transposer would be just the right primitive. Mark further implemented the stencil code using these and other primitives that are now available in CMIS. In fact, thanks to his efforts, there is no reason not to use the wires anymore. One of the more recent innovative uses of these primitives is a routing compiler written by Denny Dahl.

One communication pattern whose potential remains to be further explored is the n-rotated Gray codes hidden in one of Lennart Johnsson and Ching-Tien Ho's papers. I've written a version with Jean-Philippe Brunet for the n-body problem. A variant where you do an n-body problem along each of two axes is just right for matrix multiply. (This became obvious to me after I realized that the n-rotated Gray codes was just right for the n-body problem after which I learned that Johnsson & Ho had already written this down.) Though not sufficiently publicized, this particular communication is very simple, hence very fast. It is currently the **fastest** communication pattern that I can think of for the CM2 because it both uses the full bandwidth of the machine, and when coded properly is very simple.

Other recent uses of the cube-wires include a routing pattern for bit-reversal written by Steve Heller and myself with lots of assistance from Mark. I won't explain what this pattern is here, but I will mention that the router implementation of this communication pattern slowed down the FFT immensely. The new version is much faster. Not only are these techniques useful for the FFT, but the same code improves Marshall's IO twiddling by a factor of 4 as compared with the use of the router. Hopefully, this can be used for the framebuffer as well. Mark, Steve Heller, and I reimplemented the Vavasis spreads, obtaining a factor of 2 to 3 speedup over his pioneering work. Bigger factor over the paris code. Steve Heller is now placing these spreads into the fieldwise code. Kimberly is the newest member of the wire team. She is currently implementing a send-address to news-address converter.

With the exception of the phase 1 bit reversal (my own algorithm), all of the multiwire algorithms that have so far been implemented were either the straightforward algorithm or the creations of Lennart and his students who have published several papers on exploiting the bandwidth of the hypercube. These papers are chock full of nice ideas, though be warned that one has to sometimes penetrate deeply to find the ideas.

2 Answers to frequently asked questions

Before I get into the specifics, let me clarify some basic issues.

Q: What's a hypercube?

A: A d-dimensional hypercube refers to 2^d nodes labeled as a d bit binary string. Two nodes are adjacent, if their labels differ by exactly one bit. Since there are d bits to change, each node has exactly d neighbors, one for each dimension (hence d-dimensional).

Q: How are grids embedded in hypercubes?

A: By throwing away some of the wires, grids can be found embedded in hypercubes. The technique is known as Gray coding, and has been well understood for decades. When a two dimensional grid is embedded in a hypercube, the rows (and the columns) are subcubes. Thus a spread from column amounts to independent parallel broadcasts in smaller hypercubes. It is also true that the row wires and column wires are disjoint allowing for simultaneous uses of these wires. (Spread a row and column simultaneously, spread from diagonal across row and column simultaneously, (news!! (a 0 1)(b 1 0)), i.e., shift a along the 0th dimension and b along the 1st, you get the idea.) (Steve Heller suggested I mention something about the above points.)

Q: I now believe that for a two dimensional grid, I can use the four news wires simultaneously. I also believe that spreads along rows and columns can use other less obvious wires as well. Any other possibilities?

A: The limits are only that of the imagination. As I mentioned there is a matrix multiply algorithm that uses all the bandwidth. All that is required is to throw away your preconceptions. I admit that it is sometimes difficult to figure out how to rethink two dimensions onto higher cubes, but the payoff can be great.

Q: I heard a rumor that floating point operations are faster slicewise. What about communication?
A: Communication is more convenient slicewise. In fact, this paper assumes that your data in memory is indeed slicewise.

Q: Does that mean that if my data is fieldwise, I simply can not use the wires?
A: Not always. If the axis on which you wish to communicate is conveniently located, it may not matter whether your data is fieldwise or slicewise. (The most convenient situation is an axis that is completely off-sprint.)

Q: On the CM, can you move data both ways across a wire at the same time?
A: For purposes of the programming model that you are about to hear, the answer is an emphatic YES. In fact, as a mode of thinking it is helpful to focus on one node and consider which data is outgoing over which dimensions at that node. Usually one finds that other nodes are performing similar operations. By conceptually separating outgoing from incoming usually one finds a greater simplicity in thinking.

Q: Do I have to own a μ-code workshirt to even consider the idea of programming the wires?
A: Not at all, all you need to know follows.

Q: Who should be using the wires?
A: Anybody who wants their code to run faster. Specifically, if one is using the same routing pattern many times overshadowing the cost of any set up. People should get on the "wires" mailing list, and take the initiative to find out what is not clear or not mentioned in this document.

Q: Do I have to rewrite all my code to make use of these primitives?
A: If your code is already slicewise, you probably need not do anything. If your code is fieldwise, with a certain amount of thought you might still be able to use these primitives. Contact Mark Bromley, Steve Heller, or myself (email to CERFACS is okay starting in January 1990).

Author's note: other questions (or gaps in what you need to know) will be filled in later based on reader's questions.

3 Basic Mechanics

If you've already programmed in CMIS, you are in an excellent position to explore the wires. If not, let me say a few words about CMIS and the CM2.

CMIS has evolved into a reasonably convenient language for performing low-level operations on the Connection Machine without resorting to microcode. Though at first the names of the instructions may look intimidating (consider CMIS-FPU[-M{R,W}T{A}-%P{1-4}]-FRT{B}{C}-DYN2-DELAY-%C3-STATUS-L3) actually, once you get into the CMIS philosophy, you see that it's not so bad. Do get yourself all the nice documentation on the third floor. In Appendix 1, we list the most relevant CMIS instructions for using the cube wires.

The basic hardware unit on the CM is the Sprint Node. (CMIS calls it the Sprint Section, which hides its chief propery of being the basic processing node.) There are 2048 Sprint Nodes on a full machine; there are 16 on a board. The concept of our machine having 64k processors is essentially a lie when it comes to floating point operations. Just note that there are 2048 nodes on a full machine and 16 nodes on one board.

Each sprint node has memory for 64k or 256k 32 bit (slicewise) words known as slices. The sprint nodes are connected as a d-dimensional hypercube. Here are the values of d for various machines:

	d
full machine	11
half machine	10
eighth machine	8
board machine	4

The hypercube allows for communication of **two** slices out of each node along each dimension. This communication has lovingly become known as a **cube-swap**. Thus on a full machine, as many as 22 slices can depart from (and of course, simultaneously arrive into) a sprint node during a cube swap. The word *swap* takes the point of view of the edge. For each edge in the hypercube, four slices, two on each side of the edge, swap places during a cube swap.

Each sprint node contains devices for arranging data to depart from and arrive into the node. These devices are known as transposers because of an archaic purpose (something about turning fieldwise to slicewise). As long as your data is slicewise, you need not worry about this function of the transposers. All your data will come out just right.

If your data starts out slicewise, there is
no need to worry about it ever becoming fieldwise

There are three transposers on current machines: TA, TB, and TC. The basic mechanics of cube-swapping is to 1) load TA from memory with departing data 2)perform a cube-swap (say from TA to TB) and then 3)read the just arrived data from TB back to memory. There are other options of which transposer to use (TA-to-TB,TA-to-TC,TB-to-TA) though one should suffice.

For easy reference, all communication takes the following form:

Memory is written into TA
Cube-Swap TA-to-TB
Memory is read from TB

The hard parts are the loading and unloading of the transposers. The communication is trivial.

This is probably the hardest paragraph. I am going to say the same thing in two slightly different ways in the following two paragraphs to hopefully make it easier. The transposers each have 32 slots for loading and unloading data, though you will at most use slots 1-11 and slots 17-26 for the communication. (This would be for a full machine, in general slots 1-d apply.) The slots are specified by transposer pointers. We will give examples of setting the pointers later. The data loaded into slots 1-11 of TA in any one sprint node say farewell to each other and arrive in TB at 11 distinct sprint nodes corresponding to the 11 distinct neighbors of the sending node. Data loaded into slot 1 travels along the edge corresponding to the least significant dimension; data in 11 travels along the most significant dimension in a full machine. At its new location, the data ends up in TB at the same slot it was in when it left TA. Slots 17-26 duplicate slots 1-11 entirely, thus allowing for two slices to leave along each dimension.

I like to talk about which wire data travels on. Data can travel on any of wires 1 through d where d is the dimension of the hypercube (11 on a full machine.) The reason the wires aren't numbered from 0 through $d-1$ is that wire 0 refers to a wire connecting the two processor chips on a sprint-node. This wire is useless in this context and should be ignored. Conceptually, I like to say that two slices can depart from a node along each of the d wires during a cube swap. Specifically the data in TA slots i and $i+16$ travel along wire i arriving at the sprint node corresponding to a change in the ith least significant bit. These items end up in slots i and $i+16$ in TB of the new transposer. Note that this is happening in every sprint node, so each node has loaded data into TA and then later reads data from TB.

The following example shows how data would appear in TA before cube swapping using all four wires on a one board machine:

Transposer A

0:	
1:	data for wire 1
2:	data for wire 2
3:	data for wire 3
4:	data for wire 4
5:	
6:	
7:	
8:	
9:	
10:	
11:	
12:	
13:	
14:	
15:	
16:	
17:	more data for wire 1
18:	more data for wire 2
19:	more data for wire 3
20:	more data for wire 4
21:	
22:	
23:	
24:	
25:	
26:	
27:	
28:	
29:	
30:	
31:	

I strongly urge the reader to turn to the LISP code, and to look at simple-cube-swap example. Notice how pointer 1 gets loaded, then we write four slices of transposer A, then pointer 17 and another four slices. We then do the cube swap, and do the read transposer B analogously.

Important Point: During a cube swap, two slices leave a sprint node along each dimension. Also two new slices arrive into each sprint node along each dimension. That's two, not three, and not one. This allows for a bandwidth of 2d at each sprint-node, not 1d nor 3d. (Sorry for overemphasizing this, but I really wanted to quickly avoid any misconceptions.)

It is often not necessary to send data along all of the wires. For example, in a two dimensional stencil code, the sprint nodes are thought of as a two-dimensional grid and the wires perform a get from north, south, east, and west. Thus four wires are used in each node. The mechanism here is to load TA with only eight slices (two in each direction) instead of the $2d$ that are possible. Then TB is read into memory from only 8 of its slots. Thus you see, the cube swap occurs in all available wires, but all but the relevant slots are ignored.

Now all you need to know is how to load and unload the transposers and you can write your own communications code.

There is a certain amount of flexibility in loading and unloading the transposers and this allows for some pretty powerful communications possibilities:

1. All sprint nodes send memory from the same address over the same wire.

2. All sprint nodes send memory from the same address over different wires.

3. All sprint nodes send memory from independent addresses over the same wire.

4. All sprint nodes send different addresses over different wires.

Clearly (1.) is the simplest. The n-body communication pattern is of this form. The stencil code and off-sprint bit-reversal is performed with method (2.). Spreads and another phase of bit-reversal is performed with method (3.). The last method is never necessary.

4 Transposer Pointers

The transposer pointers specify which slots in a transposer are being written or read. In each sprint node, the transposer pointers are set up in a slightly complicated way. Each transposer pointer is a five bit word corresponding to the 32 locations which the pointer can point to. On our machine, the pointers for TA, TB, and TC all get specified at once. The method is to create a 32 bit number, where the least significant fives bits are TA's pointer, the next five, are TB's pointer, and the next five are TC's pointer. The other 17 bits are ignored on current machines. This 32 bit number needs to get to the bypass register and then the pointers can be loaded using a CMIS instruction. One approach would be

```
(CMIS-FPU-LOAD-BYPASS-CONSTANT (+ TA-POINTER (* 32 TB-POINTER)))
(CMIS-FPU-WRITE-POINTERS)
(CMIS-FPU-MWTA-%P1 count)
```

This method sets the pointer to be the same in each sprint node and then writes the transposer with count slices. Note that the pointers increment by one after each slice. The funny symbol %P1 is a CMIS register which contains the address of the data to write.

More commonly one writes the pointers into CM memory and then calls the combined instruction

`CMIS-TP-MWTA-%P1-TABLE-%A9`

with count arguments. This instruction writes transposer A with data from %P1 using a pointer table that can be found in %A9. Here TP denotes *transposer pointer*. Another method is to store one of the CMIS registers into the bypass. An instruction to do this is

`CMIS-FPU-LOAD-BYPASS-%A1`.

More comprehensive information is in the reference manual. Meanwhile look at the attached examples and start playing by yourself.

5 Indirect Addressing

Indirect addressing is a natural idea, yet it is not very well documented around here. Quite simply, it is sometimes desirable that the slice of memory to be loaded into the Sprint chip be different at each Sprint node. Thus, let's say we want to send memory location 205 over wire 1 in Sprint node 0, while memory location 347 should go over wire 1 in Sprint node 1. Somehow we have to get these different locations in memory into TA slot 1.

The ingredients for doing this are

1. Always/Conditional

2. The Base Register

3. The Bounds Register

4. Offsets

5. The Scratch Register

In the next paragraph I will mention the commands to set everything, but first let me describe what everything does. Setting the Sprint condition bits to always is completely different from turning on any context in the processors. For indirect addressing, all you need to know is that it should usually be set to always. The way indirect addressing works is that in each sprint node, the **base** plus the **offset** is the memory location written into (or read from) the transposer, so long as the **offset** is less than the **bound**. Otherwise, the memory location in the scratch register gets read or written. This system sounds a little complicated, but it is often convenient to simply update the base and keep the offsets constant.

The CMIS Instructions are

```
CMIS-FPU-ALWAYS
CMIS-IA-LOAD-BASE
CMIS-IA-LOAD-BOUND
CMIS-IA-MWTA-INDIRECT-%P1
CMIS-IA-LOAD-SCRATCH
```

1. CMIS-FPU-ALWAYS: Word of advice – do this at the top of any code that does indirect addressing, and you won't regret it.

2. CMIS-IA-LOAD-xxxx: All of these instructions are similar to the instruction for load pointers. First you have to get your data into the bypass register. This can be done with CMIS-FPU-LOAD-BYPASS-CONSTANT or with CMIS-FPU-MWB-%P1, depending on whether you want the number to be the same or different in each Sprint node. You can also use CMIS-FPU-LOAD-BYPASS-a CMIS register.

3. CMIS-IA-MWTA-INDIRECT-%P1: This is probably the most convenient method for writing transposer A based on the offsets. The offsets are stored slicewise in the memory location starting at %P1. Notice that the data to be written into the transposer is not in any way specified as an argument to this function as they are the base + offset. The base has already been loaded and the offset is in %P1.

In the attached example file is a simple instance of indirect addressing that actually works. Please see the reference manual for further information.

6 What else?

I didn't cover everything in here as well as I might have liked. Please look at the reference manual (cheat sheet) for more information.