The Burroughs FMP Machine

by

Jacob T. Schwartz

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
715 Broadway, 10th Floor
New York, NY 10003

Ultracomputer Note #5
January, 1980

ABSTRACT

This note comments on various interestings points which appear in the Burroughs Corporation final technical report NAS2-9897, March 1979, entitled *Numerical Aerodynamic Simulation Facility Feasibility Study*. The points covered include: The data movement algorithm of the interconnection network, physical structure of interconnection network and timings, programming style, applications benchmarked, and reliability considerations.

1. Introduction

This note will comment on various interesting points which appear in the Burroughs Corporation final technical report NAS2-9897, March 1979, entitled *Numerical Aerodynamic Simulation Facility Feasibility Study.*

I divide my review into the following 5 headings:

(1) The data movement algorithm of the interconnection network (cf. Section 5.7, Appendix B, Appendix H).

(2) Physical structure of the interconnection network; timings (cf. p. 5.40-5.57, Appendix B (p. 14-25).

(3) Programming style (cf. Chapter 4, especially 4.2.2.3-4.2.2.5, 4.2.2.10, Appendix A, Appendix G).

(4) Applications benchmarked (cf. Chapter 3, Appendix A).

(5) Reliability considerations (cf. Chapter 6, esp. sections 6.1.3, 6.1.5).

2. The Data-movement Algorithm of the Interconnection Network

The Burroughs FMP machine is very close to a 512 processor ultracomputer, but involves an interesting engineering/algorithmic idea for realizing dynamically generated permutations rapidly without pre-analysis. The key idea is as follows: let a permutation $n \to p_n$ be given. Then attempt to move n into position p_n, using essentially the technique currently employed for packing. That is, examine the bits of n and p_n in sequence (for which purpose a sequence of shuffles is employed, in the ordinary way). Where these bits differ on the k-th cycle, move n to adjust its k-th bit. Since no two items are allowed to move into the same position, contention will sometimes develop between pairs of processors. Thus, the rule must be:

(1) Take a pair of integers n, n' differing only in their k-th bit; call n (resp. p_n') *k-wrong* if the k-th bit of n differs from the k-th bit of p_n (resp. p_n').

(2) (Cycle in parallel through the bits of each n in the usual way using shuffles.) If (on the k-th cycle) both n and n' are k-wrong, then interchang n and n' between p_n and p_n'. If neither n nor n' is k-wrong, do not interchange. If just one of the two items n, n' is k-wrong, do not interchange, but change the k-wrong item to nil.

(3) If both items of a pair are nil, then do not interchange. If only one item is nil, then interchange if the non-nil item is k-wrong, otherwise do not interchange.

Let the permutation p be selected at random, and suppose that on the k-th cycle of the above iteration a fraction x_k of the data items being permuted remains non-nil. An item I will be nilled only if it belongs to a pair of items neither of which is nil, and only I is k-wrong and its partner is not. The probability of this event is $x_k^2 \frac{1}{2} \frac{1}{2} = \frac{1}{4} x_k^2$, so that x_k satisfies the recursive rule

$$x_{k+1} = x_k - \frac{1}{4} x_k^2.$$

The sequence generated by this recurrence is:

0	1.000	8	.300	16	.183	24	.133
1	.750	9	.278	17	.175	25	.128
2	.609	10	.259	18	.167	26	.124
3	.517	11	.242	19	.160	27	.120
4	.450	12	.227	20	.154	28	.117
5	.399	13	.214	21	.148	29	.113
6	.359	14	.203	22	.142	30	.110
7	.327	15	.193	23	.137	31	.107

Thus, in a 512-processor system, 26% of requests will be transmitted successfully thru 9 shuffle steps; in a 16K-processor system, 19% of requests will be transmitted successfully.

The bandwidth of the connection network can be increased by duplexing (or quadruplexing) and putting 2 (or 4) copies of each data item on the network at 2 (or 4) different priority levels. If this is done, then the percentage of requests handled successfully rises as follows (provided that the pattern of request addresses is random):

512 processors, duplexed switch 48% successful requests
512 processors, quadruplexed switch 73% successful requests
16K processors, duplexed switch 36% successful requests
16K processors, quadruplexed switch 60% successful requests

It is also worth considering the fraction of requests that will have been handled successfully at the end of several successive full permutation operations. This is shown in the following table:

number of processors	duplexing	permu-tation 1	permu-tation 2	permu-tation 3	permu-tation 4
512	none	.28	.53	.75	.91
16K	none	.20	.40	.58	.74
512	duplex	.48	.83	.99	1.00
16K	duplex	.36	.68	.91	1.00
512	quadruplex	.73	.99	1.00	1.00
16K	quadruplex	.60	.95	1.00	1.00

This makes it plain that in the average case quadruplexing gives quite a good interconnection bandwidth, even for large numbers of processors.

Somewhat better results can be obtained by bringing together two pairs of data items, rather than simply two items, at each node, and nilling a data item only when more than two items at a given node wish to proceed to the same successor node. Again consider a random permutation, and let x_k be the fraction of items which are non-nil when the k-th bit of all permutation addresses n are being examined. Then the expected number of items that will become nil per group of 4 is

$$
\begin{aligned}
&1 * \text{probability that 3 non-nil items appear in the group} \\
&\quad * \text{probability that all 3 have the same destination} \\
+&2 * \text{probability that 4 non-nil items appear in the group} \\
&\quad * \text{probability that all 4 have the same destination} \\
+&1 * \text{probability that 4 non-nil items appear in the group} \\
&\quad * \text{probability that 3 of 4 have the same destination}
\end{aligned}
$$

As a rough approximation we assume that one item is nil independent of another and obtain the recurrence

$$x_{k+1} = x_k - \frac{1}{4}(x_k^3 1 - x_k) + (\frac{2}{8} + \frac{1}{2})x_k^4) = x_k - \frac{1}{4}(x_k^3 - \frac{1}{4}x_k^4).$$

This recurrence generates the following sequence:

1	1.000	9	.452	17	.339	25	.283	33	.247	41	.223
2	.813	10	.432	18	.330	26	.278	34	.244	42	.220
3	.706	11	.414	19	.322	27	.273	35	.240	43	.217
4	.633	12	.398	20	.314	28	.268	36	.237	44	.215
5	.580	13	.384	21	.307	29	.263	37	.234	45	.213
6	.538	14	.371	22	.301	30	.259	38	.231	46	.210
7	.504	15	.360	23	.294	31	.255	39	.228	47	.208
8	.476	16	.349	24	.288	32	.251	40	.225	48	.205

Again one can duplex or quadruplex. The following table shows the number of data items handled successfully at the end of several successive full permutation operations:

number of processors	duplexing	permu-tation 1	permu-tation 2	permu-tation 3	permu-tation 4
512	none	.45	.81	.99	1.00
16K	none	.37	.70	.94	1.00
512	duplex	.70	.99	1.00	1.00
16K	duplex	.60	.96	1.00	1.00
512	quadruplex	.91	1.00	1.00	1.00
16K	quadruplex	.84	1.00	1.00	1.00

3. Physical Structure of the Interconnection Network; Timings

The FMP interconnection network has 24-bit wide data paths (for 48-bit words), and the successive phases of the data transmission algorithm are 'unrolled' into 9 (or 10/11 if duplexing/quadruplexing is used) successive layers of gates to avoid the additional latching expense of an iterative algorithm. The 512 processor system is estimated to have a 30' greatest physical distance. Assuming no duplexing, 9 levels of delay are involved in setting up a path thru the connection network; 120 ns, divided into a succession of 20 ns subcycles, are allowed for this. Then 2 more cycles are needed to transmit a data word, for a total time of 360 ns thru the connection network, giving a 600 ns total access time since the memory cycle time is 240 ns. This is reasonable balance with a 400 ns floating multiply time.

Processors whose requests are refused receive a reflected nondelivery notification before the end of the 120 ns request cycle.

In a 512 processor system a processor address is 9 bits, and thus two such addresses can be brought together on a pair of 24-pin chips which can accomplish the basic control and interchange function. 512 such chips will be required, and these can be accomodated on two 1.5' x l.5' boards, making a 10 ns subcycle possible. Latching is then a relatively small overhead, so that an iterative algorithm of 9 (or 10/11) cycles might as well be used, as this will reduce chip count considerably. Data motion can proceed in parallel with address processing; of course, address processing will generate all necessary data-routing control bits. If a 48-bit wide data path is provided and data is moved iteratively, then roughly 8 24-bit chips are needed per pair of words, or 2000 chips in all, which can be housed on 8 more 1.5' x 1.5' boards. Thus, an entire simplex interconnection network could be housed in a 1.5' cube, and a quadruplex network in a 2.5' cube, again confirming the possibility of a 10 ns switching subcycle. Assuming quadruplexing, switching will then require something like 150 ns, to which 60 ns delay in communicating data to/from the switch should be added for a total of 210 ns. point-to-point.

This compares favorably with the 240 ns. memory access time. In a pure register-register ultracomputer architecture providing shuffles only, a single shuffle step would require roughly approximately 70-80 ns., and thus is probably not advantageous in face of the greater logical generality provided by the FMP's 'full permutation' scheme. This indicates that *the shuffle network and its associated algorithms*

can be hidden at what is in effect a 'microcode level' within the parallel computer, making it possible for the user to think in terms of a simpler, any-processor-to-any-memory-box 'machine level' logic.

An eclectic scheme might provide very fast nearest-neighbor communication based on sharing of registers between nearest neighbors, plus the FMP scheme for remoter data communication.

4. Programming Style

Burroughs proposes a FORTRAN-based programming approach, with only a few dictions added to FORTRAN. The dictions added are a parallel DOALL/ENDDO loop construction and a rather crude structure declaration. Statements within the body of a DOALL are executable in parallel, and the DOALL body is simply executed by all available processors until all its (logically independent) iterations have been accomplished.

Public data (as distinct from data private to a particular processor) is available to all processors during a DOALL, but results are written back to the public area only at the end of the DOALL.

Assuming that the switch performs well, all that is necessary to achieve efficiency within a DOALL is to ensure that the memory accesses are distributed as uniformly as possible over the memory modules provided.

Our current ultracomputer algorithms could all be written as DOALLS, and would have the property that neither interconnection network contention nor memory contention ever developed. This is a reasonable approach for algorithms involving any significant amount of floating point arithmetic. Simpler data motion and communication algorithms (including permutation, packing, summing, and sorting) might better be built into the underlying, micro-coded hardware, and appear to the ordinary user as a set of machine-level primitives, some of which will be discussed below.

Code outside of DOALLS is strictly sequential, and is executed by just one processor. The machine can therefore be regarded as a sequential machine capable of executing DOALLS at 512 times its normal rate. This suggests timesharing the parallel hardware among a number of sequential processors, to ensure that enough DOALLS to keep the parallel hardware busy are always available.

Major synchronization points occur precisely where arrays are updated. This suggests a generalized programming approach in which synchronization is required only at those points at which public scalar quantities or arrays are modified. In such a style, the synchronization point could be indicated simply by a pair of rudimentary SYNC commands, which would hold up early-arrivers until all processors had executed it. Then the characteristic of DOALL construct would simply be that it partitions a certain iterative mass of computational work (initially written as a group of independent iterations) statically, using the fact that the number of processors available is known, so that the iterative work can simply be partitioned in to 512 parts, and executed independently by the processors, which have only to go thru the loop by steps of 512, synchronizing on exit.

Suppose that a DOALL is used to modify a PUBLIC array A, which for simplicity we suppose to be a simple linear array of dimension 512*K. Then within the body of the DOALL the new values of A

would be stored in a (implicit temporary) PRIVATE array A' of dimension K, which might be called the 'shadow' of A. Upon exit from DOALL, A' would be transmitted to A. The code for this can be written using the SYNC primitive and an ordinary FORTRAN DO-loop simply as:

```
      SYNC
      DO 1 J = MYID, 512*K, 512
1     A(J) = A' (J)
      SYNC
```

Note that with a bit more hardware, the more general programming style of the earlier Draughon-Stein-Schwart-Grishman work on parallel computation (see *Programming Considerations for Parallel Computers,* NYU Courant Institute Tech. Report IMM 362, November 1967; *Individual and Multi-Processing Performance Characteristics of Programs on Large Parallel Computers,* IMM 380, April 1970) can be made available. The semantic primitives employed in that report are as follows:

(1) PUBLIC and PRIVATE variables

(2) A primitive function operating on halfword (or even two-byte) integers, which appears in FORTRAN as NEWVAL (I,J). As described in IMM 362, 'This function has the value I + J. Moreover, each time it is called, it changes the value of I to I + J. If several processors call this function simultaneously, the effect is the same as if these processors called the function in some serial order.'

Quantities I addressed by NEWVAL will ordinarily be PUBLIC and used for semaphoring, for distribution of processors among control paths, and to control busy-waits during SYNC-like operations. Hence such quantities may well be accessed by many processors almost simultaneously, and it may be desirable to give them a hardware implementation which prevents contention problems from developing. This can be done by a technique sketched just below, but since this technique is relatively expensive one will probably want to do this for a limited number of specially declared quantities, for which purpose the declaration

PUBLIC SEMAPHORE

or simply

SEMAPHORE

is suggested.

The following is a possible treatment of SEMAPHORE variables I at the hardware level:

(1) Allow up to 512 such variables, packed one per memory box.

(2) Suppose first that such variables can only be addressed by the instruction NEWVAL(I,J). (Note that J=0 gives a simple 'LOAD' instruction).

(3) Transmit all the instructions NEWVAL(I,J) issued on a single clock cycle to a built-in ultracomputer (hidden, like the ultracomputer structure of the communications network itself, at a 'micro-coded', 'hardware' level.)

The identifier of the processor originating the command should be kept with the command itself. Using the ultracomputer sorting algorithm, sort these instructions by their distinction I. Retrieve the value held in the location I, and merge to place this value in front of the group of instructions with destination I (this can be done in time proportional to log N.) Then use the ultracomputer summing algorithm (more precisely, summing by groups) to form the partial sums of all J's associated with a given destination I. Reorder (by 'packing' and then by a communication step that can be done as rapidly as 'merging', i.e. in time O (log N)) to place the last sum in each group in proximity with the associated destination address I. Next store to update the semaphore cell I. Finally, re-sort to re-transmit to the address of originating processor, thus delivering the NEWVAL (I,J) values to the processors.

This algorithm will require 2 sorting times (roughly 100 micro-cycles) plus an overhead of roughly 50% for the various summing and merging substeps. A total of 250 30 ns cycles, or 7.5 μus, is therefore reasonable; this is equivalent to roughly 12 standard mass memory accesses, or 20 floating point instructions. The hardware needed would roughly equal that needed for a duplex communication network.

Note however that only one semaphore is required to support the very simple logic of the Burroughs DOALL construct, and that if only one is provided it can be implemented rapidly and cheaply as a 512 way OR with result available to all 512 processors; this could be generated in 60 ns.

If the NEWVAL function is provided, then an unordered version of the ultracomputer 'packing' algorithm reduces to the simple sequence

```
1  IF NEWVAL (Z,0) ≠ 0 THEN GO TO 1
   IF DATA_FLAG(I) ≠ 0 THEN PACKED (NEWVAL(Z,1)) =  DATA (I)
```

where Z should be a SEMAPHORE variable initialized to zero.

By supporting the NEWVAL function in hardware, we make it possible for the processors of the parallel computer to be used effectively in subgroups or varying size, which might otherwise not be possible. Thus, NEWVAL is probably useful for the system flexibility which it provides.

In the present Burroughs scheme, the simplest external sign that a DOALL is possible is simply the presence of a single DO or a nest of DOs in which no variable (other than a DO index) appears on both left and right, and in which no location is the target of an assignment in more than one instance.

5. Applications Benchmarked

A variety of aero and metorological applications supplied by NASA have been benchmarked, apparently with very satisfactory results. (However, it is not entirely clear that measurements of innermost loops gives an entirely fair picture of the performance of a machine that favors inner loops so very strongly.) Instruction rates of more than 1 gigaflop are reported, but with 50% falloff in a less favorable weather application. However, analysis was by hand-compilation and simulation only, without full machine simulation. Between 5 and 20 floating point operation between non-PRIVATE memory references are projected, (32K words of PRIVATE, and 64K words of PUBLIC memory per processor are

provided.) Parallelism within innermost DOALLS depends in some cases on the fact that a two-dimensional sub-grid of a 3-dimensional problem is being treated in parallel. In another somewhat more favorable case, each processor can perform an FFT independently within the innermost loop.

In all but the most favorable cases at least a bit of recoding in the parallel 'FMP FORTRAN' is needed to push system utilization up.

The parallel FFT can be executed at about .5 gigaflops with the communications hardware provided (cf. pages A-62 thru A-64). Note that the more perfect use of the perfect shuffle would only improve this by 20%. Thus, the cost of the Burroughs data routing scheme in this, a particularly favorable ultracomputer case, is quite modest.

6. Reliability Considerations

The bulk of the FMP hardware lies in its processing and memory units. The communication network will be relatively small, especially if iteration is used instead of implementing the data movement algorithm by fully unrolled hardware.

Reliability is therefore achieved by allowing defective processing units (and/or memory units) to be switched out and replaced by spares. The units are divided into 4 groups of 128 units each, and one spare, numbered 128, is provided per group. When a unit becomes defective, each unit following it is logically shifted one position to the left, and the spare unit 128 becomes unit 127. This only requires the ability to switch a unit between two adjacent connection-network ports.

All data is tagged with (single error connecting, double error detecting) correction bits. Thus, transient or hard errors on the communication network should show up immediately. If necessary, it should be possible to increase the reliability of a quadruplexed communication network simply by switching malfunctioning nodes off. Since multiple data paths always exist, this should cause only slight degradation in the communications service experienced by units which use the deleted point communication network nodes. However, this point deserves closer study.

7. Summary

Overall, the FMP design looks very attractive. Since it should be as easy to program as a parallel computer can be, and since it scales up smoothly to allow use of very large number of processors, it may well set the pattern for the coming generation of parallel machines, in much the same way that the 'von Neumann' architecture has typified the last 25 years. The degree to which the perfect shuffle network used to accomplish data motion and the ultracomputer algorithms naturally associated with this network become explicit will depend on detailed engineering parameters: e.g., the relative physical sizes of the switching network and the mass of processors surrounding it. Use of a superspeed technology to realize the switching network may also impact the architecture. Switching times which are small relative to end-to-end data delays and operation time will favor the Burroughs approach. The relatively high-level programming approach which the FMP allows, is, of course, a great advantage.

Table of Contents

1. Introduction 1
2. The Data-movement Algorithm of the Interconnection Network 1
3. Physical Structure of the Interconnection Network; Timings 5
4. Programming Style 6
5. Applications Benchmarked 8
6. Reliability Considerations 9
7. Summary 9