Preliminary Thoughts on Ultracomputer Programming Style

by

J.T. Schwartz

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

Ultracomputer Note #3
May, 1979

Generally, but not always, we will assume that the processors of the ultracomputer are operating in synchronous (or single-instruction/multiple-data) mode, i.e. all executing the same instruction during each instruction cycle. Thus, for example, in writing the instruction sequence

```
DO J = 1 TO 100;
  A (J) = B (J) + C (J) ∗ D (J) + X;
END DO;
```

we may assume that:

(1) J designates a vector of NPROC integers, all of which are simultaneously initialized, advanced, tested, etc., during execution of the loop shown above. Likewise, X designates a vector of NPROC reals.

(2) A(J), B(J), and C(J) designate arrays of vectors, of some declared intent DIM × NPROC (e.g., 100 × NPROC). All the separate additions and multiplications appearing in the above loop (e.g., the NPROC multiplications C(J) * D(J) (for a fixed value of J) are performed synchronously; likewise the NPROC stores indicated by A(J) = ... (for a fixed value of J).

The most important point here is that *once synchronous operation is established it will persist until disrupted by a conditional transfer involving a (vector valued) expression whose components may differ.*

To be specific, we shall assume the syntax and semantic environment of LITTLE (with details such as word size, real arithmetic format, signed integer format, etc., defined by the particular machine used as the individual processor of the ultracomputer).

Next consider an IF statement, such as

```
IF D ≥ NPROC/2 THEN
  J = J + 1;
ELSE
  J = J - 1;
END IF;
```

The test with which this statement begins will affect certain of the processors one way and the remainder another, thus destroying synchronization. We will often wish to reestablish synchronized operation at the end of such a conditional block (and sometimes at other program points as well). For this purpose, we provide the very special operation

```
SYNCH;
```

This operation delays all processors which execute it until all processors have arrived at the SYNCH operation, at which time they all proceed in resynchronized fashion. (A plausibly efficient hardware implementation of this operation will be suggested below.) Thus, we can reestablish synchronization after the IF-statement shown above by placing a ‘SYNCH’ immediately after the ‘END IF’. Because this construction will be rather common, we allow the combination

```
END IF;
SYNCH;
```

to be abbreviated as

ENDS IF;

Similarly, we allow ‘ENDS DO;’ and ‘ENDS WHILE;’ to reestablish synchronized operation immediately after a DO-loop or WHILE-loop which may have lost synchronization. Moreover, we allow ‘RETURNS’ to reestablish synchronization upon return from a subroutine or function.

The compiler which translates our parallel language into ultraprocessor machine code is allowed to replace any use of the SYNCH hardware operation by several cycles of delaying no-ops if the precise number of cycles required to traverse the different branches of a nest of IF statements can be precalculated. (For example, in generating code for the IF statement shown above, the compiler may know that both branches of the IF will take equally long to execute, so that this particular IF statement may preserve synchronization, even though other IF-statements may not. Having recognized this, the compiler can implement the ‘ENDS’ occurring at the end of this particular IF statement just as if it were ‘END’. Note that all this suggests a kind of synchronization-related optimization that can probably be handled by standard global analysis techniques.)

We use

LEFT I and RIGHT I

to denote the left-shift (vs. right shift) of the NPROC-dimensional vector value of I among processors. Similarly, we use

SHUF I and ISHUF I

to designate the shuffled and inverse-shuffled value of I. The component of I held in the N-th processor will be written as I[n]; thus

(SHUF I) [n] = I [if even (n) then n/2 else n/2+NPROC/2]
(ISHUF I) [n]= I [if n < NPROC/2 then 2n else 2n–NPROC+1].

The operations SHUF and ISHUF can be used even during periods of ansynchronous operation, though of course they will more ordinarily be used in procedures whose expected effect depends on synchronous operation.

1. Code for Various Primitives

We will now illustrate the flavor of the proposed programming language by writing procedures which represent various of the important primitives reviewed in [Schwartz, 79]. All of these procedures must be entered synchronously

1.1. Summing

Given X, this forms SUM(X), which is the vector of partial sums of X.

```
FUNC SUM (X);  $ PARTIAL SUMS ∅F X
REAL X;         $ PARAMETER
REAL Y;         $ SCRATCH QUANTITY
SIZE J(INDEX_SIZE);  $ AUXILIARY INDEX
$ THE SYSTEM QUANTITY 'LOGNPROC' IS THE LOGARITHM
              OF 'NPROC'.
Y = X
IF LOWBIT ID = 1 THEN  $ ODD PROCESSOR
    Y = X + LEFT X;
ENDS IF;
DO   J = 1 TO LOGNPROC – 1;
     Y = ISHUF Y;
     IF (LOWBIT ID = 1) & (HIGHBITS(J) (¬ID) = 0) THEN
        Y = Y + LEFT Y;
     ENDS IF;
END  DO;  $ SYNCHRONIZATION IS PRESERVED HERE
DO   J = 2 TO LOGNPROC;
    Y = SHUF Y
    IF (LOWBIT ID = 0) & ID> (NPROC−1−LOWBITS (J) (NPROC−1) THEN
       Y = Y + LEFT Y;
    ENDS IF;
END DO;   $ SYNCHRONIZATION IS PRESERVED HERE
RETURN Y;
END FUNC SUM;
```

It is clear that partial product, maximum, minimum, etc. routines can be written to the same pattern. Note also that the Fast Fourier Transform algorithm can be regarded as having a similar pattern.

1.2. Taking

The procedure RTAKE(X,MARK) regards the components of the quantity MARK as indicating the division of X into groups, a group beginning at each component n such that MARK[n] ≠ 0. The value Y returned is defined by Y[n] = X[m], where m is the largest integer not greater than n such that MARK[m] ≠ 0. The following code shows the necessary processing pattern (which must resembles 'summing' , cf. (a) above). Note that if no component to the right of n (i.e., with smaller index than n) is marked, then RTAKE(X,MARK)[n] = X[0].

```
FUNC RTAKE (X, MARK);
REAL X;     $ PARAMETER
REAL Y;     $ SCRATCH QUANTITY
SIZE MARK (WORD_SIZE);   $ MARK PARAMETER
SIZE MARCOP (WORD_SIZE); $ INITIALLY  THIS IS A COPY OF MARK, BUT IT
                           WILL CHANGE
SIZE J (INDEX_SIZE)      $ AUXILIARY INDEX
Y = X;  MARKCOP = MARK;
IF (LOWBIT ID = 1) THEN
  IF MARCOP = 0 THEN
    Y = LEFT Y;
    MARCOP = MARCOP  LEFT MARCOP;
  END IF;
ENDS IF;
DO J = 1 TO LOGNPROC – 1;
    Y = ISHUF Y;
    MARCOP = ISHUF MARCOP;
    IF (LOWBIT ID = 1) & (HIGHBITS(J)  (¬ID) = 0) THEN
      IF MARCOP = 0 THEN
        Y = LEFT Y;
        MARCOP = MARCOP  LEFT MARCOP;
      END IF MARCOP;
    ENDS IF;
END DO;
DO J = 2 TO LOGNPROC;
    Y = SHUF Y;
    MARCOP = SHUF MARCOP;
    IF (LOWBIT ID = 0) & (ID > NPROC – 1 –
                  LOWBITS(J–1) (NPROC – 1)) THEN
      IF MARCOP = 0 THEN
        Y = LEFT Y;
        MARCOP = MARCOP  LEFT MARCOP;
      END IF MARCOP;
    ENDS IF;
END DO;
RETURN Y;
END FUNC RTAKE;
```

Various other useful operations can be expressed in terms of RTAKE. For example, to transmit the value of X[0] to all other processors we have only to write Y = RTAKE(X,ID=0). It is easy to program a companion LTAKE(X, MARK) to RTAKE which works in much the same way but uses the first mark to the left rather than to the right of a given place. Using this operator and the operator MINS(X) (analogous to SUM) which calculates the 'partial minima' (rather than partial susm) of the components of X, we can put the minimum of all the components of X into every processor simply by writing Y = LTAKE (MINS(X), ID = (NPROC – 1)). (As a matter of fact, it is easy to write a procedure having this same effect which is almost twice as fast).

1.3. Data permutation

As noted in [Schw, 79], an arbitrary permutation f of NPROC items can be factored as

$f = g_1\sigma^{-1}g_2 \ldots \sigma^{-1}g_D\sigma g_{D+1} \ldots \sigma g_{2D-1}$,

where D = LOGNPROC, σ designates the shuffle map, and each of the permutations g_j interchanges some of the collection of all possible even-odd pairs {2n, 2n+1} of integers. Accordingly, each map g_j is described by a vector quantity G_j of dimension NPROC with single-bit components, and all the maps G_j together define a vector quantity G with bitstring components. (We assume that G[2n] = G[2n+1] for all n.) Once the quantity G has been calculated, the permutation which it describes can be effected by calling the following rather simple function.

```
FUNC PERM (X,G);
REAL X;    $ PARAMETER
REAL Y,Z;  $ SCRATCH QUANTITIES
SIZE J (INDEX_SIZE); $ AUXILIARY INDEX
SIZE ISODD (1);  $ NONZERO IF ODD PROCESSOR
ISODD F = LOWBIT ID;
Y = X;
IF (BIT(1) G) & ISODD THEN
    Y = LEFT X;
ELSEIF BIT (1) G THEN
    Y = RIGHT X;
ENDS IF;
DO J = 2 TO NPROC;
    Z = SHUF Y;  Y = Z;
    IF (BIT(J) G) & ISODD THEN
      Y = LEFT Z;
    ELSEIF (BIT(J) G) THEN
      Y = RIGHT Z;
    ENDS IF;
END DO;
DO J = NPROC + 1 TO 2∗NPROC – 1;
    Z = ISHUF Y;  Y = Z;
    IF (BIT(J) G) & ISODD THEN
      Y = LEFT Z;
    ELSEIF (BIT(J) G) THEN
      Y = RIGHT Z;
    ENDS IF;
END DO;
RETURN Y;
END FUNC PERM;
```

1.4. Bitonic murging and sorting

The following codes represent the bitonic murging and sorting operations, as described in [Schw, 79]. Note that MURGE has a structure closely related to that of PERM; cf. (c) above.

```
FUNC MURGE (X);   $ PERFORMS BITONIC MURGING;
                  $ X[n] IS ASSUMED TO RISE TILL
                  $ n = NPROC/2 – 1, THEN FALL
REAL X;   $ PARAMETER
REAL Y,Z;  $ SCRATCH QUANTITIES
SIZE J (INDEX_SIZE);  $ AUXILIARY INDEX
SIZE ISODD(1);  $ NONZERO IF ODD PROCESSOR
ISODD = LOWBIT ID;
Y = X;
DO J = 1 TO LOGNPROC;
    Z = SHUF Y;  Y = Z;
    IF ISODD & Z < LEFT Z THEN
      Y = LEFT Z;
    ELSE IF ¬ISODD & Z > RIGHT Z THEN
      Y = RIGHT Z;
    ENDS IF;
END DO;
RETURN Y;
END FUNC MURGE;
```

Bitonic sorting is a somewhat more complex operation. We can regard it as a merging sort based on the MURGE operation described by the preceding program (see [Schw, 79]; [Knuth, v. II]). However, the implementation we desire is iterative rather than recursive; also we will keep reversing the signs of the quantities being merged in order to maintain the rising/falling pattern expected by the murge subo-perations. The code is as follows:

```
PROC BITON (X);
REAL X;    $ PARAMETER TO BE SORTED
REAL Y,Z;  $ SCRATCH QUANTITIES
SIZE J(INDEX_SIZE);  $ AUXILIARY INDEX
SIZE ISODD(1);        $ NONZERO IF ODD PROCESSOR
ISODD = LOWBIT ID;
Y = X;
IF (BIT(1) ID) THEN  $ REVERSE, TO PRECONDITION
   Y = –Y;
ENDS IF;
DO J = 1 TO LOGNPROC/2;
    IF (BIT(J) ID ≠ BIT (J + 1) ID) THEN
      Y = – Y;    $ MUST REVERSE AGAIN FOR THIS CYCLE
    ENDS IF;
    DO K=1 TO J;   $ ISHUFFLE REPEATEDLY TO PUT
      Y = ISHUF Y; $ INTO POSITION FOR MURGING
    END DO;
    DO K=1 TO J;        $ THIS NOW RESEMBLES
      Z = SHUF Y;  Y = Z;  $ THE MURGE LOOP
      IF ISODD & Z < LEFT Z THEN
```

```
          Y = LEFT Z;
        ELSE IF ¬ISODD & Z > RIGHT Z THEN
          Y = RIGHT Z;
        ENDS IF;
      END DO K;
END DO J;
$ NOW WE BEGIN THE SECOND HALF OF THE PROCESSING,
$ WHICH ONLY DIFFERS FROM THE FIRST HALF TO SAVE
$ UNNECESSARY ISHUFFLES WHENEVER THE SAME EFFECT CAN
$ BE OBTAINED BY A COMPLEMENTARY NUMBER OF SHUFFLES.
DO J = LOGNPROC/2 + 1 TO LOGNPROC;
    IF (BIT (J) ID ≠ BIT (J + 1) ID) THEN
      Y = –Y;  $ MUST REVERSE AGAIN FOR THIS CYCLE
    ENDS IF;
    DO K=1 TO LOGNPROC–J; $ NOW IT IS MORE EFFICIENT
      Y = SHUF Y;       $ TO SHUFFLE REPEATEDLY
    END DO;
    DO K=1 TO J;         $ AGAIN THIS RESEMBLES
      Z = SHUF Y; Y =Z;  $ THE MURGE LOOP
      IF ISODD & Z < LEFT Z THEN
        Y = LEFT Z;
      ELSE IF ¬ISODD & Z > RIGHT Z THEN
        Y = RIGHT Z;
      ENDS IF;
    END DO K;
END DO J;
RETURN Y;
END FUNC BITON;
```

As noted in [Schw, 79] a wide variety of set-theoretic, communication, and other functions can be expressed quite directly in terms of BITON.

2. A Software Emulation Technique

A reasonably efficient and acceptably accurate software emulation for programs written in the language that we have described can be constructed by modifying the LITTLE compiler. We can proceed as follows: all operations except three of those involving interprocessor communication (namely SYNCH, SHUF, and ISHUF) essentially the ordinary manner, except that only a limited number of registers (e.g., X1, X6, A1, A6, B1, B6 on the 6600 are used). Arrays declared to be of dimension N are treated as if they were of dimension N∗PROC; simple variables are treated as quantities of dimension NPROC. Status information for all the available processors (consisting of register contents and instruction location counter) is stored in an array of appropriate size. The ID of the currently emulated processor is held in some specially allocated register.

When emulation switches to a given processor, the register contents for that processor are loaded and a jump to the current instruction location for that processor is taken. Execution then continues in standard fashion until an operation requiring synchronization is encountered. Preceding any such operation, we compile a call to a special ‘advance processor’ procedure which stores the current register and

ILC, and advances to emulation of the next processor. When all processors have reached the synchronization point we let them proceed (in the case of a SYNCH) or we execute a shuffle or an inverse shuffle (in the SHUF and ISHUF cases). Then emulation begins again with the first processor.

Note that this approach avoids the overhead of interpretation. It comes close to emulating, but does not quite capture all the details of, the hardware synchronization semantics described earlier. In particular, LEFT and RIGHT may not operate quite as synchronously as desired, whereas SHUF and ISHUF can force synchronization in a more restrictive manner than hardware is likely to. (But differences should only be seen in certain esoteric, even if conceivable, cases. In particular, all the parallel codes listed in the preceding pages will execute without difficulty.)

Deadlock will occur if two different processors attempt to execute operations requiring synchronization at two different points. Such deadlock can easily be detected and diagnosed when it does occur.

3. A Remark on Hardware Implementation of the SYNCH Primitive

The SYNCH primitive can be implemented as follows (for a large but not enormous collection of processors). Let I_j be the instruction location counter of the j-th processor, and $\overline{I}_j$ its boolean inverse. Form

$$S = \bigwedge_{k=1}^{L} \left[\bigvee_{j=1}^{NPROC} (BIT(k) I_j) \bigvee_{j=1}^{NPROC} (BIT(k) \overline{I}_j) \right],$$

where L is the length of I_j. Then S=0 if and only if the processors are synchronized. Done efficiently by dot-oring, this should not require much more than 5-6 levels of logic. Processors executing SYNCH can simply delay until S=0.

It is also easy to give a deadlock check: let each processor executing SYNCH drop a line *sy*, and tie all those lines together in a dot-or tree to get a single result *syt* which will only be zero if all processors are attempting to synchronize. If *syt* is off but S is on, a deadlock exists.

A similar technique can be used to measure the number of cycles wasted by processors attempting to synchronize.