PLUS: A PL/I Based Ultracomputer Simulator, I

by

Allan Gottlieb

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

Ultracomputer Note #10
September, 1980

ABSTRACT

This note describes a (presently implemented) ultracomputer emulator called PLUS which uses the multitasking and preprocessing features of PL/I to support a recursive ultracomputer programming style. Since the emulation to be described as written in PL/I, the powerful debugging features of the PL/I checkout compiler, as well as PL/I's separate compilation facility are available. This note also describes the emulation system and furnish a ''User's Guide''. In a subsequent part II we will discuss the system's implementation and prove both correctness and freedom from deadlock.

1. Introduction

[UC] introduced the concept of an ultracomputer and reviewed various basic algorithms for such an ensemble of processors containing ''shuffle'' [Clos, Benes, and Stone] interconnections. [UCN3] described a style for programming ultracomputers and rewrote several of the basic algorithms in this style. This new style forbids recursion so the recursive algorithms of [UC] appeared in iterative form in [UCN3]. Although removing recursion raises no theoretical obstacle, it creates obvious practical irritations. Nevertheless, we can allow recursion within an ultracomputer programming language. Just as for uniprocessors, the resulting recursive implementations of ultracomputer algorithms may require more ultracomputer cycles, memory space, and may make more synchronization requests than non-recursive variants of the same algorithm.

Thus one would not use such an implementation for the final version of a production program. Nevertheless, the simplicity of the recursive form of an algorithm will sometimes make it attractive. In particular, since ultracomputer algorithms frequently employ a divide and conquer strategy, the use of recursion often results in reduced programming effort and more natural code.

This note will describe a (presently implemented) ultracomputer emulator called PLUS which uses the multitasking and preprocessing features of PL/I [LRM] to support a recursive ultracomputer programming style. Since the emulation to be described is written in PL/I, the powerful debugging features of the PL/I checkout compiler, as well as PL/I's separate compilation faciltiy are available.

Due to the modular nature of PLUS's design, only a minor effort is needed to reconfigure it to support interconnection schemes other than the ultracomputer shuffle. In particular, the layered ultracomputer variant of [UC] and the multidimensional variants of Harrison and Kalos (see [UCN6]) are easy to emulate.

In this note we describe the emulation system and furnish a ''User's Guide''. In a subsequent part II [PLUS2] we will discuss the system's implementation and prove both correctness and freedom from deadlock.

Section II of this paper introduces the PLUS model of multiple processors and the synchronization issues that emerge. Section III is a users guide to PLUS. Sections IV and V are illustrative, and present PLUS implementations of summing and packing, two algorithms taken from [UC]. We believe that the resulting code constitutes a ''natural'' implementation of these algorithms. Finally, section VI discusses

the PLUS supplied PL/I main program.

2. Synchronization Requirements

As suggested in [UCN3], we suppose that all processors in the ultracomputer will execute the same program. Note that this does not imply an SIMD architecture since conditional statements are permitted and the processors execute asynchronously. Our basic idea is to write such programs as PL/I procedures containing an additional parameter representing the processor number. Then this procedure is invoked as a *task*, once for each processor. The PL/I multitasking facility allows the multiple invocations of the procedure thereby created to run ‘‘in parallel’’.

Communication between each processor and its four neighbors (via nearest neighbor and shuffle connections) is handled using global arrays. Consider the SUMMING procedure as an example and assume that the declaration

```
DECLARE W (0:MAX_PE) FLOAT;
```

appears global to the procedure definition for SUMMING, where here, as elsewhere, PLUS uses PE to abbreviate ‘‘processing element’’ or ‘‘processor’’. Then the task corresponding to processor N will refer to W(N) for the value stored in processor N and to W(RIGHT_PE(N)), W(LEFT_PE(N)), W(SHUFFLE_PE(N)), and W(UNSHUFFLE_PE(N)) for the values stored in its four neighbors.

Appropriate sychronization is required to insure that if one processor references a variable stored in a logical neighbor, the value obtained is current. This issue also appears in the model proposed in [UCN3] where a conditional statement[1] can cause the processors to lose synchronization and will therefore often end with a resynchronization request. Of course, it is neither necessary nor desirable for the tasks constituting our emulation to be in step at all times; as long as they are referencing only local variables, they may run completely asynchronously. But non-local references require more careful treatment. Consider once more the global declaration

```
DECLARE W(0:MAX_PE) FLOAT;
```

and assume that for each N, processor N executes

```
W(N) = W(N-1) + W(N);
```

Care is needed, since if (without the programmer intending this to be the case) processor 3 updated W(3) prior to processor 4 referencing W(3), the value assigned to W(4) would be incorrect.

Given the above global declaration we will say that task N owns the component W(N). We consider W(N) to be local to task N and non-local to all tasks M≠N. Our model forbids a task to update a non-local variable. We also insist that whenever a task references a non-local variable, it does so using a synchronization macro.

[1]With a condition depending on the processor number as implied by the use of such dictions as ‘‘each even numbered processor adds 1 to x’’

Should only a proper subset of the tasks require a non-local reference, these tasks (called ‘‘snoopers’’ since they are to examine nonlocal data) execute the macro, SYNC_SET. This macro synchronizes all these processors and assigns the nonlocal value being referenced to a local variable. The other tasks (called ‘‘observers’’) execute the macro, SYNC, that synchronizes them with the snoopers but does no assignment. When a task executes one of these macros, that task enters a wait state and remains in this state until all the tasks have begun executing either SYNC_SET or SYNC. Eventually, all the tasks are waiting. At this point, with the help of a software module called the gatekeeper (described in detail in [PLUS2]), each snooper is allowed to evaluate its nonlocal expression. The tasks wait again, assuring that all the expressions are evaluated, and finally the gatekeeper allows them to proceed once more. The snoopers are free to complete their assignments and each task may leave its macro.

We prove in [PLUS2] that this scheme is deadlock free. Naturally deadlock may occur if the system is used incorrectly. If only a proper subset of the tasks execute a macro, they will wait while the others proceed. Should these later tasks terminate, the system deadlocks. Thus another requirement is that when one task synchronizes, they all do. All the above requirements can be combined to yield:

> Non-local updates are forbidden. When non-local references are required, *every* task executes a synchronization macro. The snoopers SYNC_SET a local variable EQUAL_TO a non-local expression. The observers SYNC.

The SYNC and SYNC_SET macros referred to above are discussed below in later sections. The system is proved to function correctly and without deadlocks in [PLUS2] where the implementation is given in detail.

3. USER’S GUIDE

In this section we describe each of the PLUS facilities in a terse somewhat dry manner. Many of these facilities are also described in the next two sections where we present two sample PLUS programs.

3.1. Constants

The following preprocessor variables may be used as constants by the PLUS programmer.

3.1.1. TRUE and FALSE

These boolean constants have the obvious meaning.

3.1.2. NIL

This integer constant equals -2^{15}.

3.1.3. #_PE

This integer constant is the number of processors simulated. For n a power of 2, the statement

%INCLUDE PLUSn;

specifies that #_PE = n.

3.1.4. MAX_PE# and LOG_#_PE

These integer constants equal #_PE - 1 and $\log_2$(#_PE) respectively.

3.2. (Read Only) Arrays

PLUS predefines four arrays specifying the basic ultracomputer left, right, shuffle, and unshuffle interconnection patterns and a fifth array specifying each processor's partner when the ultracomputer is logically subdivided into even-odd pairs of processors. These arrays should only be read.

3.2.1. LEFT_PE and RIGHT_PE

LEFT_PE(N) and RIGHT_PE(N) are defined as MOD(N-1,#_PE) and MOD(N+1,#_PE) respectively.

3.2.2. SHUFFLE_PE and UNSHUFFLE_PE

SHUFFLE_PE(N) and UNSHUFFLE_PE(N) are defined as $\sigma(N)$ and $\sigma^{-1}(N)$ respectively, where σ is the perfect shuffle function.

3.2.3. PARTNER_PE

PARTNER_PE(2N) and PARTNER_PE(2M+1) are defined as 2N+1 and 2M respectively.

3.3. PL/I Builtin Functions

3.3.1. MOD and COMPLETION

Since PLUS declares these functions (as BUILTIN) the user may not redeclare them (even as BUILTIN).

3.4. Macros

3.4.1. EVEN and ODD

These trivial macros have the obvious meaning: EVEN(X) becomes (MOD(X,2)=0) and ODD(X) becomes (MOD(X,2)=1).

3.4.2. SYNC_SET

This macro is the basic synchronization mechanism in PLUS. Its PL/I (preprocessor) declaration is

```
%SYNC_SET: PROC (X, EQUAL_TO, OF_TYPE, PE) RETURNS (CHAR) STMT;
```

The simplest and most common usage of the SYNC_SET macro is

```
SYNC_SET (target) EQUAL_TO (nonlocal expr);
```

This statement synchronizes the tasks and then sets the target equal to the nonlocal expression. In this example we assume that the OF_TYPE and PE parameters have been furnished via the PLUS_DEFAULT macro described below; when the target is not of the default type, one needs the following more elaborate

statement:

SYNC_SET (target) EQUAL_TO (expr) OF_TYPE (type);

This statement generates the following declaration

DCL 1 $$TEMPnnnnn type;

where nnnnn is a unique positive integer. $$TEMPnnnnn is used to hold the value of the nonlocal expression.

The PE parameter may be used to indicate the identifier specifying the processor ID. The default mechanism is nearly always adequate, however, so this parameter is very rarely used.

3.4.3. SYNC_SHUFFLE and SYNC_UNSHUFFLE

We describe only SYNC_SHUFFLE; SYNC_UNSHUFFLE is defined analogously. The most common usage is

SYNC_SHUFFLE (nonlocal array);

which expands into the appropriate SYNC_SET. Like SYNC_SET this macro permits OF_TYPE and PE parameters.

3.4.4. SYNC

This macro is used when only a proper subset of the tasks which to invoke a SYNC_SET. The remaining tasks simply include the statement

SYNC;

The only possible parameter is PE used as above.

3.4.5. SYNC_IF

This macro, used for IF statements with nonlocal conditions, has syntax

SYNC_IF (nonlocal boolean expr) THEN

where normal PL/I rules apply after the THEN. The above invocation generates a SYNC_SET having a PLUS created boolean variable as target followed by a conventional PL/I IF statement with this variable as condition.

Although it is possible to supply a PE parameter, the syntax differs from the above macros (since the SYNC_IF macro does not have the PL/I attribute). If N is the variable used as processor identifier, the above example would be coded

SYNC_IF (nonlocal boolean expr, N) THEN

N.B. The first SYNC_IF coding above generates a preprocessor warning stating that an argument is missing; however, the macro generates correct code in this case so the message may be ignored. Nonetheless, the diagnostic is embarrassing: Future versions of PLUS may eliminate the PE parameter from the SYNC, SYNC_SET, SYNC_SHUFFLE, SYNC_UNSHUFFLE, and SYNC_IF macros and require the use of PLUS_DEFAULT. This appears to be the preferred usage.

3.4.6. PLUS_DEFAULT

This macro is used to indicate defaults for the type of the target in SYNC_SET macros and for the variable used as processor identifier. The syntax is

PLUS_DEFAULT TYPE (type) PE (id);

where either parameter may be omitted. Supplying a TYPE parameter generates a temporary variable used by all SYNC_SETs without the OF_TYPE parameter. Supplying a PE parameter eliminates the need for this parameter in all other macros and is highly recommended.

3.5. Other Reserved Words

PLUS reserves, for internal use, all identifier beginning with $$.

4. A Simple Example — SUMMING

For completeness we excerpt the following description of the summing algorithm from [UC].

(a) Replace w_n by $w_{n-1} + w_n$ for each odd n.[2]

(b) Proceeding recursively, apply summing to the odd elements. (This can be done by first unshuffling then applying summing to the upper half of the processors, then shuffling.) At the end of this step, every odd processor p_n will contain $w_0 + ... + w_n$.

(c) Replace w_n by $w_{n-1} + w_n$ for each even n>0.

The PLUS implementation of summing shown below is quite similar to this high level description. The second parameter LB (lower bound) of the summing proceedure is used to implement the recursion indicated in step (b) above. At each level of the recursion, the only processors ‘‘active’’ are those numbered LB, LB+1, ..., MAX_PE#; when summing is called initially, the second argument is 0.

[2]The actual version in [UC] is for any associative operator ⊠.

```
SUMMING: PROC (W, LB, N) RECURSIVE OPTIONS (REENTRANT);
        %INCLUDE PLUS8;
        DCL (W(0:MAX_PE#), LB, N) FIXED BINARY;
        PLUS_DEFAULT PE (N) TYPE (FIXED BIN);

        IF (LB<N & ODD(N))
                THEN SYNC_SET (W(N)) EQUAL_TO (W(N-1) + W(N));
                ELSE SYNC;

        IF (LB+1 < MAX_PE#)
                THEN DO;
                        SYNC_UNSHUFFLE (W);
                        CALL SUMMING (W, (LB+#_PE)/2, N);
                        SYNC_SHUFFLE (W);
                        END;

        IF (LB<N & EVEN(N))
                THEN SYNC_SET (W(N)) EQUAL_TO (W(N-1)+W(N));
                ELSE SYNC;
        END SUMMING;
```

The %INCLUDE statement brings in the preprocessor package that contains the macros described in the preceeding sections. A full listing of this package is given in [PLUS2]; here we are content to illustrate its *use.*

PLUS8 defines and initializes the preprocessor variables #_PE = 8 (in PLUS16, #_PE = 16, etc), MAX_PE# = #_PE - 1, and LOG_#_PE = log(#_PE). These preprocessor variables become *constants* in the PL/I program, and consequently may appear as dimensions for parameters and STATIC arrays.

In the next section we illustrate the use of STATIC arrays, but may not appear as targets of assignments.

The PLUS_DEFAULT macro declares N to be the identifier corresponding to the processor number

ODD and EVEN are trivial macros with the obvious meaning. The SYNC_SET macro was described in the previous section.

The SYNC_UNSHUFFLE macro actually expands into a SYNC_SET, namely:

```
SYNC_SET (W(N)) EQUAL_TO (W(SHUFFLE_PE(N)));
```

where SHUFFLE_PE is a global array initialized so that SHUFFLE_PE(N) is the number of the processor into which processor N shuffles. This is also the number of the processor that unshuffles into processor N. SYNC_SHUFFLE (and UNSHUFFLE_PE) are defined analogously. The auxiliary arrays SHUFFLE_PE and UNSHUFFLE_PE may be referenced by user programs but they must not be updated.

The reader should note that the PL/I code stated above satisfies the synchronization requirements stated in the previous section: No non-local updates are used, all non-local references use SYNC_SET, and when one task SYNCs they all do.

5. A Less Trivial Example — PACKING

5.1. A Description of PACKING

Consider a set

$$S = \{s_0, ..., s_{max_PE\#}\}$$

with s_n initially located in processor n. We assume that some of the elements of S are marked and we wish to rearrange S so that the marked elements are moved into the low numbered processors but maintain their original order. When rewritten in our notation, the packing algorithm presented in [UC], operates as follows.

First we determine the destination processor for the marked element in processor n. This is simply the number of marked elements in lower numbered processors, and can be determined by summing.

Then interchange adjacent even/odd elements x,y according to the following rule: If x is marked and has an even location but an odd destination, then interchange x and y. Proceed similarly if x is marked and has an odd location but an even destination.

After all the above interchanges, each marked element with an even (resp. odd) destination will be in an even (resp. odd) location. Then apply packing to the odd and even elements separately (and) in parallel; this packs S. Note that to pack the even and odd elements separately, we first unshuffle, then separately pack elements held in the lower half and the upper half of our processor assemblage, then shuffle.

We implement a slight variation of the above algorithm in which the LOG_#_PE − 1 shuffles that conclude the algorithm are replaced by a single unshuffle and the recursion is removed.

```
PACKING: PROC (S, N) OPTIONS (REENTRANT);
        %INCLUDE PLUS8;
        DCL 1 S(0:MAX_PE),
            2 VALUE FIXED BIN,
            2 MARKED BIT (1),
          N FIXED BIN;
        DCL (NBAR INIT (PARTNER_PE(N)), K) FIXED BIN,
          DEST (0:MAX_PE) FIXED BIN STATIC,
          SUMMING ENTRY;
        PLUS_DEFAULT TYPE (FIXED BIN) PE (N);

        /* CALCULATE DEST FOR MARKED ELEMENTS */
        IF S(N).MARKED
                THEN DEST(N) = 1;
                ELSE DEST(N) = 0;
        CALL SUMMING (DEST, 0, N);
        DEST(N) = DEST(N) - 1;

        EXCHANGE_UNSHUFFLE_LOOP: DO K = 1 TO LOG_#_PE;
                /* EXCHANGE PAIR IF MARKED HAS BAD PARITY */
                SYNC_IF S(N   ).MARKED & MOD(N   ,2)=MOD(DEST(N   ),2)
                    | S(NBAR).MARKED & MOD(NBAR,2)=MOD(DEST(NBAR),2)
                        THEN DO;
                                SYNC_SET (S(N)) OF_TYPE (LIKE S)
                                        EQUAL_TO (S(NBAR));
                                SYNC_SET (DEST(N)) EQUAL_TO (DEST(NBAR));
                                END;
                        ELSE DO;  SYNC;  SYNC;  END;

                /* UNSHUFFLE DEST - THEN UNSHUFFLE PROCESSORS */
                DEST(N) = UNSHUFFLE_PE(DEST(N));
                SYNC_UNSHUFFLE (S) OF_TYPE (LIKE S);
                SYNC_UNSHUFFLE (DEST);
                END EXCHANGE_UNSHUFFLE_LOOP;
        END PACKING;
```

Consider the SYNC_IF statement above. Since the tasks that satisfy the condition in the SYNC_IF execute two SYNC_SETs, those tasks that do not satisfy the condition must execute two SYNCs. Also note that in the preceeding code both the structure S(N) and the integer variable DEST(N) are targets in SYNC_SETs. Since the default type is specified as FIXED BINARY by the PLUS_DEFAULT macro occuring early in the code, all SYNC_SETs involving S must include the OF_TYPE parameter. Alternatively, we could have either specified the default type as LIKE S and used the OF_TYPE parameter when DEST was the target, or not specified a default type and used the OF_TYPE parameter with each SYNC_SET. Note also that SYNC_SHUFFLE and SYNC_SHUFFLE macros, which expand into appropriate SYNC_SETs, have an analogous OF_TYPE parameter which is used in the same way.

5.2. Using STATIC Arrays

In PL/I the distinction between STATIC and AUTOMATIC variables becomes more significant when the procedure involved is REENTRANT or RECURSIVE. Since nearly all user written PLUS procedures are REENTRANT, the PLUS programmer must be aware of the effect obtained by declaring a variable STATIC.

Each procedure invocation receives a separate copy of all locally declared AUTOMATIC variables; whereas all invocations share the same copy of each STATIC variable. When the above program is executed, eight concurrent invocations of PACKING are created. Each of these eight tasks obtains its own copy of NBAR but they all share the one array DEST.

PLUS variables used to simulate a set that is stored one element per processor are usually STATIC arrays or array parameters, exemplified by DEST and S above. Moreover, the target in a SYNC_SET, SYNC_SHUFFLE, or SYNC_UNSHUFFLE macro is also usually a STATIC array or an array parameter.

5.3. The PARTNER_PE Array and the SYNC_IF Macro

We often consider the processors in an ultracomputer to be grouped into even-odd pairs and, for any processor N, define its partner NBAR as the other member of the pair. PLUS creates a global array PARTNER_PE initialized so that PARTNER_PE(N) = NBAR.

Often the boolean expression constituting the condition in an IF statement contains a nonlocal reference. An example of this occurs in PACKING where each processor must know if its partner contains a marked element. One PLUS implementation of an IF statement with a nonlocal IF condition is

```
SYNC_SET (TEMP_BOOL) EQUAL_TO (condition) OF_TYPE (BIT(1));
IF (TEMP_BOOL)
```

Using the PLUS supplied macro SYNC_IF, one may write the above simply as

```
SYNC_IF (condition)
```

6. The Main Program and the Number of Concurrent Tasks

The actual PL/I main program is part of the PLUS system. This program first performs various initializations, then attaches the gatekeeper task (used for synchronization - see [PLUS2] for details), and finally calls the procedure USER. This last procedure is the first user written procedure to execute. *N.B.* When USER is called, two tasks are active: The main task, which consists of the main procedure and USER, and the gatekeeper task. Thus, if an N processor ultracomputer is being simulated, N+2 tasks will be executing concurrently. It may be necessary to specify this information in the JCL.[3]

[3]By default, the PL/I optimizing compiler expects that no more than 10 tasks are simultaneously active. Thus, if an ultracomputer of size greater than 8 is simulated, the default value is inadequate. No such limit exists for the checkout compiler. See the *PL/I Programmer's Guides* for details ([OPT] and [CKOUT]).

6.1. The USER Procedure for Summing

This procedure is rather stylized and changes little from one algorithm to the next. We present a version that may be used to test summing and leave as an (easy) exercise the task of producing a version to tests packing.

```
USER: PROC;
    DCL (MAX_PE) FIXED BIN STATIC INIT (7), /* AVOID PREPROCESSOR */
      (W(0:MAX_PE), PE) FIXED BIN,
      (DONE(0:MAX_PE)) EVENT,
      (SUMMING) ENTRY ((*) FIXED BIN, FIXED BIN, FIXED BIN);

    GET (W);   PUT DATA (W);
    TASKING_LOOP: DO PE = 0 TO MAX_PE;
          CALL SUMMING (W, 0, (PE)) EVENT (DONE(PE));
          END TASKING_LOOP; /* (PE) ABOVE GIVES CALL BY VALUE */
    WAIT (DONE);
    PUT DATA (W) SKIP (2);
    END USER;
```

7. References

[Benes] V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, NY, 1965.

[CKOUT] *OS PL/I Checkout Compiler: Programmer's Guide*, SC33-0007-3, IBM, 1976.

[Clos] C. Clos, ''A Study of Nonblocking Switching Networks, *Bell System Technical Journal* 32, pp. 406-424, 1953.

[LRM] *OS PL/I Checkout and Optimizing Compilers: Language Reference Manual*, GC33-0009-4, IBM, 1976.

[OPT] *OS PL/I Optmizing Compiler: Programmer's Guide*, SC33-0007-3, IBM, 1976.

[PLUS2] Allan Gottlieb, ''PLUS: A PL/I Based Ultracomputer Simulater, II'', in preparation.

[Stone] Harold S. Stone, ''Parallel Processing with the Perfect Shuffle'', *IEEE Trans.* C-20, pp. 153-161, 1971.

[UC] J. T. Schwartz, ''Ultracomputers'', to appear in *TOPLAS.*

[UCN3] J. T. Schwartz, ''Preliminary Thoughts on Ultracomputer Programming Style'', *Ultracomputer Note* #3, N.Y.U., 1979.

[UCN6] Clyde Kruskal and Larry Rudolph, ''Observations Concerning Multidimensional Ultracomputers'', *Ultracomputer Note* #6, N.Y.U., 1980.

Table of Contents

1. Introduction 1
2. Synchronization Requirements 2
3. USER'S GUIDE 3
3.1. Constants 3
3.1.1. TRUE and FALSE 3
3.1.2. NIL 3
3.1.3. #_PE 3
3.1.4. MAX_PE# and LOG_#_PE 4
3.2. (Read Only) Arrays 4
3.2.1. LEFT_PE and RIGHT_PE 4
3.2.2. SHUFFLE_PE and UNSHUFFLE_PE 4
3.2.3. PARTNER_PE 4
3.3. PL/I Builtin Functions 4
3.3.1. MOD and COMPLETION 4
3.4. Macros 4
3.4.1. EVEN and ODD 4
3.4.2. SYNC_SET 4
3.4.3. SYNC_SHUFFLE and SYNC_UNSHUFFLE 5
3.4.4. SYNC 5
3.4.5. SYNC_IF 5
3.4.6. PLUS_DEFAULT 6
3.5. Other Reserved Words 6
4. A Simple Example — SUMMING 6
5. A Less Trivial Example — PACKING 8
5.1. A Description of PACKING 8
5.2. Using STATIC Arrays 10
5.3. The PARTNER_PE Array and the SYNC_IF Macro 10
6. The Main Program and the Number of Concurrent Tasks 10
6.1. The USER Procedure for Summing 11
7. References 11