The Add-and-Lambda Operation:
An Extension of F&A

by

Malcolm C Harrison

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

Ultracomputer Note #104
July, 1986

ABSTRACT

This note describes a proposed extension to the architecture of shared memory multiprocessors with combining fetch-and-add operations, such as the NYU Ultracomputer and the IBM RPn. The extension involves addition of a small amount of hardware between the network and the memory, which permits the efficient implementation of a number of parallel operations. Examples are given.

1. Introduction

It has been shown [GGKMRS,GLR] that a number of important operations can be made completely parallel using the combining fetch-and-add (F&A) operation. These include parallel queue operations and semaphores, More recently, Dimitrovsky [D85] has shown that parallel garbage collection can be implemented using F&A, and in [D86] has proposed a new synchronization mechanism, the group lock, which can also be implemented using F&A. However, the implementation of these operations in terms of fetch-and-add, though bounded in time independent of the number of processors, is generally slow (in some cases of the order of 50 instructions). There also seem to be a number of other operations for which F&A is not quite the right primitive, sufficient but not optimal.

In this note an extension of the basic F&A is suggested, not in the direction of fetch-and-phi, as has been previously proposed, but in the direction of supplementing the operations carried out at the memory. These use the same combining network as F&A, and take the form of simple operations which are dependent both on the data values arriving from the network and also on the value in memory. We call these add-and-lambda (A&L) operations, and the smart memory which supports them a lambda-memory.

2. Hardware

We propose a new class of PE-memory operations on single words. These operations are designed to make use of a standard combining fetch-and-add network, and can be implemented by a simple FSA interspersed between the network and the memory. The function of this FSA in general terms is:

```
procedure add_and_lambda
        (in address, nvalue; out nresult);
begin
   nresult :=
        lambda1 (memory[address], nvalue);
   memory[address] :=
        lambda2 (memory[address], nvalue);
end add_and_lambda;
```

Here nvalue is the value delivered by the F&A network to the memory, and nresult is the value returned through the network.

In many of these operations, we will consider memory locations to consist of three fields, as specified in the following record structure:

```
type memword =
        record
```

```
state: integer := 0;
i: integer := 0;
j: integer := 0;
end;
```

The general idea is to implement operations as add-and-lambdas with the increment being a value of 1 in the i and/or j field. These instructions will be combined by the F&A network in the usual way, ending up at the memory location in the form of a positive increment. If the fields are large enough, this increment specifies unambiguously how many i-increments and j-increments have been requested. Furthermore, after being passed back through the network, the result is effectively two independent F&As. However, the values stored and returned are more general functions of the value delivered by the network and the memory contents.

The memory word is viewed as a FSA with state specified in its state field, so the general A&L operation is as follows:

```
procedure add_and_lambda
        (in address,
        state_req, ireq, jreq;
        out nresult);
begin
   with memory(address) do
        case state of
           ...     {compute new memory contents}
           ...     {compute result}
           end case;
        nresult.state := state;
   end with;
end add-and-lambda;
```

Here the value arriving from the network is considered to have three fields, state_req, ireq, and jreq. In most cases the state-req field will be zero, so that the new state can be returned unchanged through the network.

2.1. Compatibility with F&A

We propose to maintain compatibility with the standard F&A operation by interpreting F&A operations on memory words whose state value is either all ones or all zeros as a standard F&A. This requires that a few bits (we expect no more than 4 or 5) of the memory word be lost. We are not aware of any algorithms for which this is critical.

3. Applications

We start with a very simple example, the modulus operation, and then discuss the TIR and TDR operations used for queue implementation, one-word buffer operations, queue operations, and then the group lock operations.

3.1. The Mod operation

Some algorithms use F&A on an integer to allocate operations on an array, computing the result modulo the size of the array. To prevent this number overflowing sometimes requires a separate synchronization operation, which can be eliminated by an A&L operation. We assume the state field has been initialized to "mod", the i field to the modulus, and the j field to the initial value, using a normal write instruction. The solution is straightforward:

```
case state of
   mod:
         j := j + ireq;
         if j > i then j := j-i;
         nresult.j := j;
   end case;
```

This still requires that the processor itself compute j modulo i, since the network may itself return large values of j.

3.2. TIR, TDR operations

These operations are used in the parallel queue algorithm proposed in [GGKMRS], and also used in [D86]. (We also give another algorithm below for queue operations). An algorithm for the TIR and TDR operations is a little tricky, because it must avoid passing negative numbers through the network. We first note that TIR can be implemented by TDR, so we just need to implement TDR. The solution given below receives requests for increments and decrements in the ireq and jreq fields respectively, with the value of the counter being in i. However, the value in i is prevented from going negative, as follows:

```
case state of
   tdr;
         inc := ireq-jreq;  {increment in count}
         if i+inc > 0 then{ok}
            i := i+inc;
         else
            i := 0;
         end if;
         nresult.j := N+i+inc;
   end case;
```

Here the j field in the memory is not used at all. All increments are accepted implicitly and used to modify i. Decrements are only accepted if they do not cause the count in i to go negative. A processor can determine if its decrement was accepted by examining the value returned in the jreq field; if this is greater than N, some value chosen to be at least as large as the number of PEs, the decrement was accepted. For example, if the count would have been reduced to -2, the jreq field returned would be N-2; this will be returned to two of the requesting PEs as the values N-1 and N, which should be interpreted as rejection.

3.3. Small producer-consumer buffers

The queue implementation in [GGKMRS] must synchronize insertion and deletion in a specific location in the queue. Such a location is a buffer, acting as the target of produce and consume operations, which must alternate.

If the buffer is small, we can implement it in one word by using the state field to store the full/empty status of the buffer, the ireq field to request reads, and the jreq field to request writes. A fourth field, k, is used to store the value. A PE writes by executing an A&L(address,i1+value) and testing the returned i field for being 1 -- a returned value greater than this indicates that the store was rejected. Here i1 is a word with a value of 1 in the i field. A PE reads by executing an A&L(address,j1) and testing the returned j field for being 1, in which case the k field contains the value. A simplified algorithm is as follows:

```
case state of
   empty:
        nresult.j := 1;              {refuse reads}
        if ireq = 1 then  {write ok}
           nresult.i := 0;
           k := kreq;
        else                         {write clash}
           nresult.i := 1;
        end if;
   full:
        nresult.i := 1;              {refuse writes}
        if jreq = 1 then  {read ok}
           nresult.j := 0;
           nresult.k := k;
        else                         {read clash}
           nresult.j := 1;
        end if;
   end case;
```

This accepts a read or write only if there is only one of them, rejecting all requests when there is a clash. If there are many possible simultaneous readers or writers, this would require them to try again, with a danger of starvation if not done carefully (the rejection field could be used by the PEs to control a delay before trying again, for example).

There are a number of improvements which could be made. For example, a simultaneous multiple read and single write could be satisfied, by returning kreq as nresult.k, and letting the PEs examine the j field.

The number of bits required for the i and j fields depends on how many PEs might request simultaneous writes and reads. In the case of two PEs communicating via a buffer, only 2 bits are needed for i and j.

3.4. Parallel queue operations

In [GGKMRS] algorithms are given for implementing parallel queues, using an array, two pointers, and two bounds for the number of elements in the queue. Essentially the same algorithm can be implemented using add-and-lambdas, by using location ILE to store the insertion pointer and the lower bound on the number of empty slots, and location DLF to store the deletion pointer and the lower bound on the number of full slots. However, separate fields must be used to initiate and complete each operation, so a third field, the k field, will be needed to store the actual pointer. The i and j fields need to be only large enough to accommodate the maximum number of PEs, while the k field needs to be large enough for the queue index.

The insert operation could be written as:

```
procedure insert (data, q, overflow);
   in := A&L (ILE, j1);  {j1.j = 1}
   np := jfield(in) - N;
   if np > 0 then
         overflow := false;
         i := (np+kfield(in)) mod buffersize;
         {wait for i'th item to be empty}
         q[i] := data;;
         A&L (DLF, i1);            {i1.i = 1}
   else overflow := true;
   end;
```

The A&L solution given above for single-word buffers could be used to implement the wait.

The idea of the implementation of the ILE state is to use the k field to store the insertion index, and the i field to store the lower bound on the number of empty slots. The size of the queue is assumed to be a power of 2, so only the significant lower bits need be stored. The insert procedure uses the jreq field to request an increment in the insertion index and a decrement on the lower bound on space, while the queue delete procedure uses the ireq field to increment the lower bound on space. The original insertion index is returned through the kreq field, and the various increments through the jreq field (biased positively by the number of processors N), so back at the PE the sum of these minus N gives the correct index. Overflow occurs when the lower bound on space would become negative. In this case the computed index increment will be negative for the overflow PEs.

3.5. The Group Lock Algorithm

This is a set of general synchronization primitives proposed by Dimitrovsky [D86]. The idea is to permit a programmer to write:

```
glock(g); aaa; gsynch(g); bbb; gunlock(g);
```

with the assurance that there is no unbounded delay, and that no aaa operation will execute at the same time as a bbb operation. This has a number of applications, permitting, for example, simple solutions for parallel queues, stacks, and heaps, and a straightforward solution of the readers-writers problem.

The idea of the implementation is that PEs requesting glocks are split into groups, with the synchronization operation applying to just the group members. The next group starts only after all members have finished the current group.

We can represent a group lock by a memory location consisting of five fields, as specified in the following record structure:

```
type glocktype is
        record
        state: (init, open, closed, synched)
                        := init;
        g: boolean := true;
        i: integer := 0;
        j: integer := 0;
        k: integer := 0;
        end;
```

The i field is used to store the number of PEs still in the current group, the j field to store the number of PEs in the current group which have executed a synch operation, and the k field to store the number of PEs which have been assigned to the next group.

For the group lock operations themselves a 1 in the i field will request a lock, a 1 in the j field will request a synch, and a 1 in the k field will request an unlock. Each lock request is either accepted into the current group, or the next. Acceptance into the current group is permitted if no synchs or unlocks have been requested. The algorithm is as follows:

```
case state of
  init:
        i := 0;                 {# in current group}
        j := 0;                 {# of synchs}
        k := 0;                 {# in next group}
        state := open;
  open:
        if jreq = 0 and kreq = 0 then
          nresult.i := i;
          i := i+ireq;    {accept into group}
        else
          k := ireq;            {for next group}
          nresult.j := j;
          j := jreq;            {# synchs}
          i := i-kreq;    {# remaining}
          nresult.i := 0;
          state := closed;
        endif;
  closed:
        nresult.i := k;
        k := k+ireq;            {next group}
        nresult.j := j;
```

```
        j := j+jreq;            {synchs}
            i := i-kreq;                {# remaining}
            if i = 0 then               {end of group}
                state := open;
                g := not g;
            elsif i = j then    {all have synched}
                state := synched;
            endif;
    synched:
            nresult.i := k;
            k := k+ireq;                {next group}
            i := i-kreq;                {unlocks}
            if i = 0 then               {end of group}
                state := open;
                g := not g;
            endif;
    end case;
```

Here there are three active states, open, closed, and synched, which are executed in sequence with g = true and then in sequence with g = false, and so on. Every lock request is accepted, either for the current group (indicated by a state of open), or for the next (any other state). In each case the i field returned specifies the group position. (If a lock is not accepted the PE should busy-wait till g changes).

By examining the returned state field, the PE can determine if synchronization is complete. If not, such PEs will have to busy-read the glockword to determine synchronization.

The high-level operations on group locks will be implemented as follows:

```
function glock (a: pointer to glocktype) returns integer is
        r : glocktype;
    begin
    r := A&L (a, i1);
    if r.state <> open then         {busy wait}
        while A&L(a, 0).g = r.g loop end loop;
    end if;
    return r.i;             {position within the group}
    end glock;
function gsynch (a: pointer to glocktype) returns integer is
        r : glocktype;
    begin
    r := A&L (a, j1);
    if r.state <> synched then      {busy wait}
        while A&L(a, 0).state <> synched loop end loop;
    end if;
    return r.j;             {position within synch subgroup}
    end glock;
procedure gunlock (a: pointer to glocktype) is
    begin
    A&L (a, k1);
```

```
end glock;
```

4. References

[GGKMRS]

Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin McAuliffe, Larry Rudolph, and Marc Snir, ‘‘The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer’’, *IEEE Trans.* **C-32** (Feb. 1983), pp. 175-189; also in *Proc. 8th Annual Symp. on Computer Architecture*, April 1982.

[GLR]

Allan Gottlieb, B.D. Lubachevsky, and Larry Rudolph, ‘‘Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors’’, **ACM TOPLAS**, Vol. 5 No. 2, April 1983.

[D86]

Isaac Dimitrovsky, "A Group Lock Algorithm, with Applications", Ultracomputer System Software Note 64, Courant Institute, March 1986.

[D85]

Isaac Dimitrovsky, "Parallel Garbage Collection", Ultracomputer Software Note 65, Courant Institute, 1985.