Low-level programming for a massively parallel fine-grain computer: the Microflow approach*

Laurie Hendren
Alex Nicolau†
Jon Solworth††
Pat Xavier

87-806

March 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*This work was supported in part under ONR grant number N00014-86-K-0215.
†This author was supported in part under NSF grant number DCR-8502884.
††This author was supported in part under NSF grant number DCR-8503610.

Low-level programming for a massively parallel fine-grain computer: the Microflow approach[1]

Laurie Hendren, Alex Nicolau[2], Jon Solworth[3] and Pat Xavier

Department of Computer Science
Cornell University
Ithaca, NY 14950

(607) 255-7166
solworth@cornell

Abstract

A new programming language *MFL*[3] is described, which, while low level, combines both message passing and shared memory models. We examine both the programming style and implementation issues of such a language.

The programming style splits the computation into a computation thread (one process per processor) and several server threads. The computation thread (which performs the bulk of the computation) is deterministic, while all of the non-deterministic code is in the server threads.

Also described are several ways of making programming in message passing languages less tedious and more modular, in terms of compilation techniques, runtime structures and a new programming structure.

1. Introduction

Traditionally, there has been a taxonomic split into two classes of parallel processors: *message passing* and *shared memory*. In this paper we examine the effect of unifying these two classes. An architectural model is briefly described as a prelude to examining the programming consequences of such a language. We introduce here a systems programming language which, while low level, gives the programmer full control over the resources of the machine. This programming language is called Microflow Low Level Language (*MFL*[3]).

[1]This work was supported in part under ONR grant number N00014-86-K-0215.
[2]This author was supported in part under NSF grant number DCR-8502884.
[3]This author was supported in part under NSF grant number DCR-8503610.

The programming language exhibits several properties which differentiates it from other message passing languages. These properties are examined in terms of skew, speed and register usage.

The programming language has also led to a characteristic style of programming for such a machine. This style exhibits several notable properties:

- a natural separation of non-deterministic from deterministic computation,
- A highly static, low overhead method for invoking parallelism,
- Efficient use of fine-grain parallelism, and
- Shared memory with the ability to exploit locality.

Finally, our programming language is compared with two other parallel programming languages: Sietz's C extensions and Inmos's Occam.

2. Project goals

The goal of the Microflow project is to provide fine-grain parallelism (efficient use of as few as 2 or 3 instructions per processor) while enabling MIMD control. We have decreased the overhead of invoking parallelism through static compile-time scheduling of the processor resources and activities while tolerating and avoiding latency through architectural design. Latency toleration means the ability to perform useful work while waiting on an event to occur; latency avoidance is the reduction in the time that a processor is waiting for an event to occur.

Our goal is to make processors more effective than in other systems. The techniques we use:

- exploit locality properties of both processors and memory,
- provide a very efficient message passing architecture to enable rapid assignment of processors to computations, and
- avoid latency by the use of a high *skew* execution model.

By *skew*, we mean the degree to which different processors can operate asynchronously —in essence the degree of "MIMDness" of the processors. The less tightly coupled the processor, the higher the *skew*. Processors which operate in lockstep exhibit the lowest amount skew. Examples of skew include overlapped fetch with execution and producers which are allowed to get several elements ahead of consumers.

3. Architectural model

In this section, we describe the basic architectural model which supports efficient execution of the programming language *MFL*[3]. An architecture which implements this model is described in [SoN85].

The architectural model is for a large-scale MIMD parallel processor, containing thousands of nodes, each consisting of a processor, network interface, and memory. The nodes are connected together by a routing interconnection network which exhibits locality (such as a Hypercube or Cube-Connected Cycle), and all storage in the computer is globally addressable. See Figure 1.

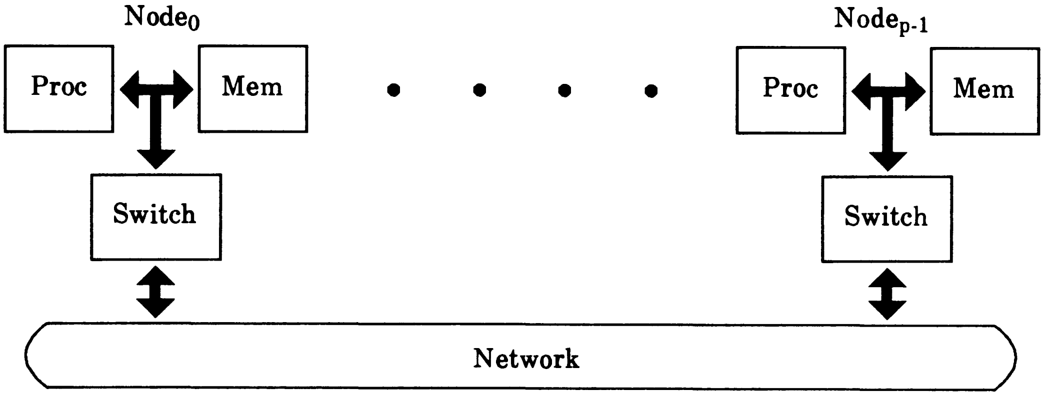

Figure 1 — the architectural model

A processor contains a number of contexts, each with its own registers and program counters. A processor communicates or synchronizes with other processors by the following instructions:

- Load(reg, addr): load into register *reg* the value in memory location *addr*.
- Store(reg, addr): store into memory location *addr* the value in register *reg*.
- Send(value, reg, node, context): send *value* to a queue associated with a *reg-context* pair on processor *node*.
- Receive(reg): Let q be the queue associated with *reg* (within the current context). Receive causes the first element of q, when it becomes defined, to be removed from the queue and loaded into the register. If another instruction uses the register as an operand before the operation completes then that instruction blocks.

For a large parallel processor, it would be prohibitively expensive to maintain a globally consistent view of events in hardware. However, the network ensures that if $processor_i$ sends two messages to $processor_j$, these messages will arrive in the order they were sent (possibly interleaved with messages from other processors).

4. Language and Compiler Issues

Our view of a program is as a single code which all processors independently execute. Processing can be tailored to a processor since each processor knows its processor number.

MFL[3] differs from other message passing languages in two principle ways. The first is the separation of logical *communications variables* (sometimes called channels) from the physical hardware-dependent properties of ports and network topology. The second is that shared memory is combined with message passing.

Message passing systems must perform two functions: they must route a message to the proper node, and then the message must be stored at the node. We will call the name of the location where the message is stored a *communications variable*, and the route *r*. Every other implementation we know about limits the number of *communications variables* and most limit the set of nodes which are reachable from a given node. Limiting the number of (physical) communications variables forces the programmer to think of a communications variable as a sequential channel, on which different (logical) variables must be explicitly sequenced. As we shall see later, limiting the number of *communications variables* also hinders modularity.

In contrast, MFL^3 allows an arbitrary (but statically allocatable) number of communications variables to be used in a program. For efficiency, the compiler attempts to map these *communications variables* to registers (which when used for this purpose will be called *communications registers*). Unfortunately, a mapping does not always exist, since the number of communications variables which are in use at a given point may exceed the number of registers available. In later sections we will describe compile-time analysis, programming language structures and backup strategies to deal with this problem.

Since our architectural model contains shared memory, messages tend to be quite short; instead of passing large blocks of information, a pointer can be passed. Shared memory tends to result in increased *skew*, since processors independently can physically access any memory location.

We have implemented about a dozen algorithms in MFL^3, including quicksort, bitonic sort, breadth-first search, LU decomposition with partial pivoting, matrix multiply, FFT and parallel B-trees. The programs, while varied, all exhibit a common style which we believe is due to the combination of shared memory and message passing in the same language. This style is characterized by several properties: the uses of message passing and shared memory, the use of non-determinism and servers, the use of static recursive unrolling, and locality.

First of all, message passing is used to coordinate processes, divide global data structures for parallel processing, and to perform pipelined computation (as in systolic processing). Once the message passing divides up the work, processors can use shared memory to access data structures which they now have exclusive access to.

The message passing functions are sometimes handled by servers. These servers are often non-deterministic while the main functions are deterministic. This segregation of deterministic from non-deterministic computations simplifies debugging and program understanding.

A static unrolling of parallel recursive routines (currently performed manually) binds processors to recursive calls. (This will be seen later in the quicksort example.) The unrolling is performed for as many processors as are available, after which the leaf routines execute the procedures by sequential recursion. More dynamic unrolling is also possible by passing around pointers to data and/or procedures, but since this has higher overhead, it is discouraged.

Finally, locality is exploited by placing processing at the site where data is located (this can be seen in the breadth-first search example). Also, we take advantage of locality by mapping processes that communicate with each other to neighboring processors.

4.1. Language Overview

A parallel computation is composed of several **threads**. A thread is a single code which runs on each of the P processors in the systems. Hence each processor executes independently within the code for the thread. A processor communicates with other processors executing the same thread or with other processor-thread pairs by message passing and shared memory accesses. Although a program is composed of possibly many threads, one is distinguished as the computation thread —the others are all known as server threads. To each thread, (and on each processor) a context is assigned. The computation thread is the main focus of control, while server threads, as the name indicates, provides services to the computation thread.

We describe the semantics of the communications constructs in MFL^3 as follows:

com <type> <var>;

declares *<var>* as a communications variable of a primitive data type *<type>*. P local variables named <var> are created, one for each processor executing the thread.

send(<value>, <var>, <thread>, <proc>);

sends a <*value*> to a <*thread*> at <*proc*>. <*Var*> is a communications variable declared in the thread (and at the processor) to which the value is sent.

receive(<var>);

removes the head of the queue associated with the communications variable <*var*> assigns it to that variable at the processor which executes the receive. If the queue is empty, the next operation on the variable will block until the data arrives at the queue and is removed.

In addition, for each processor, *proc* is a constant which equals the processor's number (from $0 ... P-1$).

4.2. Programming examples

We present here two programming examples in MFL^3. The first is a parallel version of quicksort and the second is a breadth-first search.

4.2.1. Parallel quicksort

The $O(n)$ average time quicksort proposed by Ehud Shapiro [Sha83] is often given as a parallel program example. The sort is performed by a tree of processors: non-leaf nodes partition the original array (called *source*) based on the pivot, while leaf nodes perform a sequential sort and place the elements in the global array called *sorted*. Our version differs from Shapiro's in that first, the assignment to processors of nodes in the tree is static and second, when all of the processors have been used, the leafs of the tree perform a sequential quicksort.

A processor is either a leaf or a non-leaf (internal) node. If the processor is not a leaf node, it takes its subsequence and partitions it, sending elements less than the pivot to its left subtree, and otherwise sending elements to the right subtree. The subsequence comes from the array *source* if the processor is the root and from its parent otherwise – this is an example of the combination of message passing and shared memory since there is no restriction on where the array *source* is located.

If a processor is a leaf node, it sorts its subsequence, and then puts the elements back into a global array called *sorted*. Message passing is used to pass the index (*count*) of the global array where the first element of the subsequence for that processor is to be placed. while shared memory allows the processor to directly write the elements of the global array.

The information flow in this algorithm is shown in Figure 2, and the program in Figure 3.

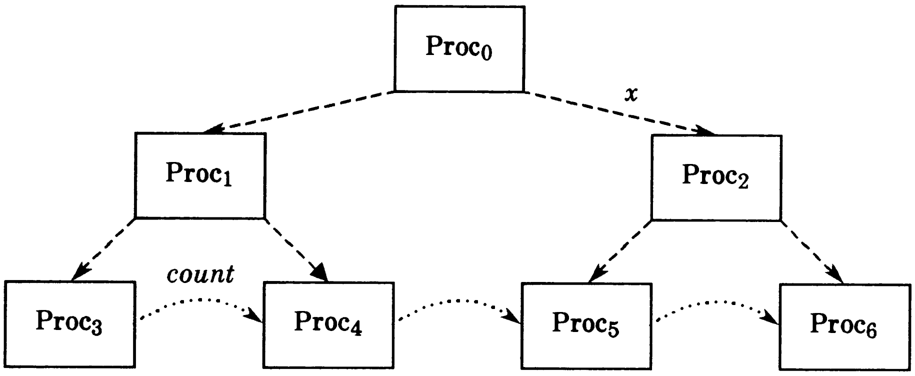

Figure 2 – message passing in quicksort algorithm

```
thread main do

    com int x;       - elements of the array to be sorted
    com int count;   - place where leaf node will start inserting
                     - its sorted subsequence

    partition(x, pivot) -    send an element to left or right subtree
        if x < pivot then
            send(x, x, main, left(proc));
        else
            send(x, x, main, right(proc));
        end;
    end;

    not_leaf() - partition all elements which are responsibility of this node

        if proc=root then
            pivot := source[1];
            for i:=2..#source do
                partition(source[i], pivot);
            end;
        else
            receive(x);
            pivot := x;
            receive(x);
            while x ≠ EOL do
                partition(x, pivot);
                receive(x);
            end;
        end;

        send(pivot, x, main, right(proc));
        send(EOL, x, main, right(proc));
        send(EOL, x, main, left(proc));
    end;
```

```
leaf()
        receive(x);
        i := 0;
        while x ≠ EOL do
                i := i+1;
                local_array[i] := x;
                receive(x);
        end;

        -       sequential quicksort at leaf
        seqsort(local_array);

        -       compute place in global array where this processors
        -       sorted sequence should be placed.
        if proc = first_leaf then
                send(i, count, main, next_leaf(proc));
        else
                receive(count);
                if proc ≠ last_leaf then
                        send(count+i, count, main, next_leaf(proc));
                end;
        end;

        -       put local sorted sequence into the global array
        for j := 1..i do
                sorted[j+count] := local_array[j];
        end;
end;


qsort()
        if !leaf(proc) then
                not_leaf();
        else
                leaf();
        end;
end;
endthread;
```

Figure 3 – parallel quicksort in MFL^3

The above program has several performance drawbacks. The first is that the algorithm is not very sophisticated. The second is that the assignment of computational tasks to processors is static. While MFL^3's static design yields a low-overhead invocation of processors, more dynamic styles (with attendant higher overheads) can be be more adaptable to data not exhibiting statistical regularity which would ensure high average processor utilization. Because MFL^3 is a low-level language, we have decided to explicitly code dynamic parallelism for those relatively few cases when it is appropriate. Both of these effects are curable by modifications to the algorithm.

A third problem is that the structure of the program potentially results in needlessly idling the processors: this occurs because the number of elements at a leaf processor is known as soon as all of the elements at the leaf have been received, but *count* (the total number of elements to the left of the leaf) is not needed until after the elements have been sorted. If *count* is computed before the elements are *seqsort*ed, then the *seqsort* must wait until the partial sum is computed on all earlier processors. If *count* is computed after the elements are sorted, then the processors must wait to insert its element in the array *sorted*.

This problem can be solved by introducing a separate thread of computation which will compute the value of *count* for each leaf, given the number of elements i at each leaf to the left. The computation of *count* proceeds independently of the main processing at the leaf subject to the following two constraints: the server cannot compute *count* until it has received i from all leaves to the left, and an element of the main thread cannot put elements in the global array until the server thread has computed the *partial sum*. The revised code for *leaf* is shown if Figure 4, and the (additional) server thread in Figure 5.

```
leaf()
        receive(x);
        i := 0;
        while x ≠ EOL do
                i := i+1;
                local_array[i] := x;
                receive(x);
        end;

        send(i, i, server, proc); — send number of elements used at this leaf noded

        seqsort(local_array); —sequential quicksort at leaf

        receive(count); —       receive index into sorted where this leaf's subsequence goes

        —       put local sorted sequence into the global array
        for j := 1..i do
                sorted[j+count] := local_array[j];
        end;
end;
```

Figure 4 — leaf without waiting

```
—       compute place in global array where this processor's sorted sequence should be placed.
thread server do
        com int i;
        com int partial_sum;

        receive(i); —number of elements seqsorted on this processor
        if proc = first_leaf then
                count := 0;
        else
                receive(partial_sum);
        end;

        —compute partial sum for next leaf
        if proc ≠ last_leaf then
                send(partial_sum+i, partial_sum, server, next_leaf(proc));
        end;

        send(count, count, main, proc); — send starting place back to main on this proc
endthread;
```

Figure 5 — server thread to compute partial sum

This program demonstrates (in a rather primitive way) some of the elements of our programming style.

First, a program is separated into a computation thread (here called *main*) and one or more server threads. The computation thread is the main notion of sequencing —the server thread is only called on to perform services for the computation thread.

Second, message passing is used to coordinate processors and to pipeline values (such as the elements of a partition). Shared memory is used to allow a processor with exclusive write or multiple read to access large (and arbitrary) amounts of information. For example, *count* is used to provide exclusive access to elements of the array *sorted.*

4.2.2. Breadth-first search

The second example is a breadth-first search (*bfs*) of a graph. Figure 6(a) shows the main thread. The outer (synchronized) loop iterates through each level of the breadth-first search. The inner loops iterate through the list of nodes which are adjacent to nodes in the current set on that processor to compute the set of nodes to search for the next level of the *bfs.* To ensure that a node shows up on the visited set at most once, access to a given node is always through a particular processor (on the server thread *node_enqueue*). The search starts with a seed node.

```
thread main do
        bfs(seed)

                com boolean done;
                com ptr queue;
                instantiate node_enqueue;
                instantiate boolean_tree;

                done := false;

                if VertexToProc(seed) = proc then
                        send(seed, elmt, node_enqueue, VertexToProc(seed));
                        send(ENDMARK, elmt, node_enqueue, proc);
                end;

                while not done do
                        receive(queue);
                        local_done := true;
                        forall v ∈ queue do
                                forall a ∈ adjacency(v) do
                                        send(a, elmt, node_enqueue, VertexToProc(a));
                                        local_done := false;
                                end;
                        end;

                        -- synchronize end-of-round
                        send(local_done, term, boolean_tree, parent(proc));
                        receive(done);
                        send(ENDMARK, elmt, node_enqueue, proc);
                end;
        end;
end;
```

Figure 6(a) Main computation thread for breadth-first search

The main thread contains a single procedure called *bfs*. The procedure begins by initializing each processor's queue: exactly one will contain the seed, all others will be empty. The non-empty processor queue will contain the seed.

The function of the inner and outer loop of *bfs* has already been described. Now the servers will be described.

```
thread node_enqueue do
        com ptr elmt;
        ptr next_queue;

        do

                next_queue := nil;

                receive(elmt);
                repeat until elmt=ENDMARK do
                        if not elmt->visited then
                                next_queue := append(next_queue, elmt);
                                elmt->visited := true;
                        end;
                        receive(elmt);
                end;

                send(next_queue, queue, round, proc);
        end;
end;
```

Figure 6(b) node_enqueue server thread (ensures that node is put on queue at most once)

Node_enqueue (Figure 6(b)) builds a queue of elements that are adjacent to elements in the current round and that have not been encountered before in the search. Each processor considers only those elements which are physically located at the processor's node. The queue built by the processor in the *node_enqueue* thread will be used by the main thread in the next round. Thus, the paradigm of exploiting locality is observed.

```
thread boolean_tree do
        com boolean term;
        com boolean result;
        boolean partial_or;

        receive(term); – receive from computation thread
        partial_or := term;

        – receive from sons in boolean_tree thread
        forall s ∈ sons do
                receive(term);
                partial_or := partial_or or term;
        end;

        if proc=root then
                result := partial_or;
        else
                send(partial_or, term, boolean_tree, parent(proc));
                receive(result);
        end;

        send(result, done, round, proc);
        forall s ∈ sons(proc) do
                send(result, result, boolean_tree, s);
        end;
end;
```

Figure 6(c) boolean tree server thread (returns the **or** of all values).

The second server thread, *boolean_tree* (Figure 6(c)), implements a voting tree which returns true if any if its leaves are true, and also synchronizes the completion signal for the next round.

The use of non-determinism may not be entirely obvious. At a given processor, *node_enqueue* can receive messages from all of the processors. These messages will be interleaved in indeterminate order. However, this is the only non-determinism in the program and is clearly isolated from the deterministic code. Therefore the exact order in which nodes are examined within a particular round of the *bfs* differs (potentially) each time that the program is run.

4.3. Language implementation issues

From an implementations point of view, MFL^3 is different from other message passing languages in two principle ways. The first is the arbitrary number of *communications variables*. The second is that shared memory is combined with message passing.

The use of shared memory yields no particular problem, although we would like the compiler to separate loads from uses of variables to mask memory latency. It is not possible, without global analysis to change the order of memory requests to shared objects. In addition, since message passing may be used to synchronize access to memory, memory accesses cannot in general be moved above these operations. (These constraints are in addition to the normal data flow constraints).

For performance reasons it is desirable to map *communications variables* to registers. A hardware *send* instruction sends a value from $processor_i$ to a particular register on $processor_j$. We shall say a communications value is *live* (potentially in use) the entire time in $processor_j$'s time frame during which the message could be received. Since processors operate asynchronously, the period of time that a variable is live is determined by the synchronization points in the program.

Another complication stemming from the asynchrony of processors (relative to other processors) is that it is not possible for $processor_i$ to know when $processor_j$ has called a subroutine without additional communication. Hence, the target of a send (a register) must remain available for that purpose *even though the subroutine using that variable has itself called a subroutine.* Since it is not possible to spill registers which are used for communications variables, the compiler must allocate registers for communications variables based on the dynamic sequence of subroutine calls.

Naively, communication variables can be allocated to registers for the whole execution of the program – but this is wasteful of registers. We will, in the next few sections, discuss both programming language structures and compilation techniques that enable the registers to be used more effectively.

4.3.1. Compilation issues

The problem of register allocation is far more critical for MFL^3 than for conventional languages, since registers —a scarce resource to start with —are now used both for local computation and for communication between processors. Furthermore, the analysis of when a *communications register* can be reused is complicated by the asynchronous behavior of different processors. Since registers used for local computation follow the same rules as in the sequential case, we concentrate on the reclamation of *communications registers.*

The problem with freeing communications registers is that both the arrival of messages and the use of messages must occur in the same order. However, the order of message arrival is only guaranteed between pairs of processors —there is no global notion of time. In fact, the only situations in which reuse of a communication register is safe are:

(1) When the register is known to be dead (i.e., no further uses of the variable corresponding to the register occurs *and* the value of the register is consumed) then the register may *only* be reused for local computation.

(2) When the corresponding sends/receives can be determined to strictly dominate each other (e.g., send(a) dom send(b) and rec(a),use(a) dom rec(b),use(b)) [AhU77].

Since these conditions are very restrictive, it appears that we will not often reclaim communication registers. In practice, the programs we have written require at most $O(log(P))$ communication registers. Furthermore, if an application does require more registers than the hardware provides, a server thread, as described in the next section can be used to handle the "spilling" correctly.

4.3.2. Runtime structures

When the number of communications variables exceeds the number of physical communications registers that exist in a context, the analysis described previously is used reduce the number of registers needed. If there are still more live variables than registers available, the excess variables must be mapped to memory. (We refer to all such communication variables as "memory mapped".) Memory-mapped communication variables are implemented with a server thread known as a Secretary.

The hardware **send** and **receive** operations cannot be directly used on memory mapped communication variables. Instead, these operations are simulated by protocols between the sender and the Secretary and between the receiver and the Secretary. A single physical communication register in each context is dedicated for communication with the Secretary. On each processor the Secretary maintains a queue of messages for each communication variable it is handling; for each variable for which the queue is empty, the Secretary keeps a record of whether there is an unfilled **receive**.

As communication through a communications variable handled by the Secretary thread will be slower than communication through one mapped to a register, it is important to assign registers to those communications variables which are most frequently used. The main heuristic we use to achieve this is similar to one commonly used for mapping conventional variables to registers: the greater the current loop depth of a variable the greater its priority during the current loop.

All of the algorithms so far use, at most, a number of registers equal to the logarithm of the number of processors. Hence, register sets containing 32 registers should be sufficient to contain the vast majority of communication variables. Only those variables which are memory mapped suffer a degradation in speed. The high likelihood of a variable being register mapped implies that presence of some memory-mapped variables will not significantly degrade performance.

4.3.3. Language extensions

The difficulty of tight liveness analysis of communications variables has led to several problems. How can these variables be more effectively reclaimed? What is the meaning of communications variables within recursive procedures?

We have recently proposed a new programming language construct, called an *epoch* [Sol87], which is a scoping mechanism that limits the time during which a communications variable is live. An epoch contains one or more communications variables, which are used only within the static scope of the epoch. Before the epoch is entered (and after it is exited), the communications variables are dead. For this mechanism to be useful, epochs are synchronized across processors, ensuring that all processors entering an epoch do so simultaneously. A partial BNF grammar is given in Figure 7.

```
<epoch> ::= epoch <name> do <c_decls>+ <stmts> end
<c_decls> ::= com <type> <var_name>;
```

Figure 7 – BNF grammar for epochs

Epochs are a construct that enables the programmer to trade off resources (registers) against skew. Since fast execution requires real resources, it is desirable to use no more virtual resources than real resources. However, if there are excess real resources it is desirable to use them to decrease (on average) the amount of wait time by allowing more asynchrony in the computation.

The epoch mechanism is implemented by explicit, compiler-inserted synchronization across all processors involved at the beginning and end of the epoch. Thus epochs should be used to enclose relatively large phases of code –rather than single operation, hence the name.

With epochs, procedure calls (including recursion) can be used while maintaining a bounded number of live *communications variables*. Moreover, these communications variables can be bound at compile time to registers.

4.3.4. Limiting the number of communications variables

Other languages limit the number of communications variables that can be simultaneously live. This has a detrimental effect on the ability to write modular message passing code.

For example a new procedure cannot simply be added to an existing program as a black box since it may cause the number of live communications variables used to exceed the number of physical communications registers. In this case, the programmer must find some place where the number of registers may be reduced. This forces the programmer to change code which might have nothing to do with the additional module in order to overload one real communications variable with two or more logical communications variables —clearly this is a bad idea since it is likely to introduce bugs into previously correct code. This is in addition to the previously described problem of sequencing multiple (logical) variables on one physical channel.

MFL^3 leaves this overloading of communications variables to the compiler, where the previously described techniques can be employed to handle it.

5. Other languages

5.1. Occam

Occam [Inm83] is a simple parallel programming language based on CSP [Hoa78]. As the programming language for the INMOS transputer it has received much attention. We shall compare MFL^3 to Occam.

Occam's model of computation is that of sequential processes each executing in their own completely independent environment and communicating through unidirectional channels by explicit, synchronous message-passing protocols. In most implementations, the number of channels are limited to the number of physical ports and communications is restricted to nearest neighbor connections.

Occam requires that the communication be carried out completely through synchronous message passing (shared memory is not supported). This form of message passing is different from MFL^3; for two processes to communicate, both the sending and the receiving processes must rendevous. If one of the processes is ready to communicate before the other, the early process freezes and waits for the other process. Hence message passing requires handshaking between communicating processes. This requirement doubles the number of messages that must be exchanged per communication.

The tight coupling of synchronization and communications has several negative effects on the performance of Occam-based systems. The first problem is that the speed of the parallel processing ensemble is limited to the slowest processor – all other processors must wait. Hence, the computation cannot overlap processing with communication, and computation speeds are not "averaged out". A second problem is that the tight coupling of communications and synchronization is most practical in nearest neighbor communications – in routing type networks the cost of synchronization doubles the load on the network and decreases the effectiveness of pipelining of messages by the number of stages in the network.

We agree with Halstead [Hal85] in the belief that CSP/Occam encourages use of few, large processes which don't exploit well fine-grain parallelism. MFL^3's combination of message passing and shared memory seems to offer advantages which are not available just using one of the methods.

5.2. Cosmic Cube C-extensions

The Cosmic Cube (and its commercial successor, the Intel iPSC) support extensions to the C programming language in the form of systems calls to perform message passing functions. We compare here those described in Intel's manual [Int86] with MFL^3.

The C extensions allow up to 1024 channels to be opened per process, each channel capable of distinguishing 32,000 messages. Although routing is performed by the system, the programmer is forced to map logical communications variables to physical resources (since the 1024 channels will have to be reused). Moreover, channels have a single buffer associated with them, which forces processors to execute a protocol if there is more than one processor uses the channel. Finally, the extensions do not support a shared memory model.

6. Conclusions and future work

We have described a programming language in which shared memory and message passing are both available. We have shown how having an arbitrary number of communications variables increases modularity and programmability. Methods for mapping *communications variables* to registers that yield increased efficiency while reducing programming overhead have been discussed.

We have also exhibited a characteristic programming style. This style segregates the uses of message passing and shared memory, limits the amount of non-determinism and isolates it from deterministic code, employs a highly-efficient static execution model, and exploits locality.

We are in the final stages of the implementation of an MFL^3 compiler, which generates code for a simulated Microflow machine. Once this implementation is complete, we will be able to answer a number of questions:

How effective is compile time analysis of programs for reclaiming communications registers?

Are the methods and constructs described here sufficient to support programming in the large?

How much effect do epochs have on skew, and how relevant is skew in terms of efficiency?

How does our approach compare with other parallel approaches in terms of total performance?

References

[AhU77] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.

[Hal85] R. H. Halstead, "Multilisp: a language for concurrent symbolic computation", *Association for Computing Machinery Transactions on Programming Languages and Systems 7*, 4 (October 1985), 501-538, Association for Computing Machinery.

[Hoa78] C. A. R. Hoare, "Communicating sequential processes", *Comun. Association for Computing Machinery 21*, 8 (Aug. 1978), 666-677.

[Inm83] Inmos, *Programming manual for Occam*, Inmos, Corp., 1983.

[Int86] Intel, *iPSC system overview*, Intel Corp., Jan., 1986.

[Sha83] E. Y. Shapiro, "A Subset of Concurrent Prolog and its interpreter", *Tech. Rept. Technical Report-003*, January, 1983.

[SoN85] J. A. Solworth and A. Nicolau, "Microflow: A fine-grain parallel processing approach", *C. S. Dept*, Nov. 1985.

[Sol87] J. A. Solworth, "Epochs", *C. S. Dept*, Jan. 1987.