Application Programming for a Processor-in-Memory Architecture

Jose Renau Paul Feautrier Josep Torrellas et.al.

August 17, 2000

1 Design Considerations for a Special Purpose Programming Language

1.1 Program Structure

Basically, a parallel computer is programmed using one of its native language compiler (most often, C) enriched with calls to a run-time which is responsible for handling the parallelism (among other things). There are many reasons why this scheme is not really satisfactory:

- Control constructs (e.g. a parallel loop) are difficult to implement directly in a run-time system. The user has to implement the construct using the base language control construct and perhaps elementary calls to the run-time system. For instance, a parallel loop has to be implemented using a sequential loop including `fork` calls or something similar. This kind of generation is easilly handled by a compiler.
- Checking the validity of parallel code written only calls to a run-time system is almost impossible at compile time and difficult at run time. An example is the use of the `barrier` primitive, which has to be executed by processors in a processor group to work correctly. This constraint can easilly be enforced by syntactic rules at the language level.

Nevertheless, some constructs do not need to be full-fledged new statements and can be implemented as calls to the run-time system. A case in point is the memory allocation subroutine. In the following, we will discuss each feature of C-Flex and decide whether it has to be in the language or in the run-time system. We will also give indications on which basic primitives are needed in the run-time system for the implementation of the new language constructs.

1.2 Overall Design

The main characteristics of FlexRAM is that it has one (physical) memory space and three level of processors: the host, the P.Mems (one per FlexRAM chip, typically less than ten per configuration) and the P.Arrays (64 per FlexRAM chip). While the memory space is unique, there are visibility rules:

- The host sees all memory, including non-FlexRAM additional chips, and the access time is uniform.
- The P.Mems see all FlexRAM memory, with a penalty (to be estimated) when accessing distant chips.
- A P.Array sees only its own memory and the memory of its two neigbors. The memory of a P.Array is 1/64th of a FlexRAM memory.

An important point is that the situation for program text is the same as for data. The host program can be anywhere; it would be better if P.Mem programs were replicated in each FlexRAM, and the P.Array programs have to be replicated to a special I-memory (only one per 4 P.Arrays). This design implies SPMD programming for P.Arrays, while MIMD programming is possible for the P.Mems. It would be best if problems of program text distribution were completely transparent to the user.

This design can be exploited only if there is a guarantee that a given data structure is implemented in the same way by each processor native compiler. At present, this is guaranteed for arrays. Consider the following structure definition:

```
struct x {
    char y;
    int z;
    double w;
    };
```

If one compiler implement the structure as is, because the target processor has no memory alignment constraints, while another one either padd or reorder the members, we are in trouble. It may also be that our compilers have different opinions on the size of an `int`. While one may inforce rules at the language level to avoid these problems (always order the members by diminishing size, use `short` or `long` but never `int`, always give values to the elements of an `enum`, it would be best if all three compilers were all derivatives of the host compiler.

We will review here the basic design issues of C-Flex.

1.3 A 3PMD language

We could have designed one special-purpose programming languages per type of processor. This would have made reading C-Flex programs difficult to read and

understand, since one would had to refer to three different listings. A C-Flex program is thus a single document. It is the work of the compiler to dispatch the various parts to the relevant native language compiler.

One open question is where to have the seams between the various sections of code. The simplest solution is to change processors only at function call and return. This is probably too restrictive. It remain to decide whether allowing a change of processor at the entry to a compound statement is feasible.

1.4 Physical and Virtual Processor

Each processor in a FlexRAM machine has a physical number, which is related to the high-order bits in the address of the memory it can access. It would be dangerous to have these numbers hard-wired in C-Flex programs, as that would forbid sharing of a configuration between different programs. One may suppose that at program loading time, the application request a given number of (P.Mem) processors, the operating system being responsible for allocating and deallocating them. Virtual processors are numbered consecutively starting from 0, and the operating system create mapping from virtual to physical, which is used by the run-time system.

It would be nice if an application could use more virtual processors than physical processors. This would allow better portability of C-Flex programs. This idea is left for future extension of the language.

1.5 Data Mapping

Remember that a C compiler divides memory into three "segments":

- The static segment, which is allocated at compile time.
- The heap segment, which is allocated at run-time by calling the `malloc` function.
- The stack, which is allocated and deallocated automatically at function entry and return.

Having the heap segment shared by all processors is easy, since they all use the same scheme of addressing. The only problem is that a call to `malloc` may modify the page table of the calling process (this is mediated by a lower level primitive, `sbreak`). The solution we propose is to forbid the use of `malloc` at the P.Mem and P.Array level. There will be instead minor subroutines, `malloc_pmem` and `malloc_parray`, whose aim is to "sub-let" zones of memory obtained by the host program either by `malloc` or by static allocation. This has the advantage of killing two birds with one stone: if the zones are in one-to-one correspondance with processors, there will be no need of critical sections. Besides, each zone can be mapped to its processor memory, thus solving the visibility problem.

The definition of static memory is best restricted to the host part of the program. Remember that the compiler allocate *virtual memory*. It can then be (physically) mapped to the memory of each interested processor. There is however a compiler probleme. Since each part of the program is compiled independently, we have to convey the decisions of the host compiler to the P.Array and P.Mem compiler in some way. This is probably best done at program linking time. If the native compilers are nealy identical, another solution is to copy static declarations in the P.Mem and P.Array code.

Lastly, each processor needs an execution stack (several if we contemplate multiprocessing). The usual design is that all stacks have the same virtual address (at the higher end of virtual memory). This is feasible for FlexRAM, since P.Mems and P.Arrays have a TLB, but the impact on the host operating system may be important. A better solution is probably to mimic the design of threads systems, where each thread has its own independent stack address.

Note that in a parallel program, data can be replicated. This can be done by combinations of other features of the language (like mapping and broadcast). It would be nice to have a special purpose statement.

Some parallel programs have moving data. This can be handled with communications primitives, and does not seem to influence the data mapping facilities of the language.

All in all, what is needed at the run-time level is a primitive for pinning a page in memory in a given range of physical adress. This is best implemented as a new statement (not a declaration or directive), as this allow to restrict its use to the host part of the program.

One important question is how to correlate the data and processor mappings. One solution is to postulate a linear array of processors (something like a template in HPF) and to specify all mappings relative to this template. Another solution is to consider only memory mappings from virtual to physical addresses, and then to deduce the processor mapping by something like an `on owner ---` clause. This solution will be explored in the following.

1.6 Special Purpose Operations

Special purpose operations are shorthand notations for complicated actions (like, e.g., synchronisation), which are provided for the convenience of the user.

1.6.1 Synchronisation

1.6.2 Communications

1.7 Caches Management

1.8 Communication

The object of this note is to investigate the use of the nearest neighbour memory access of P.Arrays for implementing arbitrary one-to-one communication. The setting is the following:

- Each P.Array holds $b = n/P$ items, where P is the number of P.Arrays. Items are arranged as a distributed array:

  ```
  item a[P][B];
  map a[ip][0:b] to p_array[ip] for ip in [0:P-1];
  ```

- Each item is to be moved to a P.Array, in such a way that after a move, each P.Array still holds n/P items. It may happen that some items do not move at all.

- On the receiving side, items are collected in an array `b` with the same properties as `a`. Since emission and reception are going to occurs asynchronously, it is not possible to use `a` as the receiving area.

- We are given a function:

  ```
  struct location {
      int processor;
      int offset;
      };

  struct location dest(location origin);
  ```

  which gives the destination an item as a location. The item at location `m` is in `a[m.ip][m.offset]`.

Such a communication takes time $O(n)$ if we use the P.Mem in the straightforward way. The question is, are there cases in which we can do better by using the P.Arrays?

The Communication Routine The following routine organize a kind of merry-go-round among the P.Arrays. The merry-go-round carry messages, which contain a value and a destination. Each P.Array grabs messages which are addressed to it and replace them by outgoing messages of its own.

```
on parray[ip] in [0:P] {
  struct msg {
      int value;
      struct location destination;
      } m;
  int i, b;

  i = 0;
  b = n/P;
  m.destination.processor = -1;
  while(true){
      if(m.destination.processor < 0)
          if(i<b){
              m.value = a[ip][i];
              m.destination = dest(ip, i);
              i++;
              }
      shift m right;
      if(m.destination.processor == ip){
          b[ip][m.destination.offset] = m.value;
          m.destination.processor = -1;
          }
      }
}
```

Obviously, there is a symmetric routine in which objects travel left instead of right. For any given global communication, the user should either use the most efficient direction, or split each communication in a move left and a move right if possible.

Termination will be discussed later.

Complexity Let d_i be the distance item i has to travel. The total communication work is $W = \sum_{i=0}^{n-1} d_i$. As long as all seats on the merry-go-round are occupied, at each step of the algorithm (i.e. each execution of the `shift` instruction), the travelling distance of each item on the merry-go-round decreases by 1, for a total work of P per step.

A seat on the merry-go-round become unoccupied when one of the P.Arrays first notice it has sent all its data away. We say in that case that the P.Array is empty. If there are already p empty P.Arrays and another one is emptied, an empty seat is generated. This seat moves around the merry-go-round until it reachs a non empty P.Array, and this takes at most $p + 1$ steps, for a work loss of $p + 1$. p takes all values from 0 to $P - 1$. When it reaches P, one needs at

most P steps to deliver remaining items. Let L be the number of steps until all items are delivered. The work equation is:

$$LP - l_1 - l_2 = W.$$

where l_1 is the work lost each time a P.Array is emptied, and l_2 is the work lost delivering the last items. We have:

$$l_1 \leq \sum_{p=0}^{P-1} = P(P-1)/2,$$

$$l_2 \leq P^2,$$

from which follows:

$$L \leq W/P + (2P-1)/2.$$

The first term can be rewritten in a more meaningfull way if we introduce the mean travelling distance, $\overline{d} = \frac{1}{n}\sum_{i=0}^{n-1} d_i$. We then get:

$$L \leq \frac{n}{P}\overline{d} + O(P).$$

The last term is overhead. It shows that one should regroup messages in order to amortize this overhead. The first may vary from $O(n/P)$ for short travelling distances (we then have full parallelism in communications) to $O(n)$ (since $\overline{d}$ is bounded by P). This shows that for long travelling distances, one should use P.Mem for communication. The latency using P.Mem is $O(n)$, and the constant factor is probably much smaller for P.Mem than for P.Arrays.

Applications For a simple shift, we have $\overline{d} = 1$ and $L = n/P$, a very favourable case. This situation is found in systolic-like algorithms, like the matrix-matrix product.

In the case of the Binary-Exchange version of the FFT, the mean travelling distance is $P/2$ for the fist step, then $P/4$, etc. Hence the total latency is $n/2 + n/4 + n/8 + \ldots \approx n$.

In the case of matrix transposition, one can prove that $\overline{d} = O(P)$, for a latency of $O(n)$.

Termination Termination is a complex question, since a processor must keep the merry-go-round running even if all its items have been sent away. The criterion for stopping is that just before the shift instruction, all seats of the merry-go-round are empty. The corresponding termination detection method is:

- All processors with `m.destination.processor < 0` set up a bit somewhere in memory.

- Synchronize the P.Mem.
- The P.Mem processor test whether all bits are set. If true, it sends a Terminated signal to all P.Arrays, which break out of the loop. If some processors are not finished yet, it sends an Unterminated signal, and the P.Arrays execute the shift.

This algorithm can be summarized as the following pseudo-code:

```
on pmem[i] in [0:0] {
  while(true) {
    while(~ _control_register);
    _control_register = 0;
    s = 1;
    for(ip=0; ip<P; ip++) s = s && nothing_to_do[ip];
    if (s){
        broadcast TERMINATED;
        break;
        }
    else broadcast UNTERMINATED;
    while (~ _control_register);
    _control_register = 0;
    broadcast 0;
 }


on parray[ip] in [0:P-1] {

  ......
  nothing_to_do[ip] = m.destination.processor < 0;
  _control_register |= (1 << ip);
  code = _broadcast_register;        /*implicit wait and reset */
  if (code == TERMINATED) break;
/* two step shift */
  [right]mm = m;
  _control_register = (1 << ip);
  code = _broadcast_register;
  m = mm;

  .......

}
```

2 Examples of FlexRAM Programming

2.1 Sparse Matrix Computation

```
float *a[NB], *y[NB], *x[NB];

   float ss[NB];

   void main(int argc, char *argv[]){

    int b, a_size, y_size, x_size;
    FILE *in;
    in = fopen(argv[1], "r");

    fscanf(in, "%d", &n);
    b = n/NB;
    a_size = NB * n;
    y_size = b;
    x_size = n;

    wait on flexram[ib] for(ib=0; ib<NB; ib++) {
        a[ib] = malloc(a_size * sizeof(float));
        y[ib] = malloc(y_size * sizeof(float));
        x[ib] = malloc(x_size * sizeof(float));
        }

    on flexram[ib] for(ib=0; ib<n; ib++) {
        in j;
        for(j=0; j<n; j++)
            x[ib][j] = 1.0;
        }

    for(ib=0; ib<NB; ib++)
        for(i=0; i<b; i++) {
            for(j=0; j<n; j++) {
                fscanf(in, %f%, &u);
                a[ib][i*b + j] = u;
                }
        flush a[ib][0 : n*NB -1];
        }

    wait on flexram[ib] for(ib=0; ib<NB; ib++);

    for(l=0; l<10; l++) {
```

```
wait on flexram[ib] for(ib=0; ib<NB; ib++) {
```

2.2 The Fast Fourier Transform

2.3 The Fast Fourier Transform

The sequential algorithm, as given in Kumar et. al., is:

```
r = log2(n);
mask = 1 << (r-1);
for(m=0; m<r; m++){
    for(i=0; i<n; i++) s[i] = r[i];
    for(i=0; i<n; i++){
        j = i &(~mask);
        k = i | mask;
        r[i] = s[j]+s[k]*twiddle_factor[bit_reverse(j) << (r-m)];
        }
    mask >>= 1;
    }
```

`r` and `s` are in fact arrays of complex numbers. The selection of the twiddle factor is not guaranteed.

In signal processing applications, one often do fixed point FFT. FFT is thus a natural for P.Arrays. As to the size parameters, there are two situations:

- The number of data points is small (i.e., arrays `r` and `s` fits in a P.Array memory). However, samples come at a very high rate from a data acquisition system (e.g. a camera or microphone), and the question is to do as many independent FFTs as possible in a given time. The solution is obviously to distribute the calculations on all P.Arrays, and the only problem is to feed the input and extract the results fast enough. This is the usual situation in real time signal processing systems.

- The number of data points is large (e.g. a whole FlexRAM is needed to hold `r` and `s`. This situation occurs in some signal processing applications, like Fourier spectroscopy, but mostly in case where the FFT is used to approximate derivatives when solving PDE. For instance, most global weather prediction models are currently converted to use "spectral methods", i.e., FFTs. This is the case we are interested in.

2.4 A P.Array Implementation

The obvious implementation is to distribute the n data points among p P.Arrays. Let us suppose for simplicity that n and p are powers of 2: $n = 2^r$ and $p =$

2^q. Each P.Array holds 2^{r-q} samples. It is easy to see that, for the first q iterations of the `m` loop, one of the accesses to `r[j]` and `r[k]` is distant, the other access being local. Begining with the q-th iteration, all accesses are local. Note that there are n twiddle factors. Hence they cannot be precomputed and replicated among the processors. The best solution is probably to compute them redundantly as needed, and this does not change the overall complexity of the algorithm (Kumar et. al. again).

There are two ways of implementing the data movements which are needed in the q initial iterations. The simplest way is to use the P.Mem:

```
#define NP 64
#define B 64*1024

int log2(int);

int r, q, b, mask1, mask2;
complex r[NP][B], s[NP][B], t[NP][B];

void fft(int n) {
 int ip;



 r = log2(n);
 q = log2(NP);
 b = n/NP;
 mask1 = 1 <<(q-1);
 mask2 = 1 <<(r - q -1);

 map r[ip][0:b-1] to p_array[ip] for(ip=0; ip<NP; ip++);
 map s[ip][0:b-1] to p_array[ip] for(ip=0; ip<NP; ip++);
 map t[ip][0:b-1] to p_array[ip] for(ip=0; ip<NP; ip++);

 for(m=0; m<q; m++) {
     wait on p_array[ip] for(ip=0; ip<NP; ip++) {
         for(i=0; i<b; i++)
             s[ip][i] = r[ip][i];
     }
     wait on pmem[0] {
         int ip, jp;
         int i;
         for(ip=0; ip<NP; ip++) {
             jp = ip ^ mask1;
             t[jp][i] = s[ip][i];
```

```
            }
        }
    wait on p_array[ip] for(ip=0; ip<NP; ip++) {
        for(i=0; i<b; i++)
            r[ip][i] = twiddle_factor[....]*s[ip][i]
                     + twiddle_factor[....]*t[ip][i];
        }
    mask1 >>= 1;
    }
 for(m=q; m<r; m++) {
    wait on p_array[ip] for(ip=0; ip<NP; ip++) {
        int j, k;
        for(i=0; i<b; i++)
            s[ip][i] = r[ip][i];
        for(i=0; i<b; i++)
            j = i & (~mask2);
            k = i | mask2;
            r[ip][i] = s[ip][j] + twiddle_factor[....]*s[ip][k];
        }
    mask2 >>= 1;
    }
}
```

The running time for this program has two components: the computation proper, taking $O(n/p \log_2 n)$, and the communication, taking about $O(n \log_2 p)$. The efficiency is:

$$\epsilon = \frac{1}{1 + \frac{p \log_2 p}{\log_2 n}}. \tag{1}$$

Since one complex number uses 8 bytes, the maximum possible value for n is about 2^{22}. With $p = 64$, the efficiency comes out slightly better than 5%.

The other solution is to use the nearest neighbor connection of the P.Arrays to move data around. The first communication ($m = 0$) move data a distance of $n/2$ in opposite directions, and thus needs $O(n)$ steps. For the next movement, the distance is halved, and only $O(n/2)$ steps are needed. Since:

$$n + n/2 + n/4 + \cdots \approx 2n,$$

the communication time becomes $O(2n)$ instead of $O(n \log_2 p)$, and the efficiency improves to:

$$\epsilon = \frac{1}{1 + \frac{2p}{\log_2 n}}, \tag{2}$$

giving about 15% efficiency for the same conditions as above.

2.5 Conclusion

The efficiency values given above are to be taken with a grain of salt, since we have not taken into account the ratio of the elementary computation time (one complex multiplication and one complex addition, or four integer multiplication and four integer addition) to the elementary communication time (two memory accesses). Nevertheless, the Fast Fourier Transform comes out to be communication bound.

There is another implementation, in which the n sample points are arranged as a $(\sqrt{n} \times \sqrt{n})$ matrix. The matrix is striped by columns on the available processors. The first $\frac{\log_2 n}{2}$ steps are then local. The matrix is then redistributed to obtain row striping, and the remaining steps are again local. If done by the P.Mem, the communication time is again $O(n)$, and the formula for the efficiency has the same shape as (2).

2.6 Sorting

```
/* This is a parallel variant of quicksort on the P.Arrays.
      The current number of p.arrays is NP. The host first sort NP-1
      elements of the input file to be used as pivots. The input elements
      are then distributed among the P.arrays according to the pivots.
Each
      P.Array sorts the elements which have been assigned to it, and
the host
      collect the results.
   */
   #define NB 4
   #define NP 64*NB

   #define N 128*1024
   #define ERROR 1

   struct params {
       int lower_limit;
       int upper_limit;
       int n;
       int mine;
       int bar_u;
       };

   struct workspace {
       struct params p;
       int arena[N];
```

```
      } wsp[NP];                /* NP number of P.Arrays in the configuration
*/

   int x;
   int n;
   int bar_s, bar_t, bar_u;

   void host_sequential_sort(int[], int);
   void p_array_sequential_sort(int[], int);

   void main(void){
       int n;                          /* number of elements to be
sorted */
       int pivot[NP];
       int ip, ib;
                                 /* NB : number of P.mems in the configuration
*/
       set_barrier bar_s = {host, p_mem[0:NB-1]};
       set_barrier bar_t = {host, p_mem[0:NB-1]};
       set_barrier bar_u = {host, p_arrays[0:NP-1]};


       scanf("%d", &n);

       map wsp[ip] to p_array[ip] for(ip=0; ip<NP; ip++);

       for(ip=1; ip<NP; ip++)
           scanf("%d", &pivot[ip]);
       host_sequential_sort(pivot+1, NP-1);
       for(ip=1; ip<NP; ip++) {
           wsp[ip-1].upper_limit = pivot[ip];
           wsp[ip].lower_limit = pivot[ip];
           wsp[ip].n = n-NP;
           wsp[ip].bar_u = bar_u;
           }
       wsp[0].n = n-NP;
       wsp[0].p.bar_u = bar_u;

       on p_array[ip] for(ip=0; ip<NP; ip++) {
           int i;
           for(i=0; i<wsp[ip].p.n; i++) {
               int x;
               receive_broadcast x;
               if((ip == 0 || x > wsp[ip].p.lower_limit)
```

```
            && (ip==NP-1 || w<= wsp[ip].p.upper_limit)) {
                wsp[ip].arena[wsp[ip].p.mine++] = x;
                if(wsp[ip].p.mine >= N) throw(ERROR);
                }
            }
        p_array_sequential_sort(wsp[ip].arena, wsp[ip].p.mine);
        barrier(wsp[ip].p.bar_u);
        }

    flush wsp[ip] for(ip=0; ip<NP; ip++);

    on pmem[ib] for(ib=0; ib<NB; ib++) {
        int i;
        for(i=0; i<n-NP; i++){
            barrier(bar_s);
            broadcast x;
            barrier(bar_t);
            }
        }

    for(i=0; i<n-NP; i++) {
        scanf("%d", &x);
        flush x;
        barrier(bar_s);
        barrier(bar_t);
        }

    r = barrier(bar_u);
    if(r == 0)
        for(ip=0; ip<NP; ip++)
            for(i=0; i<wsp[ip].p.mine; i++)
                printf("%d ", wsp[ip].arena[i]);
    else printf("Error %d\n", r);
}

/* Comments.
   This is a static version. The workspace is allocated in the
   static region of the host, then mapped to each P.Array.
   Delegating the selection process to the P.arrays is important
for
   performance, since the O(NP) comparisons are done in parallel.

*/
```

2.7 Tree Algorithms

3 Experiments

Jose

3.1 Experimental Setting

3.2 Results

4 Evaluation

The Whole Crew

A Formal Definition of C-Flex

A.1 Conventions

The C-flex language is defined as an extension to C. Hence, in the following sections, syntax rules may use apparently undefined non-terminals, which come in fact from a standard C grammar.

The shape of a production is:

```
non_terminal : terminal_and_non_terminal_list
             | terminal_and_non_terminal_list
             ....
             ;
```

Terminals are represented by uppercase names. You will never find a : in a production. This special character is tokenized first by the lexical analyser and you will find a SEMICOLON in its place. Most of the time, the token names are self-explanatory.

Each reserved word is also tokenized by the lexical analyser. Here again, the token names are self-explanatory. However, if you decide that you don't like a keyword (say you want to replace "release" by "unpin"), you do not have to change the grammar. You just change the lexical analyzer in such a way that `unpin` is tokenized to RELEASE.

The C grammar has a production for the non-terminal `statement` which is just a list of the different sorts of stateements in the language. The non-terminals of the form `*_statement` in the following are to be added to this list.

A.2 New Reserved Words

Word	Token	Comments
`on`	ON	
`parray`	PARRAY	
`pmem`	PMEM	
`phost`	PHOST	Is it necessary?
`flexram`	FLEXRAM	Is it necessary?
`waitfor`	WAITFOR	
`map`	MAP	
`release`	RELEASE	
`to`	TO	
`shift`	SHIFT	
`[|`	LBOX	
`|]`	RBOX	
`flush`	FLUSH	
`writeback`	WRITEBACK	
`invalidate`	INVALIDATE	

A.3 New Statements

A.3.1 Process Mapping

```
on_statement : ON processor processor_index range compound_statement ;

processor : PHOST
          | PMEM
          | PARRAY
          ;

processor_index :                       /* empty */
                | LPAR IDENT RPAR
                ;

range : LBOX RBOX
      | LBOX expression RBOX
      | LBOX expression COLON expression RBOX
      | LBOX expression COLON expression COLON expression RBOX
      ;

range_list : range
           | range range_list
           ;
```

Notes Combinations `[|` and `|]` are used in place of the usual convention `[ ]` to avoid ambiguity with subscripting.

`IDENT` is a generic token for strings of characters that conform to the grammar for identifiers and are not reserved words.

A range can be one value or an interval or an interval with a step. For instance, to start a process on all even numbered P.Arrays, just say:

```
on parray(i) [|0:63:2|] {
    ....
    }
```

`on pmem` can only be executed by the host.`on parray` can be executed by the host or by a P.Mem.

If there is not enough hardware to accomodate a given range, as in:

```
on parray [|0:1023|] { ...}
```

a run-time error occurs. It might be possible to relax this constraint at the price of a much more complex execution scheme (virtual processors).

The processes which are launched by this statement run in parallel with the process which launched them (no implicit wait).

One unsolved question is whether all P.Arrays in a configuration are to be considered as a whole or do we have to consider separatly the P.Arrays in each FlexRAM chip. In a four chips configuration, can we write:

```
on parray [|0 : 255 |] {
....
}
```

or

```
on pmem [|0:3|] {
    on parray [|0:63|] {
       ....
    }
}
```

The `compound_statement` can access the following variables:

- Variables that are local to the compound statement.
- Variables that are local to the enclosing function.
- Static variables, only in the case of P.Mem. Problems of synchronization are the responsability of the programmer.
- The processor index if it has been specified.

In my opinion, variables that are local to the enclosing function but not to the compound statement should be read only. The processor index *must* be read only.

A.3.2 On clause

The syntax of a function definition has to be extended to indicate which compiler is to translate the function:

```
on_clause :                  /* empty */
          | ON processor
          ;

function_definition : declaration_specifier declarator
                              on_clause compound_statement;
```

Notes An empty on clause is equivalent to `on phost`.

A.3.3 Wait Statement

```
wait_statement : WAITFOR processor range SEMICOLON ;
```

Notes `wait for pmem` can only be executed by the host. `wait for parray` can only be executed by the P.Mem.

There is no obligation that the range in a `wait` statement match any or all ranges in `on` statements. The following construction is strange but not invalid:

```
on pmem[0:1] { .... }
on pmem[2:3] { .... }
wait for pmem[1:2];
```

A.3.4 Map and Release Statements

```
map_statement : MAP object TO processor LPAR expression RPAR SEMICOLON
              | MAP object TO FLEXRAM
              ;

release_statement : RELEASE object SEMICOLON;

object : postfix_expression range_list
       ;
```

Notes These statements can only be executed by the host. A `map` statement is said to be active until the correspondibg `release` statememt has been executed.

Mapping is done in a unit of a page of the host architecture. The programmer is urged to map only large objects, after grouping if necessary. You should not map `chars`!

After a `map` statement, the operating system guarantees that, whenever the indicated processor is running on behalf of the current application, the indicated

object resides somewhere in the physical memory of the processor. There is no guarantee that:

- the physical address will stay the same as long as the `map` statement is active.
- the object will stay in memory when the application is not running.

After a `release` statement, the operating system is free to move the object anywhere in memory, if necessary.

On a machine with FlexRAM and ordinary DRAM, the construction:

```
map object to flexram ;
```

can be used to insure that an object is accessible from the P.Mems.

A.3.5 Flush Statements

A.4 New Library Functions

A.4.1 Dynamic Memory Allocation

The `malloc` function can be used from any processor with the usual semantics. If used from the P.Mems or P.Arrays, the allocated space is guaranteed to be mapped to the physical memory of the processor.

The following functions:

```
void *Flex_flexram_malloc(int size);
void *Flex_pmem_malloc(int size, int processor);
void *Flex_parray_malloc(int size, int processor);
```

can be used on the host only and return space which is mapped to the physical memory of the indicated processor.

The function:

```
void *Flex_sublet(int size, void *zone, int zone_size);
```

can be used anywhere to allocate space from the indicated memory zone, as in the following example:

```
void foo(void)   /* on the host */   {
   char free_space[NP][8 * 1024];

   for(ip=0; ip<NP; ip++) map free_space[ip][||] to pmem(ip);

   on pmem(ip)[0:NP-1] {
      int *q;
      ...
```

```
    q = (int *) Flex_sublet(n * sizeof(int),
                            (void *)free_space[ip], 8*PAGESIZE);
    ...
    }
}
```

The `Flex_sublet` function is completely local to a processor and hence is probably the most efficient one.

All other fumctions involve possible interaction with the host operating system. This interaction is local in the case of the host `malloc` and in the case of `Flex_flexram_malloc` et. al. These function are probably more efficient than `malloc` on P.Mem and P.Array.

Unsolved question: do we need as many `free` functions as there are `malloc` functions, or can we do with only one?

A.4.2 Cache Flush

```
#define F_WRITEBACK 1
#define F_INVALIDATE 2
....
void flush(void *pointer, int size, int flags);
```

This function can only be executed from the host or P.Mems: P.Arrays have no caches. The indicated range of addresses is written back to memory, or invalidated in the cache of the processor, or both.

There can be other flags.

A.4.3 Broadcast

The broadcast functions are patterned after the behaviour of the P.Mem/P.Arrays. They can be simulated, emulated or implemented on P.Host/P.Mems.

```
void broadcast(void *address, int size);
int poll();
int receive(void *address, int size);
```

The `broadcast` function can be used only on the P.Mem. The indicated address range is broadcast to all P.Arrays.

The `poll` function can be used only on the P.Arrays. It returns `true` (i.e. 1) if there is something in the broadcast register and `false` if the broadcast register is empty.

The `receive` function can be used only on the P.Arrays. It collect successive values from the broadcast register and store them in the indicated memory area. Reception stops whenever the area is full or there is nothing more to send. The return value is the number of received bytes.