Activity Counter: a New Optimization for SIMD Control Flow (extended version)

Ronan Keryell[*]
Centre de Recherche en Informatique
École des Mines de Paris
77305 FONTAINEBLEAU Cedex
FRANCE

Nicolas Paris[*]
Hyperparallel Technologies
École Polytechnique X-POLE
91128 PALAISEAU Cedex
FRANCE

11 January 1993

Abstract

SIMD computers and collection-oriented languages, like C*, are designed to perform the same computation on each data item or on just a subset of the data. Subsets of processors or data items are implemented via an *activity* bit and a stack of activity bits when subsets of subsets are supported. We present an implementation of activity stacks based on counters. At a given stack depth n, the number of memory bits required is $\log_2 n$, whereas previous implementations require n bits. The local controller is of equivalent complexity in both cases. This algorithm is useful for SIMD machines and for compilers of collection-oriented languages on MIMD computers.

1 Introduction

The data-parallel programming model is seen as an acceptable solution to efficiently program many parallel applications on massively parallel machines. In this model, a single program is applied on different instances of data, spread across different processors, to gain use of parallelism on SIMD or MIMD machines.

Data-parallel collection-oriented languages like MPL [Mas91], C* [Thi90] or PompC [Par92] and SIMD[1] computers like the MP-1 [Bla90], the CM-2 [Bla90] or POMP [HKMP91] have in common that they all deal with a set of objects and apply the same operation on each object of the set:

- in an SIMD computer there is a unique instruction flow and thus each processor performs the same operation on different data;

[*]Major parts of this work were made when the authors were with the Laboratoire d'Informatique de l'École Normale Supérieure, 45 Rue d'Ulm, 75005 PARIS, FRANCE. This research and the POMP project were partially funded by the French Research and Technology Ministry, Thomson Digital Image, the CNRS (National Center of Scientific Research), the LIENS, the École Normale Supérieure, the PRC-ANM.

[1]Single Instruction stream, Multiple Data streams.

- in a collection-oriented language, the data parallel semantics allow parallel operations on a set of data in a locked-step SIMD way: operations are applied on each element of data.

The goal of these parallel approaches is to obtain maximal performance on straight regular data parallel problems. However, it is at least as important to deal correctly with data parallel control flow.

As a matter of fact, a lot of numerical problems, like solving partial differential equation problems, often need to apply different at the boundary conditions which are different from the ones used on the interior points. The computation of many classical mathematical functions, such as the absolute value and many general algorithms also need such a control flow on parallel data. This is the extension in the space domain of the sequential `if...then...else` construct.

There seems to be an intrinsic contradiction in the commonly used SIMD control flow model and the need for a local instruction stream, *i.e.* a bounded dissynchronization in the synchronous SIMD model, to deal with parallel control flow.

Section 2 presents an abstract of the PompC language in order to display examples of data parallel control flow. Section 3 discusses background to generally related work and section 4 presents our new algorithm with some examples applied to PompC parallel control flow operators. Section 5 compares the activity stack with our method according to time and space complexity, for SIMD and MIMD, hardware and software. Section 6 presents a general view of the implementation of the method used in our POMP project and our PompC compiler. Section 7 presents related work.

2 POMPC: a collection-oriented data parallel language

Although it is not the object of this article, we briefly present a data parallel language to exhibit the problems related to parallel control flow in some languages with SIMD semantics.

PompC is a superset of the C language similar to the new version of C* and gathers some advantages of C* [Thi90], MPL [Mas91] and MULTIC [Wav91]. It is:

- a data parallel language since it can express computations on parallel data, with element by element interactions, such as additions of vectors or matrices;
- a collection-oriented language since each variable belongs to an equivalence class, a *collection* in our language, which defines its size, its geometry and its *activity*, a better suited term we introduce for the previous "mask" or "context" (detailed below).

Thus, PompC has a new keyword `collection` to declare a new collection. The collection name is a new parallel type that can be combined with the usual C type. In fact, PompC adopts a microscopic point of view of parallelism, in contrast to languages such as Fortran 90.

In Figure 1, we define a rectangular collection `an_array` with 100×200 elements and `a` and `b` two double precision variables. `a_vector` is a collection with a size and a number of dimensions unknown a compile time and `v` is a parallel double precision variable belonging to this collection.

```
collection an_array[100,200];
collection a_vector; /* The size is defined dynamically elsewhere in the program.
                                                                              */
double an_array a,b; /*  a and  b are parallel variables of the collection
                                                                  an_array. */
double a_vector v; /* The parallel size of  v is unknown at compile time. */
    ...
void div_by_0()
{
    int i = 0;
    double s = 0;
        ...
    where (a != 0)
        where (a > 0)
        {
            b = 1/a;
            i++;
            where (v > 0)
                s +<- v;
            b += s;
        }
        elsewhere
            b = -1/a;
    elsewhere
        b = 0;
}
```

Figure 1: Example in PompC with `where`/`elsewhere`s.

Belonging to the same collection allows variables to interact element by element. All other interactions are explicit and called communications, but this is not our interest here.

As in most parallel languages, PompC contains parallel control flow operators, like the classical `where`/`elsewhere` to select some variable elements. As in Actus [Per79] or C*, they can be nested. Each operator has an action only on the variables in the same collection as specified by the collection of the condition in the operator.

The function `div_by_0` of Figure 1 assigns 0 to an element of `b` when the corresponding element in `a` is 0, the absolute inverse of `a` when the element of `a` is negative and the inverse of `a` plus the sum of all the positive elements of `v` when `a` is positive.

The scalar operation `i++` is always executed since it is scalar code and does not belong to the `an_array` collection controlled by any previous `where`s. The scalar reduction `s +<- v` is always executed with the selected elements, the *active* elements, by the `where (v > 0)`. This `where` is not affected by the previous `where`s since they do not involve the `a_vector` collection.

We can consider this microscopic point of view of parallelism as if there were as many virtual SIMD machines in the computer as there are collections in the program.

Each virtual machine is naturally independent and has its own activity, its own number of virtual processors, its own geometry: it is the virtualization. An important point is that it is possible to have calls to functions in a conditional parallel block and these functions may have conditional parallel blocks depending on other collections. So it is neither always possible nor easy to have interprocedural analysis. Also such side effects may be hidden in libraries because of the mechanism of separated compilation in the C language, when source code is not available.

Therefore, activity resolution is needed at run time.

3 Classical approaches to parallel control flow

We use the same model as FLYNN [Fly66] and view an SIMD program as a sequential frame, running on a physical or virtual sequential machine, containing parallel instructions for a physical or virtual SIMD machine.

Conceptually, the parallel control flow can be seen as a jump into the future to an address which will be reached by the global instruction stream. Indeed, since a processing element (PE) cannot influence this stream, it has to wait until the stream gives it the expected instruction. In comparison with a sequential computer, it appears to be a space-time duality:

- an "SIMD branch" is a time delay since a PE must wait for the moving of the global instruction stream until the expected instruction is reached;
- a sequential branch (or on the PE of an MIMD[2] computer) is into the space since it can change the value of its program counter and instantaneously jump to another point of the program.

We need a global program counter with a value broadcasted to all the PEs which can execute or not an instruction if they have reached the branch destination or not [HLJ+91]. But behind this conceptual point of view there are some problems:

- programming with recursion is impossible since the recursion level is lost through a flat address space;
- a hardware realization requires the addition of a special bus to broadcast the global current address.

So this global address would have to be replaced by a global time, whose computation at compile time is equivalent to executing the program and is not usually possible.

Hopefully, if we restrict our interest to structured parallel programming then parallel control flow is applied on block structures: parallel `goto`s are not allowed.

3.1 Activity mask

The following properties hold for parallel instruction blocks:

- an instruction block has a beginning and an end;

[2]Multiple Instruction stream, Multiple Data stream.

- an instruction block is executed (active) or not (inactive), waiting for an active block, according to a local condition;
- each block is strictly nested into another block or is the main block of the program;
- by construction, a block inherits the activity of its ancestors, *i.e.* its surrounding blocks. Each idle block on a given collection has all the included blocks which are idle on this collection.

Thus it is useless to have a global program counter or a global time to tackle recursion. Instead the activity is stored in a local bit representing the current activity. This method is used in most SIMD [SBM62] and vector computers when parallel control flow exists [MU84].

If the programming model needs parallel control flow imbrications (nests of parallel control flow operators), the activity mask is pushed on a stack to recall the history.

3.2 Memory write control

Since each computation result can be seen as moving this result in memory, a variation of the previous method is to transform the program in such a way that a local condition controls the writing of the result, thus simulating local execution control.

This method has many drawbacks since:

- results cannot stay in registers for optimization, memory bandwidth is wasted;
- executed instructions may still have side effects like a divide by 0 exception for division.

This method is used in computers such as CRAY [Rus78] or GF11 [BDW85], along with an activity mask. Another useful but different application domain of this method is speculative execution for RISC and VLIW processors.

3.3 Local addressing

Another variation is to use local memory addressing to simulate the memory write. When a PE must be locally turned off, results are written in a dump memory cell instead of the destination cell. It is still slower than the previous method, but the necessary hardware is simpler.

3.4 Conditional instructions

Another simple method is to implement conditional instructions in the PEs like those in the GF11 microcode. Unfortunately, compilers have difficulties dealing with them when these instructions are used to build macrocode only, as in the BLITZEN [BDR87].

4 Activity counter

The activity bit stack is only used to determine the level of inactivity. Figure 2 shows an example of a nest of 6 parallel control flow statements, where the first three ones

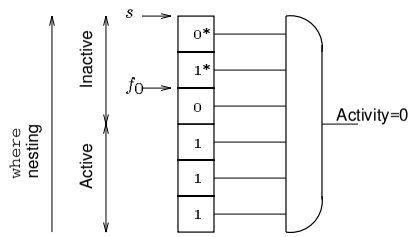

Figure 2: Example of a mask stack.

Table 1: Semantics of the `push` and `pop` operations on the activity stack.

Operation	Behavior	Precondition	Action
`push`(*cond*)	$s \leftarrow s+1$	$f_0 \neq s+1$	$f_0 \leftarrow f_0$
	$a_s \leftarrow cond$	$(f_0 = s+1) \wedge (cond = 0)$	$f_0 \leftarrow s$
		$(f_0 = s+1) \wedge (cond = 1)$	$f_0 \leftarrow s+1$
`pop`	$if^a(s>1), s \leftarrow s-1$	$f_0 \neq s+1$	$f_0 \leftarrow f_0$
	`return`(a_s)	$f_0 = s+1$	$f_0 \leftarrow s+1$

[a]Note that if the program is correct, this condition is always true.

have *true* conditions (shown as "1" in the figure) and the condition is *false* after the third one (represented by "0").

Before the first *false* condition, the stack only contains 1s, indicating that the PE is executing the code. The exit of a conditional block does not change this activity; the PE remains active. These 1s do not have any intrinsic significance in the stack.

When a PE reaches a local *false* condition, it becomes inactive for all its included blocks. The current activity is the logical *and* of the history of activity, *i.e.* all the activity bits on the stack. Once a 0 bit is pushed on the stack, all the following bits on the stack no longer have meaning (represented with a "*" in Figure 2.) since the activity is 0 (inactive).

4.1 Factorization

The only useful information in this stack is the number of imbrications of parallel conditional blocks after the first idle block, which indicates when a PE can resume execution. Therefore, it seems a waste of hardware to use a stack where a plain counter should be enough.

Let `push`(*cond*) and `pop` be the two operations controlling the stack $(a_i)_{i\in\mathbb{N}}$. We can analyze their functionality according to f_0, the rank of the first 0 on the stack, and s the current size of the stack, according to Figure 2. The activity of a PE is defined by $\mathcal{A} = \bigwedge_{i=0}^{s-1} a_i$. The PE is active if $\mathcal{A} = 1$ and idle if $\mathcal{A} = 0$.

By definition, PEs are all active at initialization time, so $s = 1$, $a_0 = 1$ (active), $f_0 = s + 1$ when there is no 0 in any stack element. A `pop` on an empty stack returns an activity *true*.

Table 1 gives an operational semantics of the activity stack. A PE is active if and only if $f_0 = s + 1$, when there is no 0 in the stack. In fact, it is more interesting to do

Table 2: Semantic of the `push` and `pop` operations on the activity counter.

Operation	Precondition	Action
`push`(*cond*)	$c \neq 0$	$c \leftarrow c + 1$
	$(c = 0) \wedge (cond = 0)$	$c \leftarrow 1$
	$(c = 0) \wedge (cond = 1)$	$c \leftarrow 0$
`pop`	$c \neq 0$	$c \leftarrow c - 1$
	$c = 0$	$c \leftarrow 0$

Table 3: Implementation of the `where`/`elsewhere` with an activity counter.

Operation	Precondition		Action
`where`(*cond*)	$c \neq 0$	*(idle)*	$c \leftarrow c + 1$
	$c = 0$	*(active)*	$c \leftarrow \neg cond$
`elsewhere`	$c \leq 1$	*(activatable)*	$c \leftarrow \neg c$
	$c \not\leq 1$		$c \leftarrow c$
End of the `where` /`elsewhere`	$c \neq 0$	*(idle)*	$c \leftarrow c - 1$
	$c = 0$	*(active)*	$c \leftarrow 0$

the variable exchange $c = s + 1 - f_0$ because only a comparison to 0 is necessary. This form is easier to implement in hardware and often even in software [Ker89, Ker92]. The basic manipulations on c are the same as on f_0: increment or decrement, load or store, as shown on Table 2.

The `push`(*cond*) when $c = 0$ can be simplified to $c \leftarrow \neg cond$. A more detailed proof of the equivalence between an activity stack and an activity counter for parallel control flow can be found in [BL92].

4.2 Application to a data parallel language

Now we can use this mechanism to implement classical parallel control flow operators such as those in the PompC language.

4.2.1 `where`

The basic operator is the `where`/`elsewhere` pair which is found in most data parallel languages from Fortran 90 to C*.

The `where` is equivalent to the `push` operator but we have to translate the `elsewhere`. A PE is active in an `elsewhere` if and only if the PE was inactive due to the last `where`, *i.e.* the inactivity level $c = 1$. The value 1 can be seen here as a special value that codes for an "activatable" state for the `where` or `elsewhere` block.

An implementation is presented in Table 3.

4.2.2 `whilesomewhere`

A parallel extension of the C language `while` is the `whilesomewhere` construct which iterates a parallel loop until all parallel conditions are false. The `continue` and `break`

Table 4: Implementation of the `whilesomewhere` with an activity counter.

Operation	Precondition		Action
`whilesomewhere`(*cond*)	$c \neq 0$	*(idle)*	$c \leftarrow c + 2$
	$c = 0$	*(active)*	$c \leftarrow 2 \times \neg cond$
Loop beginning	*all cond* $= 0$		$\rightsquigarrow$ *Loop exit*
`continue`	$c = 0$	*(active)*	$c \leftarrow 1^a$
`break`	$c = 0$	*(active)*	$c \leftarrow 2^a$
Loop end	$c = 1$	*(re-activatable)*	$c \leftarrow 0$, $\rightsquigarrow$ *Loop begin*
	$c \neq 1$		$\rightsquigarrow$ *Loop beginning*
Loop exit	$c \leq 1$	*(activatable)*	$c \leftarrow 0$
	$c \not\leq 1$	*(idle)*	$c \leftarrow c - 2$

[a] Relative value to the current `whilesomewhere` block. See Section 4.2.2.

are also extended to inactivate a PE until the next iteration or until the `whilesomewhere` exit, respectively.

A PE can be in one of the following states:

1. inactive before the entry in the `whilesomewhere` block, thus it will remain inactive in this block;

2. active at the `whilesomewhere` entry;

3. inactive at the `whilesomewhere` entry because the local condition is false; the PE will remain inactive until the `whilesomewhere` exits;

4. inactive in the `whilesomewhere` until the `whilesomewhere` exits because of a `break`.

5. inactive in the `whilesomewhere` until the next iteration after executing a `continue`;

States 3 and 4 are not different once the `break` is executed; the PE remains idle until the `whilesomewhere` exits. They can be implemented in the same way.

In comparison with the `where/elsewhere`, only point 5 requires additional mechanisms. It is implemented by reserving a protected value 1 in the counter by a double incrementation in the first row of the implementation presented on table 4.

The value $c = 1$ is used to implement state 5 (the `continue`) and the value $c = 2$ is used for the states 3 and 4 (line 2 and row 4 of Table 4.

In a real implementation on an SIMD machine, there is a scalar loop around *Loop begin/Loop end* exiting on *all cond* $= 0$ condition. On an MIMD machine, the loop is repeated on each PE.

A more complex but common case is when the `continue` or the `break` are enclosed in a `where/elsewhere`. In this case, $c \leftarrow 1$ or $c \leftarrow 2$ could be thought of as an `where/elsewhere` exit instead of a `break` or a `continue`. It is the reason why the value marked with "^{a}" must be relative to the current `whilesomewhere` block, *i.e.* augmented

Table 5: Implementation of the `switchwhere` with an activity counter.

Operation	Precondition		Action
`switchwhere`(*value*)	$c \neq 0$	*(idle)*	$c \leftarrow c + 2$
	$c = 0$	*(active)*	$c \leftarrow 1$
`case` *constant* :	$(c = 1) \wedge (value = constant)$		$c \leftarrow 0$
`break`	$c = 0$	*(active)*	$c \leftarrow 2^a$
`default` :	$c = 1$	*(activatable)*	$c \leftarrow 0$
`switchwhere` *closing*	$c \leq 1$		$c \leftarrow 0$
	$c \not\leq 1$		$c \leftarrow c - 2$

[a]Must be relative to the current `switchwhere` block, if the `break` is included in one or more `where/elsewhere`.

by the number of `where/elsewhere` imbrications between the `whilesomewhere` block and the `continue` or the `break`.

The `dowhere/whilesomewhere` version is derived from Table 4 as the transformation of a `while` into a `do/while` in the C language. Similarly, the implementation of a data parallel `for`, a `forwhere`, is derived from the `whilesomewhere` according to the translation of `for` to `while` in C.

4.2.3 `switchwhere`

The compilation of a `switchwhere`, the parallel extension of the language C `switch`, also has several states. A PE can be:

1. inactive before the `switchwhere`;
2. active in a `case` (after matching a value) or in a `default`;
3. inactive in a `case`, waiting for a matching value;
4. inactive in the `switchwhere` because of a break, until the `switchwhere` exit.

The `break` is similar to the `whilesomewhere` one. An example of state coding we use is $c = 1$ for the state 3 and $c = 2$ for the state 4, as shown in Table 5.

4.2.4 Parallel `return`

The parallel `return` allows a parallel function to return a parallel value and can be used in parallel control flow in the example of Figure 3.

In this case, we want the PEs that execute the parallel `return` to become idle until the exit of the current function. It is very similar to the `break` in a `whilesomewhere` or a `switchwhere` and has the following states:

1. a PE is active at the function entry;
2. a PE is inactive at the function entry;
3. a PE is inactive until the end of the function because it has executed a `return`.

```
collection a_coll double fabs(a_coll double x)
{
    where (x > 0)
        return x;
    elsewhere
        return -x;
}
```

Figure 3: Example of a generic parallel absolute value function in PompC.

Table 6: Implementation of the parallel `return` with an activity counter.

Operation	Precondition		Action
Function entry	$c \neq 0$	*(idle)*	$c \leftarrow c + 1$
`return`	$c = 0$	*(active)*	$c \leftarrow c + 1$[a]
Function closing	$c \neq 0$	*(idle)*	$c \leftarrow c - 1$

[a] Must be relative to the current function block, if the `return` is included in some parallel conditional blocks.

In order to constrain the visibility of the parallel `return` of the function, a value $c = 1$ is reserved in the counter, as shown in Table 6.

5 Activity counters versus activity stacks

In order to develop choice arguments, we have to analyze the time and space complexity of the activity counter method and activity stack method, both for SIMD and MIMD computers.

5.1 On an SIMD machine

The counter method needs a counter with $\log_2 c$ bits per PE if at most c levels of parallel conditional blocks are imbricated. If each PE has an L-bit operator, a PE needs $\lceil \log_L c \rceil$ cycles of duration t to do an activity counter operation.

The activity stack needs only 1-bit manipulation on each PE and takes a time t, but needs a stack pointer to manage the stack. Since the execution is SIMD, all the stacks are synchronous and the stack pointer can be:

- centralized on the scalar processor which broadcasts its value to the PEs;
- distributed with local pointers which evolve synchronously.

In the first case, it takes a time T on the scalar processor and the time is negligible on the PEs. In the second case, a time $t\lceil \log_L c \rceil$ is needed to control the stack pointer on each PE. The hardware complexity is c for a stack of 1 bit elements in each case, plus $\lceil \log_L c \rceil$ bits for the global stack pointer in the first case and $N\lceil \log_L c \rceil$ bits for the local stack pointers in the second case, for a N-PE computer.

Table 7: Complexity of the activity counter and activity stack methods.

Parallel conditioning	Computing complexity scalar	Computing complexity parallel	Hardware complexity	# broadcast
Stack (global pointer)	T	t	$Nc + \lceil \log_2 c \rceil$	1
Stack (local pointers)	ϵ	$t(1 + \lceil \log_L c \rceil)$	$N(c + \lceil \log_2 c \rceil)$	0
Activity counters	ϵ	$t\lceil \log_L c \rceil$	$N\lceil \log_2 c \rceil$	0

The complexity of the three previous methods are summarized up in Table 7.

If the computer has only fine grain PEs, typically $L = 1$ or 4 bits, it is more interesting to subcontract the computation to the scalar processor with a global stack pointer. Indeed, the scalar processor is often larger and more powerful, so the stack pointer computation only uses few cycles, and even the broadcast is often shorter than the $t\lceil \log_L c \rceil$ required to deal with a local stack pointer or activity counter by L-bit slices. Moreover, 1-bit PEs have the advantage that they easily access memory with 1-bit. This method is used on computers such as the CM-2 [Thi87] or the MP-1 [Bla90].

The activity stack with a local stack pointer is well suited for small PEs, when a broadcast is too expensive and a local indirection is available. But the local indirection in memory must be fast enough to compensate for the lack of fast broadcast. Unfortunately, a fine grain PE rarely has a large enough adder on the address bus to keep with a fast memory indirection, and more generally a sufficient data address bus throughput compared with the data bus throughput, since there often is a common address data bus which is time-multiplexed between all the PEs of a chip.

The activity counter algorithm is particularly interesting for coarse grain SIMD machines. These computers often have short cycle time and the local memory access is slow in comparison to the PE cycle time. A large data bus to memory is not adapted to deal with 1-bit accesses in memory required by the activity stack, so the activity counter clearly avoids wasting memory bandwidth. The drastic reduction in storage size allows a counter hardware implementation on the PE that minimizes latency.

The activity counter could have been applied efficiently in the ILLIAC IV [BBK+68], GF11 [BDW85], OPSILA [AB86] and more recently the CM-5 [Thi91] or the MP-2.

If we were designing a new SIMD computer, we would implement the activity counter in hardware instead of emulating it in software. However, the method is also useful in software and can improve compilers for existing architectures.

5.2 On an MIMD machine

The complexity of our method for an MIMD machine is the same as in table 7 except that since there is no scalar processor, it is not interesting to have a global activity stack pointer and thus only local pointers or activity counters are necessary.

As for the SIMD computers, the same conclusions arise according to the size of the PEs. Activity counters can avoid the 1-bit stack management, specially inefficient on the coarse grain PEs which are in most MIMD computers. Besides, the activity counter on each PE reduces to $\mathcal{O}(\log c)$ the hardware complexity to store the activity.

But unlike SIMD computers, it is not worth implementing the activity counter in hardware since local conditional jumps are used *in fine* to efficiently emulate the activity

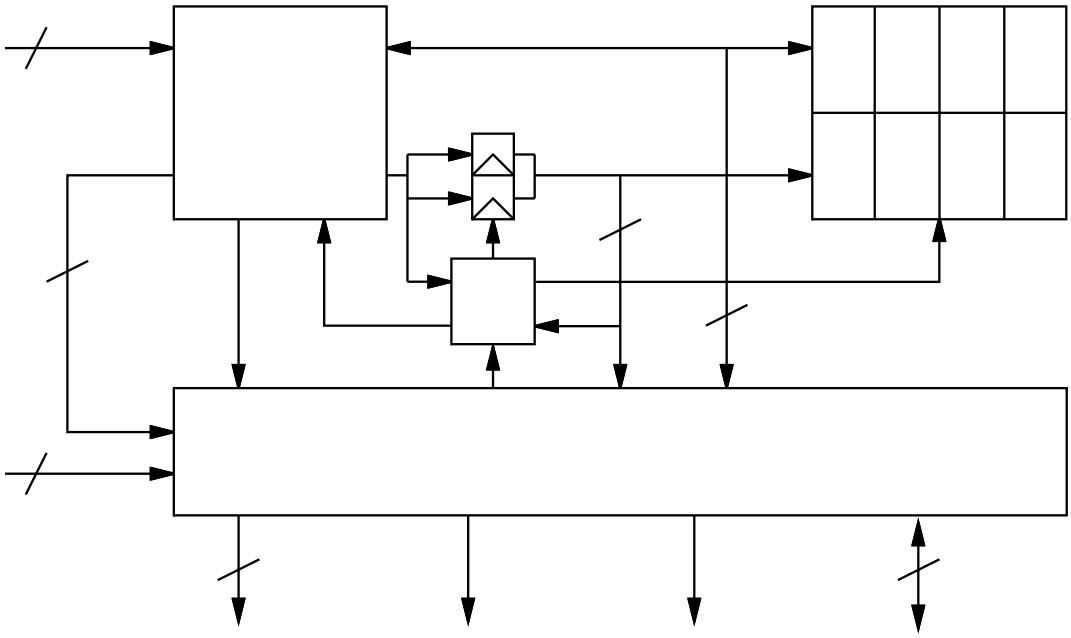

Figure 4: PE node synoptic in the POMP computer.

corresponding to the counter value. Thus, it would need incessant exchanges between a PE, its counter and its memory.

6 Implementation

In the POMP project, we have studied a hardware implementation for the POMP machine and a compiler for the PompC language which uses activity counters if they exist or emulates them if not.

6.1 Hardware application

In the POMP SIMD computer [HKMP91, Ker92], off-the-shelf RISC PEs are used. Since they are coarse grain processors (Motorola MC88100) the activity counter is valuable.

It was not possible to implement such a counter inside the processor, so it was done in a companion circuit made with a FPGA (named HyperCom in figure 4) which also has other functions that allow the machine to run in an SIMD mode (communications, broadcast, exception control,...).

The activity counter is implemented with a 32-bit counter, allowing up to 2^{32} nested parallel conditional blocks and a comparator to 0,1 and 2 to speed up `whilesomewhere` and `switchwhere`. The control of the instruction execution in the PE is performed by the FPGA which controls the ready signal (`CR0:CR1`) of the instruction bus of the PE through a fast PAL that does not slow down the pipelined processor bus.

The parallel conditional execution is supervised by the 8-bit HyperCom instruction added to the 32-bit basis instruction of the PE. These instructions are produced by

the scalar processor, another MC88100 with a VLIW coupling. Each PE can access its activity counter for complex parallel control flow and for virtualization goals through the data bus. DMA operation between the counter and the memory is done simply but optimally by initializing a read on the PE at the memory address plus a major offset indicating to the HyperCom it has to spy the bus.

6.2 Compilation

A compiler of PompC for the CM-2, the MP-1 and Unix workstations for simulation has been written and generates C*, MPL and C programs respectively.

The compiler for the CM-2 is not using activity counters since the PEs only have 1-bit operators and virtualization is present in C*.

On the contrary, the MP-1 has larger 4-bit PEs and MPL does not virtualize, *i.e.* only parallel variables with the same size as the physical machine can be used. Thus soft activity counters are used to deal with parallel control flow in the compiler of PompC to MPL.

The imbrication of parallel control flow operators, merging different collections and scalar operations, are sliced in the same manner as in the Actus compiler described in [PCMP85], but they use *extents of parallelism* to manage an activity stack instead of an activity counter. In the PompC compiler, each time there is an operation affecting another collection, the current activity counters are saved in memory and new values for the next collection are loaded instead.

Since the POMP prototype is not yet finished, we do not have a compiler which uses an activity counter in hardware.

7 Related work

In Section 4 we presented an optimization of the activity concept described in Section 3. But other methods based on a different principle can be found in the literature.

A common solution for vector computers is the use of *scatter/gather* to boost vector performance on sparse problems. Computations on subdomains are put together in different parallel variables corresponding to the subdomains of the problem. Thus, each variable is a dense vector and the vector processor can be used efficiently on it. At the end of the computations, the dense vectors are scattered back to their original place. Unfortunately, this method needs the programmer to restructure the program and sometimes to modify the algorithm.

A generalization of this method is presented in [Ble89, BS90] for the Parallation Lisp compiler to SIMD computers. Each subdomain selection is transformed in a scatter/gather couple, *i.e.* a global communication before and after the computation. The method does not implement complexity evaluation to choose between classical activity method and scatter/gather. Even when this complexity evaluation is possible at compile time, it is not trivial. But it is necessary because the scatter/gather may be negligible compared with an activity method, depending of programs and data (a big computation kernel concerning only one collection in a `where`, a `where` and an `elsewhere` well-balanced) or it may be very important due to communication overheads (a lot of `where` and many collections concerning few operations).

In the first case the scatter/gather method is better but in the second case an activity method is advisable. Since we improve the activity method, we also extend its application domain.

8 Conclusion

We have developed a new method to deal with nested parallel control flow for SIMD and MIMD computers, and compilers for languages with collection oriented data parallelism.

This technique allows a reduction to a straight logarithmic term of the size in bits of memory used to keep track of the PE history. The method is more suited to coarse grain parallel computers since it avoids an inefficient indirection in memory, 1-bit stack management, a global address broadcast and relieves the scalar processor of a henceforth superfluous load.

The optimization is also interesting for compilers targeted to modern MIMD computers when the imbricated parallel control flow cannot be resolved at compile time. For example, if different collections are mixed, interprocedural analysis is not performed or not possible, or if complex sub-array selections cannot be determined. If the activity counter method can often be replaced by MIMD local control flow, for complex imbricated case it seems a better choice.

9 Acknowledgements

The authors of this paper would like to acknowledge many useful discussions with all the members of the POMP team since the beginning of the project: Philippe MATHERAT, Philippe HOOGVORST, César DOUADY, Patrice OSSONA DEMENDEZ, Théodore PAPADOPOULO and Pierre CHICOURRAT.

Special thanks are due to Luc BOUGÉ and his team, especially Jean-Luc LEVAIRE, for their discussions on SIMD semantics in parallel control flow and for their interest for the domain and our work.

At last but not the least, the authors are indebted to Kathryn MACKINLEY, François IRIGOIN and Pierre JOUVELOT for their invaluable comments and their appropriate suggestions.

References

[AB86] M. AUGUIN and F. BOERI. "The OPSILA Computer". In INRIA, editor, *Parallel Algorithms & Architectures*, pages 143–153. North-Holland, 1986.

[BBK+68] George H. BARNES, Richard M. BROWN, Maso KATO, David J. KUCK, Daniel L. SLOTNICK, and Richard A. STOKES. "The ILLIAC IV Computer". *IEEE Transactions on Computers*, C-17(8):746–757, August 1968.

[BDR87] Donald W. BLEVINS, Edward W. DAVIS, and John H. REIF. "Processing Element and Custom Chip Architecture for the BLITZEN Massively Parallel Processor". Technical Report TR87-22 Revision 1, Microelectronics Center of NC, 1987.

[BDW85] John BEETEM, Monty DENNEAU, and Don WEINGARTEN. "The GF11 Supercomputer". In *SIGARCH 85*, pages 108–115. The Institute of Electrical and Electronics Engineers, Inc., 1985.

[BL92] Luc Bougé and Jean-Luc Levaire. "Control structures for data-parallel SIMD languages: semantics and implementation". *Future Generation Computer Systems*, 8(3-4):363–378, 1992.

[Bla90] Tom Blank. "The MasPar MP-1 Architecture". In IEEE, editor, *IEEE Compcon Spring 1990*, February 1990.

[Ble89] Guy E. Blelloch. "*Scan Primitives and Parallel Vector Models*". PhD thesis, Laboratory for Computer Science — Massachusetts Institute of Technology, October 1989. MIT/LCS/TR-463.

[BS90] Guy E. Blelloch and Gary W. Sabot. "Compiling Collection-Oriented Languages onto Massively Parallel Computers". *Journal of Parallel and Distributed Computing*, 8(2):119–134, February 1990.

[Fly66] Michael J. Flynn. "Very High-Speed Computing Systems". *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[HKMP91] Philippe Hoogvorst, Ronan Keryell, Philippe Matherat, and Nicolas Paris. "POMP or How to Design a Massively Parallel Machine with Small Developments". In *PARLE '91 Parallel Architectures and Languages Europe*, volume 505(I), pages 83–100. Lecture Notes in Computer Science, Springer-Verlag, June 1991. Available by `ftp anonymous` on `spi.ens.fr` in the file `pub/reports/liens/liens-91-5.A4.ps.Z`.

[HLJ+91] Philip J. Hatcher, Anthony J. Lapadula, Robert R. Jones, Michael J. Quinn, and Ray J. Anderson. "A Production-Quality C* Compiler for Hypercube Multicomputers". In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26(7), pages 73–82, July 1991. SIGPLAN Notices.

[Ker89] Ronan Keryell. "POMP2 : D'un Petit Ordinateur Massivement Parallèle". Rapport de magistère, LIENS — Ecole Normale Supérieure, October 1989.

[Ker92] Ronan Keryell. "*POMP : d'un Petit Ordinateur Massivement Parallèle SIMD à Base de Processeurs RISC — Concepts, Etude et Réalisation*". PhD Thesis, Laboratoire d'Informatique de l'Ecole Normale Supérieure — Université Paris XI, October 1992.

[Mas91] MasPar Computer Corporation. "*MasPar Parallel Application Language (MPL) Reference Manual*", document part number: 9302-000, revision: a4 edition, March 1991. Software Version 2.0.

[MU84] Kenichi Miura and Keiichiro Uchida. "Facom Vector Processor System: VP-100/VP-200 ". In *Proceedings of NATO Advanced Research Workshop on High Speed Computing*, volume F7, pages 59–73. IEEE Computer Society Press, 1984.

[Par92] Nicolas Paris. "Definition of POMPC (Version 1.99)". Technical Report LIENS-92-5-bis, Laboratoire d'Informatique de l'École Normale Supérieure, March 1992. Available by `ftp anonymous` on `spi.ens.fr` in the file `pub/reports/liens/liens-92-5-bis.A4.ps.Z`.

[PCMP85] Ronald H. Perrott, Danny Crookes, Peter Milligan, and W. R. Martin Purdy. "A Compiler for an Array and Vector Processing Language". *IEEE Transactions on Software Engineering*, SE-11(5):471–478, May 1985.

[Per79] R. H. Perrott. "A Language for Array and Vector Processors". *ACM Transactions on Programming Languages and Systems*, 1(2):177–195, October 1979.

[Rus78] Richard M. Russel. "The CRAY-1 Computer System". *Communications of the ACM*, 21(1):63–72, January 1978.

[SBM62] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. "The SOLOMON Computer". In *Proceedings of the Fall 1962 Eastern Joint Computer Conference*, pages 97–107, December 1962.

[Thi87] Thinking Machine Corporation. "*Connection Machine Model CM-2 Technical Summary*", April 1987. HA87-4.

[Thi90] Thinking Machine Corporation. "*C* Programming Guide*", November 1990. Version 6.0.

[Thi91] Thinking Machine Corporation. "*The Connection Machine CM-5 Technical Summary*", October 1991.

[Wav91] Wavetracer Inc. "*The multiC Programming Language — User Documentation*", pub-00001-001-1.01 edition, September 1991.