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

Ronan Keryell
Centre de Recherche en Informatique
Ecole des Mines de Paris
77305 Fontainebleau Cedex
France

Nicolas Paris
Hyperparallel Technologies
Nokia Paris

Abstract

1 Introduction

In 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 technique of activity stacks based on counters, which are efficient in terms of memory but not in terms of performance. This algorithm is used for SIMD machines and for compilers of collection-oriented languages on SIMD or MIMD architectures. Major parts of this work were made when the authors were with the Laboratoire d’Informatique de l’Ecole Normale Supérieure, 45 Rue d’Ulm, 75005 Paris, France. This research and the pomp project were partially funded by the French Ministry of Research and Technology, the Centre National de la Recherche Scientifique, and the Fondation Nationale des Sciences Exactes et Naturelles. The authors are also grateful to the CNRS, the Centre de Recherche en Informatique, and the Ecole des Mines de Paris. France.

In an SIMD computer, there is a unique instruction and thus each processor performs the same operation on different data.

11 January 1993

45 Rue d’Ulm, 75005 Paris
Centre de Recherche en Informatique
FRANCE

91128 Palaiseau Cedex
Ecole Polytechnique
FRANCE

77305 Fontainebleau Cedex
Centre de Recherche en Informatique
FRANCE

version
In a collection-oriented language, the data parallel semantics allow parallel operations on a set of data in a locked-step way; operations are applied on each variable of dimensions unknown at compile time and is a parallel double precision a-integer a-vector a-parallel collection an array with 100 x 200 elements.

In this, POMP has a new keyword collection to declare a new collection. This 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 model and the need for a local instruction stream. As a matter of fact, a lot of numerical problems like solving partial differential equations on grid need a parallel control flow. However, it is at least as important to deal correctly with data parallel control now.

Although it is not the object of this article, we briefly present a data parallel language.

POMP: a collection-oriented data parallel language

POMP is a superset of the C language similar to the new version of C and gathers some advantages of C, C++, and Fortran. It is:

1. a data parallel language since it can express computations on parallel data, with a data parallel language since it can express computations on parallel data, with
2. a collection-oriented language since each variable belongs to an equivalence class, a collection-oriented language since each variable belongs to an equivalence class,
3. a collection-oriented language which defines its size, its geometry and its activity, a collection-oriented language which defines its size, its geometry and its activity,
4. a better suited term to introduce for the previous "mask" or "context" (detailed below).

In Figure 1, we define a rectangular collection an array with 100 x 200 elements.
We can consider this instruction point of view of parallelism: if there were as
many virtual SIMD machines in the computer as there are collections in the program,
then we involve the "array" collection.

The where (a < 0) is not affected by the previous WHERE since they do not
involve the "vector" collection, whereas the collection's WHERE is always executed
with the selected elements of the collection. The collection's WHERE is always executed
before the "array" collection controlled by any previous WHERE. The scalar re-
duction of a is always executed when the scalar code and does not depend on the
selected elements of the "vector" collection. The scalar reduction of a is always executed
when the scalar of a is selected.

The function div_by_0 of Figure 1 assigns 0 to an element of b when the corre-
sponding element in a is 0, the absolute inverse of a when the denominator of a is nega-
tive, and the inverse of a plus the sum of all the positive elements of a when a is positive.

The parallelism of Figure 1 is a microscopic point of view with WHERE/ELSEWHERE.

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.

Figure 1: Example in ParC with WHERE/ELSEWHERE.

```c
double a[200];
double b[200];
int i;
double s;

void div_by_0()
{
    where (a != 0)
    {
        where (a > 0)
        {
            s = 1 / a;
        }
        else
        {
            s = -1 / a;
        }
    }
    for (i = 0; i < 200; i++)
    {
        where (i != 0)
        {
            s = s + a[i];
        }
        else
        {
            s = s - a[i];
        }
    }
}
```

/* The parallel size of a is unknown at compile time. */
double a[200];
/* The parallel size of a is defined dynamically elsewhere in the program. */
collection a[200];
/* The size is defined dynamically elsewhere in the program. */
collection an_array[100, 200];
```
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.

3.1 Activity mask

3.1.1 Classical approaches to parallel control flow

Each virtual machine is naturally independent and has its own activity. The following properties hold for parallel instruction blocks:

- An instruction block has a beginning and an end.
- A parallel instruction block has a parallel instruction block.

The following properties hold for parallel instruction blocks:

- Global control flow is applied on block structures: parallel blocks are not allowed.
- Global control flow is applied on block structures: parallel blocks are not allowed.

3.4 Classical approaches to parallel control flow
In a nested or parallel control flow statements, where the first three ones
the activity bit stack is only used to determine the level of inactivity. Figure 2 shows

4 Activity counter

Another simple method is to implement conditional instructions in the mask. Like those

3.4 Conditional instructions

As a result, the conditional instruction can be seen as

3.3 Local addressing

Another variation is to use local memory addressing to simulate the memory write.

3.2 Memory write control

Another variation is to use conditional instructions to transform the program in such a way that a local instruction

4 Activit y counter

The activity mask is used in a global program counter of a global time to track the history.

The activity bit stack is used to determine the level of inactivity. Figure 2 shows

An example of a nest of parallel control flow statements, where the first three ones

The activity bit stack is only used to determine the level of inactivity. Figure 2 shows

4 Activity counter

Another simple method is to implement conditional instructions in the mask. Like those

3.4 Conditional instructions

As a result, the conditional instruction can be seen as

3.3 Local addressing

Another variation is to use local memory addressing to simulate the memory write.

3.2 Memory write control

Another variation is to use conditional instructions to transform the program in such a way that a local instruction

4 Activit y counter

The activity mask is used in a global program counter of a global time to track the history.

The activity bit stack is only used to determine the level of inactivity. Figure 2 shows
Table 1 gives an operational semantics of the activity stack. A pe is active if and only if \( f_0 = s + 1 \), where there is no 0 in the stack. In fact, it is more interesting to do

<table>
<thead>
<tr>
<th>Operation</th>
<th>Action</th>
<th>Precondition</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>push ((c))</td>
<td>( s + )</td>
<td>( s + 1 ) ( s = ) ( s + 1 )</td>
<td>( s = ) ( s + 1 )</td>
</tr>
<tr>
<td>pop</td>
<td>( s )</td>
<td>( s &gt; 1 )</td>
<td>( s = ) ( s - 1 )</td>
</tr>
</tbody>
</table>

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

Figure 2: Example of a mask stack.

The only useful information in this stack is the number of combinations 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 would suffice. Let \( \text{push}(c) \) and \( \text{pop} \) be the two operations controlling the stack (\( N \)). 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 the current size of the stack, according to Figure 2. The activity of a pe is defined by the current size of the stack, according to Figure 2.

Before the first false condition, the stack only contains 1s, indicating that the pe is executing the code. Once a 0 is pushed on the stack, the following bits contain boolean values 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 have true conditions (shown as \( \sim \) in the figure) and the condition is false after the first idle block. Therefore, the first false condition on the stack is the only condition that the pe should be enough.
4.2.2 "elsewhere"

An implementation is presented in Table 3.

The basic operator is the "where/elsewhere" pair which is found in most data parallel languages, such as those in the "for...do...while" construct. Now we can use this mechanism to implement classical parallel control flow operators.

### Application to a data parallel language

The basic operator is the "where/elsewhere" pair which is found in most data parallel languages, such as those in the "for...do...while" construct. Now we can use this mechanism to implement classical parallel control flow operators.

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precondition</th>
<th>Action</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop</td>
<td>0 = (c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1 - c \rightarrow c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0 \neq c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push</td>
<td>(1 = \text{cond})</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1 \rightarrow c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0 = c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(c \rightarrow c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0 \neq c)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

A parallel extension of the C language which is the "where/elsewhere" construct which follows a parallel loop. All parallel condition are false. The continue and break operations.

The basic manipulations on \(c\) are the same as in C: increment or decrement, load or store, as shown on Table 2. The variable exchange \(c \rightarrow 1 - f\) because only a comparison to \(0\) is necessary. Thus, the "where/elsewhere" construct is equivalent to the "push/cond" because a special value the "elsewhere" state for the "where/elsewhere" blocks. The table can be seen here as a form of transition from PICTAX 90 to C.

The push \((\text{cond})\) when \(c = 0\) can be simplified to \(c = \text{cond}\). A more detailed proof of the equivalence between an activity stack and an activity counter for parallel control can be found in [Ker892].

The "where/elsewhere" pair is found in most data parallel languages. The "where" is equivalent to the push operator, but we have to translate the "elsewhere" into the "cond" form. This form is easier to implement in hardware and often even in software. The "push/cond" form is easier to implement in hardware and often even in software. The "cond" form is easier to implement in hardware and often even in software. The "cond" form is easier to implement in hardware and often even in software.
Table 4: Implementation of a `while some where` block.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precondition</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>continue</code></td>
<td><code>0 = c</code></td>
<td></td>
<td>The condition is false; the pe remains idle.</td>
</tr>
<tr>
<td><code>break</code></td>
<td><code>0 = c</code></td>
<td></td>
<td>The condition is false; the pe exits.</td>
</tr>
<tr>
<td><code>while some where</code> begin</td>
<td><code>1 = c</code></td>
<td><code>i = 0</code></td>
<td>The value ( i = 1 ) is used to implement state ( 1 ) (the continue) and the value ( c = 2 ) is used to implement state ( 2 ) (the break) and the value ( c = 0 ) is used to implement state ( 3 ) (the idle).</td>
</tr>
<tr>
<td><code>while some where</code> end</td>
<td></td>
<td></td>
<td>In comparison with the <code>where</code> <code>elsewhere</code> block, only point ( 5 ) requires additional mechanisms. It is implemented by reserving a protected value ( 1 ) in the counter beginning on the first row of the implementation presented on Table 4. The value ( c = 1 ) is used to implement state ( 1 ) (the continue) and the value ( c = 2 ) is used for the states ( 2 ) and ( 4 ) (line 2 and row 4 of Table 4).</td>
</tr>
</tbody>
</table>

In a real implementation on an simd machine, there is a scalar loop around loop beginning and loop exiting on all cond = 0 condition. On an spmd machine, there is a scalar loop around loop beginning. In a real implementation on a simd machine, there is a scalar loop around loop beginning.

Several values to the current `while some where` block, see section 4.2.

1. Active in the `while some where` until the next iteration after executing a `continue`.
2. Active in the `while some where` until the `while some where` exits because of a `break`.
3. Active in the `while some where` until the `while some where` exits because of a `continue`.
4. Active in the `while some where` until the `while some where` exits because of a `break`.
5. Active in the `while some where` until the `while some where` exits because of a `continue`.
6. Active in the `while some where` until the `while some where` exits because of a `break`.
7. Active in the `while some where` until the `while some where` exits because of a `continue`.

States \( 3 \) and \( 4 \) are not different once the break is executed; the pe remains idle. States \( 3 \) and \( 4 \) are not different once the break is executed; the pe remains idle. States \( 3 \) and \( 4 \) are not different once the break is executed; the pe remains idle.
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.

4. A pe is inactive because of a break until the switch closure.

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 2.

4.2.3 Switchwhere

The compilation of a switchwhere allows the following states:

1. A pe is active before the switch closure.
2. A pe is active in a case (after matching a value) or in a default.
3. A pe is inactive in a case, waiting for a matching value.
4. A pe is inactive because of a break until the switch closure.

The break is similar to the whilesomewhere one. An example of switchwhere execution was also presented above. A pe can be:

- active in the switchwhere block.
- inactive in the switchwhere block.
- inactive because of a break.
- inactive because of an inactive.

In the example of Table 5, the case/elsewhere block is executed in one or more

where/elsewhere

Table 2: Implementation of the switchwhere with an activity counter.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precondition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>switchwhere opening</td>
<td></td>
<td></td>
</tr>
<tr>
<td>switchwhere</td>
<td></td>
<td></td>
</tr>
<tr>
<td>case constant (value)</td>
<td>0 = e</td>
<td>e = e</td>
</tr>
<tr>
<td>case elsewhere (value)</td>
<td>0 ≠ e</td>
<td>e = e</td>
</tr>
<tr>
<td>default</td>
<td>1 = e</td>
<td>e = e</td>
</tr>
<tr>
<td>break</td>
<td>e ≠ e</td>
<td></td>
</tr>
<tr>
<td>switchwhere closing</td>
<td>0 = e</td>
<td>e = e</td>
</tr>
</tbody>
</table>

Table 3: Implementation of the switchwhere with an activity counter.
In the first case, it takes a time $T$ on the scalar processor and the time is negligible.

The local stack pointers in the second case, for a VPE computer, plus $\log_2 c$ bits for the global stack pointer in the first case, and $\log_2 c$ bits for each pe. The hardware complexity is $c$ for a stack of $1$ bit elements in each case, on each pe. The hardware complexity is $c$ for a stack of $1$ bit elements in each case, on each pe. In the second case, a time $\log_2 c$ is needed to control the stack pointer.

In the first case, it takes a time $T$ on the scalar processor and the time is negligible.

- distributed with local pointers which evolve synchronously
- controlled on the scalar processor which broadcasts its value to the pe's

control template and the stack pointers can be

are synchronous and the stack pointers can be

need a stack pointer to manage the stack. Since the execution is synchronous, all the stacks

need a stack pointer to manage the stack. Since the execution is synchronous, all the stacks

need an activity counter operation.

The activity stack needs only 1-bit manipulation on each pe and takes a time $T$, but

The counter method needs a counter with $\log_2 c$ bits per pe if at most $c$ levels of parallel

conditional blocks are implemented. If each pe has an L-bit operator, a pe needs $\log_2 c$ bits of parallel

blocks.

In the first case, it takes a time $T$ on the scalar processor and the time is negligible

on the pe's.

In the second case, a time $T$ is needed to control the stack pointer on each pe. The execution is

synchronous and the stack pointers can be

In order to develop choice arguments, we have to analyze the time and space complexity.

5.1. On an SIMD machine

5 Activity counters versus activity stacks

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.

<table>
<thead>
<tr>
<th>Function starting</th>
<th>Action</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c \leftarrow c - 1$</td>
<td>return</td>
<td>$0 \neq c$</td>
</tr>
<tr>
<td>$c \leftarrow c + 1$</td>
<td>return</td>
<td>$c = 0$</td>
</tr>
<tr>
<td>$c \leftarrow c + 1$</td>
<td>function entry</td>
<td>$0 \neq c$</td>
</tr>
</tbody>
</table>

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

For example, of a generic parallel absolute value function in PompC:

```c
{ return x;
    x = x + 1;
}
collection a[100][100] double x (a[0][0]) double x
```

5.1.0
The complexity of the activity counter and activity stack methods are summarized in Table 7.

<table>
<thead>
<tr>
<th></th>
<th>Activity counters</th>
<th>Stack (local pointers)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Par. Computing</td>
<td>Par. Computing</td>
</tr>
<tr>
<td></td>
<td>Complexity</td>
<td>Complexity</td>
</tr>
<tr>
<td></td>
<td>Hardware</td>
<td>Hardw.</td>
</tr>
<tr>
<td></td>
<td>Parallel</td>
<td>Parallel</td>
</tr>
<tr>
<td></td>
<td>Conditioning</td>
<td>Conditioning</td>
</tr>
<tr>
<td>τ</td>
<td>(log c + 1) N</td>
<td>(log c + 1) N</td>
</tr>
<tr>
<td></td>
<td>τ</td>
<td>τ</td>
</tr>
<tr>
<td></td>
<td>τ</td>
<td>τ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>τ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

In the Pomp SIMD computer ([HKMP]91, Ker92), on-the-shelf RISC processors (Motorola 68000) are used. Since they are coarse-grain processors, 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 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 32 nested parallel conditional blocks and a comparator to 0, 1, and 2 to speed up matching.

The control of the instruction execution in the processor is performed by the FPGA which controls the ready signal (CR0: CR1) of the instruction bus of the processor and matches the control of the instruction execution in the processor. The activity counter is implemented with a 32-bit counter allowing up to 32 nested parallel conditional blocks and a comparator to 0, 1, and 2 to speed up matching.

The parallel conditional execution is supported by the 8-bit HyperCom instruction.

The parallel conditional execution is accelerated by the FPGA and a companion circuit (HyperCom in Figure 4) which has other functions that allow the machine to run in an SIMD mode (communications, broadcast, exception control, ...).

In the Pomp SIMD computer, a parallel conditional execution which uses activity counters is used to speed up matching. These instructions are produced by the FPGA which controls the ready signal (CR0: CR1) of the instruction bus of the processor and matches the control of the instruction execution in the processor. The activity counter is implemented with a 32-bit counter allowing up to 32 nested parallel conditional blocks and a comparator to 0, 1, and 2 to speed up matching.

The control of the instruction execution in the processor is performed by the FPGA which controls the ready signal (CR0: CR1) of the instruction bus of the processor and matches the control of the instruction execution in the processor. The activity counter is implemented with a 32-bit counter allowing up to 32 nested parallel conditional blocks and a comparator to 0, 1, and 2 to speed up matching.
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 pces only have 1-bit operators and virtualization is performed in C. (Note that the C-2 is not using activity counters since the pces only have 1-bit operators and virtualization is performed in C.)

A compiler of Pomp for the C-2, the mp-1 and Unix workstations for simulation is implemented on the hypercom. It has to cope with parallelism on the compiler level.

The scalar processor, another mc88100 with a VMS coupling, can access its activity counter for complex parallel control now. The compiler is transformed in a scalar processor.

The imbrication of parallel control operations, 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 activity counters in hardware. For the next collection to be loaded instead.

An activity counter is not very important for the compiler yet. But it is necessary because the scalar processor may access its activity counter for complex parallel control now. The compiler is transformed in a scalar processor.

In Related work section 4, we present an optimization of the activity concept described in section 2.
In the first case the scatter/gather method is better than 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 forward logarithmic term of the size in bits of memory used to keep track of the history. The method is more suited to coarse-grain parallel computers since it avoids an inefficient indirection in memory. It causes a small management, a global address broadcast protocol and replaces the scatter process of the task management. A global address broadcast protocol and replaces the scatter process of the task management.

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 Mathot, Philippe Hoogvorst, Cesare Douady, Patrice Ossona Demendez, Theodore Papadopoulos and Pierre Chicourt.

Special thanks are due to Luc Beugere 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.

Finally, the authors are indebted to Kathryn MacKinley, Francois Irigoin and Pierre Jouvelot for their invaluable comments and their appropriate suggestions.