DEPARTMENT OF MATHEMATICS
TECHNICAL REPORT

ON PARALLELIZING THE CLIFFORD ALGEBRA PRODUCT FOR CLIFFORD

RAFAŁ ABŁAMOWICZ

BERTFRIED FAUSER

April 2012

No. 2012-2

TENNESSEE TECHNOLOGICAL UNIVERSITY
Cookeville, TN 38505

On Parallelizing the Clifford Algebra Product for `CLIFFORD`

Rafał Abłamowicz and Bertfried Fauser

Abstract We present, as a proof of concept, a way to parallelize the Clifford product in $\ell_{,q}$

Recent applications in engineering use real Clifford (geometric) algebras like $\ell_{8,2}$ when modeling geometric transformations in robotics. [13] Thus, there is a need for efficient and fast symbolic computations which not only take advantage of the mathematical theory, for example by using the periodici procedures, namely, the parallel `cmulWpar` against the sequential `cmulW`, `cmul` with `cmulRS`, and `cmul` with `cmulNUM` for some test computations of the most general Clifford polynomials in $\ell_{,q}$ for $+q \leq 9$. Commented code of all Maple worksheets showing these computations including parallelized `cmulNUM` and `cmulRS` is available at [4].

2 Code of `cmulW` and `cmulWpar`

The Clifford product sed on sh functions

First, we present the code of `cmulW` which we use later in the parallel procedure `cmulWpar`. The latter procedure relies on several other procedures, which we do display here for the sake of completeness, and which handle things like producing the Clifford product on basis monomials (`Walsh`) and the data conversion (`convert(<bas>,<data-type1>`) from `CLIFFORD`'s internal data structures for basis monomials and their representations as binary tuple used by the `oplus` and `Walsh` procedures. As `cmulRS` and `cmulNUM` do not have to perform these conversions, there is a slight loss of speed here due to the data conversion. `twist` provides the proper sign factor due to the grading which is easily computed from the binary (Gray code) representation of the Clifford monomials.

Listing 1 Clifford product on basis monomials `eI`, `eJ` using Walsh functions in $\ell_{,q}$

```
cmulW:=proc(eI::clibasmon,eJ::clibasmon,
            B1::{matrix,list(nonnegint)})
local a,b,ab,monab,Bsig,flag,i,dim_V_loc,ploc,qloc,
      _BSIGNATUREloc;
# -- this procedure depends on external variables
global dim_V,_BSIGNATURE,p,q;
if type(B1,list) then
   ploc,qloc:=op(B1);
   dim_V_loc:=ploc+qloc:
   _BSIGNATUREloc:=[ploc,qloc]:
   else
   ploc,qloc:=p,q;   ###<<<-- this reads global p and q
   dim_V_loc:=dim_V: ###<<<-- this reads global dim_V
   _BSIGNATUREloc:=[ploc,qloc]:
   if not _BSIGNATURE=[ploc,qloc] then _BSIGNATURE:=[p,q] end if:
end if:
# -- data structure conversion: string to binary
a,b:=convert(eI,clibasmon_to_binarytuple,dim_V_loc),
     convert(eJ,clibasmon_to_binarytuple,dim_V_loc);
# -- mod 2 binary addition
ab:=oplus(a,b);
# -- data structure conversion: binary to string
monab:=convert(ab,binarytuple_to_clibasmon);
return
 twist(a,b,_BSIGNATUREloc)*Walsh(a,hinversegGrayCode(b))*monab;
```

```
end proc:
```

Maple's threading mechanism for coarse grained parallel computing

The following example is taken from Maple's help page `?Threads:-Task:-Start`.[3] It explains how to split a computation into pieces when the computation is 'large' enough to profit from a parallel execution, and then execute the parallel tasks and use a continuation function to produce the result. The example computes $\sum_{=1}^{10^7}$.

Listing 2 Task threading example

```
continuation := proc( a, b ) # add two results
   return a + b;
end proc;
task := proc( i, j )
   # distributes the computation into tasks
   local k;
   if ( j-i < 1000 ) then
       # if the range is small, just compute
       return add( k, k=i..j );
   else
       # split computation into two parts
       k := floor( (j-i)/2 )+i;
       # produce two child tasks, by calling task recursively
       Threads:-Task:-Continue( continuation,
          Task=[ task, i, k ], Task=[ task, k+1, j ] );
   end if;
end proc;
# compute sum 1..10^7 parallel and using add
Threads:-Task:-Start(task,1,10^7) = add(i,i=1..10^7);
```

The parallelism is coarse-grained, the user does not have to deal with threads, and, for a large part, with locks. However, the involved routines have to be programmed in a thread-safe fashion.[4] Since we want to demonstrate how to parallelize the Clws

Ip froduee

The parallel procedure `cmulWpar` *for the Clifford product*

We discuss briefly the code of `cmulWpar`

```
  # -- set up multitasking
  # -- continue function, add up results of task processes
  addUp:=proc(a,b) a+b end proc
```

both lists are ‘small’ and are actually computed in their respective threads. Finally, the `Threads:-Task:-Start(...)` routine initializes the threading mechanism and starts producing the `task` in separate threads and also collects the results.

The number of tasks produced is also the number of threads Maple produces. On a 4-core cpu one would like to have 4 threads only, all takin

Table 1 Benchmarking CPU times: $_1$ **of** `cmulWpar`**;** $_2$ **of** `cmulW`**,** $_3$ **of** `cmulRS` **and** $_4$ **of** `cmulNUM`

dim V	$_1$

We suspect that `CLIFFORD` could be faster at least by an overall factor of more than 20-30, based on this current experience, by a generic rewrite using better data structures and avoiding all the repetitious parsing and type checking where it can be avoided, and using the recursive way to split (multi)linearity, etc. Optimizing `CLIFFORD` and its related packages like `Bigebra`, `Cliplus`, `Octonion`, etc. [3] is a priority whose urgency has been emphasized by this exercise in parallelizing the Clifford product.

The results discussed here are accompanied by Maple worksheets posted on [4]. These well-documented worksheets contain further results and alternatives like using the inherently parallel procedures `Add`, `Seq`, `Map` of Maple or producing threads directly. There we further discuss the efficient usage of Maple's `Threads` package. We are working to make all of `CLIFFORD` thread safe after we have succeeded parallelizing the more complex and complicated `cmulRS` and `cmulNUM` routines. While `cmulRS` is based on a provable optimal algorithm, the above discussion still sheds some light on efficiency of the implementations due to different data structures or recursive computing models (saving memory usage). In that respect, this is a very open area of research.

Acknowledgements Bertfried Fauser wants to thank Darin Ohashi from Maplesoft for his kind help with and email discussions about Maple's threading mechanism. Both authors thank referees for their comments as they have helped us extend this work and

```
   # local i,j;   # <== needed!
   # assignment of j produces a warning if not declared local
   j:=x[1];
   add(x[i],i=1..N);
end proc
```

6. ______: On the transposition anti-involution in real Clifford algebras III: the automorphism