The name of this worksheet is threads-create-cmultask-M16-RA.mws and it is posted at http://math.tntech.edu/rafal/MapleWorksheets4.html

This worksheet accompanies paper "On Parallelizing the Clifford Algebra Product for CLIFFORD" by R. Ablamowicz and B. Fauser prepared for AGACSE 2012, July 2-4, La Rochelle, France

It has been executed in Maple 16 running CLIFFORD/Bigebra package ver 16.1 on core i7-2640QM at 2.8-3.5 GHz and 8 GB RAM running ubuntu 11.10 Linux and it has a physical dual core with 4 hyper threading virtual cores seen by linux.

This worksheet illustrates that new versions of cmuNUM and cmuRS called here cmulNUM2 and cmulRS2 are thread-safe and can be parallelized.

Recall by typing ?cmulNUM and ?cmulRS that cmulNUM and cmulRS are general procedures to compute Clifford product of any two Grassmann basis monomials, which are used by default to span the Clifford algebra Cl(B) of an arbitrary bilinear form B -- to be precise, of the pair (V,B)-- in the Clifford algebra Cl(B). cmulNUM is based on recursive Chevalley's approach while cmulRS is based on Rota-Stein's Hopf algebra approach to the Clifford product.

```maple
restart: with(Clifford):

kernelopts(numcpus) ;###<<<-- this many cpus are recognized by Maple

version();
```

Procedure decomp takes a Clifford polynomial, that is, the most general element X in a Clifford algebra Cl(B) and returns a list consisting of two-element lists [coeff,monom] where coeff is a coefficient of the monomial monom in X. Thus, such list for a symbolic element X in Cl(B) contains 2^dim_V elements where dim_V is the dimension of the vector space (V,B).

We were testing three versions of that procedure with version decomp3 being the fastest. This version is then used below.

```maple
# Procedure decomp takes a Clifford polynomial, that is, the most general element X in a Clifford algebra Cl(B) and returns a list consisting of two-element lists [coeff,monom] where coeff is a coefficient of the monomial monom in X. Thus, such list for a symbolic element X in Cl(B) contains 2^dim_V elements where dim_V is the dimension of the vector space (V,B).

# We were testing three versions of that procedure with version decomp3 being the fastest. This version is then used below.
```
decomp1:=proc(p::{clibasmon,climon,clipolynom})
    local mons:
    if type(p,clibasmon) then
        # done with one term
        return [[1],[p]]
    else
        return
        [coeffs(p,convert(Clifford:-cliterms(p),list),'mons')],[mons];
    end if:
end proc:
#
#  version 2
#
decomp2:=proc(p::{clibasmon,climon,clipolynom})
    local mons,L,cofs,m:
    if type(p,clibasmon) then
        return [1],[p]
    else
        mons:=convert(Clifford:-cliterms(p),list);
        #mons:=sort(mons,bygrade);
        cofs:=[seq(coeff(p,m),m=mons)];
        return cofs,mons;
    end if:
end proc:
#
# decomp3 : caution different output format : List List
[coeff,monom]
#  NOTE: (see benchmark below)
#  -- decomp3 is much more memory efficient
#  -- decomp3 is by fare the fastest decomp routine
#  time   (factor 4 to 5)
#  memory (factor 5 to 10)
#
decomp3:=proc(p)
    local cf,term;
    if p=0 then return [0,Id] end if;
    if type(p,`+`) then
        return map(op@procname,[op(p)]);
    elif type(p,`*`) then
        term,cf:=selectremove(type,p,clibasmon);
        return [[[cf,term]];
    else
        [[1,p]];
    end if:
end proc:
Here we provide a short benchmark where we compare total times and total amount of memory needed by decomp1, decomp2, decomp3 to decompose the same 1,000 randomly created polynomials in dimension not exceeding 9. We use Maple's 'profile' function to show the times and the memory used.

```maple
unprofile(decomp1,decomp2,decomp3):
profile(decomp1,decomp2,decomp3);
for i from 1 to 1000 do
    p1:=0:
    while p1=0 do p1:=rd_clipolynom(9); end do; # -- make sure p1 is not 0
    out1:=decomp1(p1);
    out2:=decomp2(p1);
    out3:=decomp3(p1);
end do:
showprofile();
```

The above shows that decomp3 takes the least time and the least amount of memory.

A: MULTITASKING

Example 1: I am trying to use now Tasks in the module Threads with cmulNUM2, cmulRS2, and cmultask2.

My example, is based on Maple example below:

```maple
cont := proc( a, b )
    return a + b;
end proc;
task := proc( i, j )
    local k;
    if ( j-i < 1000 ) then
        return add( k, k=i..j );
    else
        k := floor( (j-i)/2 )+i;
        Threads:-Task:-Continue( cont, Task=[ task, i, k ], Task=[
```
We re-initialize first the default bilinear form $B$. Recall that when the form $B$ is undefined, CLIFFORD assumes symbolic entries in $B$.

Let's us define two random Clifford polynomials in dimension 9 -- this means that the largest possible monomial grade is 9 and that the largest possible index which may appear in a monomial is also 9. We are also using option 'unordered' which means that some of the monomials in the polynomial may have unsorted indices.

```plaintext
> B:='B': ###<<<--- we reinitialize B

> p1:=rd_clipolynom(9, 'unordered'); ###<<<--- we could define a random p1 with possibly unordered indices
p1:=-4*e1+e9+Id+e3+3*e7;       ###<<<--- for illustration purposes we take this simpler polynomial

> p2:=rd_clipolynom(5, 'unordered'); ###<<<--- we define a random p2 with possibly unordered indices
p2:=Id+e3+e5+e+e4+e5+e3;      ###<<<--- for illustration purposes we take this simpler polynomial

> ###This is my original decomp, it is replaced with Bertfried's decomp3 later
###
> decomp:=proc(p) local mons,L,cofs,m:
> if type(p, clibasmon) then return [1],[p] end if:
> else
>     #L:=[coeffs(p, convert(Clifford:-cliterms(p), list), 'mons')];
>     #return L, [mons];
>     mons:=convert(Clifford:-cliterms(p), list);
>     mons:=sort(mons, bygrade);
>     cofs:=[seq(coeff(p,m),m=mons)];
>     return cofs,mons;
> end if:
> end proc:

> decomp3:=proc(p)
>     local cf,term;
>     if p=0 then return [0,Id] end if;
>     if type(p, `+`) then
>         return map(op@procname, [op(p)]);
>     elif type(p, `*`) then
>         return
>     end if:
> end proc:
```
term, cf := selectremove(type, p, clibasmon);
return [[[cf, term]]];
else
[[1, p]];
end if;
end proc:

Observe the output from the decomp procedures:

decomp(p1);
decomp3(p1);

decomp(p2);
decomp3(p2);

Below we will eventually use decomp3 as it is faster.

We do first pre-computing regardless which internal procedure we will use: cmulNUM2 or cmulRS2 in the mutlitasking procedure cmultask2:

cofs1, mons1 := decomp(p1);
p11 := [seq([cofs1[i], mons1[i]], i = 1..nops(cofs1))];
p11 := decomp3(p1);  ###<--- Bertfried's decomp3 already does what's needed and it replaces previous two lines
N1 := nops(p11);
cofs2, mons2 := decomp(p2);
p22 := [seq([cofs2[j], mons2[j]], j = 1..nops(cofs2))];
p22 := decomp3(p2);  ###<--- Bertfried's decomp3 already does what's needed and it replaces previous two lines
N2 := nops(p22);
N := N1*N2;
Nhalf := floor(N/2);

The list LL contains smaller lists with two elements each like
[[x1, mon1], [x2, mon2]] where x1 is a coefficient of a monomial mon1 in p1 and x2 is a coefficient of a monomial mon2 in p2.
Then, cmultask2 will take the whole list LL and compute the products x1*x2*KK(mon1, mon2, BB) where
KK will be assigned either cmulNUM2 or cmulRS2, and BB will be assigned a bilinear form B.
\[ LL := \{\text{seq(seq([p11[i], p22[j]], i=1..N1), j=1..N2)}\}; \]

\[ \text{nops}(LL); \] number of elements in the list \( LL \)

\[ p1; \]

\[ p2; \]

Note that the list \( LL \) contains pairs of the type \([\text{coeff1, monom1}], [\text{coeff2, monom2}]\) where monom1 belongs to \( p1 \) and monom2 belongs to \( p2 \). Then, coeff1 and coeff2 are the coefficients of monom1 and monom2, respectively. Thus,

the length of \( LL \) = number of terms in \( p1 \) x number of terms in \( p2 \).

In the above example we have 20 terms.

Note: Essentially, \( LL \) is a Cartesian product of two lists \( \text{decomp3}(p1) \) and \( \text{decomp3}(p2) \).

First we use \text{cmulNUM2} to multitask. A thread-safe version called \text{cmulNUM2} of \text{cmulNUM} is defined this way:

\[ \text{cmulNUM2} := \text{proc}(a1, a2, lname) \]

local \( L, N, L2, x, x1, x2, S, i, ii, T1, T2, K, p1, p2, \text{coB}, \text{nameB}, a12; \text{global} \)

\( B, \text{Id}, e, w; \)

options `\text{Copyright (c) 1995-2012 by Rafal Ablamowicz and Bertfried Fauser. All rights reserved.}`;

description `\text{Last revised: March 24, 2012}`;

###This is additional code for Maple 6 version:

###if hastype({a1, a2}, clirod) then
## a12:=map(Cliplus:-clieval, [a1, a2]);
### return

\( \text{Cl}iplus:-\text{clixpand}(\text{clibilinear(a12[1], a12[2], proname, lname})) \)

end if:

### old name cmul2B: this procedure computes recursively Clifford product of any two #
### cliscalars, clibasmons, climons, and clipolynoms in Clifford algebras \( C1(lname) \) #

if nargs<>3 then error "exactly three arguments are needed" end if:
if has(0,map(simplify,[a1,a2])) then return 0 end if;
if a2=`Id` then return a1 end if:
if a1=`Id` then return a2 end if:
L:=Clifford:-extract(a1,'integers');
N:=nops(L):

### The following will allow for lname to be \( -B \), for example:
if type(lname,{name,symbol,array,matrix}) then
  coB,nameB:=1,lname:
elif type(lname,`&*`(numeric,{name,symbol,array,matrix})) then
  coB:=op(select(type,{op(lname)},numeric));
  nameB:=op(select(type,{op(lname)},name));
else
  error "third argument is of unexpected type"
end if;

###
if N=0 then return coeff(a1,Id)*a2 end if;
if N=1 then
  L2:=Clifford:-extract(a2,'integers');
  return
endif;

###
if N=2 then
  x1:=substring(a1,1..2):x2:=substring(a1,4..5);
  p2:=procname(x2,a2,lname):
  S:=clibilinear(x1,p2,procname,lname);
  return simplify(S-coB*nameB[op(L)]*a2)
end if;

x:=cat(e,L[-1]);
p1:=substring(a1,1..(3*N-4));
p2:=procname(x,a2,lname):
S:=clibilinear(p1,p2,procname,lname)-add((-1)^(i-1)*coB*nameB[L[-i], L[-1]]*procname(makeclibasmon(subs(L[-i]=NULL,L[1..-2])),a2,lname),i=2..N); return S;
return reorder(simplify(S)); ### no reorder
end proc:
Note that for now KK and BB are passed on to cmultask2 as global variables:

\[
\text{KK:='cmulNUM2'; \quad \text{###<<<--- we will use cmulNUM2 to multitask in cmultask2}}
\]
\[
\text{BB:='B'; \quad \text{###<<<--- making sure that B is blank}}
\]
\[
\text{L:=LL; \quad \text{###<<<--- assigning to L the precomputed list LL with pairs of monomials and their coefficients}}
\]
\[
\text{nops(LL);}
\]
\[
\text{_default_Clifford_product;}
\]
\[
\text{useproduct(cmulNUM); \quad \text{###<<<--- make sure cmul will also use current cmulNUM}}
\]
\[
\text{_default_Clifford_product;}
\]
\[
\text{_max_times:=10; \quad \text{###<<<--- number of times we recompute the same result}}
\]

Note: We probably should define a new global variable, say _task_length which would control the if-then-else statement in cmultask2:

Note: We should test what value is best for the given input --- maybe as some function of N = nops(LL) --- and what that best value depends on?

\[
\text{_task_length:=3; \quad \text{###<<<--- a new environmental variable whose default value should be loaded by Clifford}}
\]
\[
\text{KK:='cmulNUM'; \quad \text{###<<<--- NOTE: cmulNUM now works once reorder has been fixed!}}
\]

Note: The original procedure 'reorder' used to reorder Grassmann monomials had a line

\[
f:=\text{proc()} \text{end proc; \quad \text{###<<<--- not needed, messes up multitasking}}
\]

which was causing problems when multitasking. Once this line was removed as unnecessary, the code of 'reorder' is now thread-safe and could be used in cmulNUM. That is, cmulNUM was thread-safe yet it was using 'reorder' which was not. That's why cmulNUM initially at the time of writing 10-page version of this paper could not be parallelized.

Here is the current thread-safe code of reorder:

\[
\text{reorder:=proc(a1::algebraic) local L1,L2,N,newbas,f,a,x,K,dummy_set,n12,s12,ss; global B,dim_V;}
\]
\[
\text{options `Copyright (c) 1995-2012 by Rafal Ablamowicz and Bertfried Fauser. All rights reserved.`;}
\]
\[
\text{description `Last revised: March 24, 2012`;}
\]
if type(a1,{matrix,`+`,`*`}) then return map(simplify@procname,a1) end if; 

L1:=Clifford:-extract(a1);
N:=nops(L1); 

if N>9 then error "detected basis monomial of grade higher than 9 in the input" end if;
if N>9 then WARNING("detected basis monomial of grade higher than 9 in the input") end if;

if N=0 or N=1 then return a1 end if;
n12,s12:=selectremove(member,L1,{`1`,`2`,`3`,`4`,`5`,`6`,`7`,`8`,`9`});
#s12:=remove(member,L1,{`1`,`2`,`3`,`4`,`5`,`6`,`7`,`8`,`9`});
L2:=[op(sort(n12)),op(sort(s12))];

for ss from 1 to N do
  f(L2[ss]):=L1[ss];
end do:
dummy_set:=convert(L1,set):
K:=0:
while dummy_set <> {} do
  a:=dummy_set[1]:
dummy_set:=dummy_set[2..-1];
x:=a:
  while f(x)<>a do
    x:=f(x);
dummy_set:=dummy_set minus {x};
    K:=K+1;
  end do:
end do:
newbas:=cat(e||(op(L2[1..-2]))||w,e,L2[-1]):
return (-1)^K*newbas
end proc:

Procedure 'cmultask2' parallel's Maple's procedure 'task' shown above and it computes the Clifford product of two random Clifford polynomials using cmulNUM2 passed on as a global variable KK
Recall, that the list LL contains pairs of the type \([\text{coeff1},\text{monom1}],[\text{coeff2},\text{monom2}]\) where monom1 is a monomial from p1 and monom2 is a monomial from p2. Thus, in order to compute the Clifford product of p1 and p2, one needs to compute the following sum:

\[
\text{cmul}(p1,p2) = \text{Sum } \text{coeff1} \times \text{coeff2} \times \text{KK}(\text{monom1},\text{monom2},\text{BB})
\]

where the sum is over all elements of the list LL. That list may have, in dimension 9, up to \(2^9 \times 2^9 = 262,144\) terms.

Arguments: ii and jj each of type positive integer;
Local variable: k
Global variables:

-- BB -- assigned the bilinear form B as in Cl(V,B)
-- KK -- the name of the procedure, e.g., cmulNUM2 or cmulRS2 to be used to compute the product of Grassmann monomials
-- L -- list L is assigned the Cartesian product list LL precomputed above for p1 and p2
-- _task_length -- the maximum number of elements in a sublist subL of the list L = LL such that a subsum with that many terms each being the Clifford product of two basis monomials is computed directly, i.e., sequentially without multitasking by using the procedure assigned to KK and using the most general bilinear form B. That procedure will either be cmulNUM2 or cmulRS2

\[
\text{cmultask2} := \text{proc}(\text{ii}::\text{posint}, \text{jj}::\text{posint}) \text{ local } \text{k}; \text{global BB, KK, L, _task_length;}
\]

\[
\text{if } (\text{jj}-\text{ii} < _\text{task_length}) \text{ then}
\text{ return add(L[\text{k}][1][1] \times L[\text{k}][2][1] \times \text{KK}(L[\text{k}][1][2], L[\text{k}][2][2], \text{BB}), \text{k}=\text{ii}..\text{jj})}
\text{ else}
\text{ k := floor( (\text{jj}-\text{ii})/2 ) + \text{ii};}
\text{ Threads:-Task:-Continue( cont, Task=[ \text{cmultask2}, \text{ii}, \text{k}], Task=[ \text{cmultask2}, \text{k+1, jj}] );}
\text{ end if:}
\text{ end proc:}
\]

Now, we execute the following do loop _max_times unless there is an error and out1 computed with \text{cmul}[B](p1,p2) using _default_Clifford_product (here cmulNUM) is different from out3 computed with cmultask2 using KK=cmulNUM2:

In the next display, the list times1 will contain computation times using cmul[B](p1,p2) whereas a list times2 will containing times obtained using multitasking cmultask2 with cmulNUM2.
B:= 'B': ###<--- erase B
_prolevel:=true:

forget(Clifford): ##<--- erasing remember tables in Clifford
printlevel:=1:
flag:=true:
times1:=[[]:
times2:=[[]:
flags:=[[]:
maxtimes:=_max_times:
for i from 1 to maxtimes while flag do
  start:=time():
  printlevel:=0:
  out1:=cmul[B](p1,p2):
  out1:=reorder(out1);
  printlevel:=1:
  times1:=[op(times1),time()-start];
  start:=time():
  printlevel:=0:
  out3:=Threads:-Task:-Start( cmultask2, 1, N):
  out3:=reorder(out3);
  printlevel:=1:
  'times1'=times1;
  times2:=[op(times2),time()-start];
  flag:=evalb(simplify(out3-out1)=0);
  flags:=[op(flags),flag];
end do;
printlevel:=1:
print(`####################`):
'times1'=times1;
'times2'=times2;
ratios12:=seq(evalf(times1[i]/times2[i],3),i=1..nops(times1)):
'flags'=flags;
print(`####################`);

We display the product out1 of p1 and p2 computed using ordinary sequential cmulNUM for any B with out2 holding the product of p1 and p2 computed using multitasking to make sure results are the same:

'out1'=out1;###<--computed using cmul with cmulNUM for any B
'out3'=out3;###<--computed using cmultask2 with cmulNUM2 for any B
simplify(out1-out3); ###<--- results are the same provided this
difference is 0

> p1; ###<<<--- original polynomial p1
> p2; ###<<<--- original polynomial p2
> p12:=out1;
> ratios12:=[seq(evalf(times1[i]/times2[i],3),i=1..nops(times1))];

> average_ratio12:=evalf(add(ratios12[i],i=1..nops(ratios12))/nops(ratios12),3);

Second, we use thread-safe cmulRS2 to multitask in cmultask2:

As it has turned out, in the original code of cmulRD there were definitions of two internal functions called fun1 and fun2 which were not thread-safe:

#### prepare combinatorics for L1:
fun1:=proc(a1) a1 end proc:
for i from 1 to N1 do
  fun1(i):=L1[i];
end do:

#### prepare combinatorics for L2:
# fun2:=proc(a2) a2 end proc:
# for i from 1 to N2 do
#  fun2(i):=L2[i];
# end do:

One they got replaced with dynamically-defined functions

fun1:=i->L1[i];

and

fun2:=i->L2[i];

respectively, the new code of cmulRS2 became thread-safe.

> cmulRS2:=proc(a1,a2,1name)
  local
  max_grade,L1,N1,L2,N2,genPS,fun1,fun2,srt,cup,pList1,PN1,pList2,PN2,pSgn1,pSgn2,a,b,i,j,m,n,res,pos1,pos2,F1,F2,coB,nameB,a12;
  options `Copyright (c) 1995–2012 by Rafal Ablamowicz and Bertfried Fauser. All rights reserved.`;
  description `Last revised: March 24, 2012`;
### This is additional code for Maple 6 version:

```maple
if hastype({a1,a2},cliprod) then
    a12:=map(Cliplus:-clieval,[a1,a2]);
    return Cliplus:-cliexpand(clibilinear(a12[1],a12[2],procname,lname))
end if:
```

### This procedure computes Clifford product of any two cliscalars, clibasmons, climons, 
### and clipolynoms in Clifford algebras Cl(lname) using Rota-Sten
### Procedure cmulRS modified by Rafal to accept -K, or -B for lname.                    
### The following will allow for lname to be -B, for example:
```maple
if nargs<>3 then error "exactly three arguments are needed" end if:
    if has(0,map(simplify,[a1,a2])) then return 0 end if;
    if a1 = `Id` then return a2 end if;
    if a2 = `Id` then return a1 end if;
    L1:=Clifford:-extract(a1,'integers');
    N1:=nops(L1);
    L2:=Clifford:-extract(a2,'integers');
    N2:=nops(L2);
    if N1=1 then
        return
        reorder(simplify(makeclibasmon([L1[1],op(L2)])+add((-1)^(i-1)*coB*
            nameB[L1[1],L2[i]]*makeclibasmon(subs(L2[i]=NULL,L2)),i=1..N2)))
    return
end if;
```
simplify(makeclibasmon([L1[1], op(L2)]) + add((-1)^(i-1)*coB*nameB[L1 [1], L2[i]]*makeclibasmon(subs(L2[i]=NULL, L2)), i=1..N2))

end if;

if N2=1 then
#return
reorder(simplify(makeclibasmon([op(L1), L2[1]]) + add((-1)^(i-1)*coB*
nameB[L1[-i], L2[1]]*makeclibasmon(subs(L1[-i]=NULL, L1)), i=1..N1)))

return
simplify(makeclibasmon([op(L1), L2[1]]) + add((-1)^(i-1)*coB*nameB[L1 
[-i], L2[1]]*makeclibasmon(subs(L1[-i]=NULL, L1)), i=1..N1))

end if;

#################################################################
####Local variables are:
max_grade, L1, N1, L2, N2, genPS, fun1, fun2, srt, cup, pList1, PN1, pList2, PN 2, pSgn1, pSgn2, a, b, i, j, m, n, res, pos1, pos2, F1, F2, coB, nameB, a12
#################################################################
#### genPS ; generate a power set of 1..N; option remember

   genPS:=proc(N) local a, i, plst;
       option remember;
       a:=[seq(i, i=1..N)];
       plst:=[a]:
       for i in a do
          plst:=[op(subs(i=NULL, plst)), op(plst)]:
       end do:
       end proc:

   #################################################################
#### prepare combinatorics for L1:
# fun1:=proc(a1) a1 end proc:
# for i from 1 to N1 do
#    fun1(i):=L1[i];
# end do:
fun1:=i->L1[i];

#### here is the old code for the poweset
# a:=[seq(i, i=1..N1)];
# pList1:=[a];
# for i in a do
#     pList1 := [op(subs(i = NULL, pList1)), op(pList1)]:
# end do:
###
pList1:=genPS(N1);
   PN1:=nops(pList1)+1;  ## added 1 here
   pList1:=sort(pList1,(a,b)->evalb(nops(a)<=nops(b)));
   pSgn1 := [seq((-1)^(add(pList1[i][m]-m,m=1..nops(pList1[i]))),i=1..PN1-1)];

#### prepare combinatorics for L2:
#   fun2:=proc(a2) a2 end proc:
#   for i from 1 to N2 do
#     fun2(i):=L2[i];
#   end do:
#   fun2:=i->L2[i];

### here is the old code for the poweset
#   a:=[seq(i,i=1..N2)]:
#   pList2:=[a]:
#   for i in a do
#     pList2 := [op(subs(i = NULL,pList2)), op(pList2)]:
#   end do:

###
### Local variables are:
max_grade,L1,N1,L2,N2,genPS,fun1,fun2,srt,cup,pList1,PN1,pList2,PN2,pSgn1,pSgn2,a,b,i,j,m,n,res,pos1,pos2,F1,F2,coB,nameB,a12
###
###
pList2:=genPS(N2);
   PN2:=nops(pList2)+1;  ## added 1 here
   pList2:=sort(pList2,(a,b)->evalb(nops(a)<=nops(b)));
   pSgn2:= [seq((-1)^(add(pList2[i][m]-m,m=1..nops(pList2[i]))),i=1..PN2-1)];

### cup tangle of the rota-stein sausage tangle
   cup:=proc(lst1,lst2,coB1,nameB1) local i;
   if nops(lst1)<>nops(lst2) then return 0 end if;
   if lst1=[] then return 1 end if;
   if nops(lst1)=1 then return coB1*nameB1[lst1[1],lst2[1]] end if;
   add((-1)^(i-1)*coB1*nameB1[lst1[-1],lst2[i]]*cup(lst1[1..-2],subs(lst2[i]=NULL,lst2),coB1,nameB1),i=1..nops(lst2))
end proc:

###############################
## Rota-Stein Tangle : cliffordization
#
## compose only such terms which are potentially non zero in the
cup(...) tangle#
###############################
max_grade:=nops({op(L1),op(L2)});  ## <= new code
res:=0:
pos1:=0:
for j from 0 to N1 do  # for all j-vectors of pList1
  F1:=N1!/((N1-j)!*j!);
pos2:=0:
  for i from 0 to min(N2,max_grade-j) do  # for all i-vectors of
    pList2
      # which do not exceed
    max_grade (others are zero)
      F2:=N2!/((N2-i)!*i!);
        for n from 1 to F1 do
          for m from 1 to F2 do

            res:=res+pSgn1[pos1+n]*pSgn2[pos2+m]*cup(map(fun1,pList1[PN1-pos1-
            n]),map(fun2,pList2[pos2+m]),coB,nameB)*

            makeclibasmon([op(map(fun1,pList1[pos1+n])),op(map(fun2,pList2[PN2-
              pos2-m]))])

          end do:
        end do:
pos2:=pos2+F2;
      end do:
pos1:=pos1+F1;
    end do:
return res; ###<<<<< no reorder in cmulRS
return reorder(res); ## note that cmulRS INCLUDES already reorder
!!
end proc:

We repeat the above do-loop but this time cmul[B](p1,p2) will use cmulRS while cmultask2 will use cmulRS2:
> KK:='cmulRS2';  ### this time we will use cmulRS2 to multitask in cmultask2
BB:='B';  ### making sure that B is blank
L:=LL;  ### assigning to L precomputed LL with pairs of monomials and their coefficients
nops(LL);
_def_Clifford_product;
useproduct(cmulRS);  ### making sure that cmul will use cmulRS
_def_Clifford_product;

> KK:='cmulRS';  ### we can use cmulRS because reorder has been fixed
'_task_length'=_task_length;  ### show previously assigned value

> cmultask2:=proc(ii::posint,jj::posint) local k;global BB,KK,L, _task_length:
if (jj-ii < _task_length) then
  return
  add(L[k][1][1]*L[k][2][1]*KK(L[k][1][2],L[k][2][2],BB),k=ii..jj)
else
  k := floor( (jj-ii)/2 )+ii;
  Threads:-Task:-Continue( cont, Task=[ cmultask2, ii, k], Task=[ cmultask2, k+1,jj ] );
end if:
end proc:

> B:='B':
_prolevel:=true:

> forget(Clifford):  ### erasing remember tables in Clifford
printlevel:=1:
flag:=true:
times11:=[]:
times22:=[]:
flags:=[]:
maxtimes:=_max_times:  ### we compute as many times as when using cmulNUM2
for i from 1 to maxtimes while flag do
start:=time():
printlevel:=0:
out1:=cmul[B](p1,p2):
out1:=reorder(out1);
printlevel:=1:
times11:=[op(times11),time()-start];
start:=time():
printlevel:=0:
out3:=Threads:-Task:-Start( cmultask2, 1, N):
out3:=reorder(out3):
printlevel:=1:
'times11'=times11;
times22:=[op(times22),time()-start];
flag:=evalb(simplify(out3-out1)=0);
flags:=[op(flags),flag];
end do:
printlevel:=1:
print(`####################`);
'times11'=times11;
'times22'=times22;
ratios1122:=[seq(evalf(times11[i]/times22[i],3),i=1..nops(times11))];
'flags'=flags;
print(`####################`);

> 'out1'=out1;##<<--computed using cmul with cmulNUM for any B
> 'out3'=out3;##<<--computed using cmultask2 with cmulNUM2 for any B
> simplify(out1-out3);
> p1; ###<<<--- original polynomial p1
> p2; ###<<<--- original polynomial p2
> p12:=out1;
> ratios1122:=[seq(evalf(times11[i]/times22[i],3),i=1..nops(times11))];
> average_ratio1122:=evalf(add(ratios1122[i],i=1..nops(ratios1122))/nops(ratios1122),3);

Comparing times from:

times1 = times of cmul with cmulNUM using any B
times2 = times of cmultask2 with cmulNUM2 using any B
ratios12 = ratios of times1/times2
times11 = times of cmul with cmulRS using any B
times22 = times of cmultask2 with cmulRS2 using any B
ratios1122 = times11/times22

> print(`times of cmul with cmulNUM using any B:`);
> 'times1'=times1;
> print(`times1 deterioration (plus) or improvement (minus) in percent:`);
> evalf((times1[-1]-times1[1])/times1[1]*100,3);
In this part we experiment with

```
id1:=Threads:-Create(cmultask(lst,1,Nhalf,'K1'),'out1');
```
We use this cmultask for now:

\[
\text{cmultask} := \text{proc}(L::\text{evaln}, i::\text{posint}, j::\text{posint}, \text{KK}::\text{name})
\]
\[
\text{local } k; \quad \text{global } B;
\]
\[
\text{return } \add(L[k][1][1]*L[k][2][1]*\text{KK}(L[k][1][2], L[k][2][2], B), k=i..j);
\]

The setup is similar as in Part A above:

\[
\text{# test1}
\]
\[
\text{# -- setup}
\]
\[
\text{B} := \text{\'B\';}
\]
\[
\text{dim}_V := 9;
\]
\[
p1 := \text{rd_clipolynom}(6);
\]
\[
\text{# -- setup the data list (using decomp)}
\]
\[
\text{cofs1, mons1 := decomp}(p1);
\]
\[
\text{# -- restructure data into form List List [cf, monom]} \text{ (can be directly done by decomp3)}
\]
\[
p11 := [\text{seq}([\text{cofs1}[i], \text{mons1}[i]], i=1..\text{nops}(\text{cofs1}))];
\]
\[
p33 := \text{decomp3}(p1);
\]
\[
is(\text{convert}(p11, \text{set}) = \text{convert}(p33, \text{set}));
\]
\[
N1 := \text{nops}(p11);
\]
\[
\text{# -- setup second polynomial}
\]
\[
p2 := \text{rd_clipolynom}(6);
\]
\[
\text{cofs2, mons2 := decomp}(p2);
\]
\[
p22 := [\text{seq}([\text{cofs2}[i], \text{mons2}[i]], i=1..\text{nops}(\text{cofs2}))];
\]
\[
p33 := \text{decomp3}(p2);
\]
\[
is(\text{convert}(p22, \text{set}) = \text{convert}(p33, \text{set}));
\]
N2 := nops(p22);
# -- size of the product set, and halve size
N := N1*N2;
Nhalf := floor(N/2);
T := combinat:-cartprod([p11,p22]):
L := []:
while not T[finished] do L := [op(L), T[nextvalue]()] end do;
lst := Array(L);
# -- finally we have the set of product prestructures for compuation

Example 2: Checking out1:

> K := 'cmulNUM':
out1 := add(L[k][1][1]*L[k][2][1]*K(L[k][1][2],L[k][2][2],B), k = 1..nops(L));
_weekarnings_flag := true: #BF. -- note that _warning_flags does not turn off the warning ??
# -- current value of the global variable
useproduct(cmulNUM);
# -- selecting 'cmulNUM'
out2 := cmul(p1,p2); ###<<<--this uses current

Example 3: Checking that cmultask works well:

> out3 := cmultask(L, 1, nops(L), 'cmulRS');
simplify(out3-out1);

Example 4: Moving to use multithreading which will use thread-safe cmulNUM2 and cmulRS2:

> K1 := 'cmulRS2': ###<<<---using cmulRS2
K2 := 'cmulNUM2': ###<<<---using cmulNUM2
K1 := 'cmulRS': ###<<<--- we can use original cmulRS because we have fixed reorder to be thread-safe
K2 := 'cmulNUM': ###<<<--- we can use original cmulNUM because we have fixed reorder to be thread-safe

> id1 := Threads:-Create(cmultask(lst, 1, Nhalf, 'K1'), 'outid1');
id2 := Threads:-Create(cmultask(lst, Nhalf+1, N, 'K2'), 'outid2');
Threads:-Wait(id1, id2);
out5 := clicollect(outid1 + outid2); ##<<--combining results from two threads
out5 := reorder(out5);
simplify(out5-out1);

> id1 := Threads:-Create(cmultask(1st, 1, Nhalf, 'K2'), 'outid1');
id2 := Threads:-Create(cmultask(1st, Nhalf+1, N, 'K2'), 'outid2');
Threads:-Wait(id1, id2);
out6 := clicollect(outid1 + outid2); ##<<--combining results from two threads
out6 := reorder(out6);
simplify(out6-out1);

Example 5: Combining all of the above into a single multithreading procedure:

Note that cmulMT is called this way:

cmulMT[B](p1, p2, KK)

It uses for now Bertfried's decomp3 as external procedure defined above.

>_task_length := 3; ###<<-- our new environmental variable used by cmulMT

> cmulMT := proc(p1::{clibasmon, climon, clipolynom}, p2::{clibasmon, climon, clipolynom}, KK::name)
local BB, decomp, cmultask22, cofs1, cofs2, mons1, mons2, p11, p22, i, j, N1, N2, N, N half, T, LL, lst, id1, id2, outid1, outid2, ii, jj, out:
global decomp3, _task_length;
BB := op(1, procname);
### note that decomp3 above could be used as external routine here ###
decomp := proc(p) local mons, L, cofs, m:
  if type(p, clibasmon) then
    return [1], [p]
  else
    mons := convert(Clifford:-cliterms(p), list);
    cofs := [seq(coeff(p, m), m = mons)];
    return cofs, mons;
  end if:
end proc:

#---------------------------#-------------#--------------------------#
cmultask22 := proc(L::evaln, ii::posint, jj::posint, KK::name, BB::{name}
local k:
  return
add(L[k][1][1]*L[k][2][1]*KK(L[k][1][2],L[k][2][2],BB),k=ii..jj);
end proc:

### -- the following is automatic done by decomp3 ###
###cofs1,mons1:=decomp(p1); ###<<<--- not needed as we use decomp3
###p11:=[seq([cofs1[i],mons1[i]],i=1..nops(cofs1))]; ###<<<--- not needed as we use decomp3
p11:=decomp3(p1);
N1:=nops(p11);
###cofs2,mons2:=decomp(p2); ###<<<--- not needed as we use decomp3
###p22:=[seq([cofs2[j],mons2[j]],j=1..nops(cofs2))]; ###<<<--- not needed as we use decomp3
p22:=decomp3(p2);
N2:=nops(p22);
N:=N1*N2;
Nhalf:=floor(N/2);
LL:=[seq(seq([p11[i],p22[j]],i=1..N1),j=1..N2)];
lst:=Array(LL);
N:=nops(LL):
Nhalf:=floor(N/2);
if Nhalf<_task_length then return
clicollect(cmultask22(lst,1,N,KK,BB)); end if;

id1:=Threads:-Create(cmultask22(lst,1,Nhalf,KK,BB),outid1);
id2:=Threads:-Create(cmultask22(lst,Nhalf+1,N,KK,BB),outid2);
Threads:-Wait(id1,id2);
out:=clicollect(reorder(outid1+outid2)); ###<<<--- combining outputs from threads
return out;
end proc:

B:='B':
KK1:='cmulNUM2':
KK2:='cmulRS2';
KK1:='cmulNUM';  ###<<<--- we can use original cmulNUM because we have fixed reorder to be thread-safe
KK2:= 'cmulRS';  ###<<<--- we can use original cmulRS because we have fixed cmulRS to be thread-safe

> p1:=rd_clipolynom(7);
  p1:=e1we2we3we5we6-Id-e3+e1we2we7+e1we2we4; ###<<<--- for illustration we use this polynomial

> p2:=rd_clipolynom(7);
  p2:=6*Id+4*e6-8*e1we4we7+10*e2we5we7; ###<<<--- for illustration we use this polynomial

First, we compare cmul[B](p1,p2) using cmulNUM with cmulMT[B](p1,p2,'cmulNUM2');

NOTE: With thread-safe reorder, we can use cmulNUM and cmulRS, which in turn use reorder!

> out1:=cmulMT[B](p1,p2,KK1); ###<<<--- computation of the Clifford product using cmulMT with thread-safe cmulNUM2 or cmulNUM, and arbitrary B
> reorder(out1)-out1;
> useproduct(cmulNUM);
  out2:=cmul[B](p1,p2); ###<<<--- computation of the Clifford product using cmul with original cmulNUM and arbitrary B
  out2:=reorder(out2); ###<<<--- for some reason cmul does not return ordered output, why?
> simplify(out1-out2);
> evalb(%=0); ###<<<--- verifying that results are the same

Second, we compare cmul[B](p1,p2) using cmulRS with cmulMT[B](p1,p2,'cmulRS2');

NOTE: With thread-safe reorder, we can use cmulNUM and cmulRS, which in turn use reorder!

> out11:=cmulMT[B](p1,p2,KK2); ###<<<--- computation of the Clifford product using cmulMT with thread-safe cmulRS2 and arbitrary B
> reorder(out11)-out11;
> #decomp3(out11);
> useproduct(cmulRS);
  out22:=cmul[B](p1,p2); ###<<<--- computation of the Clifford product using cmul with original cmulRS and arbitrary B
out22:=reorder(out22); ###<<== for some reason cmul does not return ordered output, why?
> reorder(out22)-out22;
> #decomp3(out22);
> simplify(out11-out22);
> evalb(%=0); ###<<== verifying that results are the same
>
So, the above shows that it seems that cmulMT works fine too:

Example 6: Let's try a loop now for random inputs into cmulMT using multithreading:

p1:=rd_clipolynom(7,'unordered');
p1:=-4*e2-Id+e5we4we6; ##<<--simple illustration
p2:=rd_clipolynom(7,'unordered');
p2:=4*e7we5+Id-2*e7we6; ##<<--simple illustration
> _max_times:=10; ###<<== max number of times to perform the loop
>
First, we do it for cmulNUM and cmulNUM2 in multithreading:

> useproduct(cmulNUM); ###<<== product to be used by cmul
> B:='B': ###<<== erase B
  _prolevel:=true:
> forget(Clifford): ###<<-- erasing remember tables in Clifford
  printlevel:=1:
  flag:=true:
  times1:=[]:
  times2:=[]:
  flags:=[]:
  maxtimes:=_max_times: ###<<== we compute as many times as when using cmulNUM2
  for i from 1 to maxtimes while flag do
    start:=time():
    printlevel:=0:
    out1:=cmul[B](p1,p2): ###<<== this uses ordinary cmul with cmulNUM and B
    out1:=reorder(out1); ###<<== for some reason cmul does not reorder its output, why?
    printlevel:=1:
    times1:=[op(times1),time()-start];
    start:=time():
    printlevel:=0:
    #out3:=cmulMT[B](p1,p2,'cmulNUM2'); ###<<== this uses cmulMT with cmulNUM2 and B
out3:=cmulMT[B](p1,p2,'cmulNUM'); ### <<<--- this uses cmulMT with cmulNUM and B along with thread-safe reorder
out3:=reorder(out3); ### this should not be needed as cmulMT reorders its output
printlevel:=1:
'times1'=times1;
times2:=[op(times2),time()-start];
flag:=evalb(simplify(out3-out1)=0);
flags:=[op(flags),flag];
end do;
printlevel:=1:
print(`####################`);
'times1'=times1;
'times2'=times2;
ratios12:=[seq(evalf(times1[i]/times2[i],3),i=1..nops(times1))];
'flags'=flags;
print(`####################`);
average_ratio12:=evalf(add(ratios12[i],i=1..nops(ratios12))/nops(ratios12),3);

Now, we compare cmulRS used in cmul with cmulRS2 used in cmulMT:

useproduct(cmulRS); ### <<<--- product to be used by cmul
B:='B': ### <<<--- erase B
_prolevel:=true:
forget(Clifford): ## <<-- erasing remember tables in Clifford
printlevel:=1:
flag:=true:
times11:=[]:
times22:=[]:
flags:=[]:
maxtimes:=_max_times: ### <<<--- we compute as many times as when using cmulRS2
for i from 1 to maxtimes while flag do
  start:=time():
  printlevel:=0:
  out1:=cmul[B](p1,p2): ### <<<--- this uses ordinary cmul with cmulRS and B
  out1:=reorder(out1); ### <<<--- cmul for some reason does not reorder its output, why?
  printlevel:=1:
  times11:=[op(times11),time()-start];
start:=time():
printlevel:=0:
#out3:=cmulMT[B](p1,p2,'cmulRS2'); ###<<<--- this uses cmulMT with
#cmulRS2 and B
out3:=cmulMT[B](p1,p2,'cmulRS'); ###<<<--- this uses cmulMT with
#cmulRS and B, and thread-safe reorder
out3:=reorder(out3);
printlevel:=1:
'times11'=times11;
times22:=[op(times22),time()-start];
flag:=evalb(simplify(out3-out1)=0);
flags:=[op(flags),flag];
end do;
printlevel:=1:
print(`####################`);
'times11'=times11;
'times22'=times22;
ratios1122:=[seq(evalf(times11[i]/times22[i],3),i=1..nops(times11))];
'flags'=flags;
print(`####################`);
average_ratio1122:=evalf(add(ratios1122[i],i=1..nops(ratios1122))/
nops(ratios1122),3);

> Comparing times from:

times1 =  times of cmul with cmulNUM using any B
times2 =  times of cmulMT with cmulNUM using any B
ratios12 = ratios of times1/times2
times11 =  times of cmul with cmulRS using any B
times22 =  times of cmulMT with cmulRS using any B
ratios1122 = times11/times22

> print(`times of cmul with cmulNUM using any B:`);
'times1'=times1;
print(`times1 deterioration (plus) or improvement (minus) in percent:`);
evalf((times1[-1]-times1[1])/times1[1]*100,3);
print(`############################################################################`);
print(`times of cmul with cmulRS using any B:`);
'times11'=times11;
print(`times11 deterioration (plus) or improvement (minus) in percent:`);
evalf((times11[-1]-times11[1])/times11[1]*100,3);
print(`############################################################################`);
percent:`);
evalf((times11[-1]-times11[1])/times11[1]*100,3);
print(`###################################################################`);
print(`times of cmulMT with cmulNUM2 using any B:`);
'times2'=times2;
print(`times2 deterioration (plus) or improvement (minus) in percent:`);
evalf((times2[-1]-times2[1])/times2[1]*100,3);
print(`###################################################################`);
print(`times of cmulMT with cmulRS2 using any B:`);
'times22'=times22;
print(`times22 deterioration (plus) or improvement (minus) in percent:`);
evalf((times22[-1]-times22[1])/times22[1]*100,3);
print(`###################################################################`);
print(`ratios of times1 over times2:`);
'rati oslover22'=ratios12;
print(`###################################################################`);
print(`average ratio times1 over times2:`);
average_ratio12:=evalf(add(ratios12[i],i=1..nops(ratios12))/nops(ratios12),3);
print(`###################################################################`);
print(`ratios of times11 over times22:`);
'rati os11over22'=ratios1122;
print(`###################################################################`);
print(`average ratio times11 over times22:`);
average_ratio1122:=evalf(add(ratios1122[i],i=1..nops(ratios1122))/nops(ratios1122),3);
print(`###################################################################`);

Cookeville, June 28, 2012