What is new in FORM

J.A.M. Vermaseren

with contributions from M. Tentyukov

- Short introduction.
- Various commands for output reduction.
- GZIP.
- External channels.
- ParFORM.
- TFORM.

Short Introduction.

FORM has usually seen lots of additions when new calculations were undertaken. These additions then became part of the official version either during or shortly after the calculation was finished.

One problem is proper testing, and another is to make a proper abstraction of the needs of the calculations. This abstraction means that commands should not be specific for a single problem, but everybody should benefit from it. Because of this a period is needed for experimentation.

Last year we (S.Moch,A.Vogt and JV) finished a big calculation which involved lots of new facilities in FORM. Many made it already into version 3.1 and were reported on at earlier workshops. But the period after a big calculation is very good for working at FORM and putting in facilities that were overdue. Also the people in Karlsruhe had some needs and M. Tentyukov as been building in features for running FORM together with other programs and having them communicate with each other.

I expect that a number of the new features will lead to experimentation by several groups. Some things may not work properly yet under your conditions. Please send bug reports. All will benefit, and together we will go into new directions in symbolic calculations.

Various commands for output Reduction.

Can we simplify lengthy Fortran outputs? This is a science by itself. There are various approaches:

- Find another program that can do the job and feed it the expression. This may not be easy due to the size of the expression.
- Split the expression in pieces and try again. This is not optimal, but it may improve things very much already.
- Try to do this within FORM by looking for common subexpressions. This is also far from ideal, but at least it is not limited by the size of the expressions.

The first two methods can use the new external channels (see later). There are various new functions that can help in the third method. They have been developed for the GRACE effort (see talk by Fujimoto). We show here one piece of an example.

```
#procedure squeeze2(bracket,dol,d,text,par)
  #do isqueeze = 1,1
    B,‘bracket’;
    .sort:‘text’-1;
    Keep Brackets;
    MakeInteger xfact;                              *<------------NEW
    #if ( ‘par’ > 0 )
      id  ‘bracket’(x?) = ‘bracket’(nterms_(x),x); *<------------NEW
      id  ‘bracket’(1,x?) = x;
      id  ‘bracket’(n?,x?) = ‘bracket’(x);
    #endif
    B,‘bracket’;
    .sort:‘text’-2;
    Keep Brackets;
    id    ‘bracket’(x?) = ‘bracket’(-termsinbracket_(0),x); *<----NEW
    id    ‘bracket’(-1,x?) = dum_(x);
    B ‘bracket’;
    .sort:‘text’-3;
    Keep Brackets;
    #$‘dol’a = $‘dol’;
    if ( match(‘bracket’(n?,x?)) );
      $‘dol’ = $‘dol’+1;
      id  ‘bracket’(n?,x?) = ‘bracket’(n,$‘dol’,x);
    endif;
    B ‘bracket’;
    .sort:`text'-4;
    Keep Brackets;
    #if ( {`$`dol''-`$`dol'a'} > 10000 )
      #$`dol' = $`dol'a+10000;
      #redefine isqueeze "0"
    #endif
    #do i = `$`dol'a'+1,`$`dol''
      id  `bracket'(n?,`i',x?$t{`i'-`$`dol'a'}) = `d'(`i');
    #enddo
    id    `bracket'(?a,x?) = `bracket'(x);
    .sort:`text'-5;
    `OFFSTATS'
    Hide;
    .sort:`text'-6;
    #do i = `$`dol'a'+1,`$`dol''
      L [b] = $t{`i'-`$`dol'a'};
      #call subpow
      .sort:`text'-7-`i';
      #write <tmp.f> "      `d'(`i') = %e",[b](`d'(`i'))
    #enddo
    `ONSTATS'
    Unhide;
    Drop [b];
  #enddo
#endprocedure
```

This is output from a short expression. Just to show the principle.

```
*
      xu(1) = 1-xnla
      xu(2) = 3-xnla
      xu(3) = 3-2*xnla
*
      xb(1) = e2e1a2+2*e3e1a2-2*e3e1*e2e1
      xb(2) = 5*e2e1-2*e3e1
      xb(3) = 2*e2e1a2+e3e1a2-2*e3e1*e2e1
      xb(4) = e2e1-e3e1
      xb(5) = 3*e2e1-2*e3e1
      xb(6) = amtq2-2*e3e1
      xb(7) = amel2a2+e2e1a2+2*e2e1*amel2
      xb(8) = amel2+e2e1
*
      xv(1) = 2*amel2a3+xb(1)*e2e1+xb(2)*amel2a2+2*xb(3)*amel2+xb(7)*
     & amtq2
*
      xz(1) = amel2a2
      xz(2) = xz(1)*amtq2
*
      xy(1) = 2*amel2+xb(2)
      xy(2) = xb(1)+xb(5)*amel2+xb(8)*amtq2
      xy(3) = xb(1)*e2e1+2*xb(3)*amel2
      xy(4) = 2*xb(1)*e2e1+4*xb(3)*amel2-2*xb(4)*xu(2)*e3e1*amtq2+xb(6)
     & *xu(2)*amel2*amtq2+xu(2)*e2e1*amtq2a2
      xy(5) = xb(1)*xu(3)+xb(5)*xu(3)*amel2+xb(8)*xu(3)*amtq2
      xy(6) = 2*xb(4)*xu(2)*e3e1*amtq2-xb(6)*xu(2)*amel2*amtq2-xu(1)*
     & amtq2*e2e1a2-3*xu(1)*e2e1*amel2*amtq2-xu(2)*e2e1*amtq2a2
*
      ztd = ztd*(c24)
*
      xre=+c4*(xu(3)*xz(1))
      xre=xre+c2*(xy(5))
      xre= -xre
      g1e0(0,0)=g1e0(0,0)+ztd*xre
      xre=+c2*(xz(1))
      xre=xre+xy(2)
      g1e1(0,0)=g1e1(0,0)+ztd*xre
      xre=+c8*(xz(1)*xy(1)+xy(3)-(2+xnla)*(xz(2)))
      xre=xre+c4*(-(1+xnla)*(amtq2*e2e1a2)-(5+3*xnla)*(e2e1*amel2*amtq2
     &    ))
      xre= -xre
      g0e0(0,0)=g0e0(0,0)+ztd*xre
      xre=+c8*(xz(1)*xy(1)-(1+2*xnla)*(xz(2))-(1+3*xnla)*(e2e1*amel2*
     &    amtq2)-(xnla)*(amtq2*e2e1a2))
      xre=xre+c4*(xy(4))
      g0e0(0,1)=g0e0(0,1)+ztd*xre
      xre=+c8*(xu(1)*xz(2))
```

```
xre=xre+c4*(-xy(6))
xre= -xre
g0e0(0,2)=g0e0(0,2)+ztd*xre
xre=+c8*(xv(1)*xu(1))
g0e0(1,0)=g0e0(1,0)+ztd*xre
xre=+c8*(-xv(1)*xu(1))
g0e0(1,1)=g0e0(1,1)+ztd*xre
xre=+c8*(-xv(1)*xu(1))
g0e0(2,0)=g0e0(2,0)+ztd*xre
```

It should be noted that the weakest point in this all is the Fortran compiler. It needs usually far more CPU time than the FORM program that generates the code. And it also needs far more memory. Already for routines that are not very long the compiler might crash by lack of memory space.

GZIP.

The maximum size of expressions that can be handled by FORM is determined by the available file space. When there is only a single expression the total file space needed during the calculation is one scratch file for input or output and the sort file.

Often the sort file can be much larger than the completed output scratch file. These files have already some form of data compression applied to them, but this is a rather simple expression. It has the advantage though that it allows FORM to select pieces of the expression like specific brackets.

When GZIP is applied to these files one can obtain compression ratios that are typically a factor 4, although this will depend very much on the problem and the hardware. If there are mainly very lengthy coefficients, the ratio will be much less. If one works on a 64-bits computer and the coefficients are short numbers, the ratio will be larger.

Currently the sort file can be treated with the gzip library. Each 'patch' is a gzipped stream and when the file-to-file sort takes place they are simultaneously gunzipped slowly to feed the merging routine.

The drawback is that using gzip costs CPU time. Unfortunately this isn't gained back by the fact that now less writing to files and reading from files is needed. Hence one should use this facility mainly when one is really short of disk space.

Syntax:

```
    On compress GZIP 6;
```

This selects compression level 6 which seems a nice compromise between speed and compression. The highest level is 9, but this is very slow and doesn't compress much better than level 6. Level 0 means no compression.

It would have been possible to use bzip, which compresses better. It is however extremely slow. Maybe in the future a better algorithm will be developed that takes more the structure of a FORM binary file into account. The future may also see compression of the scratch files, but that will lead to a reduction of the possible commands that can be used in the next module as ‘keep brackets’ and the obtaining of bracket contents will not be possible.

External Channels (by M.Tentykov).

This is used in Karlsruhe for a program that implements the Laporta algorithm. They get often subexpressions that are sums of rational polynomials in d. They want to pull them over a common denominator and then simplify by dividing out the GCD. For this they use the program FERMAT.

The whole is described in a very recent paper: "Extension of the functionality of the symbolic program FORM by external software" (M. Tentyukov and J.V.) cs.SC/0604052.

Just an example that sends the expression withGCD to fermat and picks up the answer. We do this 1000 times just to show the speed.

```
Off statistics;
Format 250;
#define cmd "./fermat"
#define init "&d\n0\n&(M=\' \')\n&(t=0)\n&U\n&(J=d)\n"
Symbol d;
#external ‘cmd’
#toexternal "‘init’"
#do i = 1,19
  #fromexternal "tmp"
#enddo
Local withGCD =(2*d^4+3*d^3-22*d^2-13*d+30)/(d^3-11*d+10);
.sort
#do i=1,1000
  #toexternal "%E\n",withGCD
  #fromexternal "tmp"
  Local noGCD =
  #fromexternal
          ;
  Print;
  .sort
  Drop noGCD;
  .sort
#enddo
.end
```

The tail of the output file is:

```
   noGCD =
      3 + 2*d;

    .end
real	0m0.799s
user	0m0.160s
sys	0m0.010s
```

It should be clear that there is much room for creativity here. Sometimes most of the work is in building the filters/gateways that translate the notations of the various programs or manage information of programs in such a way that the others can work with it.

ParFORM.

ParFORM is a project of the university of Karlsruhe, funded by the DFG. People who have worked at it in the past are J. Staudenmaier, A. Retey, D. Fliegner, A. Onyshenko, M. Frank. Currently M. Tentyukov and M. Steinhauser are in charge of it. As seen in the above section the dexterity of M. Tentyukov with the FORM sources also gives us new features. ParFORM is a version of FORM that uses several processors or computers, connected by a network in a way that makes it look to the user as a single much faster computer.

ParFORM keeps getting improved. Its main machine at the moment is the SGI machine at Karlsruhe which has 32 Itanium 2 processors at 1300 MHz. It is used by P. Baikov for the running of 4-loop propagator diagrams, using mostly groups of 8 processors, getting execution speeds that are a factor 4-5 times better than on a single processor. The record is a factor 6.

ParFORM also runs on several other computers. The problem with the ports is that there is no rigorous standard for the MPI libraries. This means that each port is much work. They are made on demand and availability of time and resources.

TFORM.

This is a rather new project. Currently it is my pet project. The aim is to make a multi-threaded version of FORM that can run on computers with shared memory. In that case the data is divided in shared global data and thread specific local data. This way much less data has to be transfered. On the other hand: the routines have to be ‘thread-safe’. Currently this is where the major source of errors is.

The program is currently being debugged and tuned. Already some improvements have been made and mincer is already running with it. The main machine for experimentation is a quad opteron machine with 4 opteron 850 processors at 2.4 GHz and 16 Gbytes of memory.

The running of TFORM is rather simple:

```
tform -w4 calcdia
```

would run the program calcdia.frm with 4 worker threads and one master thread. The command

```
tform calcdia
```

would just run everything inside the master thread. No workers will even be started. This is on average slightly slower (2-3%) than the sequential version, although there have been cases for which it was actually slightly faster.

The treatment of the threads follows the Pthread (POSIX threads) standards which are widely available, defintely in unix systems. Hence there are very few problems with ports.

There are some features that allow the user to tune the system a bit. Terms are send to the workers in groups or 'buckets'. The optimal size of the buckets depends on the problem.

Sometimes all nasty terms can be in one bucket making one worker doing all the work. The master and the other workers have to wait for this worker to finish. There is a loadbalancing system in which the master can steal the unprocessed contents of the bucket of a thread that is still busy at the end of the module. This can be switched off if necessary. A second stage load balancing is being planned. This would be for when there is a single bad term. In that case a worker can give branches of the tree to other workers.

There will be a separate paper about version 3.2 of FORM that will explain more of the technical details. Here I will show some results on various computers. The computers that are being used are

- alfonso: a Dual Pentium IV at 1.7 GHz.
- norma: a Quad opteron system with 4 Opteron 850 processors at 2.4 GHz.
- qcmsgi: The machine at Karlsruhe with 32 itanium 2 processors at 1.3 GHz. We use a varying number of processors which we will indicate by qcmsgi4 (for 4 processors) etc. Currently no examples available as the machine was down past week......
- The use of hyperthreading processors gives only a relatively small increase of the work that can be performed, which is mostly offset by the extra overhead. Here TFORM would only be useful in crowding out other programs. We will not consider them here.

The first example program we consider is from a suite of examples that was constructed several years ago on the suggestion of D.Fliegner. They concern the construction of so-called chromatic polynomials on a lattice. The whole project from the viewpoint of FORM is to show how one can successively improve the efficiency of a program. The whole paper and the programs can be found in the FORM distribution site. The crucial part of the program is

```
#do i = 0,`N'^`D'-1
  Multiply F`i';
  repeat id  d(?a,k?,?b)*d(?c,k?,?d) = d(?a,?c,k,?b,?d);
  Symmetrize d;
  repeat id  d(?a,k?,k?,?b) = d(?a,k,?b);
  #do d = 1,`D'
    id  d(k`i',?a,k1?kk0[x],?b,k2?kk`d'[x],?c) = 0;
  #enddo
  id,ifmatch->1,d(k`i',k?) = 1;
  id,ifmatch->1,d(k`i',?b) = d(?b);
    Multiply acc([q-1]+1);
  Label 1;
  id  d = 1;
  .sort:`i';
#enddo
```

The complete program can be found in the FORM distribution site once version 3.2 of FORM will be available.

We run the program for a 13x13 lattice. The improvement factor is the improvement in the real time that it takes to run the program. When we mention 0 workers we refer to the sequential version of FORM. For one worker we refer to tform in which the master runs the whole program. This shows part of the overhead. Then there is more overhead when the workers are actually used, because now signals have to be sent and information has to be copied to and from the workers.

Workers	CPU-time	real time	CPU-master	improvement
0	1871.57	1921.14		
1	1908.60	1944.34		
2	2021.47	1089.56	159.51	1.7632
3	2041.79	1115.65	192.73	1.7220
4	2057.47	1141.83	216.61	1.6825

Program p15 running on a Dual P4(1700).

We notice that running more than two workers gives a slight penalty and makes the improvement go down a bit. But it is very well possible to run with more threads than there are processors.

Workers	CPU-time	real time	CPU-master	improvement
0	710.15	710.69		
2	738.12	386.17	50.17	1.8403
4	788.88	244.00	70.40	2.9127

Program p15 running on a Quad Opteron 850 (2.4 GHz).

The next example concerns some diagrams that are run with Mincer. I selected 3 diagrams: d1c, d10c and d11c from a dataset for calculating nonsinglet contributions to photon-quark scattering. These are all non-planar diagrams. The first is moderately simple and we can calculate the N=16 moment. The second diagram is moderately difficult and the third diagram is the most difficult of the set. Of the last two diagrams we calculate the N=10 moment.

Diagram	Workers	CPU-time	real time	CPUmaster	improvement
d1c	0	2675.80	2676.18	-	-
d1c	1	2729.62	2729.93	-	
d1c	2	2804.31	1433.28	45.53	1.8672
d1c	3	2810.25	1438.95	66.43	1.8598
d10c	0	2649.63	2651.87	-	-
d10c	2	2796.84	1440.82	77.47	1.8405
d11c	0	10344.22	10400.52	-	-
d11c	2	10870.10	5693.18	333.51	1.8268

Mincer diagrams running on a Dual P4(1700).

Diagram	Workers	CPU-time	real time	CPUmaster	improvement
d1c	0	754.05	755.73		
d1c	1	807.78	808.88		0.9343
d1c	2	827.36	424.80	15.67	1.7790
d1c	3	833.22	291.09	21.67	2.5961
d1c	4	835.19	228.79	26.72	3.3031
d10c	0	851.12	852.09		
d10c	2	928.02	471.77	30.21	1.8062
d10c	4	945.15	262.03	38.29	3.2519
d11c	0	3124.07	3125.05		
d11c	2	3408.98	1720.59	107.18	1.8163
d11c	4	3513.83	984.07	136.36	3.1756

Mincer diagrams running on a Quad Opteron 850 (2.4 GHz).

The major weakness is at the end of the module when all workers have to collect the contents of their sort file, merge it and send the results to the master thread which merges these streams into a single output file. The resulting traffic jam can bring processing almost at a halt. Sometimes it can make running on 4 processors slower than on a single one. What to do about this is still being investigated as the situation of the disk on the quad machine was far from ideal. Maybe a better (and bigger) disk will improve things. Sometimes it seems to pay to force the sort files to be written and just wait for that....

There remains the issue that is there is much writing activity, this will slow things down. The example used was the p15 program with a 15x15 lattice. In the program about 250 Gbytes of files had to be written when 4 workers were used. In the sequential version this is less as the buffers can be bigger. Hence 'only' 139 Gbytes have to be written. The result is that, even with 4 threads and much optimization concerning synchronizing the file system, TFORM is only a little bit faster:

Workers	CPU-time	real time	GBytes written
0	11614	14210	139
4	13974	10535	250

Program p15(15x15) running on a Quad Opteron 850 (2.4 GHz).

Conclusions.

As soon as the number of problems gets so low that I don't see any, I will make this version available. Most likely programs of other people will run into problems. Please report those in as concise a way as possible in such a form that I have a chance to reproduce them. Debugging multi-threaded programs is rather complicated and hence one needs as much information as possible.