What brings the future?

J.A.M. Vermaseren

Nikhef

- More processors?
- FORM experiments
- Conclusions/Outlook

More Processors?

As explained by Ettore Remiddi, FORM has its roots in SCHOONSCHIP and although its birth was not without little troubles it is doing rather well now.
Its first model of operation was based on the great example which I had been using for years. There were of course things that I wanted different and hence, when making my own program I did make them different.
Also the inside of the program is completely different from the inside of SCHOONSCHIP. The program was designed from scratch to do roughly what I wanted SCHOONSCHIP to do but that is where all similarity stops. Specially the memory model is completely different. Whereas SCHOONSCHIP was optimized for the mainframe computers of the late 1960's, FORM was designed for the workstations of the late 1980's and what was expected to follow. This expectation was reasonably accurate, because what worked very well in the late 1980's with files of up to a few Megabytes still works well in 2011 with files of 1 Terabyte.

Over the years Moores law that has computer power multiplied by a factor 10 every 5 years, has been amazingly accurate. Lately however the processors have not become much faster. The gain is to be obtained from using more processors working together as a team. This requires some special attention and the realization that not all tasks can benefit from this. But the computing model of SCHOONSCHIP in which all terms are processed sequentially and are independent objects makes life much easier. When operations take input from single terms only, it is rather easy to parallelize a program: just give each processor its share of the terms.

This has been implemented in two special versions of FORM:

- ParFORM: This is an effort of the university in Karlsruhe, paid for by the DFG. It runs on (fast) networks of (identical) computers. It uses MPI for communication between the processors.
- TFORM: This program uses the POSIX threads system on computers with multiple cores (like most people have nowadays). It uses the fact that they all share the same memory. Does not work under Windows as Microsoft is notoriously poor at adhering to standards.

Of course these implementations have many details. And as a generic property of symbolic manipulation there will be bottle necks. Also, the more popular programs have a problem here.

From the viewpoint of Mathematical elegance it looks best to see whether there are so-called common subexpressions and then store these once only as in

```
f(a+b)+g(2,a+b)
```

We store the `a+b` only once and place a pointer to this whereever this subexpression occurs.
But this means that we need a whole administration of how many times the object is used because we can only remove it when nobody needs it.
Imagine now that one term is in one processor and the other in another processor. To change something in f the worker will need to access the central administration. This gives lots of traffic and hence traffic jams. One could try to have local administrations, but then moving a term from one processor to another is a highly nontrivial operation that costs much time.
This is just one of the problems. Because in FORM all terms are independent objects FORM does not have this kind of problems.

Some FORM Experiments

Let us have a look at how the parallel version of FORM works. In particular we will look at TFORM.

First we have an expression, properly sorted, on disk. We have one master processor deciding who gets what. Depending on the structure of the input this may take from hardly any time, to a significant amount of time. There are for instance cases where it is imperative that a group of terms ends up in the same worker. The important thing here is that we have to send messages to tell the workers that there is something for them. The model here is that we set a task ready for a worker and then we send a message to the worker, saying what type of task there is and that he should pick it up.

```
while ( ( wakeupsignal = ThreadWait(identity) ) > 0 ) {
  switch ( wakeupsignal ) {
/*
    ## STARTNEWEXPRESSION :
    ## DOONEBUCKET :
    ## FINISHEXPRESSION :
    ## CLEANUPEXPRESSION :
    ## STARTNEWMODULE :
    ## TERMINATETHREAD :
    ## DOONEEXPRESSION :
    ## DOBRACKETS :
*/
    default:
      MLOCK(ErrorMessageLock);
      MesPrint("Illegal wakeup signal %d for thread %d",wakeupsignal,identity);
      MUNLOCK(ErrorMessageLock);
      Terminate(-1);
  }
}
```

If we send one message per term this slows down the program measurably. Hence the optimal strategy seems to be to make buckets with a number of terms in them. For typical jobs the optimal number is 500-1000 terms. Of course this is only a partial improvement because the master still has to read all the terms before they can be put in the buckets. An advantage of this method is that if the master has to wait for the workers it can start filling buckets in advance and hence when a worker becomes free it can immediately be given a new bucket. This way the workers do not have to wait.

Example: We run a Mellin moment of a nonplanar three loop diagram in deep inelastic scattering in QCD. N is the number of the Mellin moment.

N	FORM	TFORM(w=8)(b=500)	TFORM(w=8)(b=1)
2	0.87	0.85	1.14
4	7.54	3.75	7.24
6	54.49	15.26	36.06
8	299.52	64.39	145.97
10	1641.43	285.86	539.24

It becomes easier to understand the difference when we look at the CPU time spent by the master and the workers:

bucket	Master	All Workers	1 Worker	Real Time
500	62.48	1818.18	227.27	285.86
1	326.63	2604.08	325.51	539.24

An improvement on this system is when there is more structure in the input.

Just imagine that the input has been sorted with parts outside brackets and parts inside brackets. And imagine we have made a so-called bracketindex which points at where the brackets start on disk.

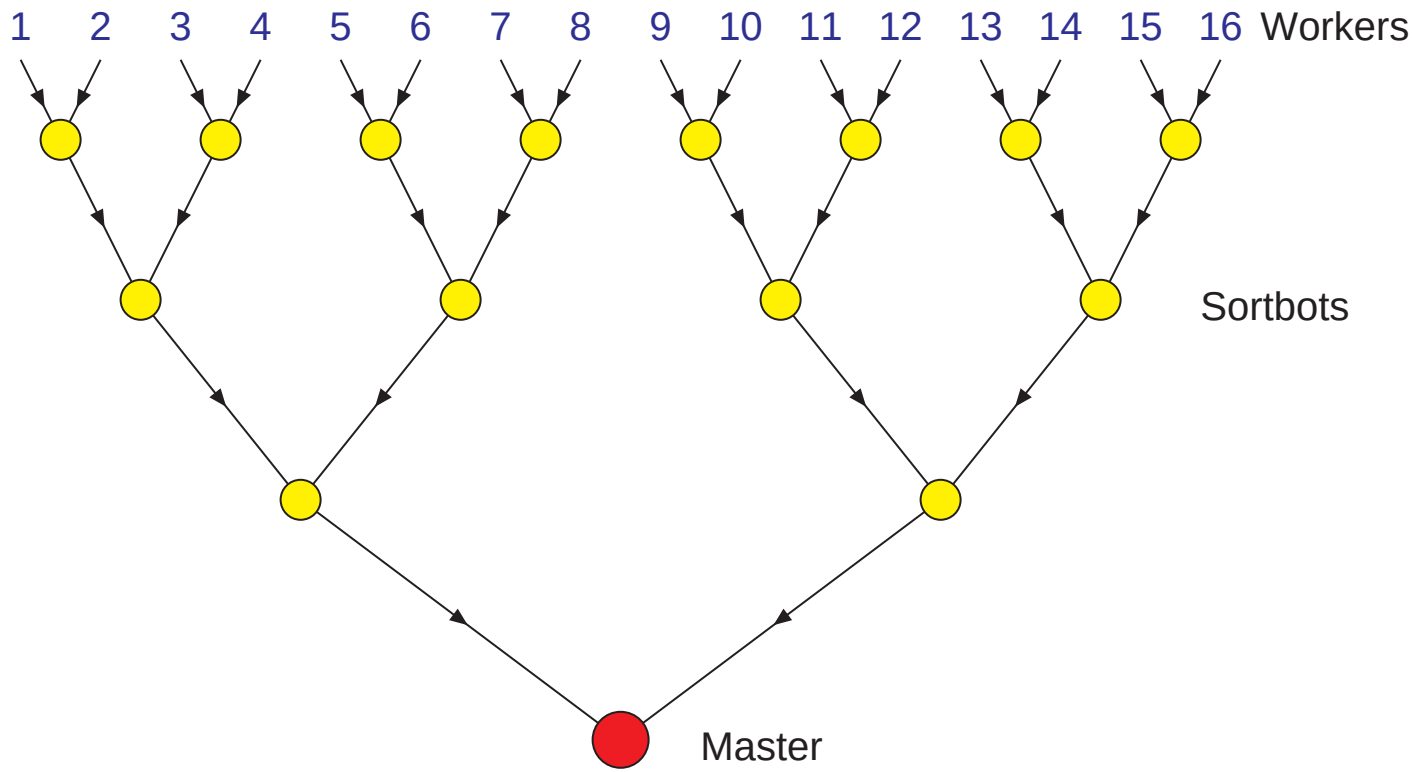
Then the master only has to tell the workers which brackets they have to do and the workers can read the terms themselves. This is particularly efficient when the input expression still fits inside the disk caches that FORM uses and hence is not on disk at all. In that case all workers can read from this cache simultaneously.

This would solve the problem completely if it were not for the fact that at some time that bracketindex has to be made, and because it belongs to the input expression it is the master who has to make it. Yet it does improve the speed measurably in a large number of programs.

Another way to speed up the input part is when we have many small expressions. In that case the master tells the workers which expression each should do. The worker is then responsible for the complete expression including writing it to output. There may be a small traffic jam at the output channel but nothing as bad as when only the master would be responsible for all the writing.

Sorting

Now we come to the real bottleneck: the collection of the output. Imagine we are calculating one big expression and we have sent terms to 16 workers. At a given moment there are no more terms and the workers get the signal that this is the case. They make then a final sort of what they have and the 16 outputs have to be merged into one complete output. The way this is done is that each two workers send their result to a special worker, called a sortbot, that merges these results and each pair of sortbots sends their results to yet another sortbot, till finally the last two sortbots send their results to the master who sorts the final two streams and writes the results to disk. This is a tree approach.


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Workers
Sortbots
Master

The idea is that when there are n processors and n workers, one can easily run an extra thread for the master and n-2 extra sortbots, because usually when one of them is working one of the workers is waiting. And in the worst case they have to share resources.

The important thing is that we loose as little CPU time as possible.

This sortbot approach gives a measurable improvement in speed over the model in which all workers give their output to the master directly and the master is responsible for the merging of 16 outputs. For 16 workers we see for N=10:

sortbots	Master	All Workers	Real Time
no	125.87	1733.26	225.43
yes	62.54	1914.57	175.22

One of the time consuming operations in the sorting is the addition of the coefficients. Because FORM works with rational numbers the addition of lengthy fractions takes quite some time. It is the calculation of the GCD's that is the culprit.

Unfortunately there are no efficient GCD algorithms.

The fastest routines I have access to are the ones of the GMP (GNU Multi Precision Library). Basically it uses the same algorithms as FORM but their implementation uses some assembler code which allows a few tricks that one cannot use in C. When the GMP is not available, (Windows, Apple) FORM runs with its own routines.

Things get even worse when the coefficient is a rational polynomial.

To compute the GCD's for them takes quite some time in the multivariate case. This means that the final stage of the sorting is at its fastest when all additions or cancellations are made in the workers and the master gets only different terms to sort out. FORM has also some facilities for that: the antibracket:

```
 Symbols a1,...,a100;
 Symbols x1,x2;
 Format NoSpaces;
 Local F = (a1+a2+a3+x1+x2+1)^2;
 Bracket x1,x2;
 Print;
 .sort
F=
   +x2*(2+2*a3+2*a2+2*a1)
   +x2^2*(1)
   +x1*(2+2*a3+2*a2+2*a1)
   +x1*x2*(2)
   +x1^2*(1)
   +1+2*a3+a3^2+2*a2+2*a2*a3+a2^2+2*a1+2*a1*a3+2*a1*a2+a1^2;
```

```
 ABracket x1,x2;
 Print;
 .end
F=
   +a3*(2+2*x2+2*x1)
   +a3^2*(1)
   +a2*(2+2*x2+2*x1)
   +a2*a3*(2)
   +a2^2*(1)
   +a1*(2+2*x2+2*x1)
   +a1*a3*(2)
   +a1*a2*(2)
   +a1^2*(1)
   +1+2*x2+x2^2+2*x1+2*x1*x2+x1^2;
```

A piece of the Mincer code:

```
  ....
ABracket p1.p3,p1.p1,p3.p3,p8.p8;
.sort
id  p1.p3 = p1.p1/2+p3.p3/2-p8.p8/2;
ABracket p2.p3,p2.p2,p3.p3,p7.p7;
.sort
id  p2.p3 = p2.p2/2+p3.p3/2-p7.p7/2;
  ....
```

Together with the bracket index complete brackets get moved to only a single worker. This means that all cancellations take place inside that worker. The spinoff is also much less use of diskspace because the workers need smaller sort files.

In an actual run the program used 800 Gbytes of total diskspace before we used this trick and only 200 Gbytes after. The complete output had a size of 100 Gbytes.

One of the very strong points of ParFORM and TFORM is that nearly all FORM programs will benefit from it.
They do not need to be modified at all.
The example we have been using (the Mincer program for three loop massless propagator-type diagrams) was originally written with only the sequential version of FORM in mind. The other versions of FORM were not even in the design phase. All runs here are with this version of Mincer. So let us see now what benefits one can obtain with the N=10 Mellin moment program:

workers	Master	All Workers	Real Time
-	1581.82	–	1582.24
0	1655.79	–	1658.94
1	52.26	1660.72	1684.96
2	70.90	1680.54	890.02
3	65.32	1678.85	614.31
4	67.53	1702.68	479.21
5	64.16	1778.53	417.08
6	63.26	1821.17	364.35
7	63.00	1783.91	310.93
8	64.46	1796.78	282.80
9	63.37	1813.19	262.12
10	64.35	1831.94	243.11
11	62.49	1835.13	223.20
12	60.09	1837.95	207.82
13	63.11	1856.82	200.78
14	61.84	1878.26	190.15
15	62.53	1907.14	184.16
16	62.54	1914.57	175.22
17	64.30	1918.81	175.27
18	63.63	1952.84	167.57
20	63.59	1987.98	159.10
23	61.67	2011.96	151.01
24	61.13	2080.46	151.46

We see here an enormous speed increase, but also that there is a levelling off.

The workers time increases slowly but the time of the master is about constant. This means that in the limit the real time can never be faster than this constant master time.

Hence the way to improve the performance is to take tasks from the master and divide them over the workers.

From the example at one worker we see how much time the master spends on the final sort (in this example): roughly one quarter. The rest of the time goes mostly in writing the output, reading the input and preparing the buckets. Hence a really significant improvement can only be made if all four of these problems are solved.

Of course one might argue that if you have 10000 Feynman graphs to calculate, it is easiest to submit 10000 jobs, each with sequential FORM. The above computer could handle 24 jobs simultaneously.

This does not work!

What happens is that at a given moment all processors are working on a difficult diagram. The easy ones take hardly any time. That puts a heavy strain on the use of memory and disk facilities with severe traffic jams at the disk. And eventually they are all waiting for the disk and the efficiency goes down tremendously. It even crashed the computer once.

The parallelization is really needed to make efficient use of such computers.

Computer algebra programs are completely different from numerical programs! The intermediate expression swell puts a heavy strain on the system.

Conclusions/Outlook.

The SCHOONSCHIP/FORM way of dealing with algebraic expressions is very suitable for use with many processors.
This is exactly the opposite of other systems. Systems like Reduce, Maple and Mathematica cannot parallelize the whole system.
What they do nowadays is build special routines that perform specific tasks like matrix inversion in a parallel way. Similarly I have been in a talk by M. Monaghan (one of the people of Maple) in which he made special routines to speed up the multiplication of polynomials. It used very clever tricks and special notations to solve this specific problem. This algorithm could not be parallelized, but it was fast. Significantly faster than FORM. But not faster than TFORM on 8 cores.

It should be clear that the way to make progress when we have to do big calculations is to use many processors. This is a relatively new field, and specially for computer algebra people are still researching what is best. And singularly amoung them is FORM. Hence not much help is to be expected from the outside.

But we manage.

Happy Birthday Tini!