The University of Manchester

# A parallel differential algebra code: tackling an apparently linear problem with openMP.

David Brett, University of Manchester/ Cockcroft Institute
with Robert Appleby, University of Manchester/ Cockcroft Institute

# Aim

- Introduce the concept of differential algebra codes.
- Introduce the use of openMP for a calculation that apparently seems linear.

# Contents

- Introduce the purpose of a differential algebra code and variety of applications relevant to accelerator physics.
- Discuss why MPI isn't a good method of parallelizing this problem.
- Introduction to openMP.
- openMP usage in a differential algebra code.
- Example results.

# Differential algebra codes

- All variables are handled as truncated power series of the unassigned initial values.
- Numerous competing codes that all do very similar things with regards DA:

  - COSY INFINITY (origins found in SixTrack) developed by Martin Berz.
  - MARYLIE developed by Alex Dragt.
  - MADX in particular subroutines in PTC developed by Etienne Forest.

# Differential algebra codes

- Matrix code = 1D differential algebra code.
- R and T matrix calculations= 2D differential algebra code.
- Natural expansion is N D differential algebra code, where as N tends to infinity the results tend towards the real result assuming no other assumptions made.

# Differential algebra codes

- A few applications:
  - One turn maps.

Proceedings of the 2001 Particle Accelerator Conference, Chicago

**Symplectic Map Tracking for the LHC**

D. T. Abell, BNL,
F. McIntosh and F. Schmidt, CERN

  - Single element integration (Both magnets and RF cavities) for use in tracking codes.

**BEAM DYNAMICS IN NS-FFAG EMMA WITH DYNAMICAL MAPS** *

Y. Giboudot†, R.Nilavalan, Brunel University, Uxbridge, UK.
R. Edgecock, STFC Rutherford Appleton Laboratory, Didcot, UK.
A. Wolski, University of Liverpool, Liverpool, UK

- Space charge effects

IEEE TRANSACTIONS ON ELECTRON DEVICES, VOL. 35, NO. 11, NOVEMBER

Differential Algebraic Description and Analysis of Trajectories in Vacuum Electronic Devices Including Space-Charge Effects

M. BERZ

# Differential algebra codes

- All codes handle each variable as an array of coefficients relating to the power series.
- Linear operations and calculus operations are able to be performed on these variables.
- An additional operation is a mapping operation which is very time consuming.

$$f(X, Y, Z) \mapsto f(x, y, z)$$

where $X = X(x, y, z)$

$$Y = Y(x, y, z)$$

$$Z = Z(x, y, z)$$

# Single threaded code

- Written in C++.
- Essentially a class with very optimized operations assuming that the array of coefficients is very sparse.
- Use of STL library:
  - Fixed size vectors to handle coefficients. (same speed as pointer double arrays, less leaks).
  - Filled array addresses stored as map variable. Allowing for easy optimization with sparse array assumptions.
  - Avoids the ridiculously large heap allocations required by fortran codes.

```
class DAobject
{
private:
    vector<double> coeffs;
    //double *coeffs;
    int order;
    int variables;
    int size;
    map<std::vector<int>,int> filled;
    friend std::ostream & operator<<(std::ostream &os, const DAobject &DA);
public:
    //Default constructor
    DAobject();
    DAobject(int N);
    DAobject(const DAobject &DA);
    ~DAobject(){ filled.clear();}

    //Access functions
    void set(double value,int v[6]);
    void set(double value, int i);
    double get(int v[6]) const;
    map<std::vector<int>,int> getmap() const {return filled;};
    void setmap(map<std::vector<int>,int> input){filled = input;}
    void clearmap(){filled.clear();}
    int getorder() const {return order;}
    int mapsize(){return (int)filled.size();}

    //Operators
    DAobject operator=(const DAobject &DA);
    DAobject operator=(const double &val);
    DAobject operator*=(const DAobject &DA);
    DAobject operator+(const DAobject &DA) const;
    DAobject operator*(const DAobject &DA) const;
    DAobject operator-(const DAobject &DA) const;
    DAobject operator+(const double &val) const;
    DAobject operator*(const double &val) const;
    DAobject operator/(const double &val) const;
    DAobject operator-(const double &val) const;
};
```

# Single threaded code - Application of class

Loop through length of integration

Coefficient of power series describing vector potential

Integration steps of 2nd order Robin-Forest-Wu integration

Mapping operation, mapping doubles onto DA variables to give a single particle track

```
84
85      ofstream out("outcrab.dat");
86      ofstream out2("outcrabmap.dat");
87      s=-0.35;
88      cout<<"***************"<<endl;
89      while (s<=0.35) {
90          //Integration
91          H4(X,s, ds,CoeffY);
92          H3(X,s, ds,CoeffX);
93          H4(X,s, ds,CoeffY);
94          H2(X,s, ds,CoeffZ);
95          H4(X,s, ds,CoeffY);
96          H3(X,s, ds,CoeffX);
97          H4(X,s, ds,CoeffY);
98          H1(X,s,ds);
99          H4(X,s, ds,CoeffY);
100         H3(X,s, ds,CoeffX);
101         H4(X,s, ds,CoeffY);
102         H2(X,s, ds,CoeffZ);
103         H4(X,s, ds,CoeffY);
104         H3(X,s, ds,CoeffX);
105         H4(X,s, ds,CoeffY);
106
107         //OUTPUT TRACK
108         for (int i=0; i<6; i++) {
109             XT2[i]=MAPAPPLY(X[i],XT);
110         }
111         out<<s<<"\t"<<XT2[0]<<"\t"<<XT2[1]<<"\t"<<XT2[2]<<"\t"<<XT2[3]<<"\t"<<XT2[4]<<"\t"<<
                XT2[5]<<endl;
112         cout<<s<<"\t"<<XT2[0]<<"\t"<<XT2[1]<<"\t"<<XT2[2]<<"\t"<<XT2[3]<<"\t"<<XT2[4]<<"\t"<<
                XT2[5]<<endl;
113     }
114     cout<<"Writing map to file"<<endl;
115     out2<<X[0]<<X[1]<<X[2]<<X[3]<<X[4]<<X[5]<<endl;
```

# Single threaded optimization

- Before even considering parallelizing code:
  - Consider where things are being calculated multiple times. For example multiple power series will say need Y x X calculating multiple times. Store results to memory and reuse.

$$x^4 = x \times x \times x \times x = (x \times x)^2$$

  - Fix array lengths where possible.
  - Consider required numerical precision. No point in going to high precision to output to file to 8 sf.

# MPI and openMP

- Two major methods for parallelizing C++ codes on single machines with many cores.
- MPI works on many iterations of the same code which communicate variables (limited to standard type and not classes), can run on very large clusters.
- openMP runs on single machine with multiple cores. Single code which can use multiple threads.

# Why not use MPI?

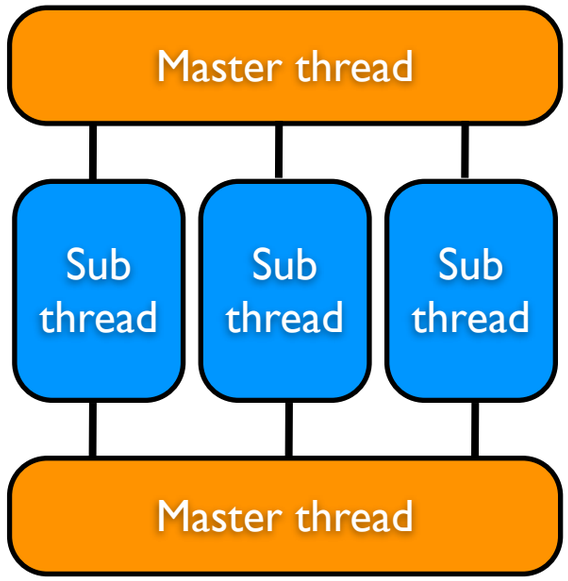
- MPI's lack of support for communicating classes means that the communication time could be long defeating the point of running parallel.
- The memory required to be passed between machines would be large.
- Only sections of the code can be parallelized leaving to a large amount of optimization required to reduce unnecessary communication.

# openMP

- Mission statement

> The OpenMP Application Program Interface (API) supports **multi-platform shared-memory parallel** programming in **C/C++** and **Fortran** on **all architectures**, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, **scalable** model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

- openMP is implemented with an additional make flag and lines are added to a single threaded code to enable parallelization of sections.




Basic code model

# openMP

Start with single threaded code.

All openmp statements begin with #pragma

parallel creates number of threads default max cpus.

for loop distributes each loop among threads as previous thread finishes

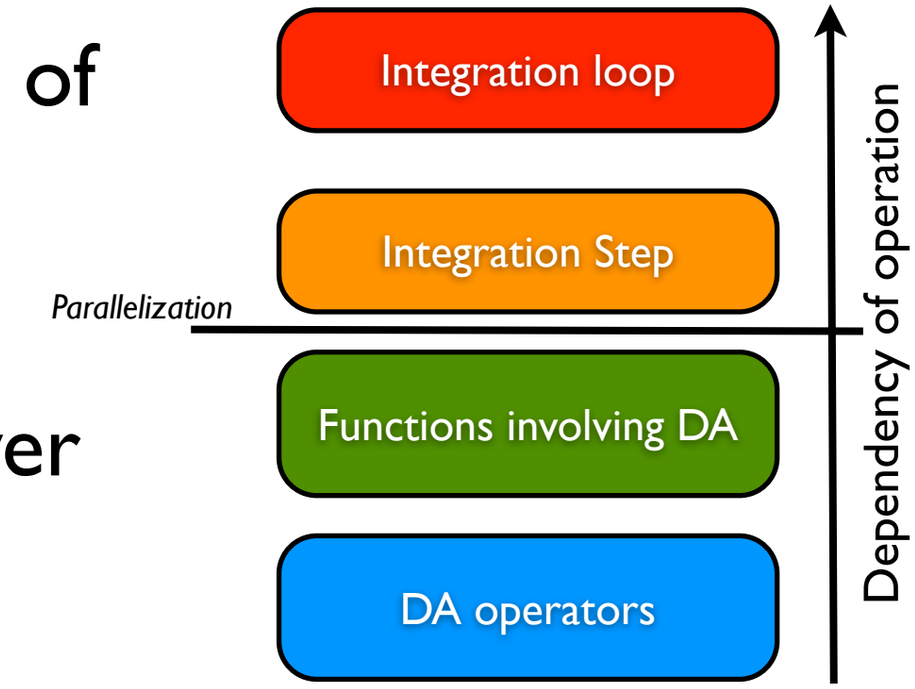
Additional code interaction with threads found in omp.h

critical prevents two threads writing to the same memory block at the same time

```
#include <iostream>
#include <vector>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>

using namespace std;

int main(int argc, const char * argv[])
{
    //Define number of test
    const long int N=100000000;
    long int count=0;

#pragma omp parallel
    {
        srand(int(time(NULL)) ^ omp_get_thread_num());
#pragma omp for
        for (long int i=0; i<N; i++) {
            double x=double(2.*rand()/double(RAND_MAX)-1.);
            double y=double(2.*rand()/double(RAND_MAX)-1.);
            if(sqrt(pow(x,2)+pow(y,2))<=1.)
            {
#pragma critical
                {
                    count++;
                }

            }
        }
    }
    double pi=4.*double(count)/double(N);
    cout<<"PI = "<<pi<<endl;
}
```

# openMP

- Application to DA code choice of how deep to implement the parallelization.
  - Lower down the code = lower amount of memory locks + higher amount of threads required.
- Choice made to apply to the map operation and each step of the integration as at intermediate level.



# openMP - for loops

Declaration of parallel section

Memory designation, either private or shared between each thread.

Thread distribution across cpus, dynamic in case some finish faster than others

Nested for loop

```
668  void AXFIELD(const vector<DAobject> X, const double &s,  vector< vector <double> >  coeff, DAobject &Ax2,  DAobject &
         DYIAXDX2,  DAobject &DZIAXDX2, const double &V)
669  {
670      DAobject Ax;
671      int ss= int((s+0.35)/0.01);
672  #pragma omp parallel shared(Ax)
673      {
674  #pragma omp for schedule(dynamic)
675          for (int x=0; x<6; x++) {
676              for (int y=0; y<6; y++) {
                     DAobject Ax_p;
                     DAobject tmp=Power(DAobject(1),x)*Power(DAobject(3),y);
679                  Ax_p=Ax_p+(tmp*coeff[ss][x+6*y]);
680  #pragma omp critical
681                  {
682                      Ax=Ax+Ax_p;
683                  }
684                  }
685          }
686      }
687      DAobject TimePart =Sin(DAobject(5)*omega/c)*cos(omega*(s+0.35)/c+phi0)+Cos(DAobject(5)*omega/c)*sin(omega*(s+0.35)/c+
             phi0);
688  Ax=Ax*TimePart*norm*V*(1.e+6*1.6e-19/omega/p0);
689  Ax2=MAPAPPLY(Ax, X);
690
```

Protection from multiple threads writing on the same block of memory at the same time

# openMP - forks

Declaration of parallel section

```
820 void H4( vector<DAobject> &X, double &s, const double &ds,const vector< vector <double> >  coeffy, const double &V)
821 {
822     DAobject DXIAYDY2, DZIAYDY2, AY2;
823 AYFIELD( X, s, coeffy, AY2,  DXIAYDY2, DZIAYDY2,V);
824 #pragma omp parallel
825     {
826 #pragma omp sections
827         {
828 #pragma omp section
829             X[1]=X[1]-DXIAYDY2;
830 #pragma omp section
831             X[3]=X[3]-AY2;
832 #pragma omp section
833             X[5]=X[5]-DZIAYDY2;
834         }
835     }
836   DAobject  DP = X[5]*(-1.)+Power(X[5],2)-Power(X[5],3)+Power(X[5],4)-Power(X[5],5)+1.;
837     X[2]=X[2]+X[3]*DP*ds/8.;
838     X[4]=X[4]-DP*DP*X[3]*X[3]*ds/8./2.;
```

Declaration of forks

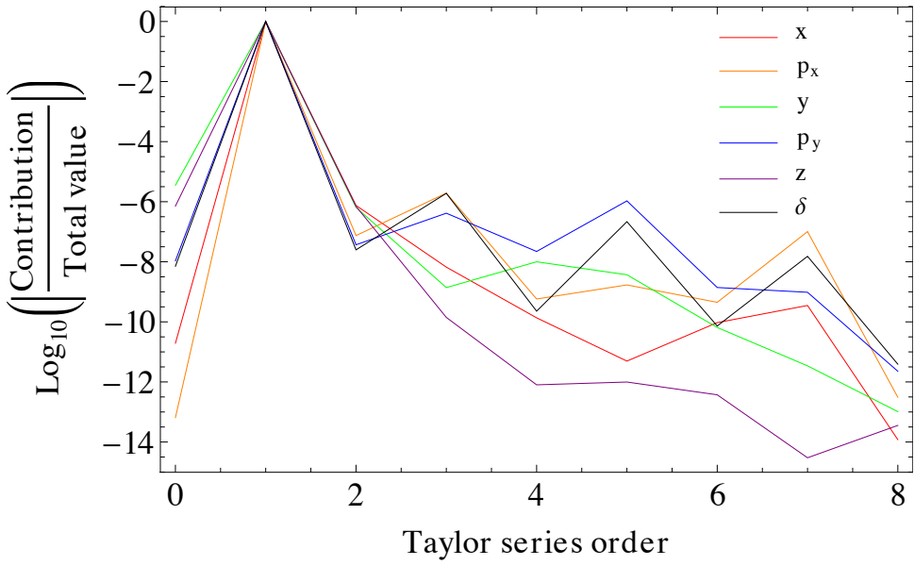
Independent calculations

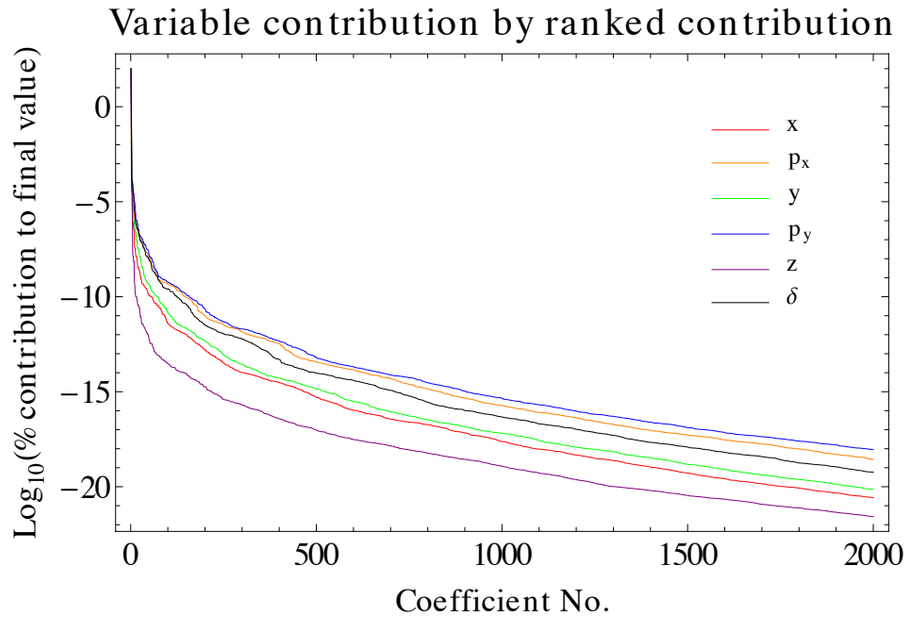
# Results

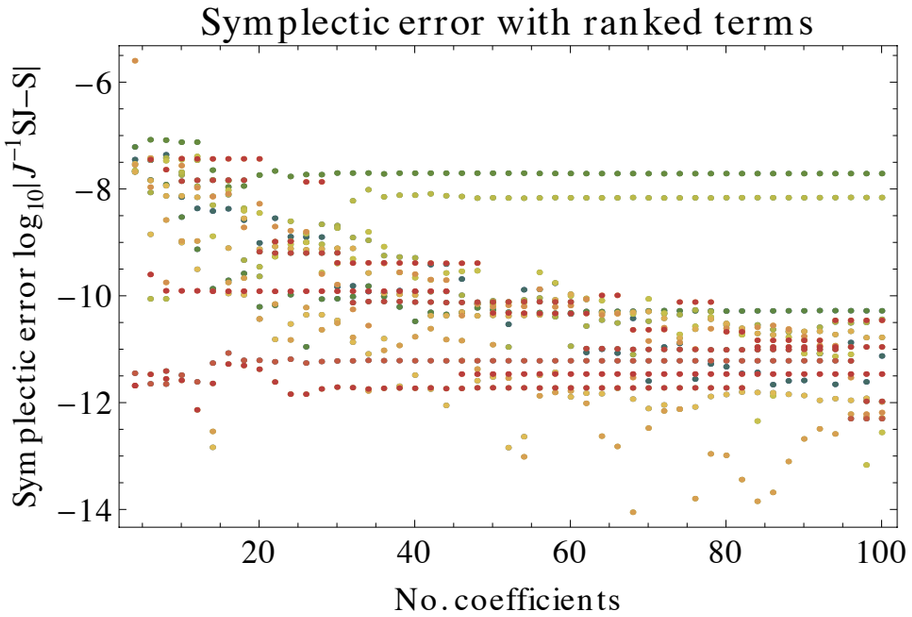
- Code is easy to edit and runs without leaks or warnings.
- Approximate speed up about 12 times when given access to 48 cores, reducing 600 cpu hours calculation down to 2 days.
- This then allows the possibility of going to higher orders or higher number of variables with the intention of avoiding the known symplecticity issues of lower order truncations.

# Results

- We can produce very higher order maps with relative ease currently highest map produced is 7 variables, 12th order truncation with 50388 terms in each series.
- This allows the Taylor series error to fall to similar to that of numerical precision.
- Even though that many terms would be ineffective to use in a tracking code it is possible to wait terms by the contribution to a particle at the edge of a bunch.



# Conclusion

- The use of STL and openMP allow for a highly optimized and hpc differential algebra application.
- openMP allows for the code to run on any machine with any number of cores.
- openMP allows problems which are on the face of it linear to be parallelized.