Speedup and quality up with Ada tasking

Solving polynomial systems faster and better

on multicore computers with PHCpack

Jan Verschelde

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
http://www.math.uic.edu/~jan
jan@math.uic.edu

Ada devroom, FOSDEM 2014, 1-2 February, Brussels, Belgium

Outline

1. Problem Statement
   - accurate numerical computations
   - motivating example: solving a triangular linear system
   - cost overhead of double double and quad double arithmetic

2. Solving with Tasking
   - coarse grain: running a pleasingly parallel computation
   - fine grain: one Newton step in parallel
   - multithreaded linear system solving

problem statement

We apply numerical methods on problems of growing size ...

- faster computers allow us to handle larger problems, but
- larger problems lead to more propagation of roundoff.

Is double precision still enough for accurate results? *No!*

Numerical polynomial system solving applies Newton's method.

Newton's method in several variables requires:

- evaluation and differentiation of the polynomials; and
- solution of a linear system to update the approximation.

solving polynomial systems with PHCpack

PHCpack is a package for Polynomial Homotopy Continuation.
ACM Transactions on Mathematical Software achived version 1.0 (Ada 83) as Algorithm 795, vol. 25, no. 2, pages 251–276, 1999.
`phc -b` computes all isolated solutions of a polynomial system.

Version 2.0 was rewritten using concepts of Ada 95
and extended with arbitrary multiprecision arithmetic.

Versions 2.1, 2.2, and 2.3 provided parallel path trackers with MPI (message passing), deflation methods for isolated singularities, and tools for positive dimensional solution sets.

Multitasking support since version 2.3.45 (2009-05-15)
with the GNAT GPL 2009 edition of the GNU-Ada compiler,
integrated quad double arithmetic since version 2.3.55 (2010-05-16).
PHCpack is an optional component of Sage.

Current version, since 2.3.84 (2013-09-26) at
`https://github.com/janverschelde/PHCpack`.

quad double precision

A quad double is an unevaluated sum of 4 doubles,
improves working precision from 2.2×10^{-16} to 2.4×10^{-63}.

- Y. Hida, X.S. Li, and D.H. Bailey: **Algorithms for quad-double precision floating point arithmetic.** In the *15th IEEE Symposium on Computer Arithmetic*, pages 155–162. IEEE, 2001. Software at `http://crd.lbl.gov/~dhbailey/mpdist/qd-2.3.9.tar.gz`.

Predictable overhead: working with `double double` is of the same cost as working with complex numbers. Simple memory management.

The QD library has been ported to the GPU by

- M. Lu, B. He, and Q. Luo: **Supporting extended precision on graphics processors.** In the *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010)*, pages 19–26, 2010.
  Software at `http://code.google.com/p/gpuprec/`.

personal supercomputers

12-core computer with NVDIA Tesla C2050:

- HP Z800 workstation running Red Hat Enterprise Linux 6.4
  The CPU is an Intel Xeon X5690 at 3.47 Ghz.
  There are two such 6-core CPUs: 12 cores.
- The processor clock of the NVIDIA Tesla C2050 Computing Processor runs at 1147 Mhz. The graphics card has 14 multiprocessors, each with 32 cores, for a total of 448 cores.

16-core computer with NVIDIA Tesla K20C:

- Microway RHEL workstation with Intel Xeon E5-2670 at 2.6 Ghz.
  There are two 8-core processors: 16 cores.
- The NVIDIA Tesla K20C has 2,496 cores (13×192) at a clock speed of 706 Mhz. The peak double precision performance of 1.17 teraflops is twice of that of the C2050.

The focus today is on multicore shared memory programming.

motivation: solving a trangular linear system

$$A\mathbf{x} = \mathbf{b}: \quad \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} & a_{1,n} \\ 0 & a_{2,2} & \cdots & a_{2,n-1} & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ 0 & 0 & \cdots & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

$$\mathbf{x} = A^{-1}\mathbf{b}: \begin{cases} x_n &= b_n/a_{n,n} \\ x_{n-1} &= (b_{n-1} - a_{n-1,n}x_n)/a_{n-1,n-1} \\ & \vdots \\ x_2 &= (b_2 - a_{2,3}x_3 - \cdots - a_{2,n-1}x_{n-1} - a_{2,n}x_n)/a_{2,2} \\ x_1 &= (b_1 - a_{1,2}x_2 - \cdots - a_{1,n-1}x_{n-1} - a_{1,n}x_n)/a_{1,1} \end{cases}$$

back substitution in double precision arithmetic

Matrix A with random elements: $|a_{i,j}| \leq 1$.
Exact solution: $\mathbf{x} = (1, 1, \ldots, 1)$, with $\mathbf{b}$ computed as $\mathbf{b} = A\mathbf{x}$.

dim	first component	error
8	1.00000000000000E+00	1.110E-16
16	1.00000000000006E+00	5.795E-14
24	1.00000000000650E+00	6.501E-12
32	9.99999513489745E-01	4.865E-07
40	9.99999112010784E-01	8.880E-07
48	1.00000025727134E+00	2.573E-07
56	1.31462587863753E+00	3.146E-01
64	2.14032668329910E+00	1.140E+00
72	-8.30854878370287E+06	8.309E+06
80	1.57365774064379E+02	1.564E+02

back substitution in double double arithmetic

Matrix A with random elements: $|a_{i,j}| \leq 1$.
Exact solution: $\mathbf{x} = (1, 1, \ldots, 1)$, with $\mathbf{b}$ computed as $\mathbf{b} = A\mathbf{x}$.

dim	first component	error
80	1.00000000000000000000000000000000E+00	0.000E+00
160	1.00000000000000000000000000000000E+00	0.000E+00
240	1.00000000000000000000000000000000E+00	0.000E+00
320	1.00000000000000000000000000000000E+00	0.000E+00
400	1.00000000000000000000000000000000E+00	0.000E+00
480	1.00000000000000000000000000000000E+00	0.000E+00
560	1.00000000000000000000000000000000E+00	0.000E+00
640	1.00000000000000000000000000000000E+00	0.000E+00
720	1.00000000000000000000000000000000E+00	0.000E+00
800	1.00000000000000000000000000000000E+00	0.000E+00

numerical interpretation

With double precision, results became unreliable already at $n = 50$.
In double double arithmetic, accuracy was maintained for large n.

Numerical conditioning measures how sensitive the problem is to numerical representation and roundoff errors.
A condition number gives a magnification factor for numerical errors.

Example (in *Matrix Computations* by Golub & Van Loan): $\kappa_\infty(A) = n2^{n-1}$ for an upper triangular $A = (a_{i,j})$ with $a_{i,i} = 1$, $a_{i,j} = 0$ if $i > j$, $a_{i,j} = -1$ if $i < j$.

Solving upper triangular systems is needed in Newton's method after a LU or QR factorization on the matrix of partial derivatives and for general polynomial systems we cannot make assumptions on the shape or structure of the matrix.

cost overhead of arithmetic

Solve 100-by-100 system 1000 times with LU factorization:

type of arithmetic	user CPU seconds
`double real`	1s 136ms
`double complex`	10s 603ms
`double double real`	10s 507ms
`double double complex`	1m 19s 908ms
`quad double real`	1m 30s 628ms
`quad double complex`	11m 16s 634ms

Fully optimized Ada code on one core of 3.47 Ghz Intel Xeon.

Overhead of complex arithmetic: `10.603/1.136 = 9.334`, `79.908/10.507 = 7.605`, `676.634/90.628 = 7.466`.

Overhead of double double complex: `79.908/10.603 = 7.536`.

Overhead of quad double complex: `676.634/79.908 = 8.467`.

speedup and quality up

Selim G. Akl, Journal of Supercomputing, 29, 89-111, 2004

How much **faster** *if we can use p cores?*

Let T_p be the time on p cores, then $\text{speedup} = \dfrac{T_1}{T_p} \to p$,

keeping the working precision fixed.

How much **better** *if we can use p cores?* Keeping the time fixed,

let Q_p be the quality on p cores, then $\text{quality up} = \dfrac{Q_p}{Q_1} \to p$.

Confusing working precision with accuracy (okay if well conditioned):

$$\text{quality up} = \frac{Q_p}{Q_1} = \frac{\#\text{ decimal places with } p \text{ cores}}{\#\text{ decimal places with 1 core}}$$

Assuming constant speedup and Q_p/Q_1 is linear in p,
then we can rescale when computing time is not constant.

solving polynomial systems by homotopy continuation

On input is a polynomial system $f(\mathbf{x}) = \mathbf{0}$.

A homotopy is a family of systems:

$$h(\mathbf{x}, t) = (1 - t)g(\mathbf{x}) + t\, f(\mathbf{x}) = \mathbf{0}.$$

At $t = 1$, we have the system $f(\mathbf{x}) = \mathbf{0}$ we want to solve.
At $t = 0$, we have a good system $g(\mathbf{x}) = \mathbf{0}$:

- solutions are known or easier to solve; and
- all solutions of $g(\mathbf{x}) = \mathbf{0}$ are regular.

Tracking all solution paths is pleasingly parallel,
although not every path requires the same amount of work.

starting worker tasks

procedure `Workers` is instantiated with a `Job` procedure, executing code based on the `id` number.

```
procedure Workers ( n : in natural ) is
   task type Worker ( id,n : natural );
   task body Worker is
   begin
     Job(id,n);
   end Worker;
   procedure Launch_Workers ( i,n : in natural ) is
     w : Worker(i,n);
   begin
     if i < n
      then Launch_Workers(i+1,n);
     end if;
   end Launch_Workers;
begin
   Launch_Workers(1,n);
end Workers;
```

managing job queues

On input is a list of start solutions.
The job queue is then the corresponding list of pointers:
each job requires the application of predictor-corrector methods
starting at the known solutions and ending at the desired solutions.

Path tracking on a list of solutions is a pleasingly parallel computation:
no communication overhead during the path tracking.

Management of the job queue:

1. an idle worker requests access to the next pointer in the queue;
2. once given access, the worker takes the job and becomes busy;
3. at the end of path, pointer to the solution of the target system.

Dynamic load balancing works well in this way.

Source of inspiration: Gem #81: GNAT Semaphores, at
`http://www.adacore.com/adaanswers/gems/gem-81`

running a pleasingly parallel computation

In the file `cyclic7` is a benchmark polynomial system.
We measure the wall clock time:

```
$ time phc -b cyclic7 /tmp/cyclic7.phc_t1

real    0m15.880s
user    0m15.744s
sys     0m0.007s
$ time phc -b -t12 cyclic7 /tmp/cyclic7.phc_t12

real    0m1.495s
user    0m13.786s
sys     0m0.010s
[jan@dezon Demo]$
```

On 12 cores, the speedup is `15.880/1.495 = 10.622`.

one Newton step in parallel

Often, we have only a couple of difficult solution paths.

The computational work in one Newton steps is in

1. the evaluation and differentiation of all polynomials:
   - arrange as multiplication of coefficient matrix with the evaluated vector of all monomials; or
   - multiply common factors (computed with a table of power products) with products of variables (the Speelpenning product), applying reverse mode of algorithmic differentiation
2. the solving of a linear system:
   - LU following the classical LINPACK routines; or
   - QR with the modified Gram-Schmidt method.

multithreaded LU factorization

Routines in PHCpack to solve linear systems are based on `ZGEFA` and `ZGESL` of LINPACK.

The multithreaded version of LU factorization does pivoting, synchronizing jobs with busy flags
and a column counter updated by first thread.

For good computational results for our first multithreaded implementation, the dimension needs to be around 80.

Because LU is $O(n^3)$, backsubstitution is $O(n^2)$, and $n \gg p$, multithreaded LU still dominates the total cost.

speedup and quality up for multithreaded LU

1000 times LU factorization of 80-by-80 matrix (8-core MacPro):

double double complex

#tasks	real	user	sys	speedup
1	1m 8.173s	1m 8.074s	0.131s	1
2	0m 36.712s	1m 13.061s	0.249s	1.857
4	0m 21.565s	1m 25.035s	0.455s	3.161
8	0m 20.986s	1m 42.156s	2.270s	3.248

quad double complex

#tasks	real	user	sys	speedup
1	10m 12.216s	10m 11.900s	0.311s	1
2	5m 12.753s	10m 24.774s	0.477s	1.958
4	2m 42.653s	10m 48.795s	0.699s	3.764
8	1m 33.234s	12m 17.653s	1.930s	6.566

Acceptable speedups with quad doubles. Quality up: with 8 cores, less than twice the time to double accuracy.

the quality up factor

Compare `68.173s` (1 core in dd) with `93.234s` (8 cores in qd).
With 8 cores, less than twice the time to double accuracy.

The speedup is close to optimal: how many cores would we need to reduce the calculation with quad doubles to `68.173s`?

$$\frac{93.234}{68.173} \times 8 = 10.941 \Rightarrow 11 \text{ cores}$$

Denote $y(p) = Q_p/Q_1$ and assume $y(p)$ is linear in p.

We have $y(1) = 1$ and $y(11) = 2$, so we interpolate:

$$y(p) - y(1) = \frac{y(11) - y(1)}{11 - 1}(p - 1).$$

and the quality up factor is $y(8) = 1 + \frac{7}{10} = 1.7$.