Efficient computations of initial-value problems involving fractional derivatives

Xing Cai & Wei Zhang

Simula Research Laboratory & University of Oslo

Ifi, UiO, November 23, 2011

Background and motivation

- Kai Diethelm, *An efficient parallel algorithm for the numerical solution of fractional differential equations*, Fractional Calculus & Applied Analysis, 14(3), pp. 475–490, 2011.
  - The very first(?) paper on using parallel computing for solving fractional differential equations
  - Interesting but somewhat unusual parallelization strategy
- We want a more straightforward parallelization
- We also want good performance, both serial and parallel

The mathematical problem

An initial-value problem involving fractional derivative:

$$D_*^\alpha y(x) = f(x, y(x))$$

where $\alpha > 0$ and initial conditions are

$$y^{(k)}(0) = y_0^{(k)} \quad (k = 0, 1, \ldots, \lceil \alpha \rceil - 1)$$

The Caputo differential operator D_*^α is defined as

$$D_*^\alpha y := J^{\lceil \alpha \rceil - \alpha} D^{\lceil \alpha \rceil} y$$

where the Riemann-Liouville integral operator J^μ is defined as

$$J^\mu y(x) = \frac{1}{\Gamma(\mu)} \int_0^x (x - t)^{\mu - 1} y(t) \, \mathrm{d}t$$

Fractional Adams-Bashforth-Moulton method

- Solving $D_*^\alpha y(x) = f(y(x))$ by time-stepping
- When $y_0, y_1, \ldots, y_j$ are known, $y_{j+1} \approx y((j+1)h)$, h being stepsize, can be computed by a predictor-corrector approach

$$y_{j+1}^P = \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{x_{j+1}^k}{k!} y_0^{(k)} + h^\alpha \sum_{k=0}^{j} b_{j-k} f(y_k)$$

$$b_\mu = \frac{(\mu+1)^\alpha - \mu^\alpha}{\Gamma(\alpha+1)}$$

$$y_{j+1} = \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{x_{j+1}^k}{k!} y_0^{(k)} + h^\alpha \left(c_j f(y_0) + \sum_{k=1}^{j} a_{j-k} f(y_k) + \frac{f(y_{j+1}^P)}{\Gamma(\alpha+2)} \right)$$

$$a_\mu = \frac{(\mu+2)^{\alpha+1} - 2(\mu+1)^{\alpha+1} + \mu^{\alpha+1}}{\Gamma(\alpha+2)}$$

$$c_\mu = \frac{\mu^{\alpha+1} - (\mu-\alpha)(\mu+1)^\alpha}{\Gamma(\alpha+2)}$$

Baseline implementation of ABM

- Three arrays:
  - array a — the a_μ coefficients
  - array b — the b_μ coefficients
  - array y — the numerical solution at discrete time levels
- If evaluation of $f(x_k, y_k)$ is not straightforward, an additional array can be used to store all the so-far computed $f(x_k, y_k)$ values
- Each c_μ coefficient is used only once, should thus be computed on-the-fly, no need for array storage
- Main code structure:
  - One outer `for`-loop of time integration, from $j = 0$ until end
  - Two inner `for`-loops: one for computing y^P_{j+1}, one for computing y_{j+1}

Computing the predictor y_{j+1}^P

$$y_{j+1}^P = \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{x_{j+1}^k}{k!} y_0^{(k)} + h^\alpha \sum_{k=0}^{j} b_{j-k} f(y_k)$$

```
sum_b = 0.0;

for (k=0; k<=j; k++)
  sum_b += b[j-k]*f(y[k]);

yp_jp1 = prefix + h_alpha*sum_b;
```

Computing y_{j+1}

$$y_{j+1} = \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{x_{j+1}^k}{k!} y_0^{(k)} + h^\alpha \left(c_j f(y_0) + \sum_{k=1}^{j} a_{j-k} f(y_k) + \frac{f(y_{j+1}^P)}{\Gamma(\alpha+2)} \right)$$

```
c_j = (pow(j,alpha+1)-(j-alpha)*pow(j+1,alpha))/gamma2;
sum_a = c_j*f(y[0]);

for (k=1; k<=j; k++)
  sum_a += a[j-k]*f(y[k]);

y[j+1] = prefix + h_alpha*(sum_a + f(yp_jp1)/gamma2);
```

Observations so far

- Majority of computation: weighted summations (involving $+$ and $*$)
- Total number of adds and mults: $2N^2$
- To compute y_{j+1}, all earlier computed values ($y_0, y_1, \ldots, y_j$) and pre-computed coefficients ($b_0, b_1, \ldots, b_j$), ($a_0, a_1, \ldots, a_{j-1}$) are needed
- Memory and cache bandwidths are important for speed
- Code optimization should aim to reduce the memory and cache traffic

Loop fusion

Merge the sum_b loop with the sum_a loop:

```
sum_b = b[j]*f(y[0]);
c_j = (pow(j,alpha+1)-(j-alpha)*pow(j+1,alpha))/gamma2;
sum_a = c_j*f(y[0]);

for (k=1; k<=j; k++) {
  sum_b += b[j-k]*f(y[k]);
  sum_a += a[j-k]*f(y[k]);
}

yp_jp1 = prefix + h_alpha*sum_b;
y[j+1] = prefix + h_alpha*(sum_a + f(yp_jp1)/gamma2);
```

In comparison with the baseline implementation, the memory and/or cache traffic related to $y_0, y_1, \ldots, y_j$ is halved.

Temporal blocking (via loop unrolling)

Compute y_{j+1} and y_{j+2} (almost) simultaneously, to decrease data traffic

```
sum_b = b[j]*f(y[0]);
c_j = (pow(j,alpha+1)-(j-alpha)*pow(j+1,alpha))/gamma2;
sum_a = c_j*f(y[0]);

sum_b1 = b[j+1]*f(y[0]);
c_jp1 = (pow(j+1,alpha+1)-(j+1-alpha)*pow(j+2,alpha))/gamma2;
sum_a1 = c_jp1*f(y[0]);

for (k=1; k<=j; k++) {
  sum_b += b[j-k]*f(y[k]);
  sum_a += a[j-k]*f(y[k]);
  sum_b1 += b[j+1-k]*f(y[k]);
  sum_a1 += a[j+1-k]*f(y[k]);
}

yp_jp1 = prefix + h_alpha*sum_b;
y[j+1] = prefix + h_alpha*(sum_a + f(yp_jp1)/gamma2);

yp_jp1p1 = prefix + h_alpha*(sum_b1 + b[0]*f(y[j+1]));
y[j+2] = prefix + h_alpha*(sum_a1 + a[0]*f(y[j+1]) + f(yp_jp1p1)/gamma
```

Now, 5 reads from L1 cache to register, 8 floating-point operations.
Data traffic from main memory is halved.

Improving cache reuse

More loop unrolling: after the “forward” k-loop, we can add a “backward” k-loop

```
sum_a = sum_b = sum_a1 = sum_b1 = 0.0;

for (k=j; k>=1; k--) {
  sum_b += b[j-k]*f(y[k]);
  sum_a += a[j-k]*f(y[k]);
  sum_b1 += b[j+1-k]*f(y[k]);
  sum_a1 += a[j+1-k]*f(y[k]);
}

yp_jp1 = prefix + h_alpha*(b[j]*f(y[0]) + sum_b);
c_j = (pow(j,alpha+1)-(j-alpha)*pow(j+1,alpha))/gamma2;
y[j+1] = prefix + h_alpha*(c_j*f(y[0]) + sum_a + f(yp_jp1)/gamma2);

yp_jp1p1 = prefix + h_alpha*(b[j+1]*f(y[0]) + sum_b1 + b[0]*f(y[j+1]))
c_jp1 = (pow(j+1,alpha+1)-(j+1-alpha)*pow(j+2,alpha))/gamma2;
y[j+2] = prefix+h_alpha*(c_jp1*f(y[0])+sum_a1+a[0]*f(y[j+1])+f(yp_jp1p
```

Cache temporal locality is improved for all three arrays: a, b and y.

Serial single-core performance (GFLOPs)

Double-precision computations, Xeon E5420, 2.5 GHz, 6MB L2 cache

Problem size	Baseline	Loop fusion	Temp. blocking	For-Backward
$N = 10^5$	1.59	2.25	2.85	3.04
$N = 2 \times 10^5$	1.57	2.23	2.80	3.04
$N = 4 \times 10^5$	0.93	1.04	1.87	2.54
$N = 10^6$	0.53	0.65	1.28	1.64

Double-precision computations, Xeon E5504, 2.0 GHz, 4MB L3 cache

Problem size	Baseline	Loop fusion	Temp. blocking	For-Backward
$N = 10^5$	1.21	1.68	2.21	2.34
$N = 2 \times 10^5$	1.17	1.59	2.16	2.36
$N = 4 \times 10^5$	1.04	1.43	2.07	2.28
$N = 10^6$	0.97	1.37	2.03	2.19

MPI Parallelization

Idea: let P MPI processes divide the work of sum_a and sum_b

```
start_k = my_rank*j/num_procs+1;
stop_k = (my_rank+1)*j/num_procs;

for (k=start_k; k<=stop_k; k++) {
  sum_b += b[j-k]*f(y[k]);
  sum_a += a[j-k]*f(y[k]);
  sum_b1 += b[j+1-k]*f(y[k]);
  sum_a1 += a[j+1-k]*f(y[k]);
}

sendbuf[0] = sum_b;  sendbuf[1] = sum_a;
sendbuf[2] = sum_b1; sendbuf[3] = sum_a1;
MPI_Allreduce (sendbuf, recvbuf, 4, MPI_type, MPI_SUM, MPI_COMM_WORLD)
sum_b  = recvbuf[0]; sum_a  = recvbuf[1];
sum_b1 = recvbuf[2]; sum_a1 = recvbuf[3];

yp_jp1 = prefix + h_alpha*sum_b;
y[j+1] = prefix + h_alpha*(sum_a + f(yp_jp1)/gamma2);

yp_jp1p1 = prefix + h_alpha*(sum_b1 + b[0]*f(y[j+1]));
y[j+2] = prefix + h_alpha*(sum_a1 + a[0]*f(y[j+1])
                           + f(yp_jp1p1)/gamma2);
```

Parallel multi-core performance

	Our result Xeon E5420 2.5GHz		*Our result* Xeon E5504 2.0GHz		*Diethelm's paper* Xeon W5580 3.2GHz	
P	Time	Speedup	Time	Speedup	Time	Speedup
1	1220.15	N/A	914.40	N/A	1186.54	N/A
2	534.30	2.28	498.80	1.83	637.71	1.86
4	182.76	6.68	302.13	3.03	477.98	2.48
8	106.09	11.50	188.71	4.85	338.62	3.50

$N = 10^6$, GNU C compiler with option `-O3 -ffast-math`

Concluding remarks

- Numerical solution of fractional differential equations can potentially be huge-scale (both computing-wise and memory-wise)
- Parallel computing is a useful tool
- Good performance relies on efficient usage of cache and memory
- There is a complex interplay between cache size, cache bandwidth, memory bandwidth, peak floating-point capability and number of CPU cores used.
- More investigations are needed!