Parallel Numerical Integration

1. Numerical Integration
   - an application of domain decomposition
   - quad double arithmetic
   - Romberg integration
2. Parallel Numerical Integration
   - using OpenMP
   - using the Intel TBB: `parallel_reduce`

MCS 572 Lecture 13
Introduction to Supercomputing
Jan Verschelde, 21 September 2016

Parallel Numerical Integration

1. Numerical Integration
   - an application of domain decomposition
   - quad double arithmetic
   - Romberg integration

2. Parallel Numerical Integration
   - using OpenMP
   - using the Intel TBB: `parallel_reduce`

domain decomposition for integration

Let $a < b$ and consider $a = c_0 < c_1 < \cdots < c_{p-2} < c_{p-1} = b$, then:

$$\int_a^b f(x)dx = \sum_{k=1}^{p} \int_{c_{k-1}}^{c_k} f(x)dx.$$

We have p subintervals of $[a, b]$ and on each subinterval $[c_{k-1}, c_k]$ we apply a quadrature formula (weighted sum of function values):

$$\int_{c_{k-1}}^{c_k} f(x)dx \approx \sum_{j=1}^{n} w_j f(x_j)$$

where the weights w_j correspond to points $x_j \in [c_{k-1}, c_k]$.

multiple integrals

Let the domain D be partitioned as $\bigcup_{i=1}^{n} \Delta_i$:

$$\int_D f(x_1, x_2) dx_1 dx_2 = \sum_{k=1}^{n} \int_{\Delta_k} f(x_1, x_2) dx_1 dx_2.$$

For a triangle Δ, an approximation of the integral of f over Δ is to take the volume between the plane spanned by the function values at the corners of Δ the $x_1 x_2$-plane.

Finer domain decompositions of D lead to more triangles Δ, more function evaluations, and more accurate approximations.

pleasing parallel computations

Like Monte Carlo simulation,
numerical integration is pleasingly parallel:

- Function evaluations can be computed independently from each other.

  No communication between processors needed once the subdomains have been distributed.

- The size of all communication is small:
  - on input: definition of subdomain,
  - on return: one weighted sum of function values.

Parallel Numerical Integration

1. Numerical Integration
   - an application of domain decomposition
   - quad double arithmetic
   - Romberg integration

2. Parallel Numerical Integration
   - using OpenMP
   - using the Intel TBB: `parallel_reduce`

quad double arithmetic

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 *15th IEEE Symposium on Computer Arithmetic* pages 155–162. IEEE, 2001. Software at
`http://crd.lbl.gov/~dhbailey/mpdist/qd-2.3.17.tar.gz`.

A quad double builds on `double double`, some features:

- The least significant part of a `double double` can be interpreted as a compensation for the roundoff error.
- Predictable overhead: working with `double double` is of the same cost as working with complex numbers.

operator overloading in C++

```
#include <iostream>
#include <iomanip>
#include <qd/qd_real.h>
using namespace std;

int main ( void )
{
   qd_real q("2");
   cout << setprecision(64) << q << endl;
   for(int i=0; i<8; i++)
   {
      qd_real dq = (q*q - 2.0)/(2.0*q);
      q = q - dq; cout << q << endl;
   }
   cout << scientific << setprecision(4);
   cout << "residual : " << q*q - 2.0 << endl;
   return 0;
}
```

compiling with a makefile

The makefile contains the entry:

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib

qd4sqrt2:
        g++ -I$(QD_ROOT)/include qd4sqrt2.cpp \
            $(QD_LIB)/libqd.a \
            -o /tmp/qd4sqrt2
```

Compiling at the command prompt `$`:

```
$ make qd4sqrt2
g++ -I/usr/local/qd-2.3.17/include qd4sqrt2.cpp \
            /usr/local/lib/libqd.a \
            -o /tmp/qd4sqrt2
```

running the code

```
$ /tmp/qd4sqrt2
2.0000000000000000000000000000000000000000000000000000000000000e+00
1.5000000000000000000000000000000000000000000000000000000000000e+00
1.4166666666666666666666666666666666666666666666666666666666667e+00
1.4142156862745098039215686274509803921568627450980392156862745098e+00
1.4142135623746899106262955788901349101165596221157440445849050192e+00
1.4142135623730950488016896235025302436149819257761974284982894987e+00
1.4142135623730950488016887242096980785696718753772340015610131332e+00
1.4142135623730950488016887242096980785696718753769480731766797380e+00
1.4142135623730950488016887242096980785696718753769480731766797380e+00
residual : 0.0000e+00
$
```

Parallel Numerical Integration

1. Numerical Integration
   - an application of domain decomposition
   - quad double arithmetic
   - Romberg integration

2. Parallel Numerical Integration
   - using OpenMP
   - using the Intel TBB: `parallel_reduce`

approximating π

We approximate π via $\pi = \int_0^1 \frac{16x - 16}{x^4 - 2x^3 + 4x - 4} dx$:

1. Apply the composite Trapezoidal rule doubling in each step the number of subintervals of $[0, 1]$.

   Recycling the function evaluations, the next approximation requires as many function evaluations as in the previous step.

2. To accelerate the convergence, extrapolate on the errors:

$$T[i][j] = \frac{T[i][j-1]2^{2j} - T[i-1][j-1]}{2^{2j} - 1}, \quad T[i][0] = T\left(\frac{h}{2^i}\right).$$

where $T(h) = \frac{h}{2}(f(a) + f(b)) + h\sum_{k=1}^{n-1} f(a + kh)$, $\quad h = \frac{b-a}{n}$.

running the program

```
$ /tmp/romberg4piqd
Give n : 20
Trapezoidal rule :
2.0000000000000000000000000000000000000000000000000000000000000000000000e+00  -1.1e+00
2.8285714285714285714285714285714285714285714285714285714285714285714286e+00  -3.1e-01
3.0599849140217096656334342993859535399012339858804671938901787924e+00  -8.2e-02
3.1208799149782192753090608934264402720707593687963124445027369600e+00  -2.1e-02
3.1363921117379842496423260900707118017245006984692475608300990573e+00  -5.2e-03
3.1402910615222379836674014569717873479608997416673717997690893828e+00  -1.3e-03
3.1412671635291865246174301601168385100304285866900551637643442050e+00  -3.3e-04
3.1415112753058333508223495008044322351701491319859264242480853344e+00  -8.1e-05
3.1415723086580001130741506654555989016008550279324178879004173905e+00  -2.0e-05
3.1415875673342908067733429372872267595388334364564318736724256675e+00  -5.1e-06
3.1415913820245079343519112519075857885126411600435940845928207730e+00  -1.3e-06
3.1415923356983838054575966637759377218669127222763583195386129930e+00  -3.2e-07
3.1415925741169353735102088131178638490686580423772000985781988075e+00  -7.9e-08
3.1415926337215784280554756890689392085987403031281282606384169231e+00  -2.0e-08
3.1415926486227395143502815854605592727657670704677224154970568916e+00  -5.0e-09
3.1415926523480298060901422591278382431441533032555099184797306336e+00  -1.2e-09
3.1415926532793523802854924341737272725049748286771339948938222906e+00  -3.1e-10
3.1415926535121830239131040417347661807897041512988564129230237347e+00  -7.8e-11
3.1415926535703906848249303226263308523017737071795040403161676085e+00  -1.9e-11
3.1415926535849426000531946040370197046578880393737659063268947933e+00  -4.9e-12
```

running the program continued

```
Romberg integration :
2.000000000000000000000000000000000000000000000000000000000000e+00  -1.1e+00
3.104761904761904761904
```

the composite Trapezoidal rule in C++

```
vector<qd_real> comptrap
   ( qd_real f ( qd_real x ),
     qd_real a, qd_real b, int n )
{
   vector<qd_real> t(n);
   qd_real h = b - a;

   t[0] = (f(a) + f(b))*h/2;
   for(int i=1, m=1; i<n; i++, m=m*2)
   {
      h = h/2;
      t[i] = 0.0;
      for(int j=0; j<m; j++)
         t[i] += f(a+h+j*2*h);
      t[i] = t[i-1]/2 + h*t[i];
   }
   return t;
}
```

Romberg integration in C++

```
void romberg_extrapolation ( vector<qd_real>& t )
{
   int n = t.size();
   qd_real e[n][n];
   int m = 0;

   for(int i=0; i<n; i++) {
      e[i][0] = t[i];
      for(int j=1; j<n; j++) e[i][j] = 0.0;
   }
   for(int i=1; i<n; i++) {
      for(int j=1, m=2; j<=i; j++, m=m+2) {
         qd_real r = pow(2.0,m);
         e[i][j] = (r*e[i][j-1] - e[i-1][j-1])/(r-1);
      }
   }
   for(int i=1; i<n; i++) t[i] = e[i][i];
}
```

Parallel Numerical Integration

1. Numerical Integration
   - an application of domain decomposition
   - quad double arithmetic
   - Romberg integration

2. Parallel Numerical Integration
   - using OpenMP
   - using the Intel TBB: `parallel_reduce`

using 2 cores on `kepler`

```
$ time /tmp/romberg4piqd 20
...
elapsed time : 2.040 seconds

real    0m2.046s
user    0m2.042s
sys     0m0.000s
$
```

Using two threads with OpenMP (speed up: `2.046/1.052 = 1.945`)

```
$ time /tmp/romberg4piqd_omp 20
...
elapsed time : 2.080 seconds

real    0m1.052s
user    0m2.081s
sys     0m0.003s
$
```

code that needs to run in parallel

The most computational intensive stage is in
the computation of the composite Trapezoidal rule:

```
for(int i=1, m=1; i<n; i++, m=m*2)
{
   h = h/2;
   t[i] = 0.0;
   for(int j=0; j<m; j++)
      t[i] += f(a+h+j*2*h);
   t[i] = t[i-1]/2 + h*t[i];
}
```

All function evaluations in the `j` loop
can be computed independently from each other.

parallel code with OpenMP

```
int id,jstart,jstop;
qd_real val;
for(int i=1, m=1; i<n; i++, m=m*2)
{
   h = h/2;
   t[i] = 0.0;
   #pragma omp parallel private(id,jstart,jstop,val)
   {
      id = omp_get_thread_num();
      jstart = id*m/2;
      jstop = (id+1)*m/2;
      for(int j=jstart; j<jstop; j++)
         val += f(a+h+j*2*h);
      #pragma omp critical
         t[i] += val;
   }
   t[i] = t[i-1]/2 + h*t[i];
}
```

benefits of using OpenMP

The command

```
omp_set_num_threads(2);
```

is executed before `comptrap`,
the function with the composite Trapezoidal rule is called.

Benefits:

1. The threads are computing inside the `j` loop,
   inside the `i` loop of the function `comptrap`.
   $\Rightarrow$ OpenMP does not create, join, destroy all threads for every different value of `i`, reducing system time.
2. The threads must wait at the end of each loop to update the approximation for the integral and to proceed to the next `i`.
   $\Rightarrow$ OpenMP takes care of the synchronization of the threads.

defining the makefile

We have to tell `gcc` that we are using

- the OpenMP library: `gcc -fopenmp`; and
- the QD library, including headers and `libqd.a` file.

The file `makefile` contains the lines:

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib

romberg4piqd_omp:
        g++ -fopenmp -I$(QD_ROOT)/include \
            romberg4piqd_omp.cpp \
            $(QD_LIB)/libqd.a \
            -o romberg4piqd_omp
```

To build the executable, we type `make romberg4piqd_omp`.

Parallel Numerical Integration

1. Numerical Integration
   - an application of domain decomposition
   - quad double arithmetic
   - Romberg integration

2. Parallel Numerical Integration
   - using OpenMP
   - using the Intel TBB: `parallel_reduce`

the class `SumIntegers` for quad doubles

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <qd/qd_real.h>
#include "tbb/tbb.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_reduce.h"
#include "tbb/task_scheduler_init.h"

using namespace std;
using namespace tbb;

class SumQuadDoubles
{
   qd_real *data;
   public:
      qd_real sum;
      SumQuadDoubles ( qd_real *d ) : data(d), sum(0.0) {}
```

defining the parallelism

```
void operator()
   ( const blocked_range<size_t>& r )
{
   qd_real s = qd_real(sum[0],sum[1],sum[2],sum[3]);
   // must accumulate !
   qd_real *d = data;
   size_t end = r.end();
   for(size_t i=r.begin(); i != end; ++i)
      s += d[i];
   sum = qd_real(s[0],s[1],s[2],s[3]);
}
// the splitting constructor
SumQuadDoubles ( SumQuadDoubles& x, split ) :
   data(x.data), sum(0.0) {}
// the join method does the merge
void join ( const SumQuadDoubles& x ) { sum += x.sum; }
```

using the class

```
qd_real ParallelSum ( qd_real *x, size_t n )
{
   SumQuadDoubles S(x);

   parallel_reduce(blocked_range<size_t>(0,n), S);

   return S.sum;
}
```

In the `main`:

```
qd_real *d;
d = (qd_real*)calloc(n,sizeof(qd_real));
for(int i=0; i<n; i++) d[i] = qd_real((double)(i+1));

task_scheduler_init init(task_scheduler_init::automatic);
qd_real s = ParallelSum(d,n);
```

compiling with a makefile

To compile `parsumqd_tbb`, we have to tell `g++` to

- use the QD library (include headers, link `libqd.a`); and
- use the Intel TBB library (headers and `-ltbb`).

Therefore, the `makefile` must contain the following:

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib
TBB_ROOT=/usr/local/tbb40_233oss

parsumqd_tbb:
        g++ -I$(TBB_ROOT)/include \
            -I$(QD_ROOT)/include \
            parsumqd_tbb.cpp -o /tmp/parsumqd_tbb \
            $(QD_LIB)/libqd.a -L$(TBB_ROOT)/lib -ltbb
```

summing quad doubles

We sum as many as `n` numbers in an array of quad doubles, starting with 1,2,3... so the sum equals $n(n+1)/2$.

```
$ time /tmp/parsumqd_tbb 20160921
 S = 2.0323137786458100 0 ... omitted ... 000e+14
 T = 2.0323137786458100 0 ... omitted ... 000e+14

real    0m0.765s
user    0m8.231s
sys     0m0.146s
$
```

We estimate the speed up comparing user time to wall clock time:
`8.231/0.765 = 10.759.`

recommended reading

- Ali Yazici: **The Romberg-like Parallel Numerical Integration on a Cluster System**. In the proceedings of the 24th International Symposium on Computer and Information Sciences, ISCIS 2009, pages 686-691, IEEE 2009. Available to UIC via IEEE *Xplore*.
- David H. Bailey and Jonathan M. Borwein: **Highly Parallel, High-Precision Numerical Integration**. April 2008. Report LBNL-57491. Available at `http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/`.

Summary + Exercises

In the book of Wilkinson and Allen, numerical integration is in §4.2.2. The work stealing scheme for `parallel_reduce` is explained in the Intel Threading Building Blocks tutorial, in §3.3.

Exercises:

1. Make the OpenMP implementation of `romberg4piqd_omp.cpp` more general by prompting the user for a number of threads and then using those threads in the function `comptrap`.
   Compare the speed up for 2, 4, 8, and 16 threads.
2. Write an elaborate description on the thread creation and synchronization issues with Pthreads to achieve a parallel version of `romberg4piqd.cpp`.
3. Use the Intel Threading Building Blocks to write a parallel version of the composite trapezoidal rule in quad double arithmetic.