Parallel Numerical Integration

1 Numerical Integration

In numerical integration we consider the problem of approximating the definite integral of a function over a domain. By domain decomposition we naturally arrive at parallel algorithms.

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

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

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):

\[
\int_{c_{k-1}}^{c_k} f(x)dx \approx \sum_{j=1}^n w_j f(x_j) \tag{2}
\]

where the weights \( w_j \) correspond to points \( x_j \in [c_{k-1}, c_k] \). Let the domain \( D \) be partitioned as \( \bigcup_{i=1}^n \Delta_i \):

\[
\int_D f(x_1, x_2)dx_1dx_2 = \sum_{k=1}^n \int_{\Delta_k} f(x_1, x_2)dx_1dx_2. \tag{3}
\]

For a triangle \( \Delta \), an approximation of the integral of \( f \) over \( \Delta \) is to take the volume between the plane spanned by the function values at the corners of \( \Delta \) the \( x_1, x_2 \)-plane. Finer domain decompositions of \( D \) lead to more triangles \( \Delta \), more function evaluations, and more accurate approximations.

Like Monte Carlo simulation, numerical integration is pleasingly parallel. The 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 we have the definition of the subdomain, and on return is one weighted sum of function values.

To obtain highly accurate values when applying extrapolation on the trapezoidal rule (so-called Romberg integration), we use quad double arithmetic. A quad double is an unevaled sum of 4 doubles, improves working precision from \( 2.2 \times 10^{-16} \) to \( 2.4 \times 10^{-63} \). A quad double builds on the double double. 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. The QD library supports operator overloading in C++, as shown in the example code below.

```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) << "residual : " << q*q - 2.0 << endl;
    return 0;
}
```

Jan Verschelde
UIC, Dept of Math, Stat & CS
Lecture 13, page 1
Compiling and running the program could go as below.

$ g++ -I/usr/local/qd-2.3.9/include qd4sqrt2.cpp /usr/local/qd-2.3.9/src/libqd.a -o /tmp/qd4sqrt2
$ /tmp/qd4sqrt2

2.0000000000000000000000000000000000000000000000000000000000000000e+00
1.5000000000000000000000000000000000000000000000000000000000000000e+00
1.4166666666666666666666666666666666666666666666666666666666666666e+00
1.4125682745098039215666666666666666666666666666666666666666666666e+00
1.41213562374699106262955788901349101165596221157440445849050192e+00
1.4121356237309540880168692350230523614981925776197428498299487e+00
1.4121356237309540880168672409698078696718753772340015610131332e+00
1.41213562373095408801686724096980786967187537769480731766797380e+00
residual : 0.0000e+00

Returning to the problem of approximating $\pi$ we consider

$$\pi = \int_0^1 \frac{16x - 16}{x^4 - 2x^3 + 4x - 4} \, dx.$$  

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

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

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 produces the following approximations for $\pi$, improved with Romberg integration.

$ /tmp/romberg4piqd

Give n : 20

Trapezoidal rule :

<table>
<thead>
<tr>
<th>n</th>
<th>Approximation</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.0000000000</td>
<td>-1.1e+00</td>
</tr>
<tr>
<td>4</td>
<td>2.8285714286</td>
<td>-3.1e-01</td>
</tr>
<tr>
<td>8</td>
<td>3.0599964093</td>
<td>-8.2e-02</td>
</tr>
<tr>
<td>16</td>
<td>3.1208799149</td>
<td>-2.1e-02</td>
</tr>
<tr>
<td>32</td>
<td>3.1320879130</td>
<td>-5.2e-03</td>
</tr>
<tr>
<td>64</td>
<td>3.1402910615</td>
<td>-1.3e-03</td>
</tr>
<tr>
<td>128</td>
<td>3.1412671635</td>
<td>-3.3e-04</td>
</tr>
<tr>
<td>256</td>
<td>3.1415723086</td>
<td>-8.1e-05</td>
</tr>
<tr>
<td>512</td>
<td>3.1415875673</td>
<td>-2.0e-05</td>
</tr>
<tr>
<td>1024</td>
<td>3.1415926337</td>
<td>-3.2e-06</td>
</tr>
<tr>
<td>2048</td>
<td>3.1415926535</td>
<td>-7.9e-08</td>
</tr>
<tr>
<td>4096</td>
<td>3.1415926535</td>
<td>-1.9e-09</td>
</tr>
</tbody>
</table>

Romberg integration :

<table>
<thead>
<tr>
<th>n</th>
<th>Approximation</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.0000000000</td>
<td>-1.1e+00</td>
</tr>
<tr>
<td>4</td>
<td>2.8284270123</td>
<td>-3.1e-01</td>
</tr>
<tr>
<td>8</td>
<td>3.0597351586</td>
<td>-8.2e-02</td>
</tr>
<tr>
<td>16</td>
<td>3.1206207244</td>
<td>-2.1e-02</td>
</tr>
<tr>
<td>32</td>
<td>3.1320869342</td>
<td>-5.2e-03</td>
</tr>
<tr>
<td>64</td>
<td>3.1402686510</td>
<td>-1.3e-03</td>
</tr>
<tr>
<td>128</td>
<td>3.1412662521</td>
<td>-3.3e-04</td>
</tr>
<tr>
<td>256</td>
<td>3.1415717762</td>
<td>-8.1e-05</td>
</tr>
<tr>
<td>512</td>
<td>3.1415863300</td>
<td>-2.0e-05</td>
</tr>
<tr>
<td>1024</td>
<td>3.1415925741</td>
<td>-3.2e-06</td>
</tr>
<tr>
<td>2048</td>
<td>3.1415926535</td>
<td>-7.9e-08</td>
</tr>
</tbody>
</table>

Jan Verschelde  
UIC, Dept of Math, Stat & CS  
Lecture 13, page 2
The functions defining the composite Trapezoidal rule and Romberg integration in C++ are listed below.

```cpp
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;
}

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];
    }
    return;
}
```

elapsed time : 0.998 seconds

Jan Verschelde  UIC, Dept of Math, Stat & CS  Lecture 13, page 3
2 Parallel Numerical Integration

Running the OpenMP implementation using 2 cores on Mac OS X:

$ time /tmp/romberg4piqd 20
...
elapsed time : 0.992 seconds
real 0m0.997s
user 0m0.993s
sys 0m0.003s
$

Using two threads with OpenMP (speedup: $.997/.509 = 1.959)

$ time /tmp/romberg4piqd_omp 20
...
elapsed time : 0.980 seconds
real 0m0.509s
user 0m0.979s
sys 0m0.003s
$

The code that needs to run in parallel is the most computational intensive stage, which is in the computation of the composite Trapezoidal rule.

```c
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. The parallel code with OpenMP is listed below.

```c
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];
}
```
The command `omp_set_num_threads(2);` is executed before `comptrap`, the function with the composite Trapezoidal rule is called. The benefits of OpenMP are twofold. (1) The threads are computing inside the `j` loop, inside the `i` loop of the function `comptrap`. ⇒ 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`. ⇒ OpenMP takes care of the synchronization of the threads.

### 2.1 using the Intel Threading Building Blocks

The Intel TBB implementation provides an opportunity to illustrate the `parallel_reduce` construction. We start out by listing the class `SumIntegers`:

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

int ParallelSum ( int *x, size_t n )
{
    SumIntegers S(x);
    parallel_reduce(blocked_range<size_t>(0,n), S);
    return S.sum;
}
```

The `ParallelSum` is called in the main program as follows:

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

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

For our numerical integration, we sum as many as `n` numbers in an array of quad doubles, starting with 1,2,3... so the sum equals \( n(n + 1)/2 \).
|$ \text{time /tmp/parsumqd_tbb 200330002} \\
S = 2.006605495082500300 \ldots \text{omitted} \ldots 00e+16 \\
T = 2.006605495082500300 \ldots \text{omitted} \ldots 00e+16 \\
\text{real 0m6.050s} \\
\text{user 0m44.685s} \\
\text{sys 0m0.924s} \\
\text{Done on 12-core machine, the estimated speedup comparing user time to wall clock time is} \\
44.685/6.050 = 7.386. \\
\text{The work stealing scheme for} \ \text{parallel\_reduce} \ \text{is explained in the Intel Threading Building Blocks} \\
tutorial, \ \text{in §3.3.} \\

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

\text{References} \\
[1] \text{Y. Hida, X.S. Li, and D.H. Bailey.} \ \text{Algorithms for quad-double precision floating point arithmetic.} \ \text{In 15th IEEE Symposium on Computer Arithmetic} \ \text{pages 155–162. IEEE, 2001.} \\
[2] \text{A. Yazici.} \ \text{The Romberg-like Parallel Numerical Integration on a Cluster System.} \ \text{In the} \\
\text{proceedings of the 24th International Symposium on Computer and Information Sciences, ISCIS 2009,} \\
\text{pages 686-691, IEEE 2009. Available to UIC via IEEE Xplore.} \\
[3] \text{D.H. Bailey and J.M. Borwein.} \ \text{Highly Parallel, High-Precision Numerical Integration.} \ \text{April} \\