

# 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, 8 February 2012

# 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  $a = c_0 < c_1 < \dots < c_{p-2} < c_{p-1} = b$ , 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  $\Delta$ , 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.

# 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  $2.2 \times 10^{-16}$  to  $2.4 \times 10^{-63}$ .

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-2.3.9.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-2.3.9
```

```
qd4sqrt2:
```

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

```
$ make qd4sqrt2
```

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



# 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 $\pi$

We approximate  $\pi$  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:

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

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





## 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 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 (speed up:  $.997 / .509 = 1.959$ )

```
$ time /tmp/romberg4piqd_omp 20
...
elapsed time : 0.980 seconds

real    0m0.509s
user    0m0.979s
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`.  
⇒ 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.

# 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

```
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;
    }
}
```

## split and join methods

```
// 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;
}
```

## code in the main program

```
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);
```

## 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 200330002  
S = 2.006605495082500300 ... omitted ... 00e+16  
T = 2.006605495082500300 ... omitted ... 00e+16
```

```
real    0m6.050s  
user    0m44.685s  
sys     0m0.924s
```

Done on 12-core machine, estimate speed up comparing user time to wall clock time:  $44.685/6.050 = 7.386$ .

## 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, and 8 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.

Homework will be collected on Friday 10 February at noon.