

# ACM/CS 114

## Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

# Approximating $\text{Li}_2$ using a numerical quadrature

- ▶ the second homework assignment involved  $\text{Li}_2(z)$ , defined by

$$\text{Li}_2(z) := - \int_0^z dz' \frac{\log(1 - z')}{z'}$$

- ▶ the assignment asked for approximating this integral using a simple quadrature based on the mid-point rule

$$\text{Li}_2(z) \approx \text{Li}_2(z, N) := - \frac{z}{N} \sum_{n=0}^{N-1} \frac{\log(1 - z')}{z'} \bigg|_{z' = (n + \frac{1}{2}) \frac{z}{N}}$$

- ▶ three implementations
  - ▶ sequential: to get a feeling for how to convert the algorithm into a functioning program
  - ▶ parallel using threads: to walk through the parallelization steps and use `pthread`s to get better performance
  - ▶ parallel using MPI: to get a feel for how MPI-based programs solve the task partitioning problem
- ▶ let's walk through composing, building and running my solutions
  - ▶ on my desktop, on `mind-meld.cacr.caltech.edu`, and on `shc.cacr.caltech.edu`

# Sequential implementation - part 1

## ► the preamble

```
1 #include <getopt.h> // for getopt and friends
2 #include <cstdlib> // for atof
3 #include <cmath> // for the correct abs, log
4
5 #include <map>
6 #include <iostream>
7 #include <iomanip>
```

## ► quadrature using the midpoint rule to avoid the singularities

```
8 // dilog
9 double dilog(double z, long N) {
10     // initialize
11     double dx = z/N;
12     double x = dx/2;
13     double sum = 0;
14     // loop
15     for (long i=0; i < N; i++) {
16         sum += std::log(1-x)/x;
17         x += dx;
18     }
19     // return; don't forget the sign
20     return -dx * sum;
21 }
```

# Sequential implementation - part 2

- using the command line to set  $z$  and the number of subdivisions  $N$

```
23 // main program
24 int main(int argc, char* argv[]) {
25     // default values for the command line options
26     long N = 1000;
27     double z = 1.0;
28
29     // read the command line
30     int command;
31     while ((command = getopt(argc, argv, "z:N:")) != -1) {
32         switch (command) {
33             // get the argument of the dilogarithm
34             case 'z':
35                 z = atof(optarg);
36                 break;
37             // get the number of subdivisions
38             case 'N':
39                 N = (long) atof(optarg);
40                 break;
41         }
42     }
```

# Sequential implementation - part 3

## ► error checking and computation of the numerical integral

```
43  // error checking
44  // abort if N < 1
45  if (N < 1) {
46      std::cout
47          << "the number of subdivisions must be positive"
48          << std::endl;
49      return 0;
50  }
51
52  // abort for z > 1 to avoid dealing with the imaginary part
53  if (z > 1.0) {
54      std::cout << "math domain error: z > 1" << std::endl;
55      return 0;
56  }
57
58  // compute
59  double value = dilog(z, N);
```

# Sequential implementation - part 4

- computing the error and printing out the results

```
60 // build a database of the known dilogarithm values
61 const double pi = M_PI;
62 std::map<double, double> answers;
63 answers[1.0] = pi*pi/6;
64 answers[-1.0] = -pi*pi/12;
65
66 // print out the value
67 std::cout << "Li2(" << z << ")="
68     << std::setprecision(17) << std::endl
69     << " computed: " << value << std::endl;
70 // check whether we know the right answer
71 std::map<double,double>::const_iterator lookup = answers.find(z);
72 if (lookup != answers.end()) {
73     // and if we do, print it out
74     double exact = lookup->second;
75     std::cout << " exact: " << exact << std::endl;
76     // compute the approximation error and print it out
77     double error = std::abs(exact-value)/exact;
78     std::cout
79         << std::setiosflags(std::ios_base::scientific)
80         << " error: " << error << std::endl;
81 }
82
83 return 0;
84 }
```

# Building and running the sequential driver

```
1 #> g++ dilog_sequential.cc -o dilog_sequential
2 #> dilog_sequential -N 1e7 -z 1.0
3 Li2(1)=
4   computed: 1.6449340282186398
5   exact: 1.6449340668482264
6   error: 2.34839726522278546e-08
7 #> time dilog_sequential -N 1e9 -z 1.0
8 Li2(1)=
9   computed: 1.6449339414016682
10  exact: 1.6449340668482264
11  error: 7.62623625958871898e-08
12
13 real    0m19.885s
14 user    0m19.877s
15 sys     0m0.003s
16 #>
```

# Threaded implementation - part 1

## ► the preamble

```
1 #include <getopt.h> // for getopt and friends
2 #include <pthread.h>
3
4 #include <cstdio>
5 #include <cstdlib> // for atof
6 #include <cmath>
7
8 #include <map>
9 #include <iostream>
10 #include <iomanip>
```



# Threaded implementation - part 2

## ► private and shared data structures

```
12 // shared information
13 struct problem {
14     int workers;          // total number of threads
15     double dz;           // the width of each subdivision
16     double sum;          // storage for the partial computations
17
18     pthread_mutex_t lock; // mutex to control access to the sum
19 };
20
21 // thread specific information
22 struct context {
23     // thread info
24     int id;
25     pthread_t descriptor;
26     // the workload for this thread
27     long subdivisions; // number of subdivisions
28     double z_low;      // the lower limit of integration
29     double partial;    // record the partial sum computed by this thread
30     // the shared problem information
31     problem* info;
32 };
```

# Threaded implementation - part 3

## ► the coarse grain task

```
33 // worker
34 void* worker(void* arg) {
35     context* ctxt = (context *) arg;
36     // pull the problem information from the thread context
37     double dz = ctxt->info->dz;
38     double z = ctxt->z_low + dz/2;
39     // loop over the subdivisions assigned to this thread
40     double sum = 0.0;
41     for (long i=0; i < ctxt->subdivisions; i++) {
42         sum += std::log(1-z)/z;
43         z += dz;
44     }
45     // multiply by the width of each subdivision and adjust the sign
46     sum *= -dz;
47
48     // grab the lock
49     pthread_mutex_lock(&(ctxt->info->lock));
50     // store the result
51     ctxt->info->sum += sum;
52     // and release the lock
53     pthread_mutex_unlock(&(ctxt->info->lock));
54
55     // all done
56     return 0;
57 }
```

# Threaded implementation - part 4

- the task master – interface and allocation of storage

```
58 // driver
59 double dilog(double z, long N, int threads) {
60     // the width of each interval subdivision
61     const double dz = z/N;
62
63     // setup the problem context
64     problem info;
65     info.workers = threads;
66     info.dz = dz;
67     info.sum = 0.0;
68     pthread_mutex_init(&info.lock, 0);
69
70     // and an array to hold the thread contexts
71     context thr_info[threads];
72     // partition the number of subdivisions
73     long nominal_load = N/threads;
```

# Threaded implementation - part 5

- the task master – spawning the threads

```
75 // spawn the workers
76 for (int tid=0; tid<threads; tid++) {
77     // store the thread id
78     thr_info[tid].id = tid;
79     // point to the shared problem info
80     thr_info[tid].info = &info;
81
82     // compute the starting point of the partial integral
83     thr_info[tid].z_low = tid*nominal_load*dz;
84     // compute the number of subdivisions for this thread
85     if (tid == threads - 1) {
86         // the last thread gets the leftovers
87         thr_info[tid].subdivisions = N - tid*nominal_load;
88     } else {
89         thr_info[tid].subdivisions = nominal_load;
90     }
91
92     // create the thread
93     int status = pthread_create(
94         &(thr_info[tid].descriptor), 0, worker, &thr_info[tid]);
95     if (status) {
96         printf("error %d in pthread_create\n", status);
97     }
98 }
```

# Threaded implementation - part 6

- the task master – harvesting the threads and returning the result

```
99      // harvest the threads
100     for (int tid=0; tid<threads; tid++) {
101         pthread_join(thr_info[tid].descriptor, 0);
102     }
103
104     // all done
105     return info.sum;
106 }
```

# Threaded implementation - part 7

- ▶ the main program – reading the command line

```
107 // main program
108 int main(int argc, char* argv[]) {
109     // default values for the command line options
110     long N = 1000;
111     double z = 1.0;
112     int threads = 8;
113
114     // read the command line
115     int command;
116     while ((command = getopt(argc, argv, "z:N:t:")) != -1) {
117         switch (command) {
118             case 'z':
119                 // get the argument of the dilogarithm
120                 z = atof(optarg);
121                 break;
122             case 'N':
123                 // get the number of subdivisions
124                 N = (long) atof(optarg);
125                 break;
126             case 't':
127                 // get the number of threads
128                 threads = atoi(optarg);
129                 break;
130         }
131     }
```

# Threaded implementation - part 8

## ► error checking and the invocation of the task master

```
133 // error checking
134 // abort if N < 1
135 if (N < 1) {
136     std::cout
137         << "the number of subdivisions must be positive"
138         << std::endl;
139     return 0;
140 }
141
142 // abort for z > 1 to avoid dealing with the imaginary part
143 if (z > 1.0) {
144     std::cout << "math domain error: z > 1" << std::endl;
145     return 0;
146 }
147
148 // compute
149 double value = dilog(z, N, threads);
```

# Threaded implementation - part 9

- ▶ the task master – printing out the answers

```
151 // build a database of the known dilogarithm values
152 const double pi = M_PI;
153 std::map<double, double> answers;
154 answers[1.0] = pi*pi/6;
155 answers[-1.0] = -pi*pi/12;
156
157 // print out the value
158 std::cout << "Li2(" << z << ")="
159     << std::setprecision(17) << std::endl
160     << " computed: " << value << std::endl;
161 // check whether we know the right answer
162 std::map<double,double>::const_iterator lookup = answers.find(z);
163 if (lookup != answers.end()) {
164     // and if we do, print it out
165     double exact = lookup->second;
166     std::cout << " exact: " << exact << std::endl;
167     // compute the approximation error and print it out
168     double error = std::abs(exact-value)/exact;
169     std::cout
170         << std::setiosflags(std::ios_base::scientific)
171         << " error: " << error << std::endl;
172 }
173
174 return 0;
175 }
```



# Building and running the threaded driver

```
1 #> g++ dilog_threads.cc -o dilog_threads -pthread
2 #> dilog_threads -N 1e7 -z 1.0 -t 4
3 Li2(1)=
4   computed: 1.6449340301295035
5   exact: 1.6449340668482264
6   error: 2.23223068274304058e-08
7 #> time dilog_threads -N 1e9 -z 1.0 -t 8
8 Li2(1)=
9   computed: 1.6449340044883614
10  exact: 1.6449340668482264
11  error: 3.79102520315773892e-08
12
13 real    0m2.803s
14 user    0m20.693s
15 sys     0m0.006s
16 #>
```

# MPI implementation - part 1

## ► the preamble

```
1 #include <getopt.h> // for getopt and friends
2 #include <mpi.h>
3
4 #include <stdio>
5 #include <stdlib> // for atof
6 #include <cmath>
7
8 #include <map>
9 #include <iostream>
10 #include <iomanip>
```

# MPI implementation - part 2

## ► coarse grain task

```
12 double dilog(double zprime, long N, int id, int processes) {
13     // the width of each interval subdivision
14     const double dz = zprime/N;
15     // compute the starting point of the partial integral
16     const double z_low = id*zprime/processes;
17     // partition the number of subdivisions
18     long nominal_load = N/processes;
19     // the last process gets the leftovers
20     if (id == processes - 1) {
21         nominal_load = N - id*nominal_load;
22     }
23     // initialize the partial sum
24     double sum = 0.0;
25     double z = z_low + dz/2;
26     // loop over the subdivisions assigned to this thread
27     for (long i=0; i < nominal_load; i++) {
28         sum += std::log(1-z)/z;
29         z += dz;
30     }
31     // collect the partial answers from all the processes
32     double value;
33     MPI_Allreduce(
34         &sum, &value, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
35     // multiply by the width of each subdivision and adjust the sign
36     return -dz*value;
37 }
```

# MPI implementation - part 3

- the main program – setting up MPI

```
38 // main program
39 int main(int argc, char* argv[]) {
40     // initialize MPI
41     int status = MPI_Init(&argc, &argv);
42     if (status != MPI_SUCCESS) {
43         std::cout << "error in MPI_Init; aborting..." << std::endl;
44         return status;
45     }
46     // get process information from the world communicator
47     int id, processes;
48     MPI_Comm_rank(MPI_COMM_WORLD, &id);
49     MPI_Comm_size(MPI_COMM_WORLD, &processes);
```

# MPI implementation - part 4

## ► reading the command line

```
51 // default values for the command line options
52 long N = 1000;
53 double z = 1.0;
54 // read the command line
55 int command;
56 while ((command = getopt(argc, argv, "z:N:")) != -1) {
57     switch (command) {
58         case 'z':
59             // get the argument of the dilogarithm
60             z = atof(optarg);
61             break;
62         case 'N':
63             // get the number of subdivisions
64             N = (long) atof(optarg);
65             break;
66     }
67 }
```

# MPI implementation - part 5

## ► error checking and computation

```
68 // error checking
69 // abort if N < 1
70 if (N < 1) {
71     if (id == 0) {
72         std::cout
73             << "the number of subdivisions must be positive"
74             << std::endl;
75     }
76     MPI_Finalize();
77     return 0;
78 }
79 // abort for z > 1 to avoid dealing with the imaginary part
80 if (z > 1.0) {
81     if (id == 0) {
82         std::cout << "math domain error: z > 1" << std::endl;
83     }
84     MPI_Finalize();
85     return 0;
86 }
87 // compute
88 double value = dilog(z, N, id, processes);
89 if (id != 0) { // let all but processor 0 die
90     // shut down MPI
91     MPI_Finalize();
92     return 0;
93 }
```

# MPI implementation - part 6

## ► printing out the results

```
94 // build a database of the known dilogarithm values
95 const double pi = M_PI;
96 std::map<double, double> answers;
97 answers[1.0] = pi*pi/6;
98 answers[-1.0] = -pi*pi/12;
99
100 // print out the value
101 std::cout << "Li2(" << z << ")=" << std::setprecision(17) << std::endl;
102 std::cout << " computed: " << value << std::endl;
103 // check whether we know the right answer
104 std::map<double,double>::const_iterator lookup = answers.find(z);
105 if (lookup != answers.end()) {
106     // and if we do, print it out
107     double exact = lookup->second;
108     std::cout << " exact: " << exact << std::endl;
109     // compute the approximation error and print it out
110     double error = std::abs(exact-value)/exact;
111     std::cout
112         << std::setiosflags(std::ios_base::scientific)
113         << " error: " << error << std::endl;
114 }
115
116 // shut down MPI
117 MPI_Finalize();
118 return 0;
119 }
```

# Building and running the MPI driver

- ▶ on my desktop, or `mind-meld.cacr.caltech.edu`
  - ▶ where there is no queue manager

```
1 #> mpic++ dilog_mpi.cc -o dilog_mpi -lmpi_cxx -lmpi
2 #> mpirun -np 4 dilog_mpi -N 1e7 -z 1.0
3 Li2(1)=
4   computed: 1.6449340301295035
5   exact: 1.6449340668482264
6   error: 2.23223068274304058e-08
7 #> time mpirun -np 8 dilog_mpi -N 1e9 -z 1.0
8 Li2(1)=
9   computed: 1.6449340044883614
10  exact: 1.6449340668482264
11  error: 3.79102520315773892e-08
12
13 real    0m3.697s
14 user    0m0.018s
15 sys     0m0.015s
16 #>
```



# Running the MP I driver on a shared resource

- ▶ on `shc.cacr.caltech.edu` there is a queue manager
  - ▶ don't use `mpirun`: you are running on the head node
  - ▶ instead, request a dedicated node

```
1 # shc-a> mpic++ dilog_mpi.cc -o dilog_mpi
2 # shc-a> qsub -I -l nodes=1:core8 -l walltime=0:15:00
3 qsub: waiting for job 105059.mistress to start
4 qsub: job 105059.mistress ready
5 Logging in as aivazis on shc168, a Linux-2.x_x86_64 system
6 setting up: (environment) (aliases) (machines) (tools: Linux-2.x_x86_64)
7 # shc168> time mpirun -np 8 dilog_mpi -N 1e9 -z 1.0
8 Li2(1)=
9   computed: 1.6449340044883614
10    exact: 1.6449340668482264
11    error: 3.79102520315773892e-08
12
13 real    0m10.501s
14 user    1m14.642s
15 sys     0m0.273s
16 # shc168> exit
17 logout
18 qsub: job 105059.mistress completed
19 # shc-a>
```