# ACM/CS 114
# Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2012

# 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

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

# Threaded implementation - part 6

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

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

# Threaded implementation - part 7

- the main program – reading the command line

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

# Threaded implementation - part 8

- error checking and the invocation of the task master

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

# Threaded implementation - part 9

- the task master – printing out the answers

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

# Building and running the threaded driver

```
#> g++ dilog-threads.cc -o dilog-threads -pthread
#> dilog-threads -N 1e7 -z 1.0 -t 4
Li2(1)=
 computed: 1.6449340301295035
    exact: 1.6449340668482264
    error: 2.2322306827430458e-08
#> time dilog-threads -N 1e9 -z 1.0 -t 8
Li2(1)=
 computed: 1.6449340044883614
    exact: 1.6449340668482264
    error: 3.7910252031577389e-08

real  0m2.803s
user  0m20.693s
sys   0m0.006s
#>
```

# MPI implementation - part 1

- the preamble

```
1  #include <getopt.h> // for getopt and friends
2  #include <mpi.h>
3
4  #include <cstdio>
5  #include <cstdlib> // 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

```
#> mpic++ dilog-mpi.cc -o dilog-mpi -lmpi_cxx -lmpi
#> mpirun -np 4 dilog-mpi -N 1e7 -z 1.0
Li2(1)=
 computed: 1.6449340301295035
    exact: 1.6449340668482264
    error: 2.2322306827430458e-08
#> time mpirun -np 8 dilog-mpi -N 1e9 -z 1.0
Li2(1)=
 computed: 1.6449340044883614
    exact: 1.6449340668482264
    error: 3.7910252031577389e-08

real   0m3.697s
user   0m0.018s
sys    0m0.015s
#>
```

# Running the MPI 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

```
# shc-a> mpic++ dilog-mpi.cc -o dilog-mpi
# shc-a> qsub -I -l nodes=1:core8 -l walltime=0:15:00
qsub: waiting for job 105059.mistress to start
qsub: job 105059.mistress ready
Logging in as aivazis on shc168, a linux.x86 system
  setting up: (environment) (aliases) (machines) (tools: Linux-2.x_x86_64)
# shc168> time mpirun -np 8 dilog-mpi -N 1e9 -z 1.0
Li2(1)=
 computed: 1.6449340044883614
    exact: 1.6449340668482264
    error: 3.7910252031577389e-08

real  0m10.501s
user  1m14.642s
sys   0m0.273s
# shc168> exit
logout
qsub: job 105059.mistress completed
# shc-a>
```