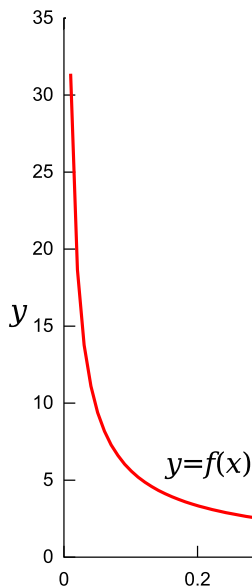


MPI profiling with Alinea MAP

Alexander Gaenko

June 6, 2016

Problem: Calculation of a definite integral.



The problem:

- We need to compute:

$$F(a, b) = \int_a^b f(x) dx$$

Where $f(x)$ is some (presumably “slow to compute”) function.

- We use this approach:

① Split $[a, b]$ by points $\{x_1, x_2, \dots, x_k\}$

②
$$F(a, b) = \sum_k f\left(\frac{x_k + x_{k+1}}{2}\right)(x_{k+1} - x_k)$$

- The integrand $f(x)$ and the integration routine are hidden inside a library.

Sequential program code

```
1  #include <stdio.h>
2  #include <math.h>
3
4  // Declare integrand() and integral() from 'mymath' library
5  #include "mylib/mymath.hpp"
6
7  int main(int argc, char** argv)
8  {
9      unsigned long int n;
10     if (argc!=2 || sscanf(argv[1], "%lu", &n)!=1) {
11         fprintf(stderr, "Usage: %s integration_steps\n\n", argv[0]);
12         return 1;
13     }
14
15     // Integration limits.
16     const double global_a=1E-5;
17     const double global_b=1;
18
19     // Perform integration
20     const double y=integral(integrand, n, global_a, global_b);
21
22     const double y_exact=4*(pow(global_b,0.25)-pow(global_a,0.25));
23     printf("Result=%lf Exact=%lf Difference=%lf\n", y, y_exact, y-y_exact);
24
25     return 0;
26 }
```

Problem size behavior: how to measure.

Do we even need to parallelize?

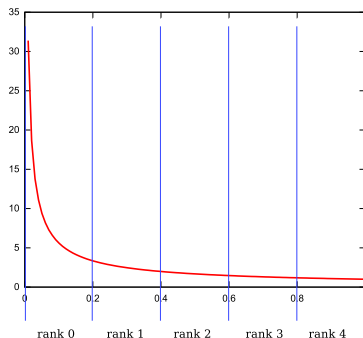
```
1 # Compilation:
2 $ gcc -O3 -o integral_seq.x integral_seq.cxx\
3     -L./mylib -lmymath
4 # Timed runs:
5 $ time -p ./integral_seq.x 1000000
6 Result=3.775045 Exact=3.775063 Difference=-0.000019
7 real 2.12
8 $ time -p ./integral_seq.x 2000000
9 Result=3.775058 Exact=3.775063 Difference=-0.000005
10 real 4.28
11 $ time -p ./integral_seq.x 8000000
12 Result=3.775062 Exact=3.775063 Difference=-0.000001
13 real 17.61
```

The time grows linearly with the problems size. Acceptable accuracy at 8M points. Can we speed it up?

Parallelization: domain decomposition.

Approach:

- Split $[a, b]$ into several domains;
- Compute integrals **independently**.



$$F(a, b) = \int_a^b f(x) dx$$

- 1 Assign a process to each domain $[x_k, x_{k+1}]$
- 2 Let each process compute $F(x_k, x_{k+1})$
- 3 $F(a, b) = \sum_k F(x_k, x_{k+1})$

“Embarassingly parallel” problem,
high speedup is expected.

A sketch of the parallel code

```
1  MPI_Init(&argc, &argv);
2
3  int rank, nprocs;
4  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
6
7  // ...Get total number of steps, broadcast it...
8  // ...
9
10 // Each rank figures out its integration limits and number of steps
11 unsigned long my_stepbase, my_nsteps;
12 get_steps(nsteps_all, nprocs, rank, &my_stepbase, &my_nsteps);
13
14 const double per_step=(global_b-global_a)/nsteps_all;
15 const double x1=global_a + my_stepbase*per_step;
16 const double x2=x1 + my_nsteps*per_step;
17
18 // Compute my own part of the integral
19 double my_y=integral(integrand, my_nsteps, x1, x2);
20
21 // Sum all numbers on master
22 double y=0;
23 MPI_Reduce(&my_y, &y, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
24
25 // ... print results ...
26 // ...
27
28 MPI_Barrier(MPI_COMM_WORLD);
29 // Here we could start another computation.
30 MPI_Finalize();
```

Parallel performance: how to measure.

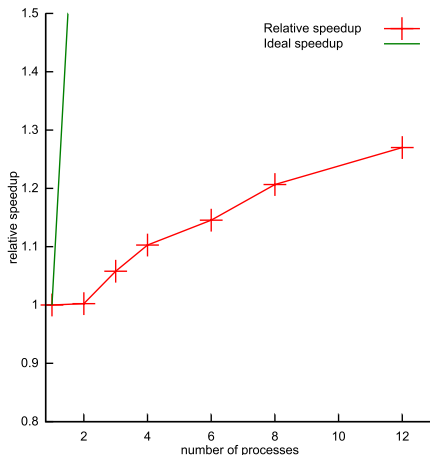
Now let's see how much we achieved...

- **Strong scaling**: as we add processes, how do we fare?
- **Weak scaling**: as we add *both* processes and work?

```
1 $ mpicc -O3 -o integral_par.x integral_par.cxx \  
2     -L./mylib -lmymath  
3 $ time -p mpirun -np 1 ./integral_par.x 8000000  
4 Result=3.775062 Exact=3.775063 Difference=-0.000001  
5 real 17.23  
6 user 17.08  
7 sys 0.02  
8 $ time -p mpirun -np 2 ./integral_par.x 8000000  
9 Result=3.775062 Exact=3.775063 Difference=-0.000001  
10 real 17.24  
11 user 31.98  
12 sys 0.05
```

Parallel performance: results

Is there a performance problem?



- Relative speedup:
$$s(p) = \frac{(\text{time with 1 process})}{(\text{time with } p \text{ processes})}$$
- Ideal relative speedup:
$$s_{\text{ideal}}(p) = p$$
- Our speedup is
25% on 12 nodes!
- I'd call it "*dismal*".
We **do** have a problem!
- Why? How to figure it out?

How does performance analysis work?

How to collect data?

- **Instrumentation:**
 - Insert timers & counters in the code
 - Requires source or binary processing
- **Sampling:**
 - Interrupt & check the program at regular intervals
 - Introduces statistical error

What kind of data?

- **Profile:**
 - Summary information only
 - Relatively small file
- **Trace:**
 - Detailed recording during the run
 - Potentially huge file
 - Profile can be restored

Allinea MAP does **tracing** by **sampling**.

Prepare for profiling

To prepare for profiling/tracing, one needs to:

- Compile with full optimization
- Generate debugging symbols
- Link with system libs dynamically
 - Usually the default
 - Notable exception: Cray
- On Flux: load ddt module

```
1 $ mpicc -g -O3 -o integral_par.x \  
2     integral_par.cxx -L ./mylib -lmymath  
3 $ module add ddt
```

Running Map: simple way (demo)

- 1 Get interactive access to a compute node
- 2 Change to your working directory
- 3 Optionally, set *sampling interval*
- 4 Run as you would, prefixed by `map`

```
1 $ qsub -V -I -X -q flux -l qos=flux,nproc=12 \  
2     -l walltime=10:0:0 -A account_flux  
3 $ cd $PBS_O_WORKDIR  
4 $ export ALLINEA_SAMPLER_INTERVAL=5  
5 $ map mpirun -np 12 ./integral_par.x 10000
```

Caution:

- Too small interval: large overhead!
- Too large interval: not enough samples!
- *Allinea* recommends at least 1000 samples/process

Running Map: other options

What if you can not or would not run a GUI?

- Have slow or non-existing X connection to compute nodes.
- Do not want to wait for interactive session.

Use `-profile` option.

```
1 #PBS -V
2 #PBS -q flux -l qos=flux -A account_flux
3 #PBS -l nproc=12,walltime=10:0:0
4 cd $PBS_O_WORKDIR
5 export ALLINEA_SAMPLER_INTERVAL=5
6 map -profile mpirun -np 12 ./integral_par.x 8000000
```

This will create a `*.map` file. Then run from the login node:

```
1 $ map integral_par_even_12p_*.map
```

If you are submitting to a Flux queue...

① Run map from the login node:

```
$ map ./integral_par.x 8000000
```

② Set number of processes

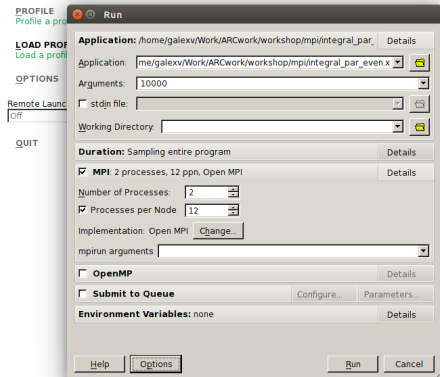
③ Check **Submit to queue**

④ Click **Configure...**

⑤ Load a proper *submission template file* (see next page)

⑥ Click **OK**

⑦ Click **Run**

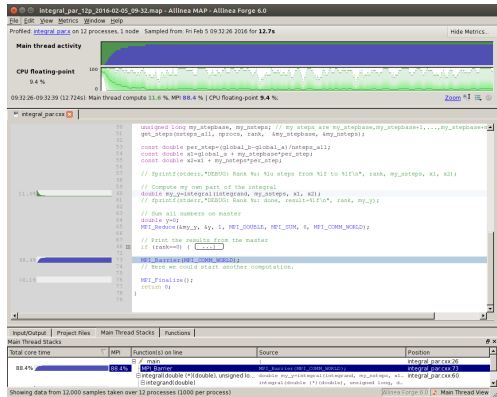


Submission template for Flux

```
1 #PBS -V
2 #PBS -l walltime=WALL_CLOCK_LIMIT_TAG
3 #PBS -l nodes=NUM_NODES_TAG:ppn=PROCS_PER_NODE_TAG
4 #PBS -q QUEUE_TAG -l qos=flux -A account_flux
5 #PBS -o PROGRAM_TAG-allinea.stdout
6 #PBS -e PROGRAM_TAG-allinea.stderr
7
8 cd $PBS_O_WORKDIR
9 AUTO_LAUNCH_TAG
```

Time for a live demo!

- Most of the time is spent in MPI;
- As run progresses, *even more* time is spent in MPI;
- Problem: some processes spend more time computing $f(x)$ than others!
- It's called "Load Imbalance".



Possible solutions:

- Distribute work unevenly (but how?)
- Implement *Dynamic Load Balancing*.

Dynamic load balancing

Idea: If a process has nothing to do, make it to do something.

Manager-Workers approach:

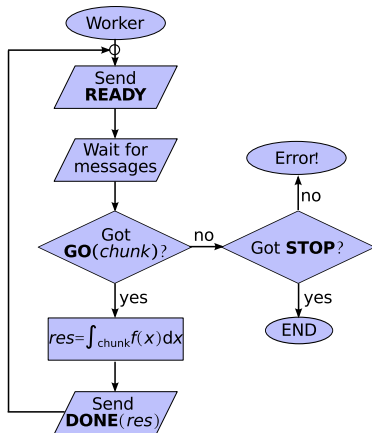
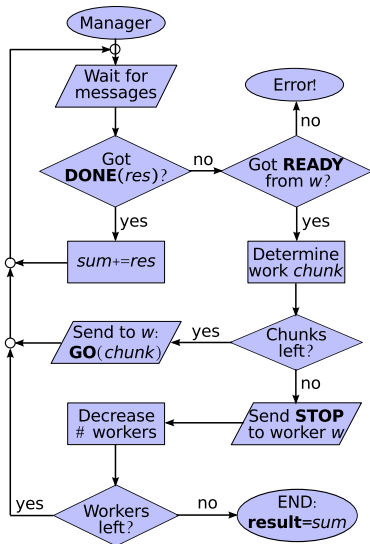
Manager

- ① Listen to all workers
- ② Worker sent **READY** ?
 - send **GO** with a job chunk
- ③ Worker sent **DONE**?
 - add result to the sum
- ④ No more job chunks?
 - send **STOP** to the worker
- ⑤ No more workers?
 - we are done!
 - Otherwise, go to (1)

Worker

- ① Send **READY** to the Manager
- ② Listen to the Manager
- ③ Manager sent **GO** ?
 - Get job chunk
 - Do the calculation
 - Send **DONE** with result to the Manager
 - Go to (1)
- ④ Manager sent **STOP**?
 - exit.

Dynamic load balancing



Dynamic Load Balancing: Worker code

```
1 void do_worker(int manager, int rank)
2 {
3     for (;;) {
4         // Inform manager that i am ready:
5         MPI_Send(NULL,0, MPI_DOUBLE, manager, TAG_READY, MPI_COMM_WORLD);
6         // ...and wait for the task.
7         double range_data[3]; // expect x1, x2 and the number of steps (as double!)
8         MPI_Status stat;
9         MPI_Recv(range_data,3,MPI_DOUBLE, manager, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
10        switch (stat.MPI_TAG) {
11            case TAG_GO:
12                break; // do normal work
13            case TAG_STOP:
14                return;
15            default:
16                fprintf(stderr,"Rank %d: Got unexpected tag=%d, aborting.\n",rank,stat.MPI_TAG);
17                MPI_Abort(MPI_COMM_WORLD, 2);
18        }
19        // we are here on TAG_GO.
20        const double x1=range_data[0];
21        const double x2=range_data[1];
22        const unsigned long my_nsteps=range_data[2];
23        // Compute my own part of the integral
24        double my_y=integral(integrand, my_nsteps, x1, x2);
25
26        // Send the result to manager
27        MPI_Send(&my_y,1,MPI_DOUBLE, manager, TAG_DONE, MPI_COMM_WORLD);
28    }
29 }
```

Dynamic Load Balancing: Manager code

```
1 double do_manager(const double global_a, const double global_b,
2                   const unsigned long nsteps_all, const unsigned long points_per_block,
3                   const int nprocs, const int rank)
4 {
5     const double per_step=(global_b-global_a)/nsteps_all;
6
7     int nworkers_left=nprocs-1;
8     unsigned long ipoint=0; // next point to be processed
9     double y=0;
10    for (;;) {
11        // Get a tagged message and possibly a result from any worker
12        double y_worker=0;
13        MPI_Status stat;
14        MPI_Recv(&y_worker,1,MPI_DOUBLE, MPI_ANY_SOURCE,MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
15        const int rank_worker=stat.MPI_SOURCE;
16        switch (stat.MPI_TAG) {
17            case TAG_READY:
18                // Do we have any work for this worker?
19                if (ipoint>=nsteps_all) {
20                    // if not, stop the worker
21                    MPI_Send(NULL,0,MPI_DOUBLE, rank_worker, TAG_STOP, MPI_COMM_WORLD);
22                    --nworkers_left;
23                    break;
24                }
```

Dynamic Load Balancing: Manager code (cont.)

```
1  { // Prepare chunk of work for the worker
2      const unsigned long ns_worker=
3          (ipoint+points_per_block > nsteps_all)? (nsteps_all - ipoint) : points_per_block;
4      const double x1=global_a+ipoint*per_step;
5      const double x2=x1+ns_worker*per_step;
6      double range_data[3];
7      range_data[0]=x1;
8      range_data[1]=x2;
9      range_data[2]=ns_worker;
10     // Send the chunk
11     MPI_Send(range_data,3,MPI_DOUBLE, rank_worker, TAG_GO, MPI_COMM_WORLD);
12     // adjust the amount of points
13     ipoint += ns_worker;
14 }
15 break;
16 case TAG_DONE:
17     y += y_worker;
18     break;
19 default:
20     fprintf(stderr,"Rank %d (manager): Got unexpected tag=%d from %d,"
21             " aborting.\n",rank,rank_worker,stat.MPI_TAG);
22     MPI_Abort(MPI_COMM_WORLD,1);
23 }
24 if (nworkers_left<=0) { // Any active workers left?
25     return y;
26 }
27 }
28 }
```

Dynamic Load Balancing: main

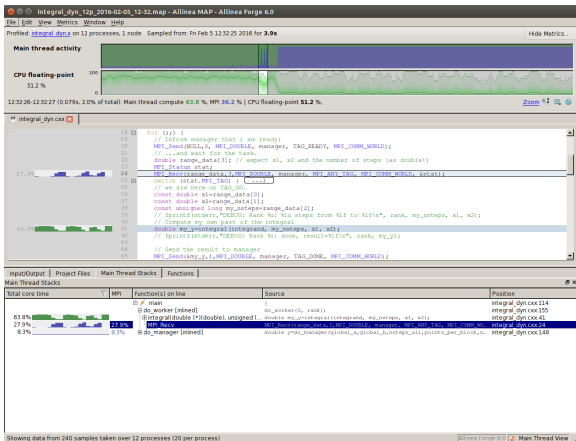
```
1  int main(int argc, char** argv)
2  {
3      MPI_Init(&argc, &argv);
4
5      int rank, nprocs;
6      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
8
9      // Get command line arguments, broadcast
10     unsigned long int nsteps_all, points_per_block;
11     // ....
12
13     // Global integration limits.
14     const double global_a=1E-5;
15     const double global_b=1;
16
17     // Split into workers and manager:
18     if (rank==0) { // Run as the manager and get the result:
19         double y=do_manager(global_a,global_b,nsteps_all,points_per_block,nprocs,rank);
20
21         const double y_exact=4*(pow(global_b,0.25)-pow(global_a,0.25));
22         printf("Result=%lf Exact=%lf Difference=%lf\n", y, y_exact, y-y_exact);
23     } else { // Run as a worker
24         do_worker(0, rank);
25     }
26
27     MPI_Barrier(MPI_COMM_WORLD);
28     // Here we could start another computation.
29     MPI_Finalize();
30     return 0;
31 }
```

Dynamic Load Balancing: large block size (2500)

Run with:

```
$ map mpirun -np 12 ./dyn_integral.x 8000000 2500
```

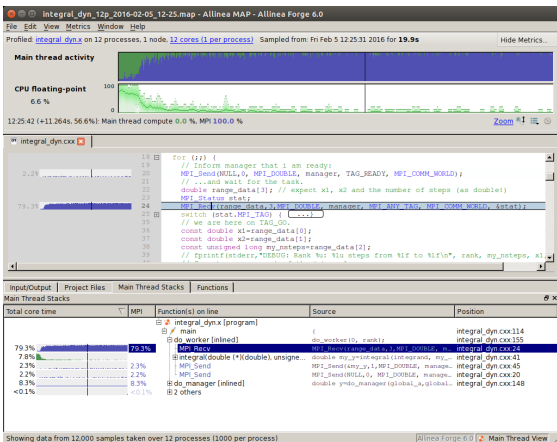
- CPU occupied till half-way
- Spikes in MPI use:
Manager receiving data
- Looks like the last worker was holding everyone
- Other workers:
worker starvation
(no work to do)



Dynamic Load Balancing: small block size (2)

Run with:

```
$ map mpirun -np 12 ./dyn_integral.x 8000000 2
```



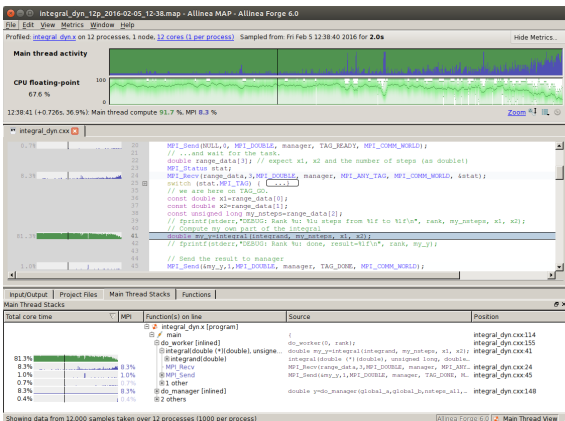
- Brief useful CPU work, then all time is in MPI
- Just moving data around:
 - Workers receiving data
 - 8% (1/12) of time: Manager sending data.
- Very low *computation/communication* ratio.

Dynamic Load Balancing: good block size (100)

Run with:

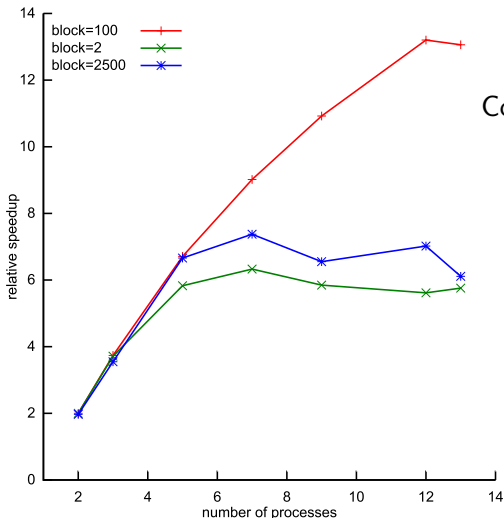
```
$ map mpirun -np 12 ./dyn_integral.x 8000000 100
```

- 80% time CPU is busy!
- 8% (1/12) of time: Manager work.
- Mostly MPI by the end of the run.
- OK computation/communication ratio.
- Still room for improvement!



Dynamic Load Balancing: Strong scaling graph.

How does it look now?



Conclusions:

- Block size does affect performance.
- Block size 100 grows up to node size (12).
- Too small block: MPI communication overhead.
- Too large block: workers starvation.

Concluding slide

Take-home message

- Once you made your program parallel, do simple scaling experiments.
- If scaling is bad, use profiling tools to understand why.
- *Allinea Map* is available for all Flux users.
- *Map* can analyze not only parallel, but single-node and OpenMP performance. (Can show something if time permits!)
- If you need any advise and/or help with your parallel programming, ARC provides [free consulting service](#)
 - Just send a mail to HPC support...
 - ...or directly to: "Alexander Gaenko" <galexv@umich.edu>

Thank you for your attention!