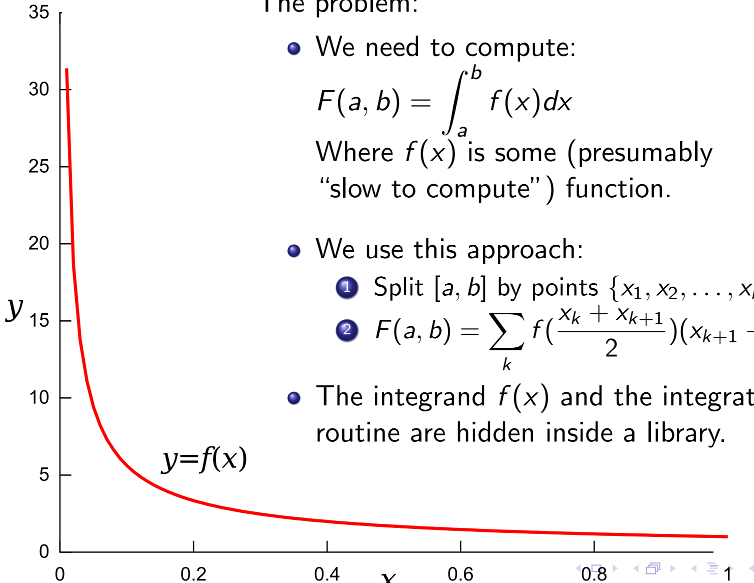
MPI profiling with Allinea MAP

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 $\{x_1, x_2, \ldots, x_k\}$
  2. $F(a, b) = \sum_k f(\frac{x_k + x_{k+1}}{2})(x_{k+1} - x_k)$
- The integrand $f(x)$ and the integration routine are hidden inside a library.

Sequential program code

```
#include <stdio.h>
#include <math.h>

// Declare integrand() and integral() from ‘‘mymath‘‘ library
#include "mylib/mymath.hpp"

int main(int argc, char** argv)
{
  unsigned long int n;
  if (argc!=2 || sscanf(argv[1],"%lu",&n)!=1) {
    fprintf(stderr,"Usage:\n%s integration_steps\n\n\n",argv[0]);
    return 1;
  }

  // Integration limits.
  const double global_a=1E-5;
  const double global_b=1;

  // Perform integration
  const double y=integral(integrand, n, global_a, global_b);

  const double y_exact=4*(pow(global_b,0.25)-pow(global_a,0.25));
  printf("Result=%lf Exact=%lf Difference=%lf\n", y, y_exact, y-y_exact);

  return 0;
}
```

Problem size behavior: how to measure.

Do we even need to parallelize?

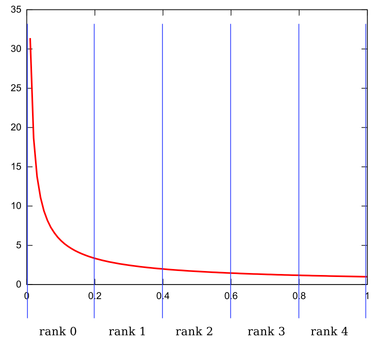
```
# Compilation:
$ gcc -O3 -o integral_seq.x integral_seq.cxx\
      -L./mylib -lmymath
# Timed runs:
$ time -p ./integral_seq.x 1000000
Result=3.775045 Exact=3.775063 Difference=-0.000019
real 2.12
$ time -p ./integral_seq.x 2000000
Result=3.775058 Exact=3.775063 Difference=-0.000005
real 4.28
$ time -p ./integral_seq.x 8000000
Result=3.775062 Exact=3.775063 Difference=-0.000001
real 17.61
```

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

Parallellization: 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

```
MPI_Init(&argc, &argv);

int rank, nprocs;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

// ...Get total number of steps, broadcast it...
// ...

// Each rank figures out its integration limits and number of steps
unsigned long my_stepbase, my_nsteps;
get_steps(nsteps_all, nprocs, rank,  &my_stepbase, &my_nsteps);

const double per_step=(global_b-global_a)/nsteps_all;
const double x1=global_a + my_stepbase*per_step;
const double x2=x1 + my_nsteps*per_step;

// Compute my own part of the integral
double my_y=integral(integrand, my_nsteps, x1, x2);

// Sum all numbers on master
double y=0;
MPI_Reduce(&my_y, &y, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

// ... print results ...
// ...

MPI_Barrier(MPI_COMM_WORLD);
// Here we could start another computation.
MPI_Finalize();
```

Parallel performance: how to measure.

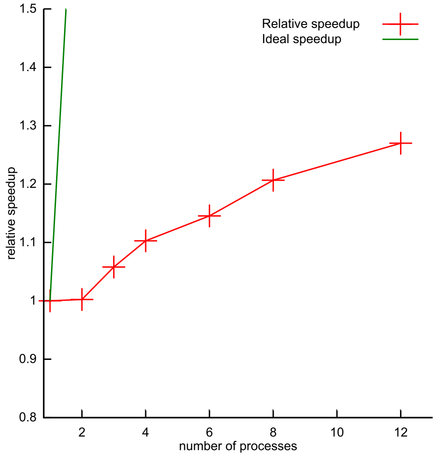
Now let's see how much we achieved...

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

```
$ mpicc -O3 -o integral_par.x integral_par.cxx \
        -L./mylib -lmymath
$ time -p mpirun -np 1 ./integral_par.x 8000000
Result=3.775062 Exact=3.775063 Difference=-0.000001
real 17.23
user 17.08
sys 0.02
$ time -p mpirun -np 2 ./integral_par.x 8000000
Result=3.775062 Exact=3.775063 Difference=-0.000001
real 17.24
user 31.98
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 recovered

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

```
$ mpicc -g -O3 -o integral_par.x \
        integral_par.cxx -L ./mylib -lmymath
$ 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`

```
$ qsub -V -I -X -q flux -l qos=flux,nproc=12 \
        -l walltime=10:0:0 -A account_flux
$ cd $PBS_O_WORKDIR
$ export ALLINEA_SAMPLER_INTERVAL=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.

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

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

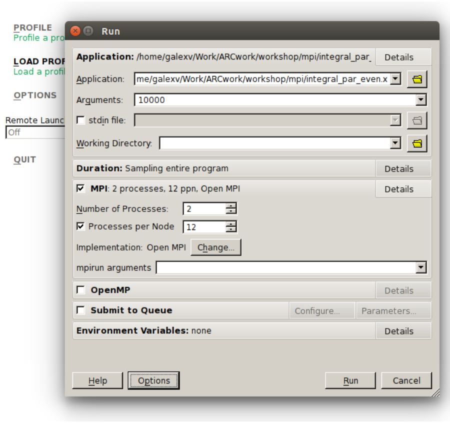
```
$ map integral_par_even_12p_*.map
```

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**



PROFILE
Profile a pro
LOAD PROF
Load a profil
OPTIONS
Remote Launc
Off
QUIT

Run

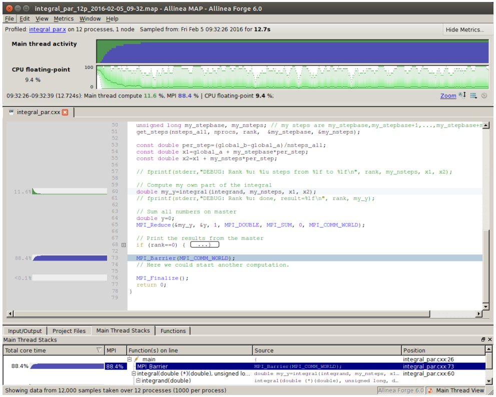
Application: /home/galexv/Work/ARCwork/workshop/mpi/integral_par_ Details
Application: me/galexv/Work/ARCwork/workshop/mpi/integral_par_even.x
Arguments: 10000
☐ stdin file:
Working Directory:
Duration: Sampling entire program Details
☑ **MPI**: 2 processes, 12 ppn, Open MPI Details
Number of Processes: 2
☑ Processes per Node 12
Implementation: Open MPI Change...
mpirun arguments
☐ **OpenMP** Details
☐ **Submit to Queue** Configure... Parameters...
Environment Variables: none Details

Submission template for Flux

```
#PBS -V
#PBS -l walltime=WALL_CLOCK_LIMIT_TAG
#PBS -l nodes=NUM_NODES_TAG:ppn=PROCS_PER_NODE_TAG
#PBS -q QUEUE_TAG -l qos=flux -A account_flux
#PBS -o PROGRAM_TAG-allinea.stdout
#PBS -e PROGRAM_TAG-allinea.stderr

cd $PBS_O_WORKDIR
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 $f(x)$ then others!
- It's called "Load Imbalance".



Profiled: integral_par.x on 12 processes, 1 node Sampled from: Fri Feb 5 09:32:26 2016 for **12.7s**

09:32:26-09:32:39 (12.724s): Main thread compute 11.6 %, MPI 88.4 % | CPU floating-point **9.4** %;

```
  unsigned long my_stepbase, my_nsteps; // my steps are my_stepbase,my_stepbase+1,...,my_stepbase+m
  get_steps(nsteps_all, nprocs, rank,  &my_stepbase, &my_nsteps);

  const double per_step=(global_b-global_a)/nsteps_all;
  const double x1=global_a + my_stepbase*per_step;
  const double x2=x1 + my_nsteps*per_step;

  // fprintf(stderr,"DEBUG: Rank %u: %lu steps from %lf to %lf\n", rank, my_nsteps, x1, x2);

  // Compute my own part of the integral
  double my_y=integral(integrand, my_nsteps, x1, x2);
  // fprintf(stderr,"DEBUG: Rank %u: done, result=%lf\n", rank, my_y);

  // Sum all numbers on master
  double y=0;
  MPI_Reduce(&my_y, &y, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

  // Print the results from the master
  if (rank==0) { ...}

  MPI_Barrier(MPI_COMM_WORLD);
  // Here we could start another computation.

  MPI_Finalize();
  return 0;
}
```

Main Thread Stacks

Total core time	MPI	Function(s) on line	Source	Position
		main	{	integral_par.cxx:26
88.4%	88.4%	MPI_Barrier	MPI_Barrier(MPI_COMM_WORLD);	integral_par.cxx:73
		integral(double (*)(double), unsigned lo...	double my_y=integral(integrand, my_nsteps, x1...	integral_par.cxx:60
		integrand(double)	integral(double (*)(double), unsigned long, d...	

Showing data from 12,000 samples taken over 12 processes (1000 per process)

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:

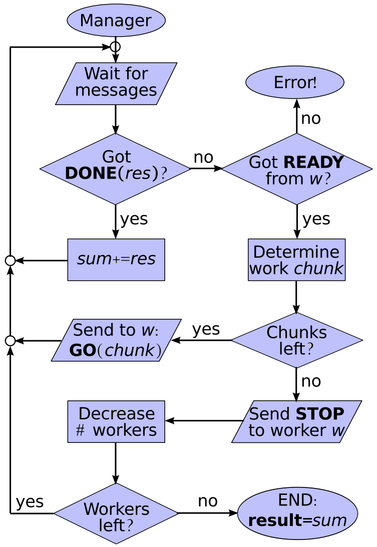
Manager

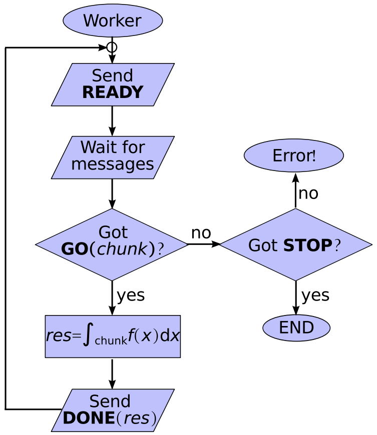
1. Listen to all workers
2. Worker sent `READY` ?
   - send `GO` with a job chunk
3. Worker sent `DONE`?
   - add result to the sum
4. No more job chunks?
   - send `STOP` to the worker
5. No more workers?
   - we are done!
   - Otherwise, go to (1)

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

```c
void do_worker(int manager, int rank)
{
  for (;;) {
    // Inform manager that i am ready:
    MPI_Send(NULL,0, MPI_DOUBLE, manager, TAG_READY, MPI_COMM_WORLD);
    // ...and wait for the task.
    double range_data[3]; // expect x1, x2 and the number of steps (as double!)
    MPI_Status stat;
    MPI_Recv(range_data,3,MPI_DOUBLE, manager, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
    switch (stat.MPI_TAG) {
    case TAG_GO:
      break; // do normal work
    case TAG_STOP:
      return;
    default:
      fprintf(stderr,"Rank %d: Got unexpected tag=%d, aborting.\n",rank,stat.MPI_TAG);
      MPI_Abort(MPI_COMM_WORLD, 2);
    }
    // we are here on TAG_GO.
    const double x1=range_data[0];
    const double x2=range_data[1];
    const unsigned long my_nsteps=range_data[2];
    // Compute my own part of the integral
    double my_y=integral(integrand, my_nsteps, x1, x2);

    // Send the result to manager
    MPI_Send(&my_y,1,MPI_DOUBLE, manager, TAG_DONE, MPI_COMM_WORLD);
  }
}
```

Dynamic Load Balancing: Manager code

```
double do_manager(const double global_a, const double global_b,
                  const unsigned long nsteps_all, const unsigned long points_per_block,
                  const int nprocs, const int rank)
{
  const double per_step=(global_b-global_a)/nsteps_all;

  int nworkers_left=nprocs-1;
  unsigned long ipoint=0; // next point to be processed
  double y=0;
  for (;;) {
    // Get a tagged message and possibly a result from any worker
    double y_worker=0;
    MPI_Status stat;
    MPI_Recv(&y_worker,1,MPI_DOUBLE, MPI_ANY_SOURCE,MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
    const int rank_worker=stat.MPI_SOURCE;
    switch (stat.MPI_TAG) {
    case TAG_READY:
      // Do we have any work for this worker?
      if (ipoint>=nsteps_all) {
        // if not, stop the worker
        MPI_Send(NULL,0,MPI_DOUBLE, rank_worker, TAG_STOP, MPI_COMM_WORLD);
        --nworkers_left;
        break;
      }
```

Dynamic Load Balancing: Manager code (cont.)

```
      { // Prepare chunk of work for the worker
        const unsigned long ns_worker=
           (ipoint+points_per_block > nsteps_all)? (nsteps_all - ipoint) : points_per_block;
        const double x1=global_a+ipoint*per_step;
        const double x2=x1+ns_worker*per_step;
        double range_data[3];
        range_data[0]=x1;
        range_data[1]=x2;
        range_data[2]=ns_worker;
        // Send the chunk
        MPI_Send(range_data,3,MPI_DOUBLE, rank_worker, TAG_GO, MPI_COMM_WORLD);
        // adjust the amount of points
        ipoint += ns_worker;
      }
      break;
    case TAG_DONE:
      y += y_worker;
      break;
    default:
      fprintf(stderr,"Rank %d (manager): Got unexpected tag=%d from %d,"
                     " aborting.\n",rank,rank_worker,stat.MPI_TAG);
      MPI_Abort(MPI_COMM_WORLD,1);
    }
    if (nworkers_left<=0) { // Any active workers left?
      return y;
    }
  }
}
```

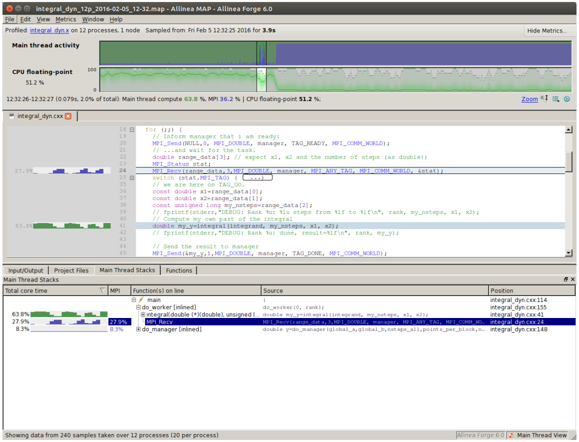
Dynamic Load Balancing: main

```c
int main(int argc, char** argv)
{
  MPI_Init(&argc, &argv);

  int rank, nprocs;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  // Get command line arguments, broadcast
  unsigned long int nsteps_all, points_per_block;
  // ....

  // Global integration limits.
  const double global_a=1E-5;
  const double global_b=1;

  // Split into workers and manager:
  if (rank==0) { // Run as the manager and get the result:
    double y=do_manager(global_a,global_b,nsteps_all,points_per_block,nprocs,rank);

    const double y_exact=4*(pow(global_b,0.25)-pow(global_a,0.25));
    printf("Result=%lf Exact=%lf Difference=%lf\n", y, y_exact, y-y_exact);
  } else { // Run as a worker
    do_worker(0, rank);
  }

  MPI_Barrier(MPI_COMM_WORLD);
  // Here we could start another computation.
  MPI_Finalize();
  return 0;
}
```

Dynamic Load Balancing: large block size (2500)

Run with:

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


integral_dyn_12p_2016-02-05_12-32.map - Allinea MAP - Allinea Forge 6.0

Profiled: integral_dyn.x on 12 processes, 1 node   Sampled from: Fri Feb 5 12:32:25 2016 for **3.9s**

Main thread activity

CPU floating-point 51.2 %

12:32:26-12:32:27 (0.079s, 2.0% of total): Main thread compute 63.8 %, MPI 36.2 % | CPU floating-point 51.2 %;

integral_dyn.cxx

```
  for (;;) {
    // Inform manager that i am ready:
    MPI_Send(NULL,0, MPI_DOUBLE, manager, TAG_READY, MPI_COMM_WORLD);
    // ...and wait for the task.
    double range_data[3]; // expect x1, x2 and the number of steps (as double!)
    MPI_Status stat;
    MPI_Recv(range_data,3,MPI_DOUBLE, manager, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
    switch (stat.MPI_TAG) { ...}
    // we are here on TAG_GO.
    const double x1=range_data[0];
    const double x2=range_data[1];
    const unsigned long my_nsteps=range_data[2];
    // fprintf(stderr,"DEBUG: Rank %u: %lu steps from %lf to %lf\n", rank, my_nsteps, x1, x2);
    // Compute my own part of the integral
    double my_y=integral(integrand, my_nsteps, x1, x2);
    // fprintf(stderr,"DEBUG: Rank %u: done, result=%lf\n", rank, my_y);

    // Send the result to manager
    MPI_Send(&my_y,1,MPI_DOUBLE, manager, TAG_DONE, MPI_COMM_WORLD);
```

Main Thread Stacks

Total core time	MPI	Function(s) on line	Source	Position
		main	{	integral_dyn.cxx:114
		do_worker [inlined]	do_worker(0, rank);	integral_dyn.cxx:155
63.8%		integral(double (*)(double), unsigned l...	double my_y=integral(integrand, my_nsteps, x1, x2);	integral_dyn.cxx:41
27.9%	27.9%	MPI_Recv	MPI_Recv(range_data,3,MPI_DOUBLE, manager, MPI_ANY_TAG, MPI_COMM_WO...	integral_dyn.cxx:24
8.3%	8.3%	do_manager [inlined]	double y=do_manager(global_a,global_b,nsteps_all,points_per_block,n...	integral_dyn.cxx:148

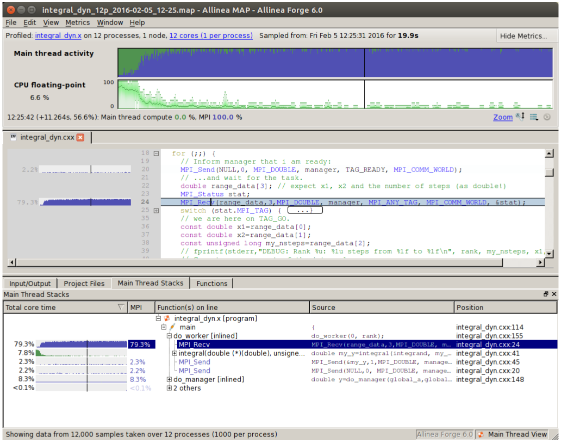
Showing data from 240 samples taken over 12 processes (20 per process)

- 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
```



integral_dyn_12p_2016-02-05_12-25.map - Allinea MAP - Allinea Forge 6.0

File Edit View Metrics Window Help

Profiled: integral_dyn.x on 12 processes, 1 node, 12 cores (1 per process) Sampled from: Fri Feb 5 12:25:31 2016 for **19.9s**

12:25:42 (+11.264s, 56.6%): Main thread compute 0.0 %, MPI 100.0 %

```
18   for (;;) {
19     // Inform manager that i am ready:
20     MPI_Send(NULL,0, MPI_DOUBLE, manager, TAG_READY, MPI_COMM_WORLD);
21     // ...and wait for the task.
22     double range_data[3]; // expect x1, x2 and the number of steps (as double!)
23     MPI_Status stat;
24     MPI_Recv(range_data,3,MPI_DOUBLE, manager, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
25     switch (stat.MPI_TAG) { ...}
35     // we are here on TAG_GO.
36     const double x1=range_data[0];
37     const double x2=range_data[1];
38     const unsigned long my_nsteps=range_data[2];
39     // fprintf(stderr,"DEBUG: Rank %u: %lu steps from %lf to %lf\n", rank, my_nsteps, x1,
```

Total core time	MPI	Function(s) on line	Source	Position
		integral_dyn.x [program]		
		main	{	integral_dyn.cxx:114
		do_worker [inlined]	do_worker(0, rank);	integral_dyn.cxx:155
79.3%	79.3%	MPI_Recv	MPI_Recv(range_data,3,MPI_DOUBLE, m...	integral_dyn.cxx:24
7.8%		integral(double (*)(double), unsigne...	double my_y=integral(integrand, my_...	integral_dyn.cxx:41
2.3%	2.3%	MPI_Send	MPI_Send(&my_y,1,MPI_DOUBLE, manage...	integral_dyn.cxx:45
2.2%	2.2%	MPI_Send	MPI_Send(NULL,0, MPI_DOUBLE, manage...	integral_dyn.cxx:20
8.3%	8.3%	do_manager [inlined]	double y=do_manager(global_a,global...	integral_dyn.cxx:148
<0.1%	<0.1%	2 others		

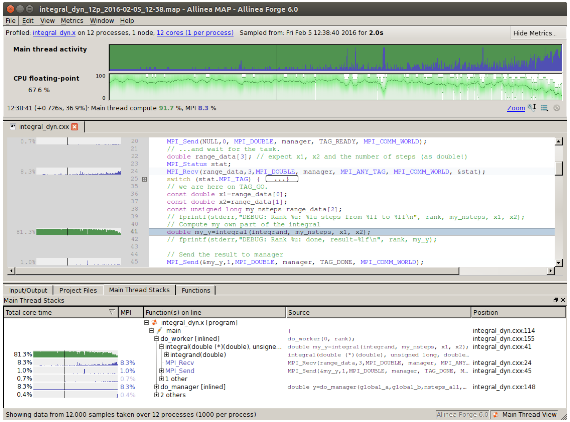
Showing data from 12,000 samples taken over 12 processes (1000 per process)

- 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
```


integral_dyn_12p_2016-02-05_12-38.map - Allinea MAP - Allinea Forge 6.0

Profiled: integral_dyn.x on 12 processes, 1 node, 12 cores (1 per process) Sampled from: Fri Feb 5 12:38:40 2016 for **2.0s**

12:38:41 (+0.726s, 36.9%): Main thread compute 91.7 %, MPI 8.3 %

```
 0.7%  20     MPI_Send(NULL,0, MPI_DOUBLE, manager, TAG_READY, MPI_COMM_WORLD);
       21     // ...and wait for the task.
       22     double range_data[3]; // expect x1, x2 and the number of steps (as double!)
       23     MPI_Status stat;
 8.3%  24     MPI_Recv(range_data,3,MPI_DOUBLE, manager, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
       25     switch (stat.MPI_TAG) { ...}
       35     // we are here on TAG_GO.
       36     const double x1=range_data[0];
       37     const double x2=range_data[1];
       38     const unsigned long my_nsteps=range_data[2];
       39     // fprintf(stderr,"DEBUG: Rank %u: %lu steps from %lf to %lf\n", rank, my_nsteps, x1, x2);
       40     // Compute my own part of the integral
81.3%  41     double my_y=integral(integrand, my_nsteps, x1, x2);
       42     // fprintf(stderr,"DEBUG: Rank %u: done, result=%lf\n", rank, my_y);
       43
       44     // Send the result to manager
 1.0%  45     MPI_Send(&my_y,1,MPI_DOUBLE, manager, TAG_DONE, MPI_COMM_WORLD);
```

Main Thread Stacks

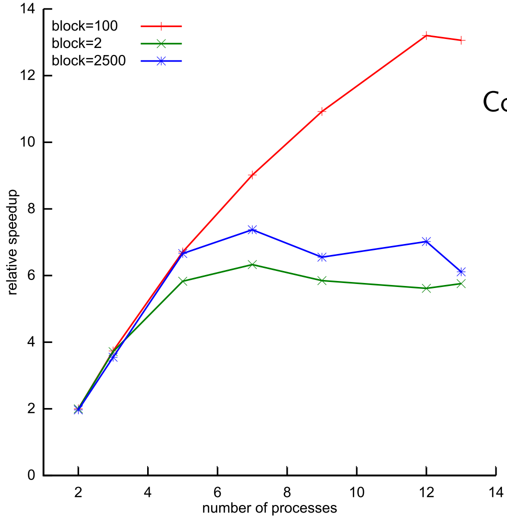
Total core time	MPI	Function(s) on line	Source	Position
		integral_dyn.x [program]		
		main	{	integral_dyn.cxx:114
		do_worker [inlined]	do_worker(0, rank);	integral_dyn.cxx:155
		integral(double (*)(double), unsigne...	double my_y=integral(integrand, my_nsteps, x1, x2);	integral_dyn.cxx:41
81.3%		integrand(double)	integral(double (*)(double), unsigned long, double...	
8.3%	8.3%	MPI_Recv	MPI_Recv(range_data,3,MPI_DOUBLE, manager, MPI_ANY...	integral_dyn.cxx:24
1.0%	1.0%	MPI_Send	MPI_Send(&my_y,1,MPI_DOUBLE, manager, TAG_DONE, M...	integral_dyn.cxx:45
0.7%	0.7%	1 other		
8.3%	8.3%	do_manager [inlined]	double y=do_manager(global_a,global_b,nsteps_all,...	integral_dyn.cxx:148
0.4%	0.4%	2 others		

Showing data from 12,000 samples taken over 12 processes (1000 per process)

- 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!