

Load Balancing

- 1 the Mandelbrot set and Granularity
 - computing a file with grayscales
 - grain size and granularity
- 2 Static Work Load Assignment
 - granularity considerations
 - static work load assignment with MPI
 - an implementation with mpi4py
- 3 Dynamic Work Load Balancing
 - scheduling jobs to run in parallel
 - dynamic work load balancing with MPI
 - probing in Python and Julia

MCS 572 Lecture 7
Introduction to Supercomputing
Jan Verschelde, 11 September 2024

Load Balancing

1 the Mandelbrot set and Granularity

- computing a file with grayscales
- grain size and granularity

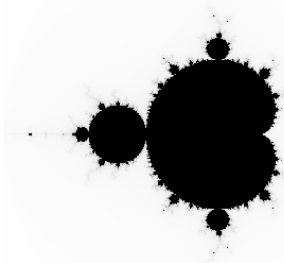
2 Static Work Load Assignment

- granularity considerations
- static work load assignment with MPI
- an implementation with mpi4py

3 Dynamic Work Load Balancing

- scheduling jobs to run in parallel
- dynamic work load balancing with MPI
- probing in Python and Julia

the Mandelbrot set



A pixel with coordinates (x, y) is mapped to $c = x + iy$, $i = \sqrt{-1}$. Consider the map $z \mapsto z^2 + c$, starting at $z = 0$. The grayscale for (x, y) is the number of iterations it takes for $|z| \geq 2$ under the map.

The number n of iterations ranges from 0 to 255.

The grayscales are plotted in reverse, as $255 - n$.

Grayscales for different pixels are calculated independently
 \Rightarrow pleasingly parallel.

the function `iterate`

The prototype of the function `iterate` is

```
int iterate ( double x, double y );  
/*  
 * Returns the number of iterations for  $z^2 + c$   
 * to grow larger than 2, for  $c = x + i*y$ ,  
 * where  $i = \text{sqrt}(-1)$ , starting at  $z = 0$ . */
```

We call `iterate` for all pixels (x,y) ,
for x and y ranging over all rows and columns of a pixel matrix.

In our plot we compute 5,000 rows and 5,000 columns.

code for iterate

```
int iterate ( double x, double y )
{
    double wx,wy,v,xx;
    int k = 0;

    wx = 0.0; wy = 0.0; v = 0.0;
    while ((v < 4) && (k++ < 254))
    {
        xx = wx*wx - wy*wy;
        wy = 2.0*wx*wy;
        wx = xx + x;
        wy = wy + y;
        v = wx*wx + wy*wy;
    }
    return k;
}
```

computational cost

In the code for `iterate` we count

- 6 multiplications on doubles,
- 3 additions and 1 subtraction.

On a Mac OS X laptop 2.26 Ghz Intel Core 2 Duo,
for a 5,000-by-5,000 matrix of pixels:

```
$ time ./mandelbrot
```

```
Total number of iterations : 682940922
```

```
real      0m15.675s
```

```
user      0m14.914s
```

```
sys       0m0.163s
```

Performed $682,940,922 \times 10$ flops in 15 seconds
or 455,293,948 flops per second.

optimizing with -O3

```
$ make mandelbrot_opt  
gcc -O3 -o mandelbrot_opt mandelbrot.c
```

```
$ time ./mandelbrot_opt  
Total number of iterations : 682940922
```

```
real      0m9.846s  
user      0m9.093s  
sys       0m0.163s
```

With full optimization, the time drops from 15 to 9 seconds.

After compilation with -O3, performed 758,823,246 flops per second.

input and output

Input parameters:

- $(x, y) \in [a, b] \times [c, d]$, e.g.: $[a, b] = [-2, +2] = [c, d]$;
- number n of rows (and columns) in pixel matrix determines the resolution of the image and the spacing between points:
 $\delta x = (b - a)/(n - 1)$, $\delta y = (d - c)/(n - 1)$.

The output is a postscript file. Why?

- standard format, direct to print or view,
- allows for batch processing in an environment without visualization capabilities.

Load Balancing

1 the Mandelbrot set and Granularity

- computing a file with grayscales
- grain size and granularity

2 Static Work Load Assignment

- granularity considerations
- static work load assignment with MPI
- an implementation with mpi4py

3 Dynamic Work Load Balancing

- scheduling jobs to run in parallel
- dynamic work load balancing with MPI
- probing in Python and Julia

grain size and granularity

Definition

A *grain* is a sequence of computational steps for sequential execution on a single processor.

Depending on the grain size, we distinguish between

- small grain size: *fine granularity*,
- large grain size: *coarse granularity*.

There is a tradeoff to make:

- Coarse granularity has little communication overhead, but may limit the amount of parallelism; while
- fine granularity promotes parallelism, but may lead to an excessive communication overhead.

Load Balancing

1 the Mandelbrot set and Granularity

- computing a file with grayscales
- grain size and granularity

2 Static Work Load Assignment

- **granularity considerations**
- static work load assignment with MPI
- an implementation with mpi4py

3 Dynamic Work Load Balancing

- scheduling jobs to run in parallel
- dynamic work load balancing with MPI
- probing in Python and Julia

static work load assignment

Static means: the decision which pixels are computed by which processor is fixed in advance by some algorithm.

For the granularity in the communication, we have two extremes:

- 1 Matrix of grayscales is divided up into p equal parts and each processor computes part of the matrix.

For example: 5,000 rows among 5 processors,
each processor takes 1,000 rows.

Communication happens after all calculations are done,
at the end all processors send their big submatrix to root node.

- 2 Matrix of grayscales is distributed pixel-by-pixel.
Entry (i, j) of the n -by- n matrix is computed by processor
with label $(i \times n + j) \bmod p$.

Communication is completely interlaced with all computation.

choice of granularity

→ Problem with all communication at end:

total cost = computational cost + communication cost.

The communication cost is not interlaced with the computation.

→ Problem with pixel-by-pixel distribution:

To compute the grayscale of one pixel requires at most 255 iterations, but may finish much sooner.

Even in the most expensive case, processors may be mostly busy handling send/recv operations.

→ Compromise: distribute work load along rows:

- 1 row i is computed by node $1 + (i \bmod (p - 1))$.
- 2 root node 0 distributes row indices and collects the computed rows.

Load Balancing

1 the Mandelbrot set and Granularity

- computing a file with grayscales
- grain size and granularity

2 Static Work Load Assignment

- granularity considerations
- **static work load assignment with MPI**
- an implementation with mpi4py

3 Dynamic Work Load Balancing

- scheduling jobs to run in parallel
- dynamic work load balancing with MPI
- probing in Python and Julia

manager/worker algorithm for static load assignment

Given n jobs to be completed by p processors, $n \gg p$.

Processor 0 is in charge of

- 1 distributing the jobs among the $p - 1$ compute nodes,
- 2 collecting the results from the $p - 1$ compute nodes.

Assuming n is a multiple of $p - 1$, let $k = n/(p - 1)$.

The manager executes the following algorithm:

```
for  $i$  from 1 to  $k$  do
  for  $j$  from 1 to  $p - 1$  do
    send the next job to compute node  $j$ ;
  for  $j$  from 1 to  $p - 1$  do
    receive result from compute node  $j$ .
```

run of an example program in C

```
$ mpirun -np 3 ./static_loaddist
reading the #jobs per compute node...
1
sending 0 to 1
sending 1 to 2
node 1 received 0
-> 1 computes b
node 1 sends b
node 2 received 1
-> 2 computes c
node 2 sends c
received b from 1
received c from 2
sending -1 to 1
sending -1 to 2
The result : bc
node 2 received -1
node 1 received -1
$
```


the main program

```
int main ( int argc, char *argv[] )
{
    int i,p;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&i);
    if(i != 0)
        worker(i);
    else
    {
        printf("reading the #jobs per compute node...\n");
        int nbjobs; scanf("%d",&nbjobs);
        manager(p,nbjobs*(p-1));
    }
    MPI_Finalize();
    return 0;
}
```

code for the compute node

```
int worker ( int i )
{
    int myjob;
    MPI_Status status;
    do
    {
        MPI_Recv(&myjob,1,MPI_INT,0,tag,
                MPI_COMM_WORLD,&status);
        if(v == 1) printf("node %d received %d\n",i,myjob);
        if(myjob == -1) break;
        char c = 'a' + ((char)i);
        if(v == 1) printf("-> %d computes %c\n",i,c);
        if(v == 1) printf("node %d sends %c\n",i,c);
        MPI_Send(&c,1,MPI_CHAR,0,tag,MPI_COMM_WORLD);
    }
    while(myjob != -1);
    return 0;
}
```

manager distributes jobs

```
int manager ( int p, int n )
{
    char result[n+1];
    int job = -1;
    int j;
    do
    {
        for(j=1; j<p; j++) /* distribute jobs */
        {
            if(++job >= n) break;
            int d = 1 + (job % (p-1));
            if(v == 1) printf("sending %d to %d\n", job, d);
            MPI_Send(&job, 1, MPI_INT, d, tag, MPI_COMM_WORLD);
        }
        if(job >= n) break;
    }
}
```

manager collects results

```
for(j=1; j<p; j++) /* collect results */
{
    char c;
    MPI_Status status;
    MPI_Recv(&c,1,MPI_CHAR,j>tag,
             MPI_COMM_WORLD,&status);
    if(v == 1) printf("received %c from %d\n",c,j);
    result[job-p+1+j] = c;
}
} while (job < n);
```

end of job queue

```
job = -1;
for(j=1; j < p; j++) /* termination signal is -1 */
{
    if(v==1) printf("sending -1 to %d\n",j);
    MPI_Send(&job,1,MPI_INT,j,tag,MPI_COMM_WORLD);
}
result[n] = '\0';
printf("The result : %s\n",result);
return 0;
}
```

Load Balancing

1 the Mandelbrot set and Granularity

- computing a file with grayscales
- grain size and granularity

2 Static Work Load Assignment

- granularity considerations
- static work load assignment with MPI
- an implementation with mpi4py

3 Dynamic Work Load Balancing

- scheduling jobs to run in parallel
- dynamic work load balancing with MPI
- probing in Python and Julia

the specifications

```
from mpi4py import MPI
```

```
COMM = MPI.COMM_WORLD
```

```
def manager(npr, njobs, verbose=True):
```

```
    """
```

```
    The manager distributes njobs jobs to npr-1  
    workers and prints the received results.
```

```
    The njobs must be a multiple of npr-1.
```

```
    """
```

```
def worker(i, verbose=True):
```

```
    """
```

```
    The i-th worker receives a number.
```

```
    The worker terminates if the number is -1,  
    otherwise it sends to the manager the  
    corresponding character following the letter 'a'.
```

```
    """
```

the main program

```
def main(verbose=True):  
    """  
    Runs a manager/worker static load distribution.  
    """  
    rank = COMM.Get_rank()  
    size = COMM.Get_size()  
    if rank > 0:  
        worker(rank, verbose)  
    else:  
        manager(size, size*(size-1), verbose)
```


code for the worker

```
def worker(i, verbose=True):
    if verbose:
        print('Hello from worker', i)
    while True:
        nbr = COMM.recv(source=0, tag=11)
        if verbose:
            print('-> worker', i, 'received', nbr)
        if nbr == -1:
            break
        chrnbr = chr(ord('a') + nbr)
        if verbose:
            print('-> worker', i, 'computed', chrnbr)
        COMM.send(chrnbr, dest=0, tag=11)
```

the manager distributes jobs

```
def manager(npr, njobs, verbose=True):
    if verbose:
        print('Manager distributes', njobs,
              'jobs to', npr-1, 'workers')
    result = ''
    jobcnt = 0
    while jobcnt < njobs:
        for i in range(1, npr):
            jobcnt = jobcnt + 1
            nbr = 1 + (jobcnt % (npr-1))
            if verbose:
                print('-> manager sends job', jobcnt,
                      'to worker', i)
            COMM.send(nbr, dest=i, tag=11)
```

the manager receives results

code continued:

```
for i in range(1, npr):
    data = COMM.recv(source=i, tag=11)
    if verbose:
        print('-> manager received', data,
              'from worker', i)
    result = result + data
for i in range(1, npr):
    if verbose:
        print('-> manager sends -1 to worker', i)
    COMM.send(-1, dest=i, tag=11)
print('the result :', result)
print('number of characters :', len(result))
print('        number of jobs :', njobs)
```

Load Balancing

1 the Mandelbrot set and Granularity

- computing a file with grayscales
- grain size and granularity

2 Static Work Load Assignment

- granularity considerations
- static work load assignment with MPI
- an implementation with mpi4py

3 Dynamic Work Load Balancing

- **scheduling jobs to run in parallel**
- dynamic work load balancing with MPI
- probing in Python and Julia


an example

Consider scheduling 8 jobs on 2 processors:

serial 

static 

$p = 2$ 

dynamic 

$p = 2$ 

manager/worker algorithm for dynamic load balancing

Scheduling n jobs on p processors, $n \gg p$:

- node 0 manages the job queue,
- nodes 1 to $p - 1$ are compute nodes.

The manager executes the following algorithm:

```
for  $j$  from 1 to  $p - 1$  do
  send job  $j - 1$  to compute node  $j$ ;
while not all jobs are done do
  if a node is done with a job then
    collect result from node;
  if there is still a job left to do then
    send next job to node;
  else send termination signal.
```

Load Balancing

1 the Mandelbrot set and Granularity

- computing a file with grayscales
- grain size and granularity

2 Static Work Load Assignment

- granularity considerations
- static work load assignment with MPI
- an implementation with mpi4py

3 Dynamic Work Load Balancing

- scheduling jobs to run in parallel
- **dynamic work load balancing with MPI**
- probing in Python and Julia

probing for incoming messages with `MPI_Iprobe`

To check for incoming messages, the nonblocking (or Immediate) MPI command has the syntax:

```
MPI_Iprobe(source, tag, comm, flag, status)
```

where the arguments are

<code>source</code>	:	rank of source or <code>MPI_ANY_SOURCE</code>
<code>tag</code>	:	message tag or <code>MPI_ANY_TAG</code>
<code>comm</code>	:	communicator
<code>flag</code>	:	address of logical variable
<code>status</code>	:	status object

If `flag` is true on return, then `status` contains the rank of the source of the message and can be received.

code for the manager

The manager starts with distributing the first $p - 1$ jobs:

```
int manager ( int p, int n )
{
    char result[n+1];
    int j;

    for(j=1; j<p; j++) /* distribute first jobs */
    {
        if(v == 1) printf("sending %d to %d\n", j-1, j);
        MPI_Send(&j, 1, MPI_INT, j, tag, MPI_COMM_WORLD);
    }
}
```

probing and receiving messages

```
int done = 0;
int nextjob = p-1;
do
    /* probe for results */
{
    int flag;
    MPI_Status status;
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG,
               MPI_COMM_WORLD, &flag, &status);
    if(flag == 1)
    {
        /* collect result */
        char c;
        j = status.MPI_SOURCE;
        if(v == 1) printf("received message from %d\n", j);
        MPI_Recv(&c, 1, MPI_CHAR, j, tag,
                 MPI_COMM_WORLD, &status);
    }
}
```

sending the next job

```
if(v == 1) printf("received %c from %d\n",c,j);
result[done++] = c;
if(v == 1) printf("#jobs done : %d\n",done);
if(nextjob < n) /* send the next job */
{
    if(v == 1) printf("sending %d to %d\n",
                      nextjob,j);
    MPI_Send(&nextjob,1,MPI_INT,j>tag,
             MPI_COMM_WORLD);
    nextjob = nextjob + 1;
}
```

at the end of the queue

```
else /* send -1 to signal termination */
{
    if(v == 1) printf("sending -1 to %d\n", j);
    flag = -1;
    MPI_Send(&flag, 1, MPI_INT, j, tag,
            MPI_COMM_WORLD);
}
}
} while (done < n);
result[done] = '\0';
printf("The result : %s\n", result);
return 0;
}
```

Load Balancing

1 the Mandelbrot set and Granularity

- computing a file with grayscales
- grain size and granularity

2 Static Work Load Assignment

- granularity considerations
- static work load assignment with MPI
- an implementation with mpi4py

3 Dynamic Work Load Balancing

- scheduling jobs to run in parallel
- dynamic work load balancing with MPI
- probing in Python and Julia

probing in Python and Julia

With mpi4py:

```
iprobe(comm, source=ANY_SOURCE, tag=ANY_TAG,  
        status=None)
```

With MPI.jl:

```
ismessage, (status|nothing) =  
Iprobe(src::Integer, tag::Integer,  
        comm::Comm)
```

specification of a Julia program

```
using MPI
MPI.Init()
COMM = MPI.COMM_WORLD
```

```
"""
```

```
    function manager(p::Int, n::Int,
                    verbose::Bool=true)
```

distributes n jobs to $p-1$ workers
and prints the received results.
Assumed is that $n \geq p-1$.

```
"""
```

```
"""
```

```
    function worker(i::Int, verbose::Bool=true)
```

The i -th worker receives a number.
The worker terminates if the number is -1 ,
otherwise it sends to the manager
the corresponding character following 'a'.

```
"""
```

the main program

```
"""
    function main(verbose::Bool=true)

runs a manager/worker dynamic load distribution.
"""
function main(verbose::Bool=true)
    myid = MPI.Comm_rank(COMM)
    size = MPI.Comm_size(COMM)
    if myid == 0
        print("Give the number of jobs : ")
        line = readline(stdin)
        njobs = parse{Int, line}
    end
    MPI.Barrier(COMM)
    if myid == 0
        manager(size, njobs)
    else
        worker(myid)
    end
    MPI.Barrier(COMM)
end
```


definition of the worker

```
function worker(i::Int, verbose::Bool=true)
    println("Worker ", i, " says hello.")
    while true
        nbr = MPI.recv(COMM; source=0, tag=11)
        println("-> worker ", i, " received ", nbr)
        if nbr == -1
            break
        end
        chrnbr = Char(Int('a') + nbr)
        MPI.send(chrnbr, COMM; dest=0, tag=11)
    end
end
```

the manager sends the first jobs

```
function manager(p::Int, n::Int,  
                verbose::Bool=true)  
    if verbose  
        println("Manager distributes ", n,  
                " jobs to ", p-1, " workers ...")  
    end  
    result = ""  
    for j=1:p-1  
        println("-> manager sends job ", j,  
                " to worker ", j)  
        MPI.send(j, COMM; dest=j, tag=11)  
    end
```

the start of the manager loop

```
jobcnt = p-1 # sent already p-1 jobs
done = 0     # counts workers that are done
while done < p-1
    for i=1:p-1
        messageSent = MPI.Iprobe(COMM; source=i)
        if messageSent
            data = MPI.recv(COMM; source=i)
            println("-> manager received ", data,
                    " from ", i)
```

Observe the use of the `Iprobe`.

sending the next jobs

code continued:

```
result = string(result, data)
jobcnt = jobcnt + 1
if jobcnt > n
    MPI.send(-1, COMM; dest=i, tag=11)
    done = done + 1
else
    nbr = 1 + (jobcnt % p)
    MPI.send(nbr, COMM; dest=i, tag=11)
end
end
end
end
println("result : ", result)
println("number of characters : ", length(result))
println("      number of jobs : ", n)
```

bibliography

- Selim G. Akl.
Superlinear performance in real-time parallel computation.
The Journal of Supercomputing, 29(1):89–111, 2004.
- George Cybenko.
Dynamic Load Balancing for Distributed Memory Processor.
Journal of Parallel and Distributed Computing, 7, 279–301, 1989.
- Yu-Kwong Kwok and Ishfaq Ahmad.
Static scheduling algorithms for allocating directed task graphs to multiprocessor.
ACM Computing Surveys, 31(4):406–469, 1999.
- C. McCreary and H. Gill. **Automatic determination of grain size for efficient parallel processing.**
Communications of the ACM, 32(9):1073–1078, 1989.

Summary + Exercises

Pleasingly parallel computations may need dynamic load balancing.

Exercises:

- 1 Apply the manager/worker algorithm for static load assignment to the computation of the Mandelbrot set. What is the speedup for 2, 4, and 8 compute nodes? To examine the work load of every worker, use an array to store the total number of iterations computed by every worker.
- 2 Apply the manager/worker algorithm for dynamic load balancing to the computation of the Mandelbrot set. What is the speedup for 2, 4, and 8 compute nodes? To examine the work load of every worker, use an array to store the total number of iterations computed by every worker.
- 3 Compare the performance of static load assignment with dynamic load balancing for the Mandelbrot set. Compare both the speedups and the work loads for every worker.