Getting computing into the classroom: building a cluster

Erik Spence

SciNet HPC Consortium

2 April 2015

Today's class

High Performance Computing (HPC) involves parallel programming. This is SciNet's specialty.

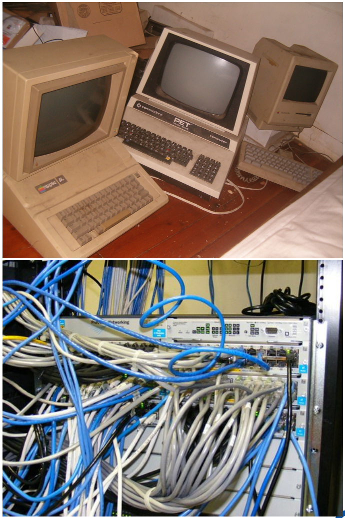
The plan for today:

- Introduce distributed-memory clusters.
- Introduce a cluster that you can create in your own classroom.
- Introduce some basic parallel programming.

Distributed Memory: Clusters

Clusters are among the simplest types of parallel computer to build:

- Take existing powerful standalone computers,
- and network them.
- Easy to build and easy to expand.
- This is what we've done with some of the computers in this lab.


(source: http://flickr.com/photos/eurleif)

Today's cluster

What about this cluster?

- The cluster is based on BCCD (Bootable Cluster CD).
- The computers boot off of a CD, DVD, USB stick, or in this case, inside a virtual machine.
- The cluster is automatically created, with nodes given the means to communicate with each other.
- In theory this can be done on top of a classroom's existing network. In practise I have never gotten this to work.

BCCD

TEACHING PARALLELISM MADE EASY

http://bccd.net

- Test to see if it'll work for your classroom. If it doesn't, another solution is to separate the computers from the school network using a hub.

BCCD

The BCCD cluster has been specifically developed for educational purposes.

- It comes with various parallel-programming approaches built in (MPI, OpenMP, CUDA).
- It comes with various software packages built-in, to learn parallel programming.
  - Monte Carlo simulation of a game of darts (Parameter-space).
  - Cellular Automata (Game of LIfe).
  - Area under the curve.
  - N-body gravitational calculations (GalaxSee).
  - Introductory CUDA programming examples.
  - And many others.
- Because it's built on Debian Linux, it's not too difficult to expand.

Setting up our cluster

```
bccd@node000:~$ pwd
/home/bccd
```

```
bccd@node000:~$ bccd-snarfhosts
```

```
bccd@node000:~$ ls
 Area-under-curve   GalaxSee-v2          Pandemic          Templates
 BW-Modules         HPL-benchmark        Parameter-space   Tests
 CUDA               Hello-world          Pictures          Tree-sort
 Desktop            Life                 Public            Videos
 Documents          Makefile             Readme            btdevices
 Downloads          Molecular-dynamics   Sieve             machines-openmpi
 GalaxSee           Music                StatKit
```

```
bccd@node000:~$ cat machines-openmpi
node011.bccd.net slots=2
node010.bccd.net slots=2
node009.bccd.net slots=2
node000.bccd.net slots=2
```

```
bccd@node000:~$
```

bccd-snarfhosts sets up the list of other computers on the network that you will be allowed to use.

We'll start on our own machines

By default your code will run on the computers ('nodes') in the order they are listed in your machines-openmpi file. We want to run on our own machines first. I've written a script that will modify your machines-openmpi file, and create a new file called 'mefirst'.

```
bccd@node000:~$ cat machines-openmpi
node011.bccd.net slots=2
node010.bccd.net slots=2
node009.bccd.net slots=2
node000.bccd.net slots=2
```

```
bccd@node000:~$ bin/make_me_first.sh
```

```
bccd@node000:~$ cat mefirst
node000.bccd.net slots=2
node011.bccd.net slots=2
node010.bccd.net slots=2
node009.bccd.net slots=2
```

We can now indicate to use our own machine first.

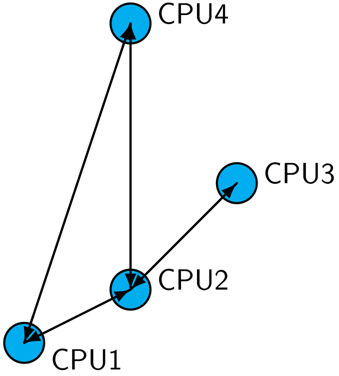
Setting up our cluster, continued

```
bccd@node000:~$ ls -F
 Area-under-curve/     GalaxSee/      Music/            StatKit/
 BW-Modules/           GalaxSee-v2/   Pandemic/         Templates/
 CUDA/                 HPC-class/     HPL-benchmark/    Tests/
 Desktop/              Hello-world/   Pictures/         Tree-sort/
 Documents/            Life/          Public/           Videos/
 Downloads/            Makefile       Readme            btdevices
 Molecular-dynamics/   Sieve/         machines-openmpi  mefirst
bccd@node000:~$ cd HPC-class
bccd@node000:~/HPC-class$ ls
bccd@node000:~/HPC-class$ mkdir Dr.S
bccd@node000:~/HPC-class$ cd Dr.S
bccd@node000:~/HPC-class/Dr.S$ ~/bin/setupMyDirectory.sh
bccd@node000:~/HPC-class/Dr.S$
```

Whatever you call your directory, make sure that it's unique. If it's not you'll end up conflicting with other computers on the network.

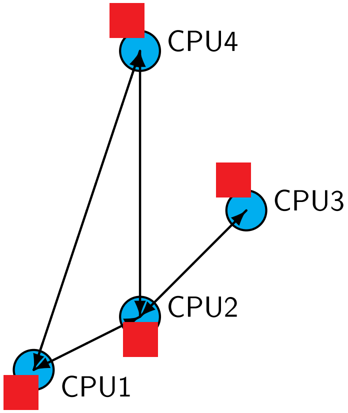
Distributed Memory: Clusters

- Each processor core is independent! Programs run on separate processors, communicating with each other when necessary.



Distributed Memory: Clusters

- Each processor core is independent! Programs run on separate processors, communicating with each other when necessary.
- Each processor has its own memory! (Or we should assume so.) Whenever it needs data from another processor, that processor needs to send it.
- All communication between processors must be hand-coded.
- MPI programming is used in this scenario.



How does MPI programming work?

MPI = Message Passing Interface.

When an MPI job is started, each processor gets the same code to run. Each processor is assigned a unique processor number by the program that launches the MPI job. The steps in the code are, in general:

1. Start the MPI session.
2. Find out how many processors are part of the job.
3. Find out which processor you are.
4. Do your work, doing different things depending upon which processor you are.
5. Close the MPI session.

After this, carry on as usual, but remember that you are part of a larger job...

Our first MPI program

```
bccd@node000:~/HPC-class/Dr.S$ pwd
/home/bccd/HPC-class/Dr.S
```

```
bccd@node000:~/HPC-class/Dr.S$ ls
```

```
bccd@node000:~/HPC-class/Dr.S$ emacs
  firstMPI.py &
```

```
bccd@node000:~/HPC-class/Dr.S$
```

Emacs is a text editor. You may use any text editor you prefer.

The & is needed to push the process into the background, otherwise you lose control over the command line until emacs is finished.

\ is a line continuation symbol. You don't need it if you write the command as one line.

```
# firstMPI.py
import pypar

# The 'size' is the number of
# processors
numprocs = pypar.size()

# 'rank' is the processor number
myid = pypar.rank()

# Some implementations will let
# you have the processor name
node = pypar.get_processor_name()

print "Hello from processor", \
  myid, "of", numprocs, \
  "on node", node

pypar.finalize()
```

Our first MPI program, continued

- Import the pypar module. This initiates the MPI session.
- Find out how many processors there are.
- Find out which processor I am.
- Close the MPI session.

```
# firstMPI.py
import pypar

# The 'size' is the number of
# processors
numprocs = pypar.size()

# The 'rank' is the processor number
myid = pypar.rank()

# Some implementations will let
# you have the processor name
node = pypar.get_processor_name()

print "Greetings from processor", \
  myid, "of", numprocs, "on node", \
  node

pypar.finalize()
```

What does it output?

So what happens when we run it?

```
bccd@node000:~/HPC-class/Dr.S$
bccd@node000:~/HPC-class/Dr.S$ python firstMPI.py
Pypar (version 2.1.5) initialised MPI OK with 1 processors
Greetings from processor 0 of 1 on node node000.bccd.net
bccd@node000:~/HPC-class/Dr.S$
```

Not bad, but that's just one processor. How do we run with multiple processors?

Running on multiple processors

We use the "mpirun" command to run on more than one processor.

```
bccd@node000:~/HPC-class/Dr.S$ mpirun -np 2 -machinefile ~/mefirst
  python firstMPI.py
Pypar (version 2.1.5) initialised MPI OK with 2 processors
Greetings from processor 0 of 2 on node node000.bccd.net
Greetings from processor 1 of 2 on node node000.bccd.net
bccd@node000:~/HPC-class/Dr.S$
```

Running on multiple processors

We use the "mpirun" command to run on more than one processor.

```
bccd@node000:~/HPC-class/Dr.S$ mpirun -np 2 -machinefile ~/mefirst
  python firstMPI.py
Pypar (version 2.1.5) initialised MPI OK with 2 processors
Greetings from processor 0 of 2 on node node000.bccd.net
Greetings from processor 1 of 2 on node node000.bccd.net
```

```
bccd@node000:~/HPC-class/Dr.S$ mpirun -np 3 -machinefile ~/mefirst
  python firstMPI.py
Pypar (version 2.1.5) initialised MPI OK with 3 processors
Greetings from processor 0 of 3 on node node000.bccd.net
Warning: Permanently added 'node009' (RSA) to the list of known hosts.
Greetings from processor 2 of 3 on node node009.bccd.net
Greetings from processor 1 of 3 on node node000.bccd.net
```

Note that the first time you run on a new node you will get a warning message letting you know that you are connecting to that node for the first time.

What is happening?

```
bccd@node000:~/HPC-class/Dr.S$ mpirun -np 2 -machinefile ~/mefirst
  python firstMPI.py
Pypar (version 2.1.5) initialised MPI OK with 2 processors
Greetings from processor 0 of 2 on node node000.bccd.net
Greetings from processor 1 of 2 on node node000.bccd.net
bccd@node000:~/HPC-class/Dr.S$
```

The command `mpirun -np 2` launches the command 'python firstMPI.py' on 2 processors. The argument `-machinefile` indicates the file containing the list of nodes to use, in this case ~/mefirst.

```
bccd@node000:~/HPC-class/Dr.S$ cat ~/mefirst
node000.bccd.net slots=2
node009.bccd.net slots=2
node011.bccd.net slots=2
node010.bccd.net slots=2
bccd@node000:~/HPC-class/Dr.S$
```

Why does it work?

Why did running "off-node" work at all?

- There's another problem. How did mpirun on the other machines know what code to run?
- Did you put the code on the other machines? If not, you likely got a "file not found" error.
- I've written a script that will copy your code to the other nodes.
- Run this script every time you edit your code and wish to run on nodes other than your own.

```
bccd@node000:~/HPC-class/Dr.S$
bccd@node000:~/HPC-class/Dr.S$ ~/bin/rsyncMyFiles.sh
Syncing files to node009
Syncing files to node011
Syncing files to node010
bccd@node000:~/HPC-class/Dr.S$
```

Sending messages

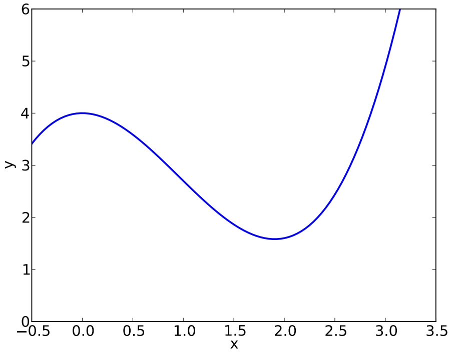
```
# secondMPI.py
import pypar
myid = pypar.rank()    # This var. is the only difference between processors.
numprocs = pypar.size()
msg = myid * 2

# Where I'm sending my message, and from whom I'm receiving a message.
sendto = (myid + 1)
recvfrom = (myid - 1)

if (sendto == numprocs): sendto = 0
if (recvfrom == -1): recvfrom = numprocs - 1

print "Processor", myid, "is sending a message to Processor", sendto
pypar.send(msg, sendto)

msg2 = pypar.receive(recvfrom)
print "Proc.", myid, "recieved the message", msg2, "from Proc.", recvfrom

pypar.finalize()
```

What does the output look like?

```
bccd@node000:~/HPC-class/Dr.S$ ~/bin/rsyncMyFiles.sh
bccd@node000:~/HPC-class/Dr.S$
bccd@node000:~/HPC-class/Dr.S$ mpirun -np 5 -machinefile ~/mefirst
  python secondMPI.py
Proc. 1 is sending a message to Proc. 2
Pypar (version 2.1.5) initialised MPI OK with 5 processors
Warning: Permanently added 'node011' (RSA) to the list of known hosts.
Processor 0 is sending a message to Processor 1
Processor 2 is sending a message to Processor 3
Processor 3 is sending a message to Processor 4
Processor 4 is sending a message to Processor 0
Proc. 3 has received message 4 from Proc. 2
Proc. 1 has received message 0 from Proc. 0
Proc. 2 has received message 2 from Proc. 1
Proc. 0 has received message 8 from Proc. 4
Proc. 4 has received message 6 from Proc. 3
bccd@node000:~/HPC-class/Dr.S$
```

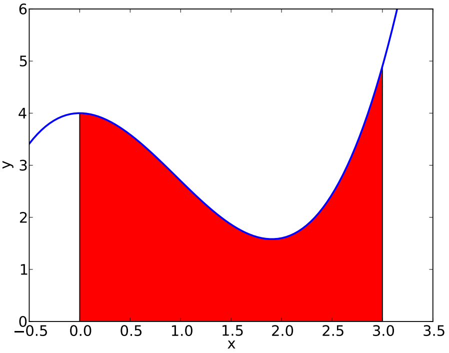
The area-under-a-curve problem

- Suppose you have a curve. How about $y = 0.7x^3 - 2x^2 + 4$?



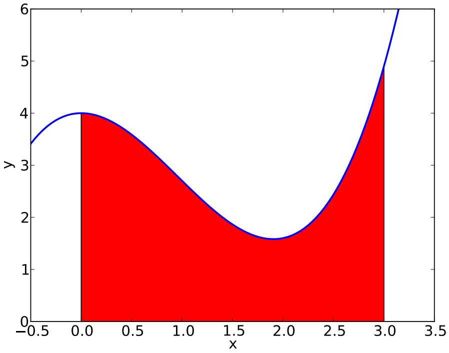
The area-under-a-curve problem

- Suppose you have a curve. How about $y = 0.7x^3 - 2x^2 + 4$?
- And suppose you want the area under the curve, between 0.0 and 3.0.



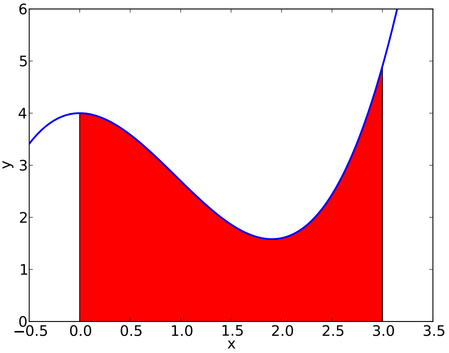
The area-under-a-curve problem

- Suppose you have a curve. How about $y = 0.7x^3 - 2x^2 + 4$?
- And suppose you want the area under the curve, between 0.0 and 3.0.
- Who cares? Well, it actually shows up all the time in scientific calculations.



The area-under-a-curve problem

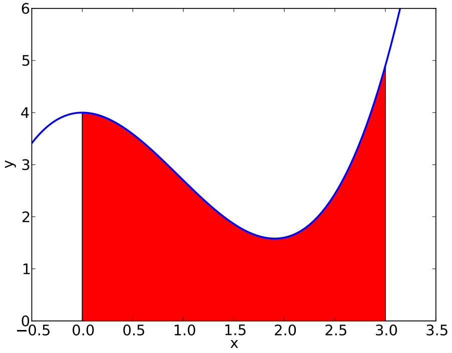
- Suppose you have a curve. How about $y = 0.7x^3 - 2x^2 + 4$?
- And suppose you want the area under the curve, between 0.0 and 3.0.
- Who cares? Well, it actually shows up all the time in scientific calculations.
- Easy! Just use calculus!



$$\begin{aligned}\text{area} &= \int_0^3 \left(0.7x^3 - 2x^2 + 4\right) dx \\ &= \left[\frac{0.7}{4}x^4 - \frac{2}{3}x^3 + 4x\right]_0^3 \\ &= 8.175\end{aligned}$$

Area under a curve, continued

- However, it's only easy to do with calculus if you CAN do it with calculus. This isn't always the case.



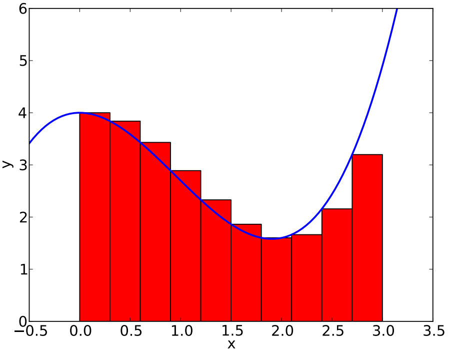
$$y = 0.7x^3 - 2x^2 + 4$$

$$\text{correct area} = 8.175$$

Area under a curve, continued

- However, it's only easy to do with calculus if you CAN do it with calculus. This isn't always the case.
- Instead, let's approximate the area under the curve using a Riemann sum.

$$\text{area} = \sum_{i=0}^{n-1} y(x_i)\Delta x$$



$$y = 0.7x^3 - 2x^2 + 4$$

$$\text{correct area} = 8.175$$

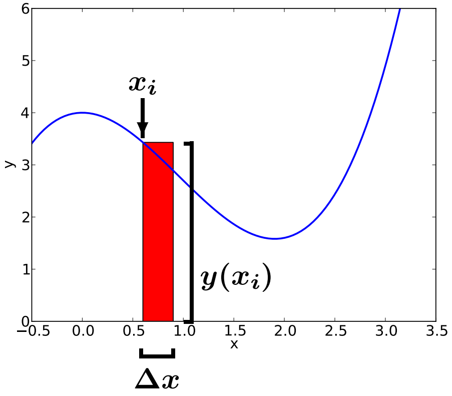
$$\text{this area} = 8.09175$$

Area under a curve, continued

- However, it's only easy to do with calculus if you CAN do it with calculus. This isn't always the case.
- Instead, let's approximate the area under the curve using a Riemann sum.

$$\mathbf{area} = \sum_{i=0}^{n-1} y(x_i)\Delta x$$

- This means chopping up the range $0 \leq x \leq 3$ into n chunks and summing over the area of the rectangles.



$$y = 0.7x^3 - 2x^2 + 4$$

$$\mathbf{correct\ area} = 8.175$$

$$\mathbf{this\ area} = 8.09175$$

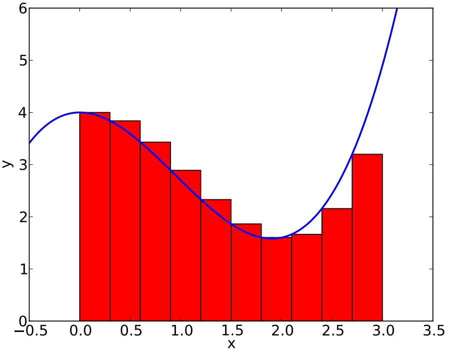
Area under a curve, continued

- As the number of bars increases, the accuracy of the estimate improves.

$$\mathbf{y = 0.7x^3 - 2x^2 + 4}$$

$$\mathbf{correct\ area = 8.175}$$

$$\mathbf{area, 10\ bars = 8.09175}$$



Area under a curve, continued

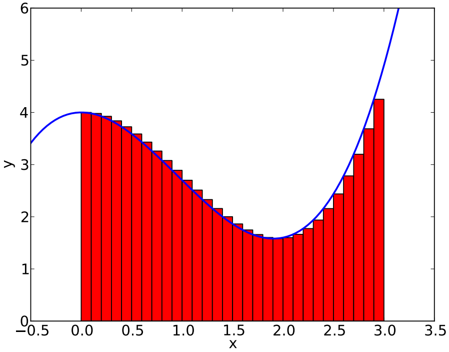
- As the number of bars increases, the accuracy of the estimate improves.

$$y = 0.7x^3 - 2x^2 + 4$$

$$\textbf{correct area} = 8.175$$

$$\textbf{area, 10 bars} = 8.09175$$

$$\textbf{area, 30 bars} = 8.13575$$



Area under a curve, continued

- As the number of bars increases, the accuracy of the estimate improves.

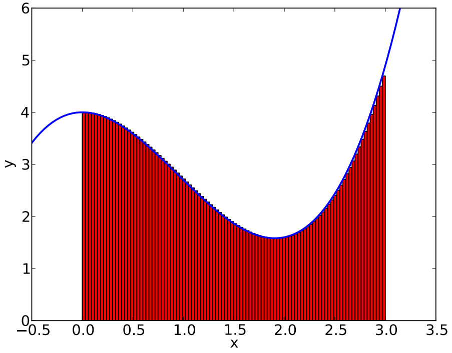
$$y = 0.7x^3 - 2x^2 + 4$$

$$\text{correct area} = 8.175$$

$$\text{area, 10 bars} = 8.09175$$

$$\text{area, 30 bars} = 8.13575$$

$$\text{area, 100 bars} = 8.1620175$$



The serial version

At the right is a code which calculates the area. It takes an optional command-line argument for the number of bars.

```
bccd@node000:~/HPC-class/Dr.S$
  python AUC.serial.py
The area is 8.09175
```

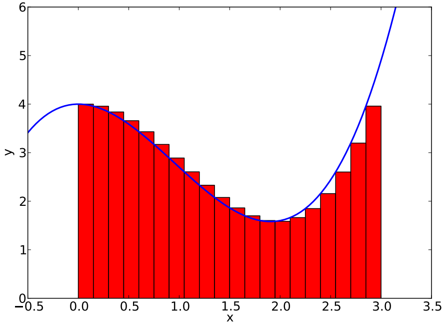
```
bccd@node000:~/HPC-class/Dr.S$
  python AUC.serial.py 100
The area is 8.1620175
```

```
bccd@node000:~/HPC-class/Dr.S$
```

It works!

```python
# AUC.serial.py
import sys

# Get the n from the command line.
if (len(sys.argv) == 2):
  n = int(sys.argv[1])
else: n = 10

area = 0.0
x = 0.0
dx = 3.0 / n

for i in range(n):
  y = 0.7 * x**3 - 2 * x**2 + 4
  area = area + y * dx
  x = x + dx

print "The area is", area
```

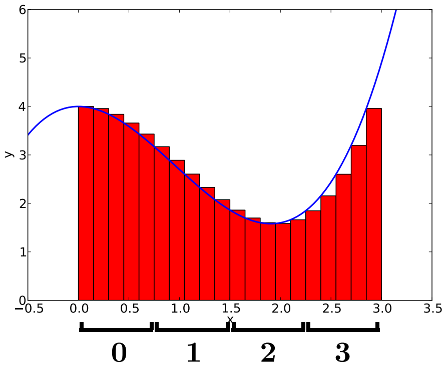
Parallelizing your code



Suppose that $n = 20$.

```
# AUC.serial.py
import sys

# Get the n from the command line.
if (len(sys.argv) == 2):
  n = int(sys.argv[1])
else: n = 10

area = 0.0
x = 0.0
dx = 3.0 / n

for i in range(n):
  y = 0.7 * x**3 - 2 * x**2 + 4
  area = area + y * dx
  x = x + dx

print "The area is", area
```

Parallelizing your code



Suppose that $n = 20$.

```
# AUC.serial.py
import sys

# Get the n from the command line.
if (len(sys.argv) == 2):
  n = int(sys.argv[1])
else: n = 10

area = 0.0
x = 0.0
dx = 3.0 / n

for i in range(n):
  y = 0.7 * x**3 - 2 * x**2 + 4
  area = area + y * dx
  x = x + dx

print "The area is", area
```

Different for each proc.

Needs to change.

Parallelizing your code, continued

How would you parallelize this?

```
# AUC.serial.py
import sys

# Get the n from the command line.
if (len(sys.argv) == 2):
  n = int(sys.argv[1])
else: n = 10

area = 0.0
x = 0.0
dx = 3.0 / n

for i in range(n):
  y = 0.7 * x**3 - 2 * x**2 + 4
  area = area + y * dx
  x = x + dx

print "The area is", area
```

Parallelizing your code, continued

How would you parallelize this?

1. Break the x axis up into numprocs pieces, and have each processor work on its piece. Each processor gets:
   - Its own starting value of x.
   - Its own number of bars to work on.

```
# AUC.serial.py
import sys

# Get the n from the command line.
if (len(sys.argv) == 2):
  n = int(sys.argv[1])
else: n = 10

area = 0.0
x = 0.0
dx = 3.0 / n

for i in range(n):
  y = 0.7 * x**3 - 2 * x**2 + 4
  area = area + y * dx
  x = x + dx

print "The area is", area
```

Different for each proc.

Parallelizing your code, continued

How would you parallelize this?

1. Break the x axis up into numprocs pieces, and have each processor work on its piece. Each processor gets:
   - Its own starting value of x.
   - Its own number of bars to work on.
2. Once each processor has calculated its part of the answer, send the totals back to processor number 0.
3. Have processor 0 sum the sub-answers and print out the answer.

```python
# AUC.serial.py
import sys

# Get the n from the command line.
if (len(sys.argv) == 2):
  n = int(sys.argv[1])
else: n = 10

area = 0.0
x = 0.0
dx = 3.0 / n

for i in range(n):
  y = 0.7 * x**3 - 2 * x**2 + 4
  area = area + y * dx
  x = x + dx

print "The area is", area
```

Different for each proc.

Your parallel code

```
# AUC.parallel.py
import pypar, sys
from numpy import zeros

numprocs = pypar.size()
myid = pypar.rank()

# Holds the sub-answers.
answer = zeros(numprocs)

# Get the n from the command line.
if (len(sys.argv) == 2):
  n = int(sys.argv[1])
else: n = 10

dx = 3.0 / n   # Width of each bar.
area = 0.0

# My starting x value.
x = myid * 3.0 / numprocs
```

```
# Number of bars for each processor.
numbars = n / numprocs

# Each proc. just works on numbars.
for i in range(numbars):
  y = 0.7 * x**3 - 2 * x**2 + 4
  area = area + y * dx
  x = x + dx

if (myid != 0): pypar.send(area, 0)

if (myid == 0):
  answer[0] = area
  for i in range(1, numprocs):
    answer[i] = pypar.receive(i)

  print "The area is", sum(answer)

pypar.finalize()
```

What's the output?

```
bccd@node000:~/HPC-class/Dr.S$ ~/bin/rsyncMyFiles.sh
```

```
bccd@node000:~/HPC-class/Dr.S$ mpirun -np 2 -machinefile ~/mefirst
  python AUC.parallel.py
Pypar (version 2.1.5) initialised MPI OK with 2 processors
The area is 8.09175
```

```
bccd@node000:~/HPC-class/Dr.S$
```

What's the output?

```
bccd@node000:~/HPC-class/Dr.S$ ~/bin/rsyncMyFiles.sh
```

```
bccd@node000:~/HPC-class/Dr.S$ mpirun -np 2 -machinefile ~/mefirst
  python AUC.parallel.py
Pypar (version 2.1.5) initialised MPI OK with 2 processors
The area is 8.09175
```

```
bccd@node000:~/HPC-class/Dr.S$ mpirun -np 2 -machinefile ~/mefirst
  python AUC.parallel.py 100
Pypar (version 2.1.5) initialised MPI OK with 2 processors
The area is 8.1620175
```

```
bccd@node000:~/HPC-class/Dr.S$
```

What's the output?

```
bccd@node000:~/HPC-class/Dr.S$ ~/bin/rsyncMyFiles.sh
```

```
bccd@node000:~/HPC-class/Dr.S$ mpirun -np 2 -machinefile ~/mefirst
 python AUC.parallel.py
Pypar (version 2.1.5) initialised MPI OK with 2 processors
The area is 8.09175
```

```
bccd@node000:~/HPC-class/Dr.S$ mpirun -np 2 -machinefile ~/mefirst
 python AUC.parallel.py 100
Pypar (version 2.1.5) initialised MPI OK with 2 processors
The area is 8.1620175
```

```
bccd@node000:~/HPC-class/Dr.S$ time mpirun -np 2 -machinefile ~/mefirst
 python AUC.parallel.py 100
Pypar (version 2.1.5) initialised MPI OK with 2 processors
The area is 8.1620175

real 0m1.154s
user 0m0.156s
sys 0m0.072s
```

```
bccd@node000:~/HPC-class/Dr.S$
```

Scaling study

We are going to perform a scaling study on the parallel area-under-the-curve code. What's a scaling study?

- A scaling study examines how much faster a code becomes as you add more and more processors. It answers the question: "how does the code scale?".
- Perfect scaling means that if you double the number of processors your code runs twice as fast.
- This is rare, due to the serial portions of the code.
- Such studies are performed on all codes used on high-performance systems, to make sure that resources are being used efficiently.
- This is a good exercise for students, to examine the utility of parallel coding.

Scaling study

We are going to perform a scaling study using the 'time' command.

```
bccd@node000:~/HPC-class/Dr.S$ time mpirun -np 1 -machinefile ~/mefirst
 python AUC.parallel_args.py 20000000
Pypar (version 2.1.5) initialised MPI OK with 1 processors
The area is [ 8.17499993]

real 0m20.979s
user 0m20.617s
sys 0m0.348s
```

```
bccd@node000:~/HPC-class/Dr.S$ time mpirun -np 2 -machinefile ~/mefirst
 python AUC.parallel_args.py 20000000
Pypar (version 2.1.5) initialised MPI OK with 2 processors
The area is [ 8.17499993]

real 0m11.827s
user 0m21.005s
sys 0m0.472s
```

```
bccd@node000:~/HPC-class/Dr.S$
```

Plotting in Python

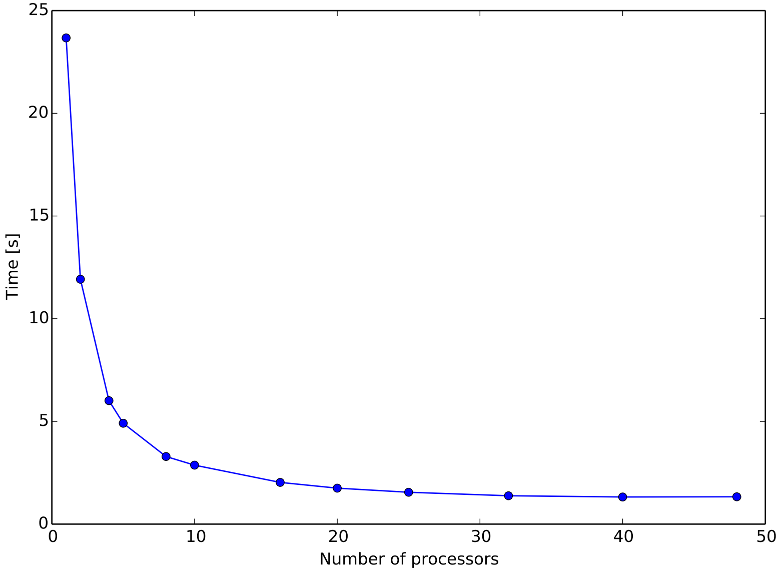
If you've never plotted in python you can use the code to the right as an example of what to do.

```
bccd@node000:~/HPC-class/Dr.S$
  python my_scaling_plot.py
bccd@node000:~/HPC-class/Dr.S$ ^C
bccd@node000:~/HPC-class/Dr.S$
```

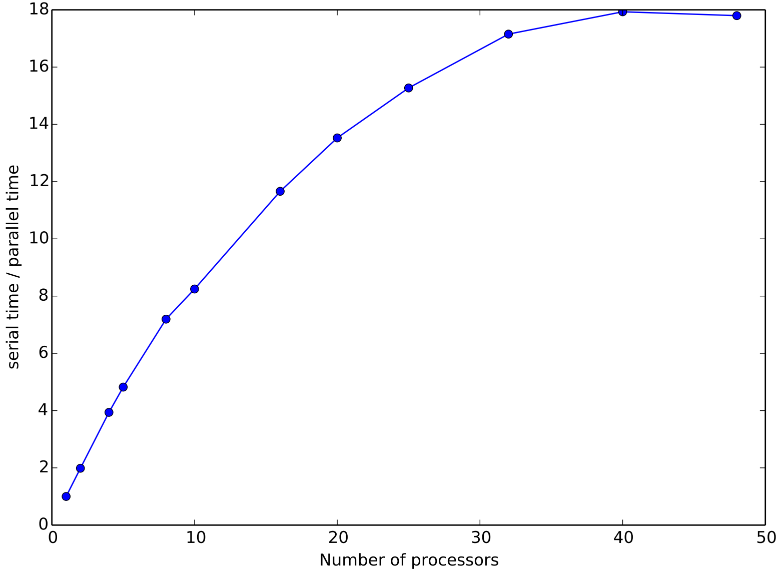
Note that by running this code you lose control over the terminal. You either need to type Ctrl-C (ˆC), or just close the figure window with the mouse.

```
# my_scaling_plot.py
from matplotlib import pylab as p

numprocs = [1, 2, 4, 5, 8, 10, 16,
  20, 25, 32, 40]
data = [23.42, 11.85, 6.09, 4.98,
  3.3, 2.77, 1.97, 1.7, 1.5, 1.36,
  1.29]

p.plot(procs, data, '-o')
p.xlabel("Number of processors")
p.ylabel("Time [s]")

p.show()
```

Scaling plot



Speedup plot


serial time / parallel time
Number of processors