

www.csiro.au

Thinking in parallel

High performance computing and the next generation of GIS

Brett Bryan

Principal Research Scientist
CSIRO Ecosystem Sciences

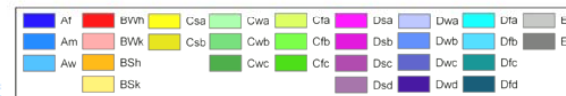
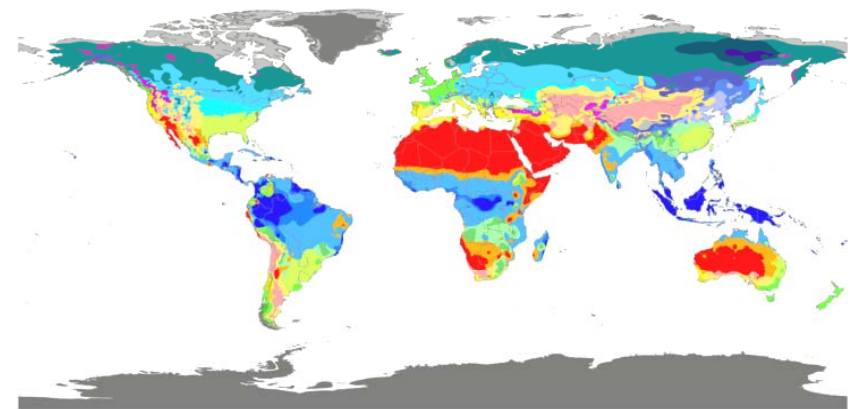
National Research
FLAGSHIPS
Sustainable Agriculture



Complex new science challenges

- Increasingly complex and interconnected problems
 - Climate, carbon, agriculture and food, economics, energy, population, water, land and soils, biodiversity
- Need to begin next generation of integrated modelling and assessment:
 - Raster-based
 - High resolution
 - National and global extents
 - Spatial and temporal processes
 - Uncertainty and sensitivity
- Huge computational demands

World map of Köppen-Geiger climate classification



DATA SOURCE: GHCN v2.0 station data
temperature (N = 4,844) and
Precipitation (N = 12,396)

PERIOD OF RECORD: All available

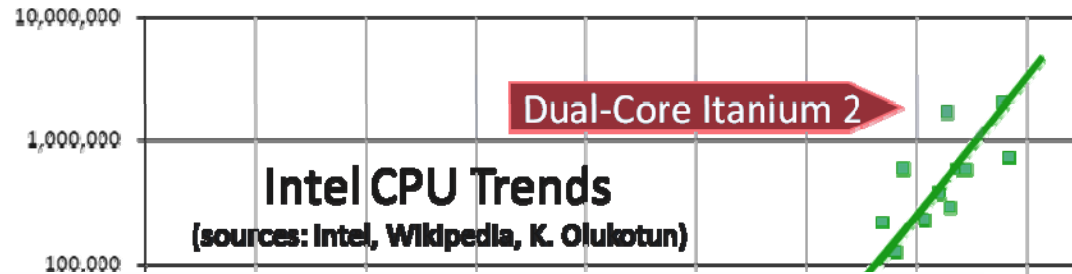
MIN LENGTH: ≥30 for each month.

RESOLUTION: 0.1 degree lat/long

Contact: Murray C. Peel (mpeel@unimelb.edu.au) for further information

Hardware trends

- Moore's law
 - Transistor density doubles every 2 years



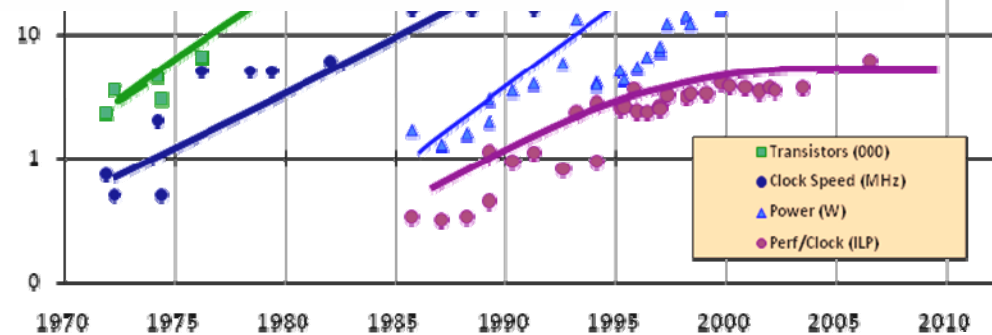
LETTER

doi:10.1038/nature09541

Ultrathin compound semiconductor on insulator layers for high-performance nanoscale transistors

Hyunhyub Ko^{1,2,3*†}, Kuniharu Takei^{1,2,3*}, Rehan Kapadia^{1,2,3*}, Steven Chuang^{1,2,3}, Hui Fang^{1,2,3}, Paul W. Leu^{1,2,3}, Kartik Ganapathi¹, Elena Plis⁵, Ha Sul Kim⁵, Szu-Ying Chen⁴, Morten Madsen^{1,2,3}, Alexandra C. Ford^{1,2,3}, Yu-Lun Chueh⁴, Sanjay Krishna⁵, Sayeef Salahuddin¹ & Ali Javey^{1,2,3}

- Number of cores increasing



Sutter 2009. <http://www.gotw.ca/publications/concurrency-ddj.htm>

Hardware trends

- Ongoing CPU advances
 - Cache, architecture, optimisation
 - Multiple cores
- Multiple connected workstations
 - High speed networks
 - Clouds
 - Clusters
- Memory is cheap and abundant
- Graphics processing units (GPUs)
 - Massively parallel
 - Able to be accessed for general purpose scientific computing



Cray-2 Supercomputer, 1985. Musée des Arts et Métiers, Paris - 2 gigaflops



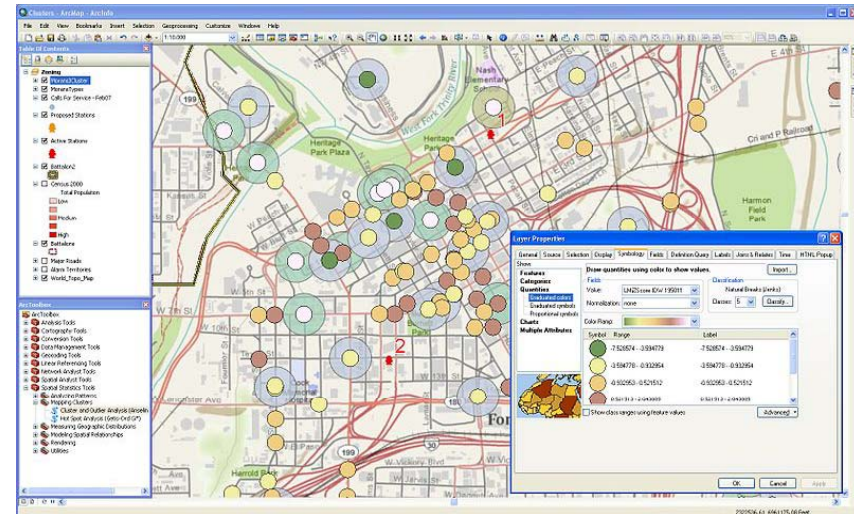
NVIDIA Tesla GPU - 2 teraflops

The problem – limitations with existing tools

- The free lunch is over...
- ESRI - Arc Macro Language
 - Comprehensive tools – vector, raster, mapping, etc.
 - Rapid development and prototyping
 - Stable, roughly same since 1986

But...

- Deprecated
- Slow – relies on disk IO, serial processing
- Language is very limited
- ESRI - ArcGIS
 - Python
 - Still relies on disk IO
 - Still serial



My conclusion

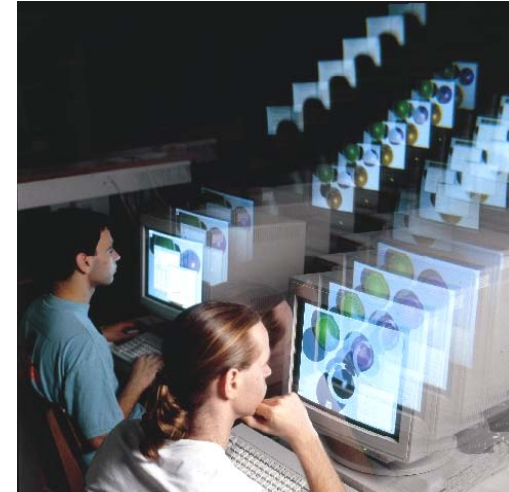
GIS software proprietors have not responded to trends in computer processing technology

Our traditional software tools cannot meet the demands of the major new science challenges

Need to adopt new, enduring software tools that can utilise, and scale with, high performance computing technology

Hardware options

- Several high-performance computing options
- Low level – desktop
 - 64-bit OS, multiple cores, lots of memory
- Mid level – WRON
 - Ticks some boxes
 - But... multiple concurrent users
- High level – cluster
 - Several options
 - GPU cluster - 1028 cores, 256 GPUs, 4 TB RAM, 50+ teraflops
 - Amongst world's fastest ~100 computers
- Other options – condor, cloud, AWS



CSIRO GPU cluster

Operating systems

- OS are critical in high performance computing
 - 32-bit OS are memory limited (3-4 GB)
 - 64-bit OS essential
- Windows (64-bit, XP, 7, Server)
 - Desktop option, easy
 - Microsoft HPC Server
 - Submit multiple jobs, parametric sweep
 - Limited flexibility and support
- Linux
 - Robust, flexible
 - Access to highly skilled support
 - In-code parallelisation possible
 - Things just seem to work better
- Support is crucial



Parallel computing - processing*

Concurrency is the next major revolution in how we write software

Herb Sutter, 2005

- **In serial:**

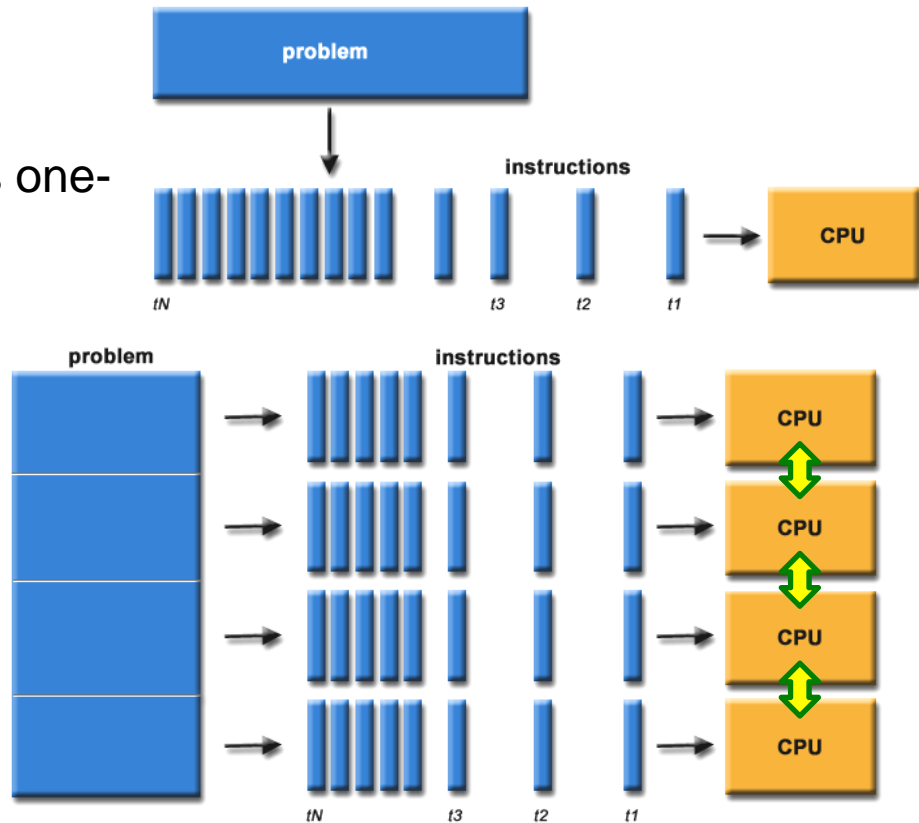
- Programs execute instructions one-by-one, like following a recipe

- **In parallel:**

- Programs execute multiple instructions simultaneously on multiple “workers”

- **Communication**

- Embarrassingly parallel
- Coarse and fine-grain parallel

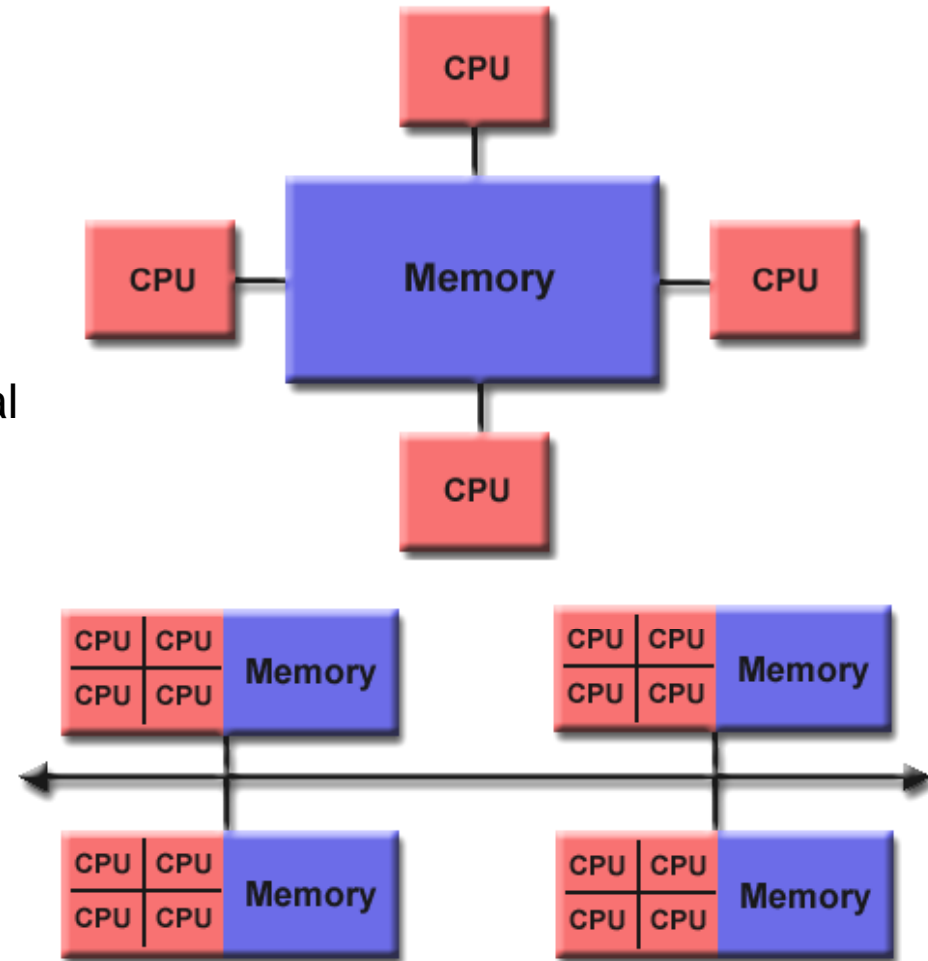


Source: Barney 2010. Introduction to parallel computing.
https://computing.llnl.gov/tutorials/parallel_comp/

*Thanks to Lawrence Murray, Sam Moskwa, CSIRO IM&T

Parallel computing - memory

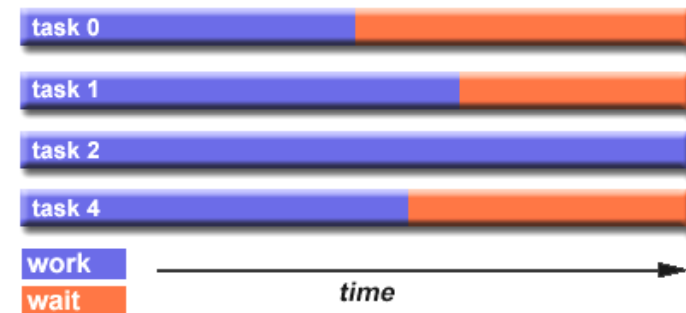
- **Shared memory**
 - Multiple processes can operate independently but share same memory
- **Distributed memory**
 - Each processor has own local memory
 - Communication via network
- **Hybrid memory**
 - Shared and distributed
 - Multiple SMP machines
- **Cache**
 - CPUs and GPUs have small but fast onboard memory



Source: Barney 2010. Introduction to parallel computing.
https://computing.llnl.gov/tutorials/parallel_comp/

Parallel computing* - implementation

- Several ways to parallelise your code
 - Data parallel
 - Split data and perform entire analysis on data subset
 - Task parallel
 - Split analysis and perform analysis subset on entire dataset
- Flynn's taxonomy
 - Single-instruction, multiple data
 - Simple to program and submit
 - Only as fast as slowest process
 - Less to go wrong
 - Multiple-instruction, multiple data
 - Master distributes jobs to workers
 - Load-balanced
 - Harder to program
 - More flexible but more to go wrong



Source: Barney 2010. Introduction to parallel computing.
https://computing.llnl.gov/tutorials/parallel_comp/

*Thanks to Lawrence Murray, Sam Moskwa, CSIRO IM&T

Software options

- Need tools that are powerful, scalable, fast, easy
- Open source (e.g. R, Python, etc.)
 - Free of charge
 - Many add-on packages provide ***diverse functionality***
 - Large, active, friendly, helpful, expert user community
 - Extensive online help
 - Often well linked (e.g. Rpy2, bindings for GRASS, QGIS, SAGA, ArcGIS)
- Proprietary (Envi IDL, Matlab, ESRI, others)
 - Expensive, especially cluster-scale installs
 - Specific, problem-oriented (image processing, engineering, GIS)



(Not so?) obvious choices

- Many choices and combinations:
 - Hardware
 - Operating system
 - Software
- Several dead-ends
- R and Python obvious choices



| Criteria | R | Python |
|-----------------------------|------------------|-------------------|
| Ease of use | Moderate | Excellent |
| Speed | Moderate | Excellent |
| Array processing | Excellent | State of the art |
| Large datasets | Limited | State of the art |
| Math/Stats | State of the art | Excellent |
| Parallelisable (CPUs) | Limited | Excellent |
| GPUs | Limited | Excellent |
| Spatial data functionality | Excellent | State of the art* |
| Visualisation ability | State of the art | Excellent |
| Flexibility and scalability | Moderate | State of the art |

Strategic direction

- Small to medium complexity jobs

- R or Python
- PC or WRON
- Can parallelise to speed code

- Large, complex jobs

- Python linking to R via **Rpy2** where needed
- GPU cluster, parallelised over multiple nodes, cores, GPUs

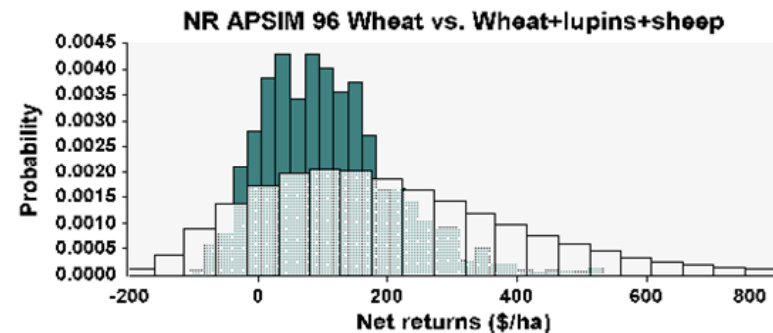
Scalability is key
With minimal effort code
can run laptop to cluster

| Criteria | R | Python |
|------------------------------------|-------------------------------|--------------------------------------------------|
| Integrated development environment | Revolution R, ESS | Spyder, Emacs, Ipython |
| Array processing | Core R | Numpy/Scipy |
| Large datasets | Revolution R | Core Python |
| Math/Stats | Core R, + + + | Numpy/Scipy, + + |
| Parallel processing (CPUs) | NWS, foreach, Rmpi | Ipython |
| GPU processing | - | PyCUDA |
| Spatial data analysis | rGDAL, GRASS, raster, sp, + + | OsGeo GDAL, Numpy, Scipy, GRASS, SAGA, ArcGIS |
| Visualisation | Lattice, ggplot2, sp, + + | Matplotlib |

Benchmarking example

- Typical example:

- Annual variation in crop yield, price, cost of production
- Large extent at high resolution – 100M cells
- 70 year time horizon
- 1,000 iterations
- Mean and variance in NPV returns for each grid cell



- Implementations:

Hardware

- 1 CPU core
- 1 – 256 CPU cores
- 1 – 64 GPUs
- 1 – 64 GPUs

Software

- ESRI Arc Macro Language
- lpython, Numpy
- lpython, Numpy, PyCUDA
- lpython, Numpy, PyCUDA ElementwiseKernel

ESRI Arc Macro Language

```
&do sim = 1 &to %nSims%  
  &do i = 0 &to %nIter%  
    &if [exists r%i% -grid] &then kill r%i%  
    r%i% = (%priceNm% * %yielNm% * spYield - %costNm%) / pow(%r%, %i%)  
  &end  
  
  &if [exists npv1 -grid] &then kill npv1  
  npv1 = sum(r0, r1, ..., r34)  
  &if [exists npv2 -grid] &then kill npv2  
  npv2 = sum(r35, r36, ..., r69)  
  &if [exists npv -grid] &then kill npv  
  npv = npv1 + npv2  
  
  &if [exists sum_new -grid] &then kill sum_new  
  sum_new = npv + sum_old  
  &if [exists sum_old -grid] &then kill sum_old  
  sum_old = sum_new  
  &if [exists std_new -grid] &then kill std_new  
  std_new = pow((npv - sum_new / %sim%), 2) + std_old  
  &if [exists std_old -grid] &then kill std_old  
  std_old = std_new  
  
&end
```

1,000 iterations

70 years

Discounted returns

Sums NPV

Sums over iterations, mean, variance

Python + Numpy

```
import numpy as np
```

Load library

```
simVals = {}
```

```
for sim in xrange(nSims):
```

```
    for i in xrange(nIter):
```

```
        simVals[sim,i,'p'] = float(max(0,np.random.randn(1)*priceSD+priceMn))
```

```
        simVals[sim,i,'y'] = float(max(0,np.random.randn(1)*yieldSD+yieldMn))
```

```
        simVals[sim,i,'c'] = float(max(0,np.random.randn(1)*costSD+costMn))
```

Populate dictionary with simulated yield, price, and cost data

```
def process(nSims, nIter, r, simVals, spatialYield, avgNum, stdNum, retNum):
```

```
    for sim in xrange(nSims):
```

```
        retNum = retNum * np.float32(0)
```

1,000 simulations over 70 years

```
        for i in xrange(nIter):
```

```
            retNum = (spatialYield * simVals[sim,i,'y'] * simVals[sim,i,'p']
```

```
                    - simVals[sim,i,'c']) / pow(r,i) + retNum
```

Discounted returns

```
            avgNum = retNum + avgNum
```

```
            stdNum = pow((retNum - avgNum / (sim + 1)), 2) + stdNum
```

```
    avgNum = avgNum / nSims
```

```
    stdNum = np.sqrt(stdNum / nSims)
```

Calculate mean and variance

```
    return avgNum, stdNum
```

```
avgNum, stdNum = process(nSims, nIter, r, simVals, spatialYieldSubset,
```

```
    zerosTemplate, zerosTemplate, zerosTemplate)
```

Call process

Python + Numpy + PyCUDA

```
import pycuda.driver as cuda
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
import pycuda.cumath as cumath
```

Load PyCUDA libraries

```
def processCUDA(nSims, nIter, r, simVals, spatialYieldNum, zerosTemplate):
    spatialYieldGPU = gpuarray.to_gpu(spatialYieldNum)
    avgGPU = gpuarray.to_gpu(zerosTemplate)
    stdGPU = gpuarray.to_gpu(zerosTemplate)
    retGPU = gpuarray.to_gpu(zerosTemplate)
```

Load Numpy arrays
into GPU memory

```
    for sim in xrange(nSims):
        retGPU = retGPU * np.float32(0)
        for i in xrange(nIter):
            retGPU = (spatialYieldGPU * simVals[sim, i, 'y'] *
                    simVals[sim, i, 'p'] - simVals[sim, i, 'c']) / pow(r, i) + retGPU
            avgGPU = retGPU + avgGPU
            stdGPU = pow((retGPU - avgGPU / (sim + 1)), 2) + stdGPU
        avgGPU = avgGPU / nSims
        stdGPU = cumath.sqrt(stdGPU / nSims)
    return avgGPU.get(), stdGPU.get()
```

Runs same calculations
- except on GPU

Copy Numpy arrays
back to main memory

```
avgNum, stdNum = processCUDA(nSims, nIter, r, simVals, spatialYieldSubset,
                             zerosTemplate)
```

Call process

Python + Numpy + PyCUDA + Elementwise

```
from pycuda.elementwise import ElementwiseKernel
```

Load library

```
npvKernel = ElementwiseKernel("int iter, float r, float *Yield, float *NPV,  
float p, float y, float c", "NPV[i] += (Yield[i] * y * p - c) / pow(r,  
iter);", "npv")  
sumKernel = ElementwiseKernel("float *retGPU, float *avgGPU", "avgGPU[i] +=  
retGPU[i];", "sum")  
stdevKernel = ElementwiseKernel("int sim, float *retGPU, float *avgGPU,  
float *stdGPU", "stdGPU[i] += pow(retGPU[i] - avgGPU[i] / (sim + 1),  
2);", "stdev")
```

Specify elementwise kernels

```
spatialYieldGPU = gpuarray.to_gpu(spatialYieldSubset)  
avgGPU = gpuarray.to_gpu(zerosTemplate)  
stdGPU = gpuarray.to_gpu(zerosTemplate)  
retGPU = gpuarray.to_gpu(zerosTemplate)
```

Load Numpy arrays
into GPU memory

```
for sim in xrange(nSims):  
    retGPU = retGPU * np.float32(0)  
    for i in xrange(nIter):
```

Call elementwise kernels

```
        npvKernel(i, r, spatialYieldGPU, retGPU, simVals[sim,i,'p'],  
                  simVals[sim, i, 'y'], simVals[sim,i,'c'])  
        sumKernel(retGPU, avgGPU)  
        stdevKernel(sim, retGPU, avgGPU, stdGPU)
```

Copy Numpy arrays
back to main memory

```
avgNum = (avgGPU / nSims).get()  
stdNum = (cumath.sqrt(stdGPU / nSims)).get()
```

Parallel CPUs - Ipython + Numpy

```
ipcontroller
```

```
mpirun -np $WORKERS --bynode ipengine
```

Start ipcontroller and ipengine in job script or at cmd prompt

```
from IPython.kernel import client
```

```
mec = client.MultiEngineClient()
```

Import and start Ipython ME client

```
mec.activate()
```

Starts the magic commands

```
%px import numpy as np
```

```
%px import math
```

Uses magic commands to import required modules on each worker

```
<"process" function code as before>
```

```
<code to split/tile data>
```

Push data and functions to workers

```
mec.push(dict(nSims=nSims, nIter=nIter, nRows=nRows, simVals=simVals, r=r))
```

```
mec.push_function(dict(process=process))
```

```
%px avgNum, stdNum = process(nSims, nIter, r, simVals, spatialYieldSubset,  
    zerosTemplate, zerosTemplate, zerosTemplate)
```

Magic to run function on workers

```
<code to collect and reassemble data>
```

Parallel CPUs – Collect and reassemble data

```
count = 0  
avgList = []  
stdList = []
```

Start counter and container lists

```
for nColsSubset in nColsSubsetList:
```

Loop over each worker/data tile

```
nArray = mec.pull('avgNum', targets=[count])  
nArray = numpy.array(nArray)
```

Pulls array tile from worker to master

```
nArray.shape = (nRows, nColsSubset)
```

Reshape Numpy array

```
avgList.append(nArray)
```

Append to list

```
nArray = mec.pull('stdNum', targets=[count])  
nArray = numpy.array(nArray)
```

```
nArray.shape = (nRows, nColsSubset)
```

```
stdList.append(nArray)
```

```
count += 1
```

Increment counter to target workers

```
avgNum = numpy.hstack(avgList)  
stdNum = numpy.hstack(stdList)
```

Reassemble result tiles to single array

Parallel GPUs - Ipython + Numpy + PyCUDA

```
ipcontroller  
mpirun -np $WORKERS --bynode ipengine
```

Start ipcontroller and ipengine in job script or at cmd prompt

```
from IPython.kernel import client  
mec = client.MultiEngineClient()  
mec.activate()
```

Start Ipython MultiEngineClient as for parallel CPU computing

```
< define "processCUDA" function code as before >  
< code to split/tile data >  
< pass data and functions to workers >
```

```
%autopx
```

Magic execute on all workers

```
import pycuda.driver as cuda  
import pycuda.autoint  
import pycuda.gpuarray as gpuarray  
import pycuda.cumath as cumath
```

Import PyCUDA libraries on each worker

Execute function on GPU workers

```
avgNum, stdNum = processCUDA(nSims, nIter, r, simVals,  
    spatialYieldSubset, zerosTemplate)
```

```
%autopx
```

End magic execute

```
< code to collect and reassemble data >
```

Parallel GPUs - Ipython + Numpy + PyCUDA + Elementwise

```
%autopx
```

Magic execute on all workers

```
< Import PyCUDA and ElementwiseKernel libraries on workers >  
< Specify ElementwiseKernels on all workers >
```

```
spatialYieldGPU = gpuarray.to_gpu(spatialYieldSubset)  
avgGPU = gpuarray.to_gpu(zerosTemplate)  
stdGPU = gpuarray.to_gpu(zerosTemplate)  
retGPU = gpuarray.to_gpu(zerosTemplate)  
for sim in xrange(nSims):  
    retGPU = retGPU * numpy.float32(0)  
    for i in xrange(nIter):  
        npvKernel(i, r, spatialYieldGPU, retGPU, simVals[sim,i,'p'],  
                 simVals[sim,i,'y'], simVals[sim,i,'c'])  
        sumKernel(retGPU, avgGPU)  
        stdevKernel(sim, retGPU, avgGPU, stdGPU)  
avgNum = (avgGPU / nSims).get()  
stdNum = (cumath.sqrt(stdGPU / nSims)).get()
```

Load Numpy arrays
into GPU memory

Call ElementwiseKernels

Copy Numpy arrays
back to main memory

```
%autopx
```

End magic execute

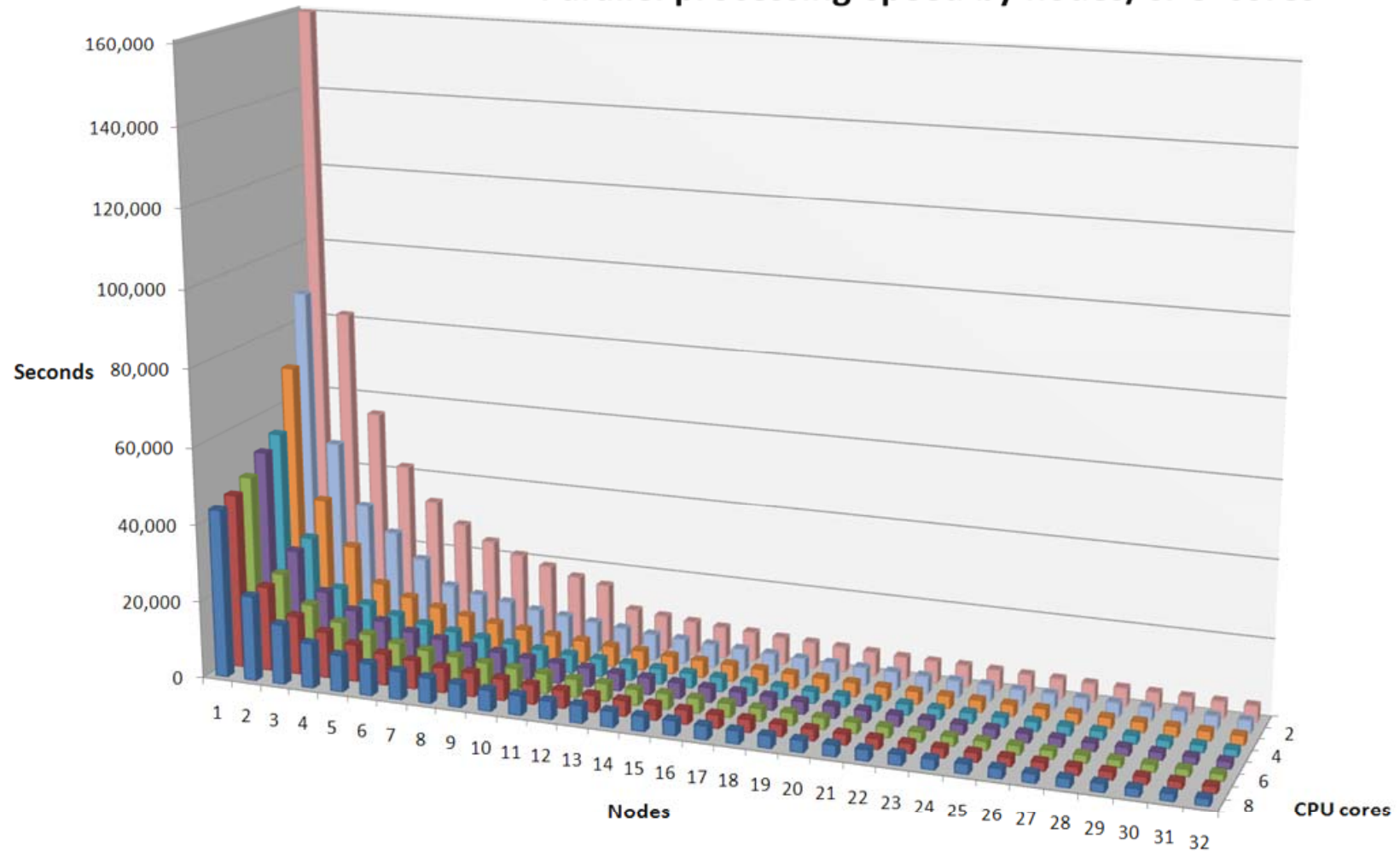
Actual speed-ups achieved

- Performance improvements:
 - GIS processing took 109 days or 15.5 weeks (equivalent)
 - Python and Numpy offer a substantial advantage
 - Further improvements in:
 - Migrating to GPU, especially using ElementwiseKernels
 - Parallelisation over multiple CPU cores, GPUs, nodes

| Hardware | Software | Time | Iterations | Effective time for 1,000 iterations | Speedup from AML | Speedup from single CPU core | Speedup from single GPU |
|---------------|-------------------------------------------|--------|------------|-------------------------------------|------------------|------------------------------|-------------------------|
| 1 CPU core | ESRI Arc Macro Language | 94,103 | 10 | 9,410,300 | | | |
| 1 CPU core | lpython, Numpy | 16,042 | 100 | 160,425 | 59 | | |
| 1 GPU | lpython, Numpy, PyCUDA | 6,388 | 1,000 | 6,388 | 1,473 | 25 | |
| 1 GPU | lpython, Numpy, PyCUDA elementwise kernel | 1,921 | 1,000 | 1,921 | 4,898 | 83 | 3.3 |
| 256 CPU cores | lpython, Numpy | 187 | 100 | 1,865 | 5,046 | 86 | 3.4 |
| 64 GPUs | lpython, Numpy, PyCUDA | 293 | 1,000 | 293 | 30,553 | 521 | 21.8 |
| 64 GPUs | lpython, Numpy, PyCUDA elementwise kernel | 148 | 1,000 | 148 | 63,643 | 1,085 | 43.2 |

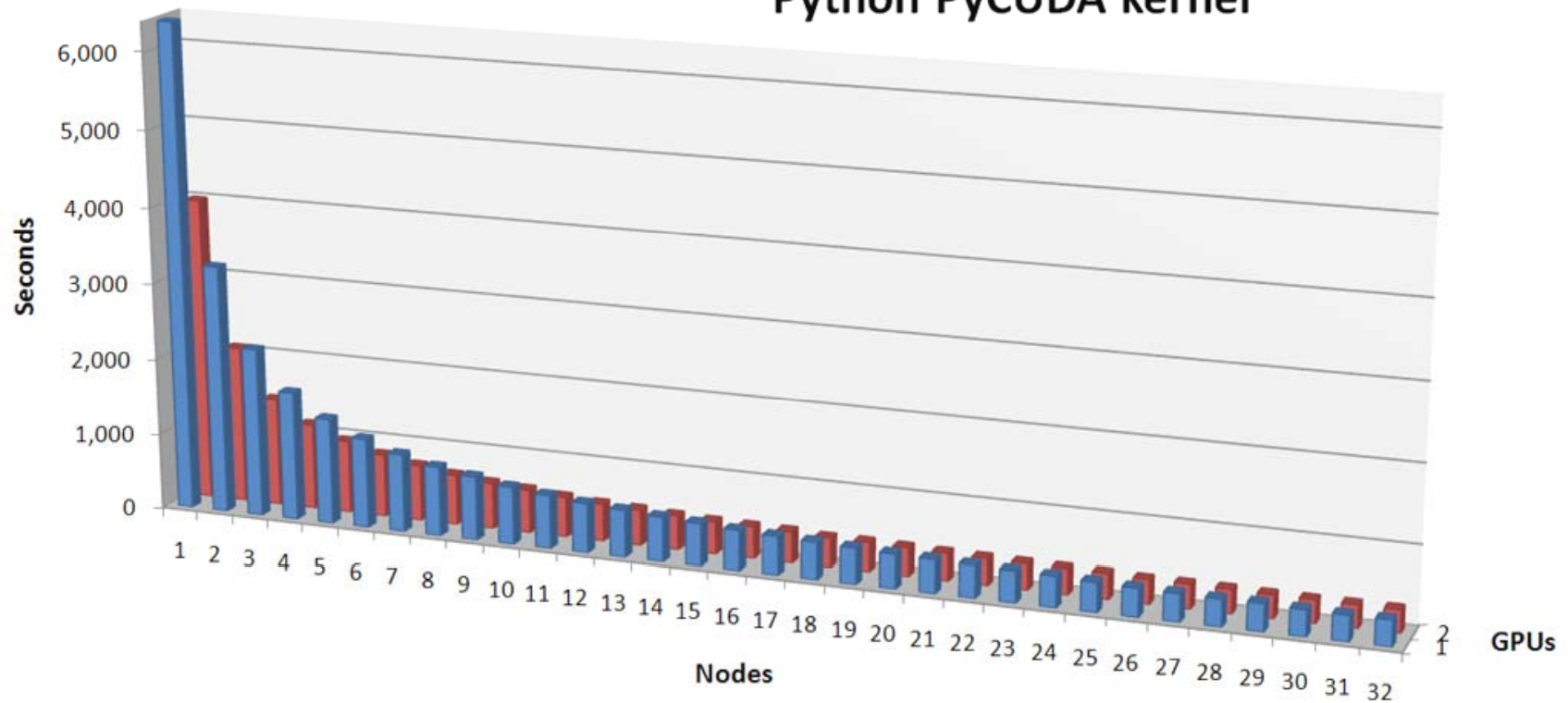
Parallel processing - multiple nodes/CPU

Parallel processing speed by nodes/CPU cores



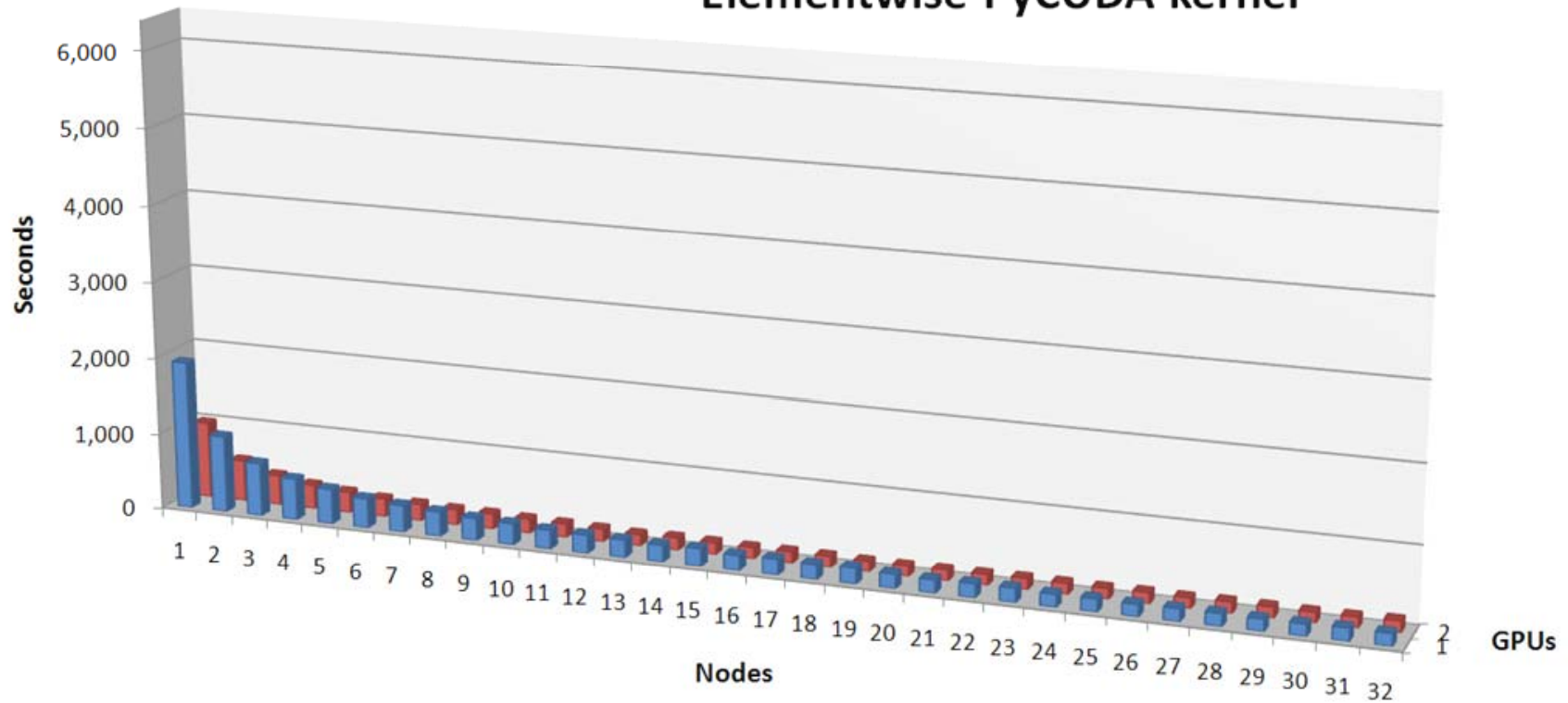
Parallel processing - multiple nodes/GPUs

Parallel computing speedup - nodes and GPUs
Python PyCUDA kernel



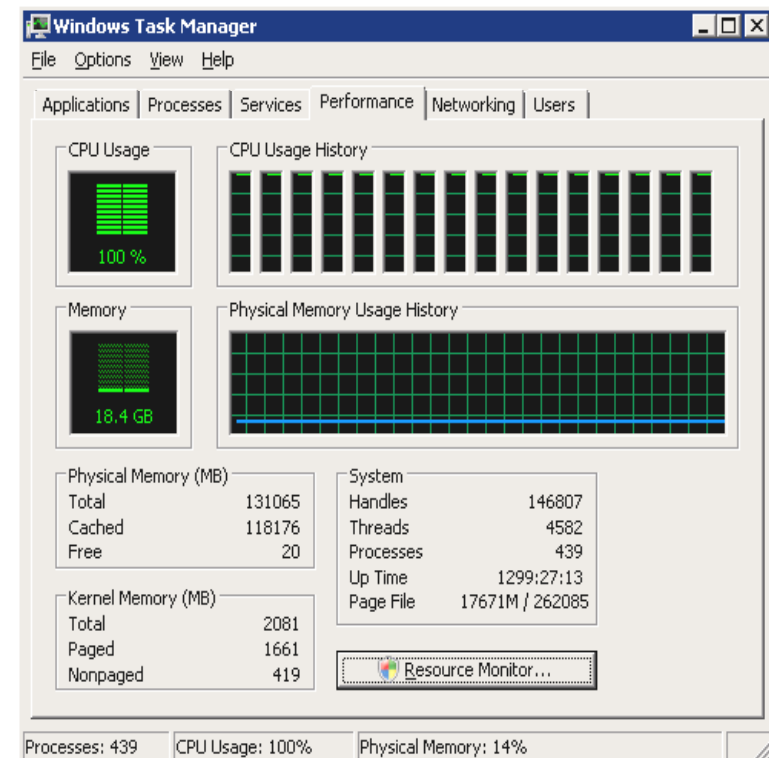
Parallel processing - multiple nodes/GPUs

Parallel computing speedup - nodes and GPUs
Elementwise PyCUDA kernel



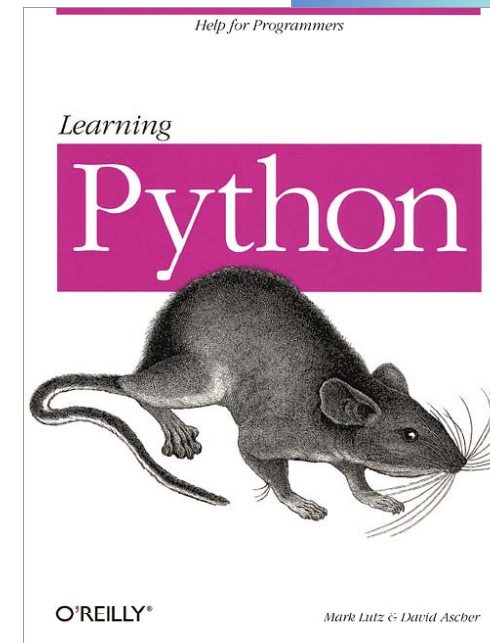
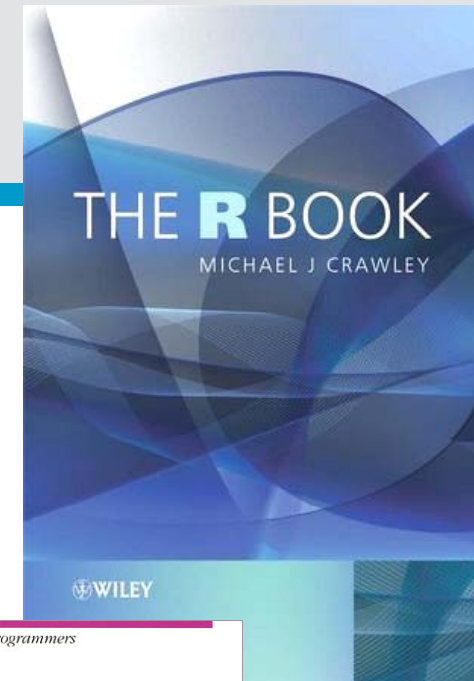
Trade-offs with parallelisation

- Benefits for scientific programming:
 - Speed and throughput
 - Data parallel is better
 - Extra access to distributed memory
 - Especially with GPUs
- The less good bits...
 - Requires thinking in parallel
 - Ahmdahl & Gustaffson
 - Waiting for queue
 - More scope for errors e.g. mutual exclusion, race conditions, deadlock, starvation, etc.
 - More difficult to develop and debug



New software tools

- The good...
 - Incredibly powerful – two stop shop
 - Use same tool for more tasks
 - Potential for breathtaking performance improvements
 - Long term, scalable solution
- The bad...
 - Steep learning curve
 - Initially slow and unproductive
- The way to do it...
 - Professional training
 - Teach yourself - buy books
 - Discipline
 - Peer support



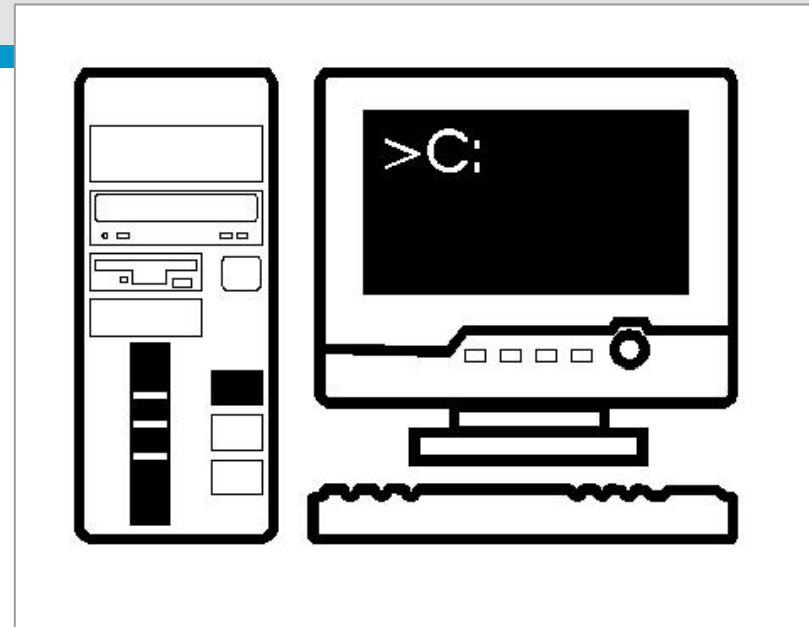
To infinity and beyond (GIS)

- In scientific computing, GIS will remain useful for:
 - Data management
 - Vector and raster integration
 - Publication quality maps
 - Smaller, once-off analyses
- But, can you afford to wait for GIS software providers to:
 - Port code to linux?
 - Access multiple CPU cores, nodes in parallel?
 - Utilise GPUs or future parallel processing technologies?
- Modern, complex spatial problems need:
 - High performance parallel computing
 - Fast, powerful, scalable, open source software tools



Conclusions

- Hardware - horses for courses
- Software
 - Powerful, open source tools
 - Scalability is king
- Most efficiencies achieved by:
 - Changing software - moving from disk IO to in-memory
 - Moving to GPU processing
 - Elementwise kernels (equivalent to parallelising over 256 cores)
- Parallelising:
 - Extra performance for extra effort
 - Extra vigilance required against bugs/errors



Brett Bryan
Principal Research Scientist
CSIRO Ecosystem Sciences
Sustainable Agriculture Flagship

Phone: +61 8 8303 8581
Email: brett.bryan@csiro.au
Web: <http://www.csiro.au>

Useful links:

CSIRO IM&T ASC Website
<http://intra.hpssc.csiro.au/>

Barney, B. 2010. Introduction to parallel computing
https://computing.llnl.gov/tutorials/parallel_comp/

www.python.org

<http://ipython.scipy.org>

<http://cran.r-project.org/>

www.csiro.au

Thank you

Aknowledgements: Thanks to Sam Moskwa and others at CSIRO IM&T ASC for their expert help.

Contact Us

Phone: 1300 363 400 or +61 3 9545 2176
Email: Enquiries@csiro.au **Web:** www.csiro.au

National Research
FLAGSHIPS
Sustainable Agriculture





TS2 ©Disney/Pixar. All Rights Reserved
©TS Disney. All Rights Reserved

 Disney.com