UNIVERSITY OF COPENHAGEN

Faculty of Science

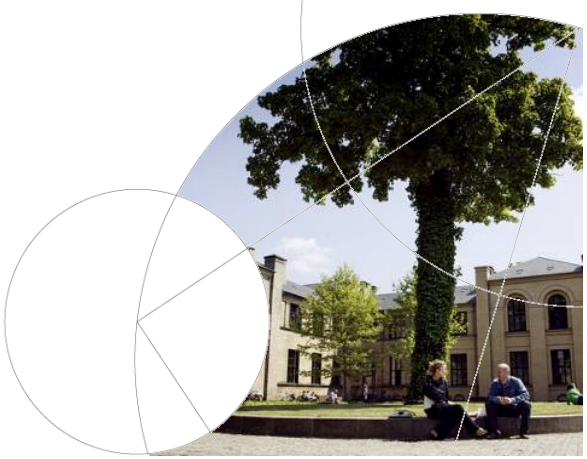
Automatic Code Orchestration from Descriptive Implementations

Professor Brian Vinter
Niels Bohr Institute

~~Automatic Code Orchestration from Descriptive Implementations~~

Prototyping for Exascale

Professor Brian Vinter
Niels Bohr Institute



Domain of interest

Users are scientists – not programmers

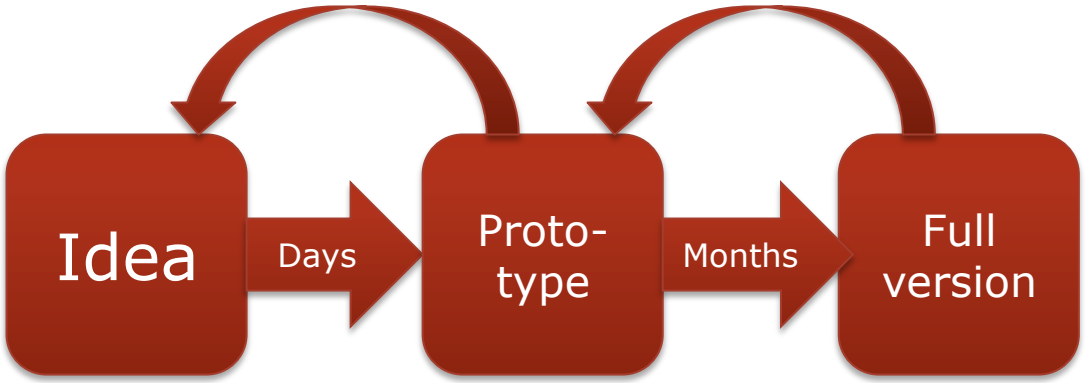
Written and maintained by a small group of people

Changes often

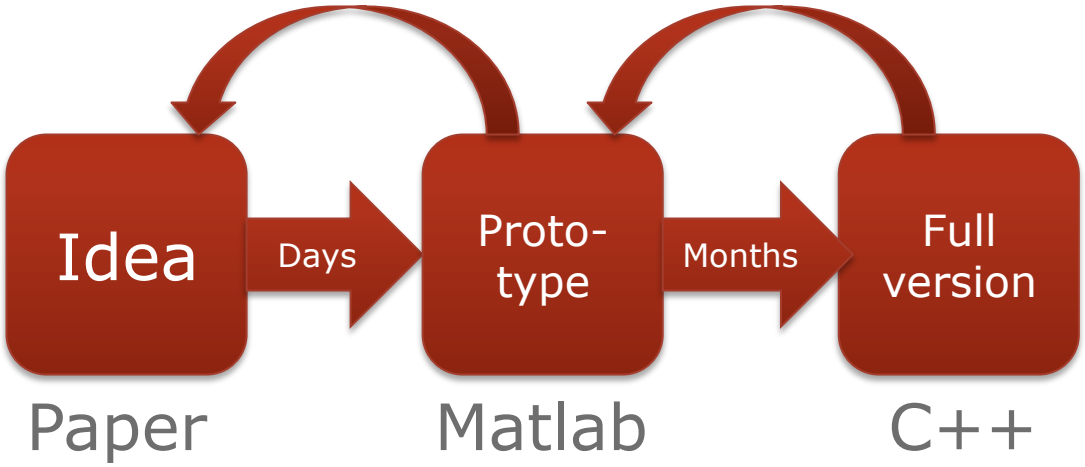
We don’t think of

- Large community codes
- Codes where the programmer is not the scientist
- Code that will run millions of times

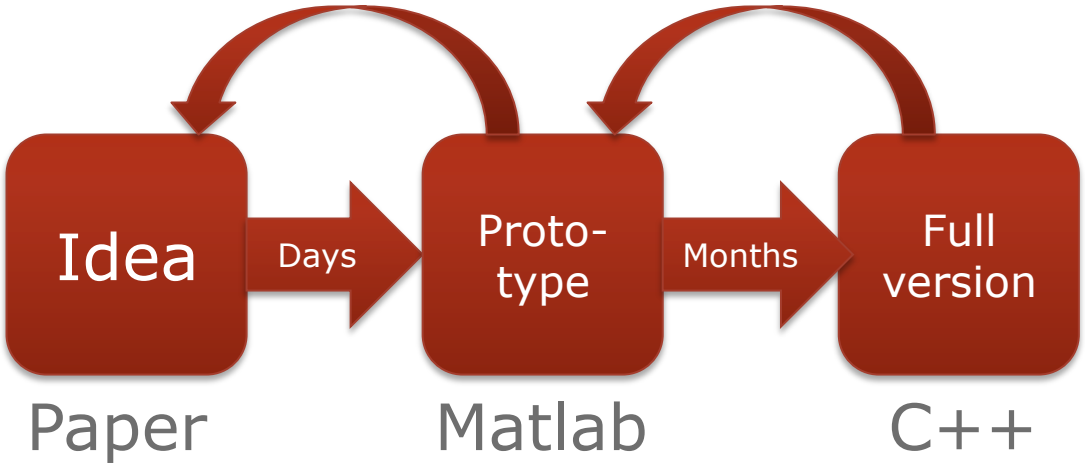
Prototyping today



Prototyping today



Prototyping today



Prototyping: TFlops

Target: PFlops

Prototyping tomorrow

Now

- Tflops => Pflops

2020

- Tflops => Pflops => Eflops

or

- Pflops => Eflops

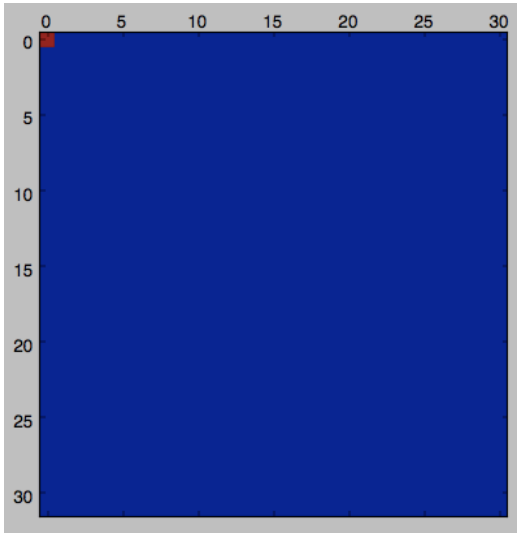
Observations

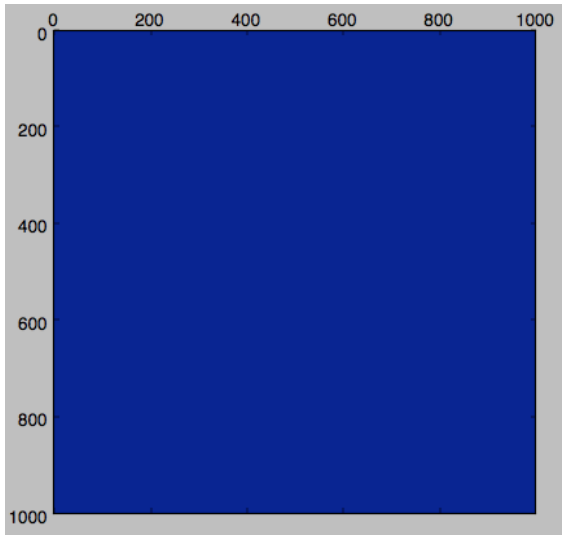
Prototyping is a fundamental requirement

Prototyping works on small datasets

- 1,000 x smaller typically
- 1,000,000 – is less realistic

We need a new approach for prototyping for Exascale





Requirements for Exascale Prototyping

Prototype must be high productivity
- Like Matlab

Must run on Petascale machines

Must have a speed in the same order as a naïve C++ implementation

Bohrium

Build around a flexible n-dimentional tensor data-structure

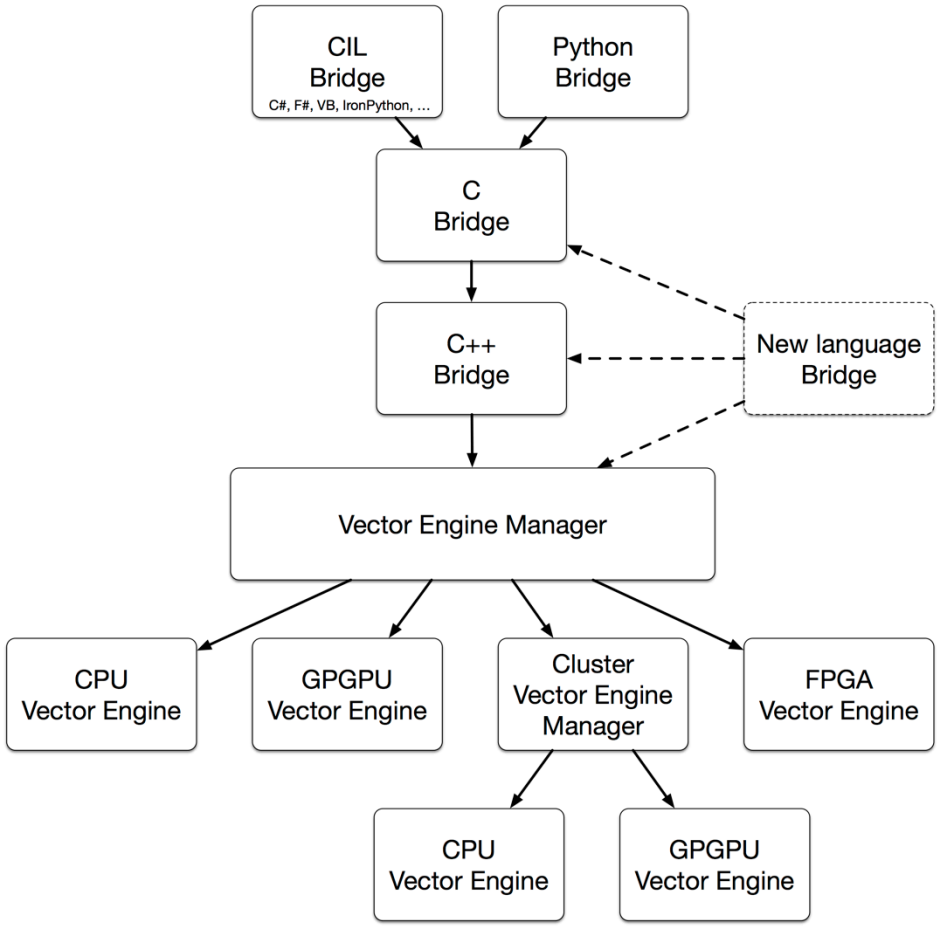
Designed to support all NumPy operations

Supported languages (for now):

- **Python/Numpy**
- C/C++
- .Net (C#, F#, …)

Supported hardware:

- Multicores
- GPGPUs
- Clusters/MPP
- (FPGAs)



Implementation

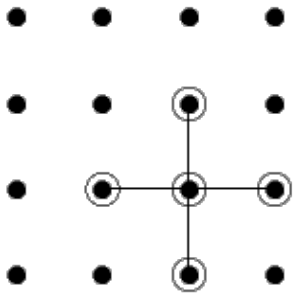
Bohrium is a Just-In-Time compiler

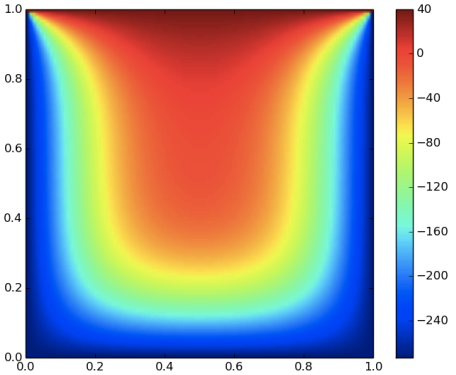
Batches all nd-array operations until a flush is required

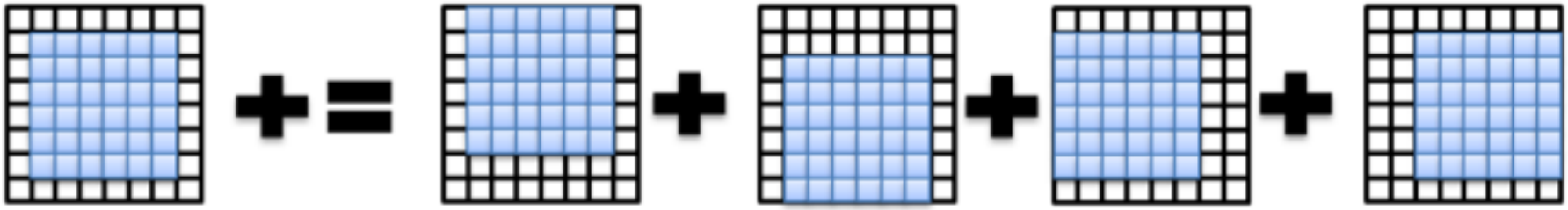
Then analyzes the batched operations and compiles target-specific code

- If it does not already exist

Programming Approach







```
bh.py
1  def solve(grid, epsilon):
2      delta  = epsilon+1
3      while delta > epsilon:
4          tmp = (grid[1:-1,1:-1]+grid[:-2,1:-1]+grid[2:,1:-1]+grid[1:-1,:-2]+grid[1:-1,2:])/5
5          delta = numpy.sum(numpy.abs(tmp-center))
6          center[:] = tmp
Line 1, Column 1                    Spaces: 2          Python
```

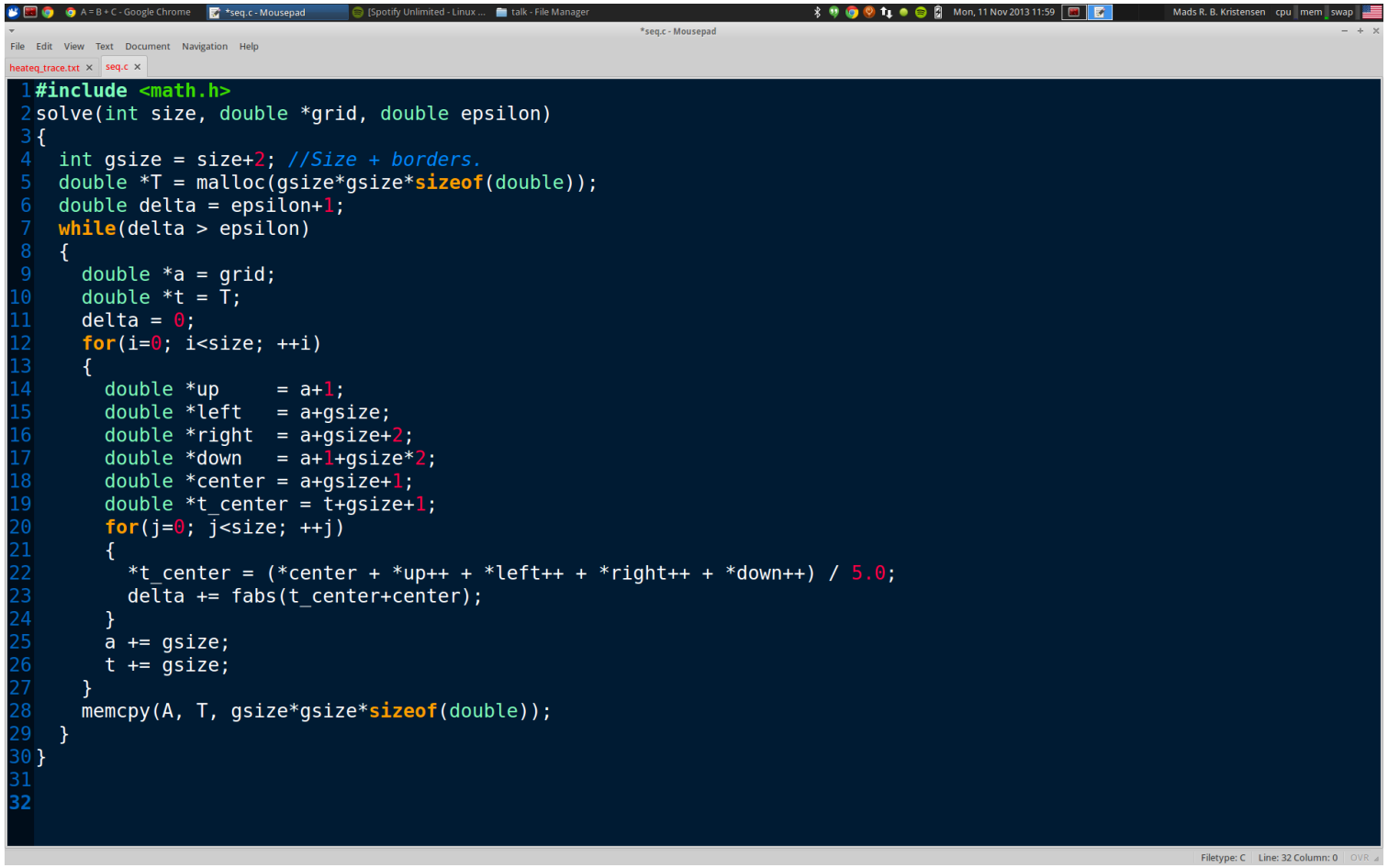
C++/OMP

jacobi.cpp

```
1   const int xdim=14000;
2   const int ydim=14000;
3
4   void solve(auto grid, double epsilon){
5       auto temp = new double[ydim][xdim];
6       double delta = epsilon + 1.0;
7       while(delta>epsilon) {
8           delta = 0.0;
9           #pragma omp parallel for reduction(+:delta) collapse(2)
10          for(int i=1; i<ydim-1; i++){
11              for(int j=1;j<xdim-1;j++){
12                  temp[i][j] = (grid[i-1][j] + grid[i+1][j] + grid[i][j] + grid[i][j-1] + grid[i][j+1])/5.0;
13                  delta += abs(temp[i][j] - grid[i][j]);
14              }
15          }
16          #pragma omp parallel for collapse(2)
17          for(int i=1;i<ydim-1; i++){
18              for(int j=1;j<xdim-1;j++){
19                  grid[i][j] = temp[i][j];
20              }
21          }
22      }
23  }
```

Line 5, Column 40 Spaces: 4 C++

Optimized C




```c
#include <math.h>
solve(int size, double *grid, double epsilon)
{
  int gsize = size+2; //Size + borders.
  double *T = malloc(gsize*gsize*sizeof(double));
  double delta = epsilon+1;
  while(delta > epsilon)
  {
    double *a = grid;
    double *t = T;
    delta = 0;
    for(i=0; i<size; ++i)
    {
      double *up     = a+1;
      double *left   = a+gsize;
      double *right  = a+gsize+2;
      double *down   = a+1+gsize*2;
      double *center = a+gsize+1;
      double *t_center = t+gsize+1;
      for(j=0; j<size; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ + *down++) / 5.0;
        delta += fabs(t_center+center);
      }
      a += gsize;
      t += gsize;
    }
    memcpy(A, T, gsize*gsize*sizeof(double));
  }
}
```

C/OMP/MPI

*omp_mpi_latency_hiding.c - Mousepad

File Edit View Text Document Navigation Help

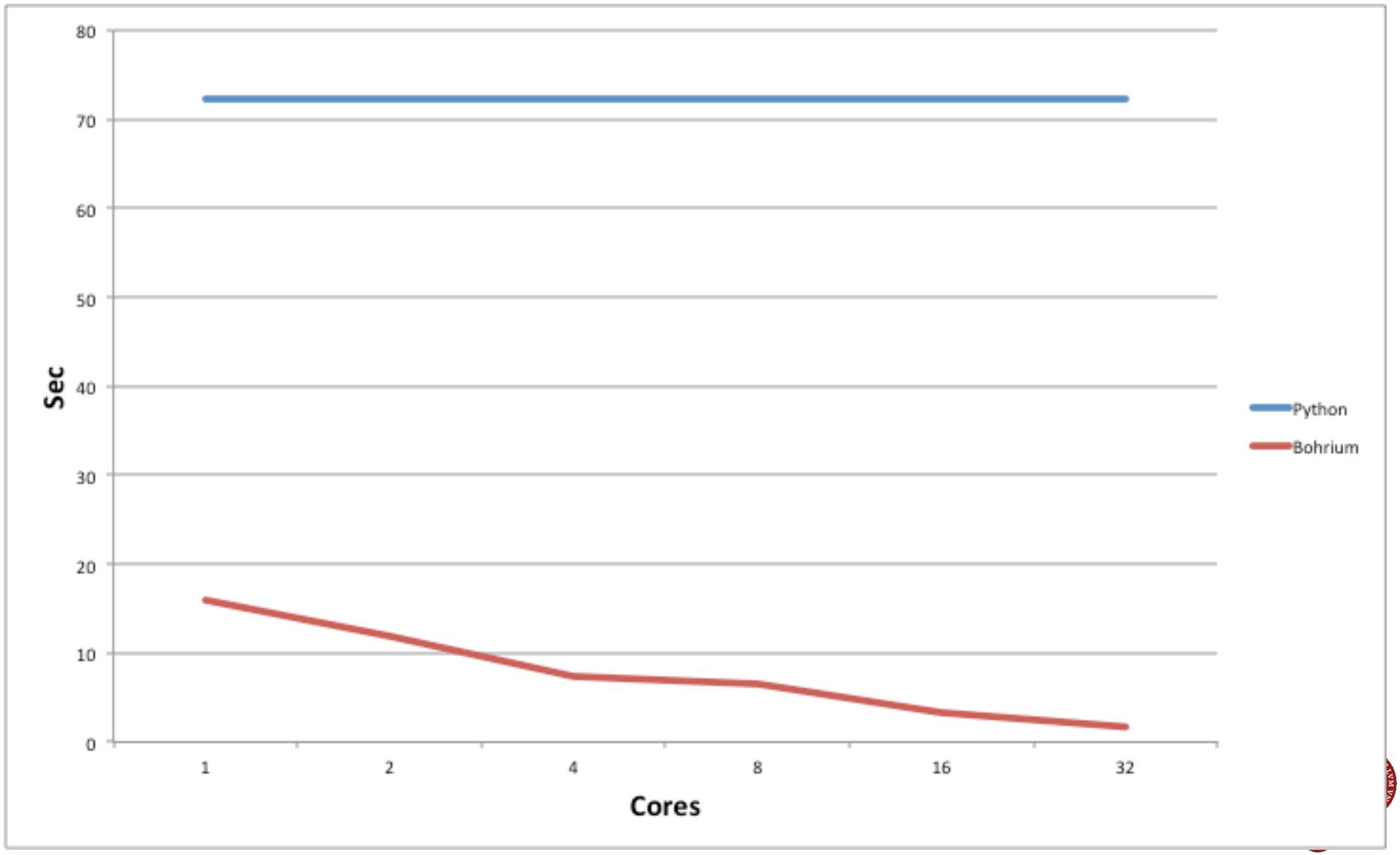
```c
#include <math.h>
#include <mpi.h>
solve(int size, double *grid, double epsilon)
{
  int gsize = SIZE+2; //Size + borders.
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
  MPI_Comm comm;
  int periods[] = {0};
  MPI_Cart_create(MPI_COMM_WORLD, 1, &worldsize,
                  periods, 1, &comm);
  int l_size = SIZE / worldsize;
  if(myrank == worldsize-1)
     l_size += SIZE % worldsize;
  int l_gsize = l_size + 2;//Size + borders.

  double delta = epsilon+1;
  while(delta > epsilon)
  {

    int p_src, p_dest;
    MPI_Request reqs[4];
    //Initiate send/receive - neighbor above
    MPI_Cart_shift(comm, 0, 1, &p_src, &p_dest);
    MPI_Isend(grid+gsize, gsize, MPI_DOUBLE, p_dest,
              1, comm, &reqs[0]);
    MPI_Irecv(grid, gsize, MPI_DOUBLE, p_src,
              1, comm, &reqs[1]);
    //Initiate send/receive - neighbor below
    MPI_Cart_shift(comm, 0, -1, &p_src, &p_dest);
    MPI_Isend(grid+(l_gsize-2)*gsize, gsize,
              MPI_DOUBLE,
              p_dest, 1, comm, &reqs[2]);
    MPI_Irecv(grid+(l_gsize-1)*gsize, gsize,
              MPI_DOUBLE,
              p_src, 1, comm, &reqs[3]);
    //Handle the non-border elements.
    delta = 0;
    #pragma omp parallel for shared(grid,T) reduction(+:delta)
    for(i=0; i<size-1; ++i)
    {
      int a = i * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size-1; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ + *down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    //Handle the upper ghost line
    MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
    {
      int a = 0 * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size-1; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ + *down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    //Handle the lower ghost line
    MPI_Waitall(2, reqs+2, MPI_STATUSES_IGNORE);
    {
      int a = (size-1) * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size-1; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ + *down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    memcpy(grid, T, gsize*gsize*sizeof(double));
    MPI_Allreduce(delta, delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  }
}
```

Display: HDMI1
Resolution: 1200 x 1920

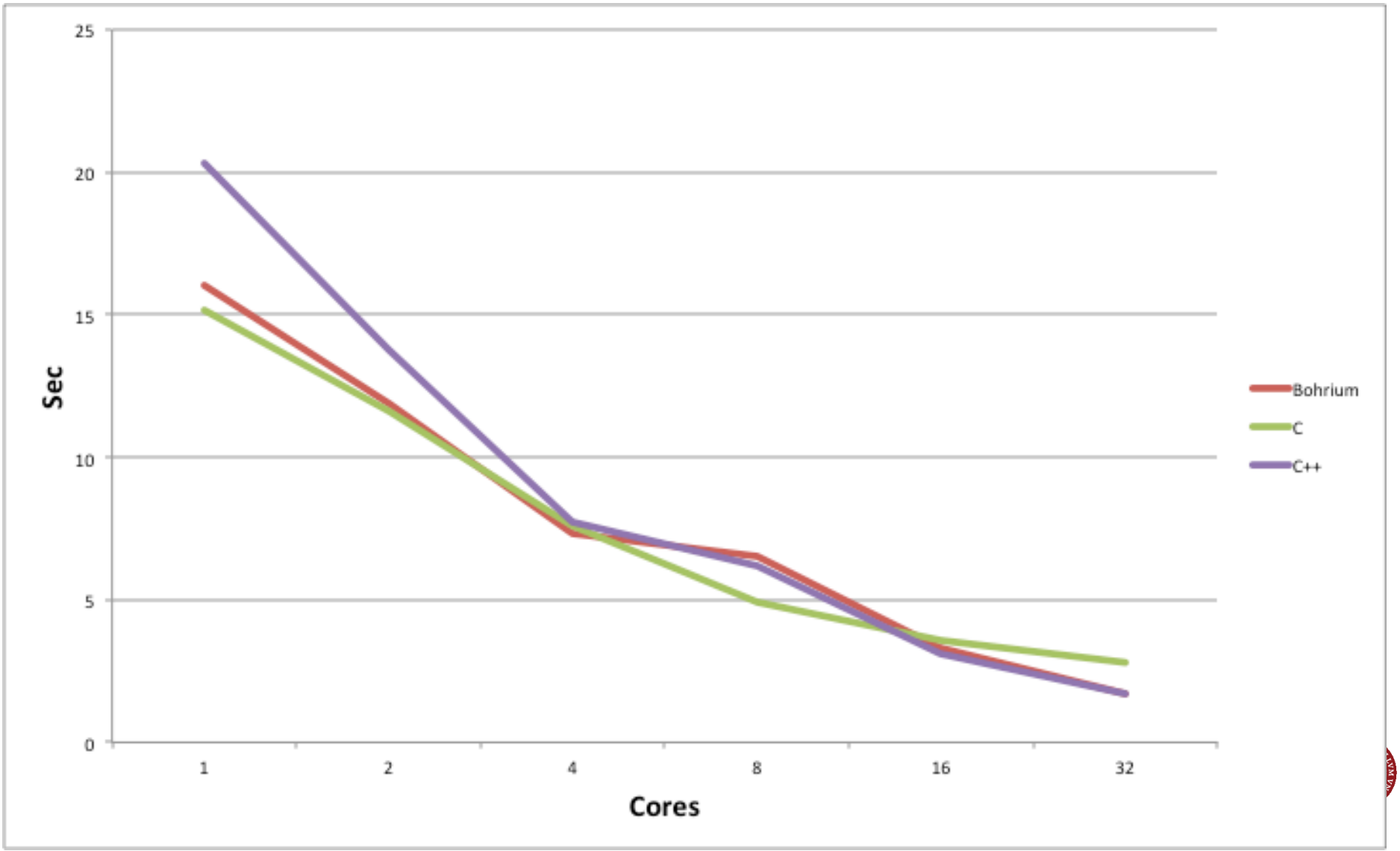
OpenCL

```cpp
#include <iostream>
#include <sys/time.h>
#include <CL/cl.hpp>
#include <stdexcept>
#include <sys/time.h>
#include <cstdio>
#include <fstream>
#include <sstream>
#include <utility>

cl::Context context;
std::vector<cl::Device> devices;
cl::CommandQueue commandQueue;
cl::Kernel kernel;

#define DTYPE double
#define STR(s) #s
#define xSTR(s) STR(s)

double solve(int size, double *grid, double epsilon)
{
    // Setup OpenCL execution system
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);
    bool foundPlatform = false;
    std::vector<cl::Platform>::iterator it;
    for (it = platforms.begin(); it != platforms.end(); ++it)
    {
        try {
            cl_context_properties props[] = {CL_CONTEXT_PLATFORM, (cl_context_properties)(*it)(),0};
            context = cl::Context(CL_DEVICE_TYPE_GPU, props);
            foundPlatform = true;
            break;
        }
        catch (cl::Error e)
        {
            foundPlatform = false;
        }
    }
    if (foundPlatform)
    {
        devices = context.getInfo<CL_CONTEXT_DEVICES>();
        commandQueue = cl::CommandQueue(context,devices[0],0);
    } else {
        throw std::runtime_error("Could not find valid OpenCL platform.");
    }

    // Read source file and compile code
    std::ifstream file("heat_eq_jacobi.cl", std::ios::in);
    if (!file.is_open())
    {
        throw std::runtime_error("Could not open source file.");
    }
    std::ostringstream srcstrs;
    srcstrs << "#define DTYPE " << xSTR(DTYPE) << "\n";
    srcstrs << file.rdbuf();
    std::string srcstr = srcstrs.str();
    cl::Program::Sources source(1,std::make_pair(srcstr.c_str(),0));
    cl::Program program(context, source);
    try {program.build(devices);}
    catch (cl::Error e)
    {
        std::cout << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices[0]) << std::endl;
    }
    kernel = cl::Kernel(program, "heat_eq_jacobi");

    // Set up model
    unsigned int w = width+2;
    unsigned int h = height+2;
    size_t grid_size = w*h*sizeof(DTYPE);
    size_t delta_size = w*sizeof(DTYPE);
    DTYPE* grid = (DTYPE*)malloc(grid_size);
    memset(grid, 0, grid_size);
    for(int j=0; j<w;j++)
    {
        grid[j] = 40.0;
        grid[j+(h-1)*w] = -273.15;
    }
    for(int j=1; j<h-1;j++)
    {
        grid[j*w] = -273.15;
        grid[j*w+w-1]= -273.15;
    }
    // Start timing
    timeval t_start,t_end;
    gettimeofday(&t_start,NULL);

    // Set up buffers and copy data
    cl::Buffer inDev = cl::Buffer(context, CL_MEM_READ_WRITE, grid_size, NULL);
    cl::Buffer outDev = cl::Buffer(context, CL_MEM_READ_WRITE, grid_size, NULL);
    cl::Buffer deltaDev = cl::Buffer(context, CL_MEM_READ_WRITE, delta_size, NULL);
    cl::Buffer deltaHost = cl::Buffer(context, CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR, delta_size, NULL);
    DTYPE* deltaHostPtr = (DTYPE*)commandQueue.enqueueMapBuffer(deltaHost, CL_TRUE, CL_MAP_WRITE, 0, delta_size);
    commandQueue.enqueueWriteBuffer(inDev, CL_FALSE, 0, grid_size, grid);
    commandQueue.enqueueCopyBuffer(inDev, outDev, 0, 0, grid_size);

    // Iterating
    for(int n=0; n<iter; ++n)
    {
        kernel.setArg(0,width);
        kernel.setArg(1,height);
        kernel.setArg(2,inDev);
        kernel.setArg(3,outDev);
        kernel.setArg(4,deltaDev);
        commandQueue.enqueueNDRangeKernel(kernel, cl::NullRange,
                                          cl::NDRange(width), cl::NullRange);
        commandQueue.enqueueReadBuffer(deltaDev, CL_FALSE, 0, delta_size, deltaHostPtr);
        commandQueue.finish();
        DTYPE delta = 0.0;
        for (unsigned int i = 0; i < width; ++i)
            delta += deltaHostPtr[i];
        cl::Buffer tmp = inDev;
        inDev = outDev;
        outDev = tmp;
    }
    commandQueue.enqueueReadBuffer(inDev, CL_TRUE, 0, grid_size, grid);
    gettimeofday(&t_end,NULL);
    double d_time = (t_end.tv_sec - t_start.tv_sec) + (t_end.tv_usec - t_start.tv_usec)/1000000.0;
    std::cout << "OpenCL - iter: " << iter << " size: " << width << "*" << height << " time: " <<
        d_time << std::endl;
    return 0;
}
```

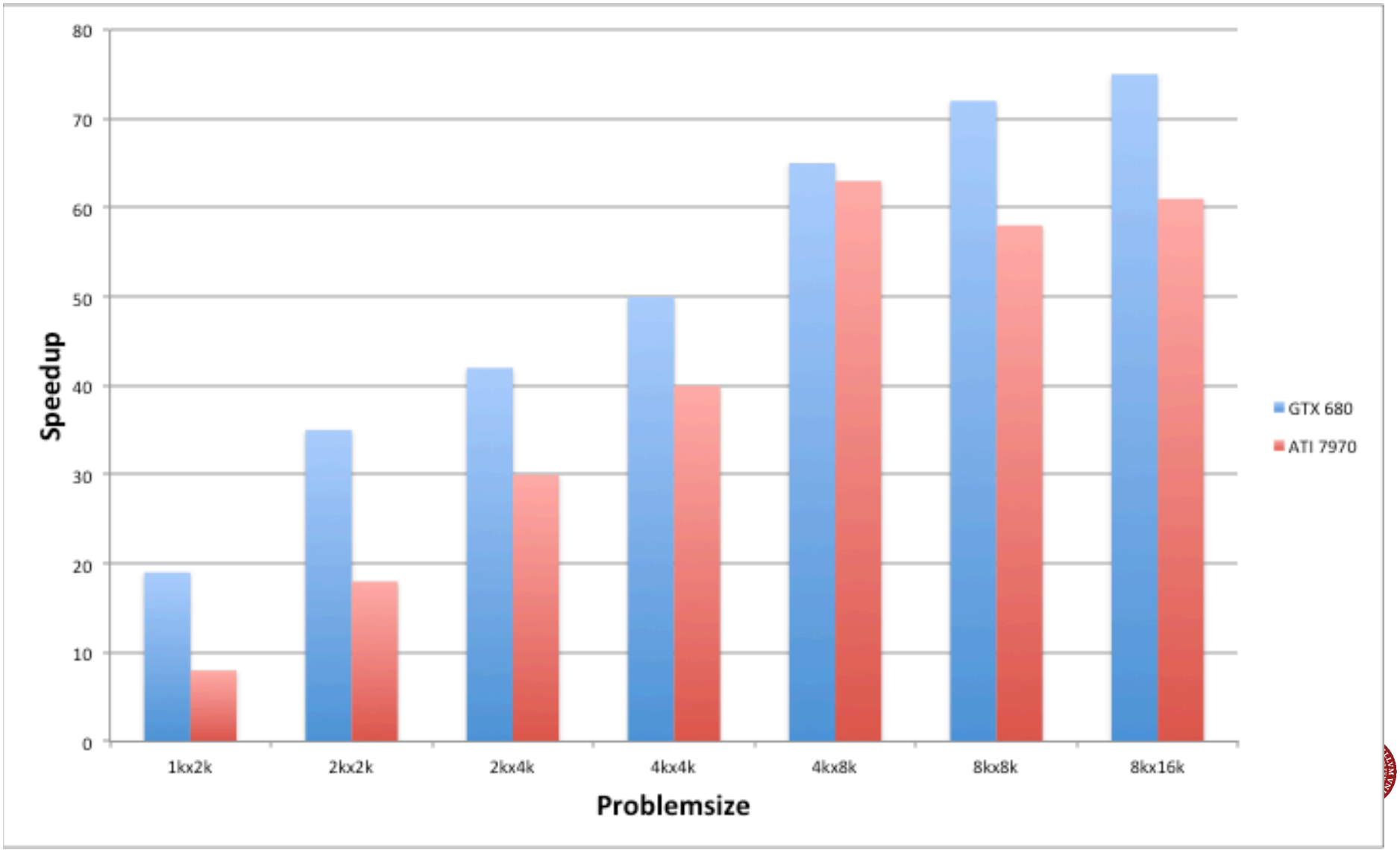
Multicore



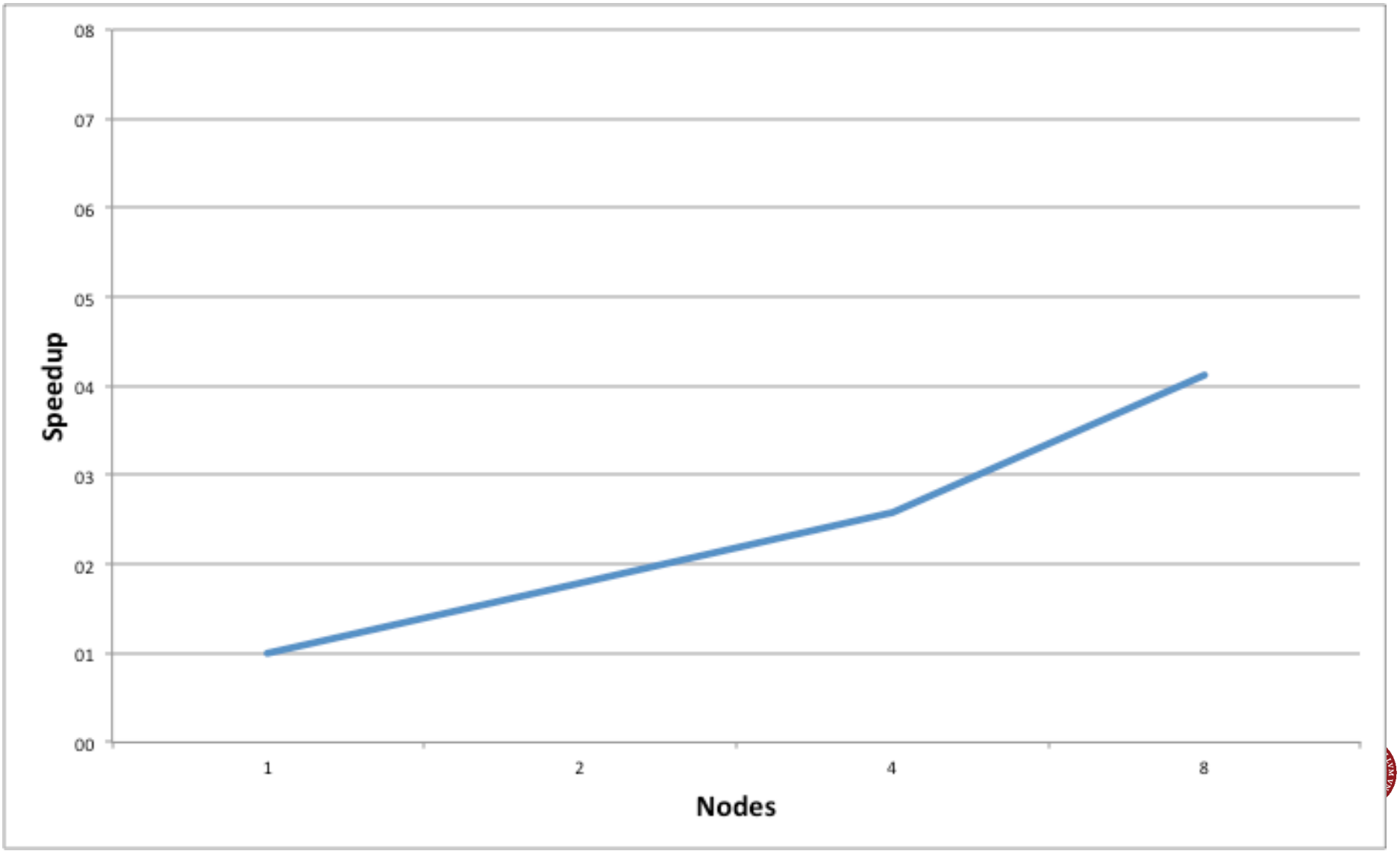
Multicore – C and C++



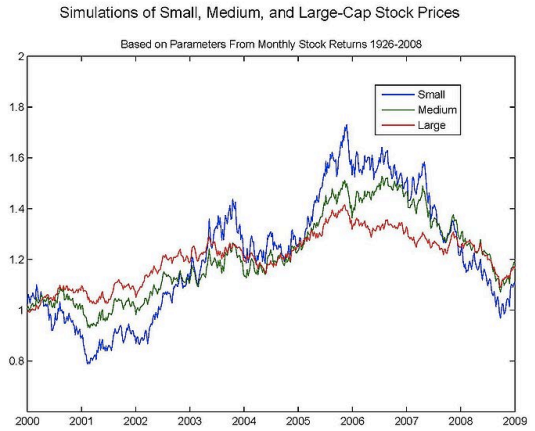
GPU


80
70
60
50
40
30
20
10
0
Speedup
1kx2k
2kx2k
2kx4k
4kx4k
4kx8k
8kx8k
8kx16k
Problemsize
GTX 680
ATI 7970

Cluster



Black-Scholes

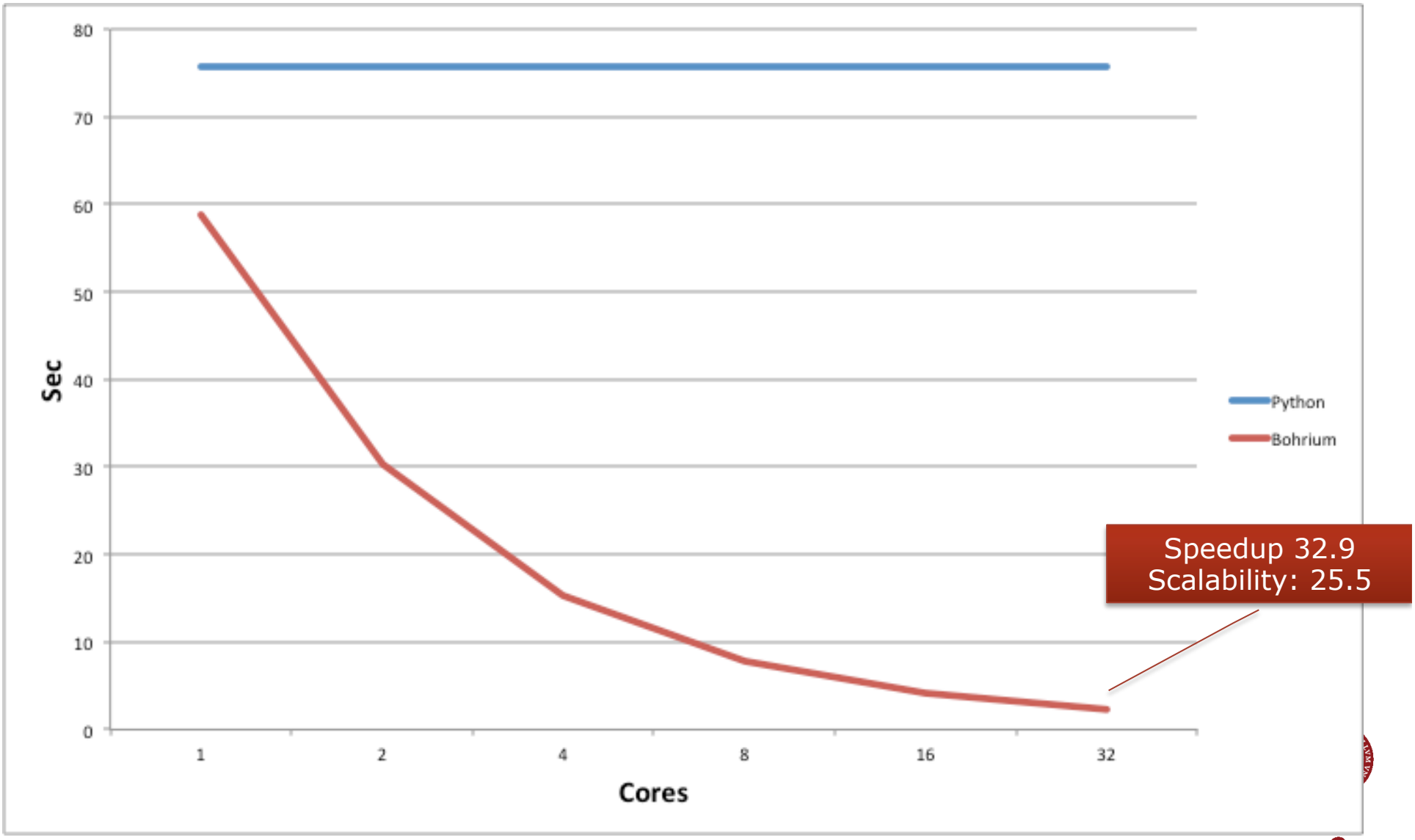


BS.py

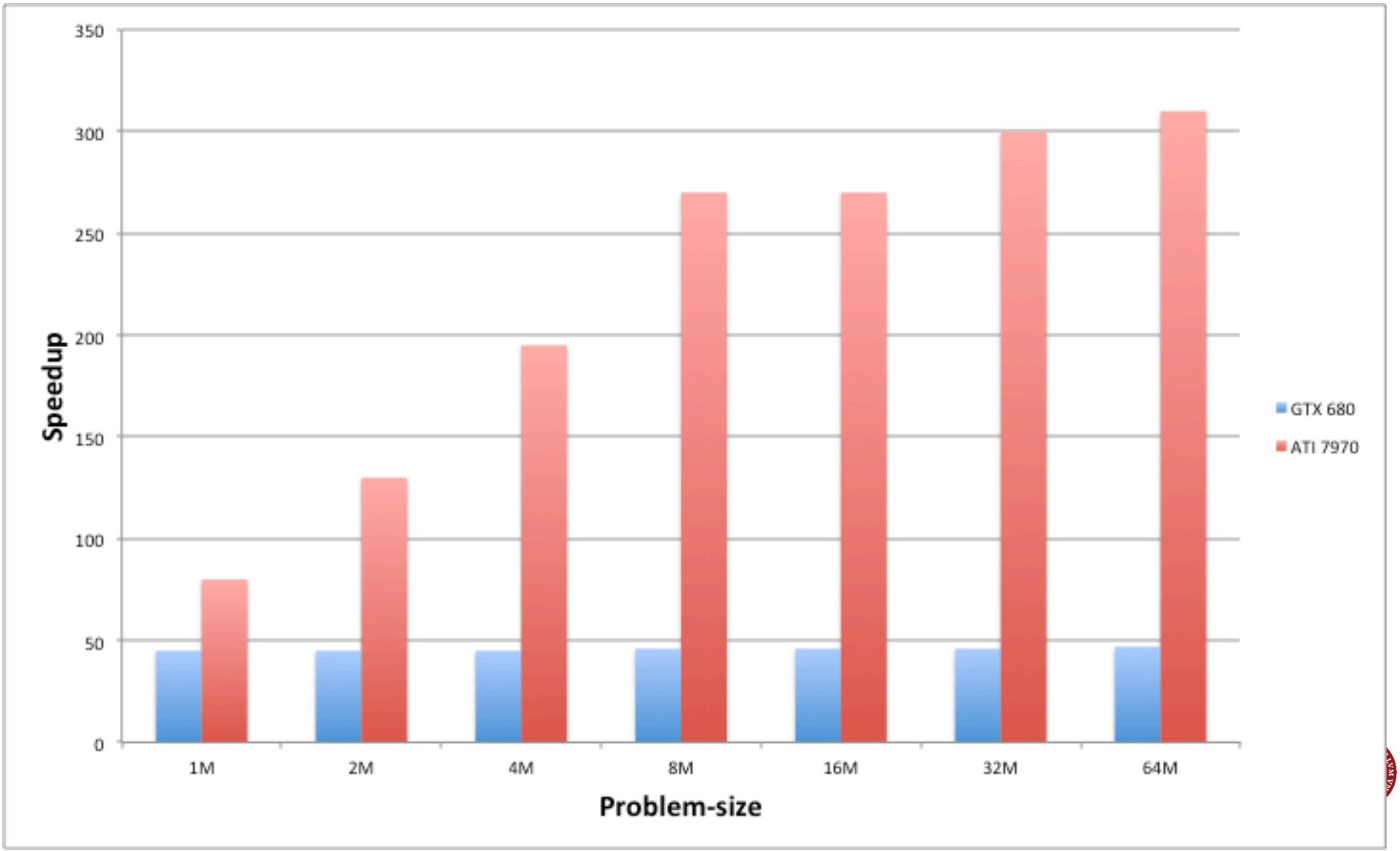
```python
def CND(X):
    """
    Cumulative normal distribution.
    Helper function used by BS(...).
    """

    (a1,a2,a3,a4,a5) = (0.31938153, -0.356563782, 1.781477937, -1.821255978, 1.330274429)
    L = np.absolute(X)
    K = 1.0 / (1.0 + 0.2316419 * L)
    w = 1.0 - 1.0 / np.sqrt(2*np.pi)*np.exp(-L*L/2.) * (a1 * K + a2 * (K**2) + a3 * (K**3) + a4 * (K**4) +
    mask = X<0
    w = w * ~mask + (1.0-w)*mask

    return w

def BS(CallPutFlag, S, X, T, r, v):
    """Black Scholes Function."""

    d1 = (np.log(S/X)+(r+v*v/2.)*T)/(v*np.sqrt(T))
    d2 = d1-v*np.sqrt(T)
    if CallPutFlag=='c':
        return S*CND(d1)-X*np.exp(-r*T)*CND(d2)
    else:
        return X*np.exp(-r*T)*CND(-d2)-S*CND(-d1)
```

Line 17, Column 1 Spaces: 4 Python

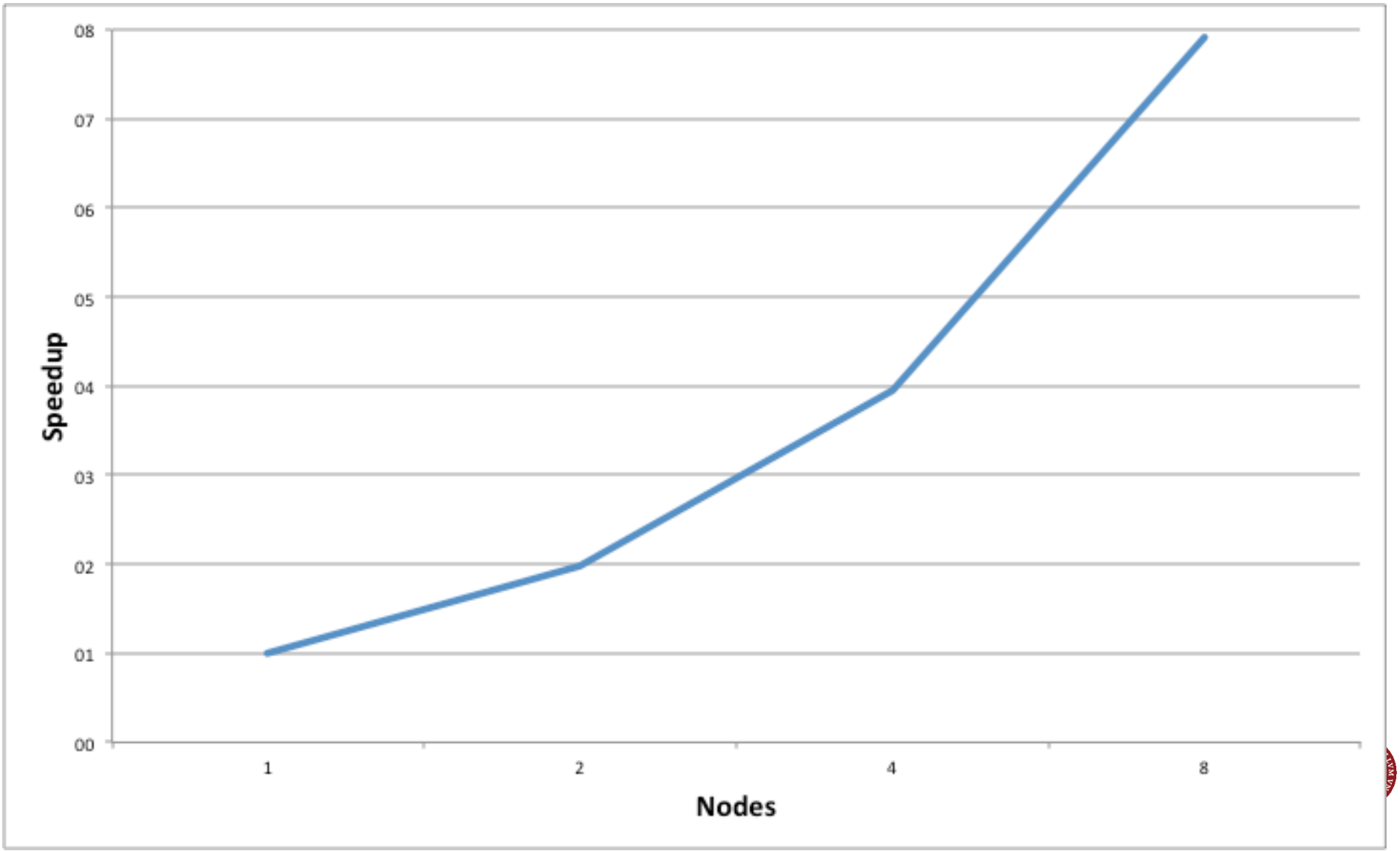
Multicore


80
70
60
50
40
30
20
10
0
Sec
1
2
4
8
16
32
Cores
Python
Bohrium
Speedup 32.9
Scalability: 25.5

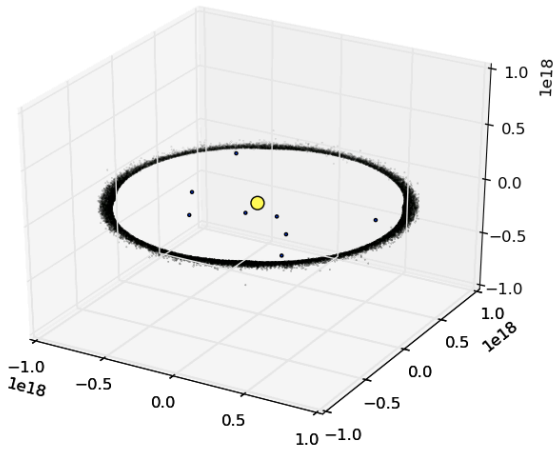
GPU


350
300
250
200
150
100
50
0
Speedup
1M
2M
4M
8M
16M
32M
64M
Problem-size
GTX 680
ATI 7970

Cluster



NICE

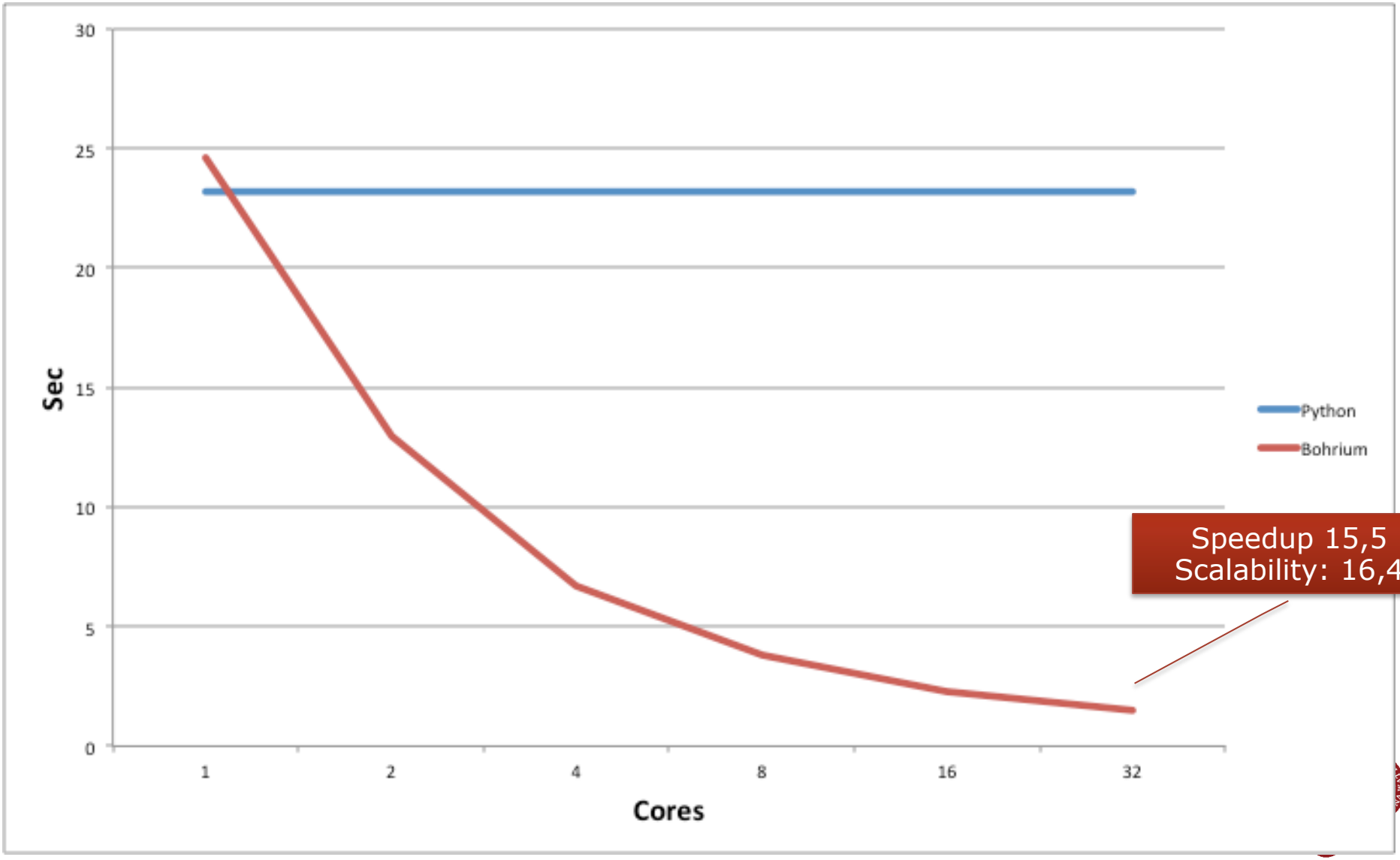


untitled

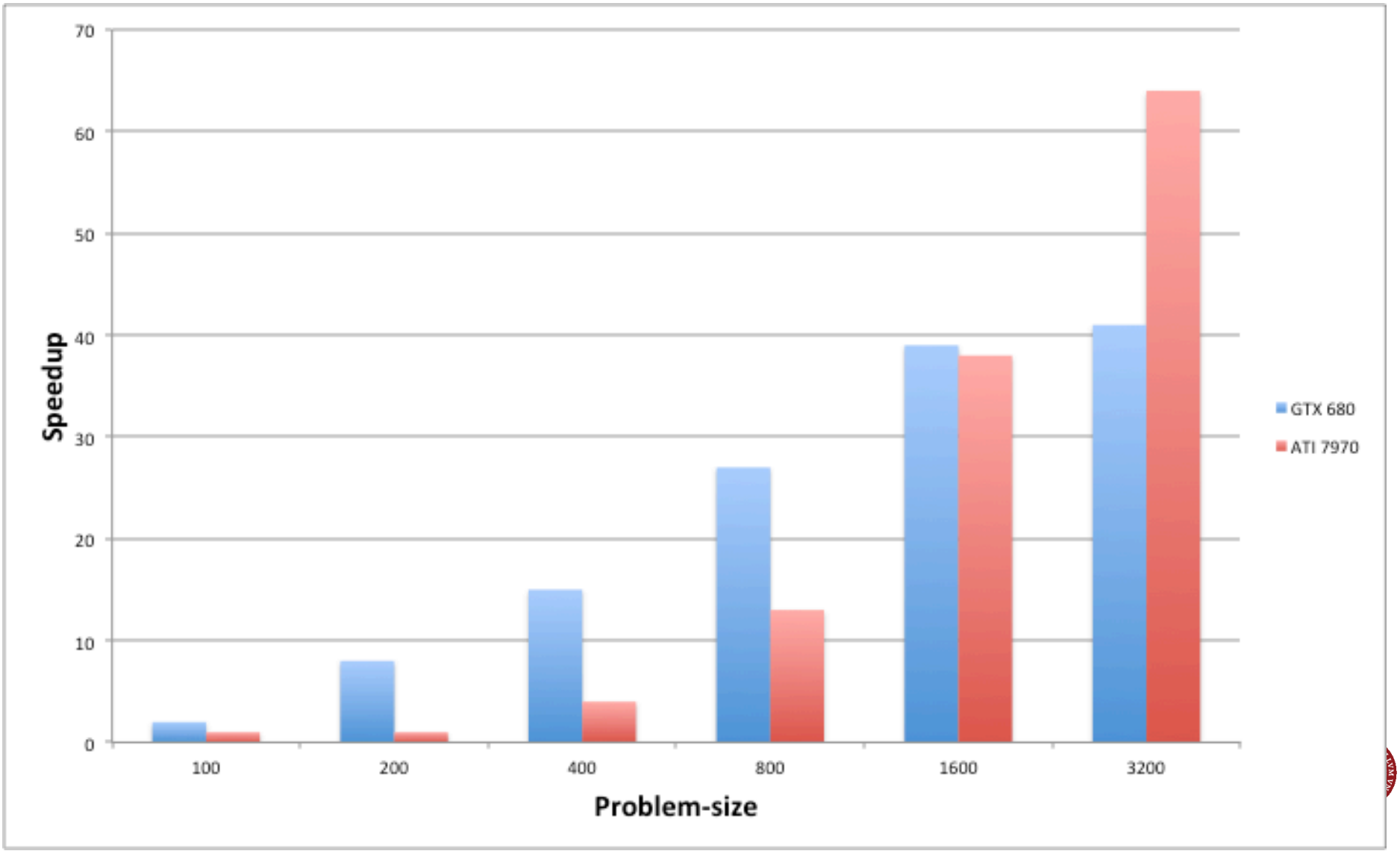
```python
def calc_force(a, b, dt):
    """
    Calculate forces between bodies
    F = ((G m_a m_b)/r^2)/((x_b-x_a)/r)
    """

    G = 6.673e-11

    dx = b['x'] - a['x'][np.newaxis,:].T
    dy = b['y'] - a['y'][np.newaxis,:].T
    dz = b['z'] - a['z'][np.newaxis,:].T
    pm = b['m'] * a['m'][np.newaxis,:].T

    if a is b:
        fill_diagonal(dx,1.0)
        fill_diagonal(dy,1.0)
        fill_diagonal(dz,1.0)
        fill_diagonal(pm,0.0)

    r = ( dx ** 2 + dy ** 2 + dz ** 2) ** 0.5

    # In the below calc of the the forces the force of a body upon itself
    # becomes nan and thus destroys the data

    Fx = G * pm / r ** 2 * (dx / r)
    Fy = G * pm / r ** 2 * (dy / r)
    Fz = G * pm / r ** 2 * (dz / r)

    # The diagonal nan numbers must be removed so that the force from a body
    # upon itself is zero
    if a is b:
        fill_diagonal(Fx,0)
        fill_diagonal(Fy,0)
        fill_diagonal(Fz,0)

    a['vx'] += np.add.reduce(Fx, axis=1)/ a['m'] * dt
    a['vy'] += np.add.reduce(Fy, axis=1)/ a['m'] * dt
    a['vz'] += np.add.reduce(Fz, axis=1)/ a['m'] * dt

def move(solarsystem, asteroids, dt):
    """
    Move the bodies
    first find forces and change velocity and then move positions
    """
    calc_force(solarsystem, solarsystem, dt)
    calc_force(asteroids, solarsystem, dt)

    solarsystem['x'] += solarsystem['vx'] * dt
    solarsystem['y'] += solarsystem['vy'] * dt
    solarsystem['z'] += solarsystem['vz'] * dt

    asteroids['x'] += asteroids['vx'] * dt
    asteroids['y'] += asteroids['vy'] * dt
    asteroids['z'] += asteroids['vz'] * dt

```

Line 178, Column 11 Spaces: 4 Python

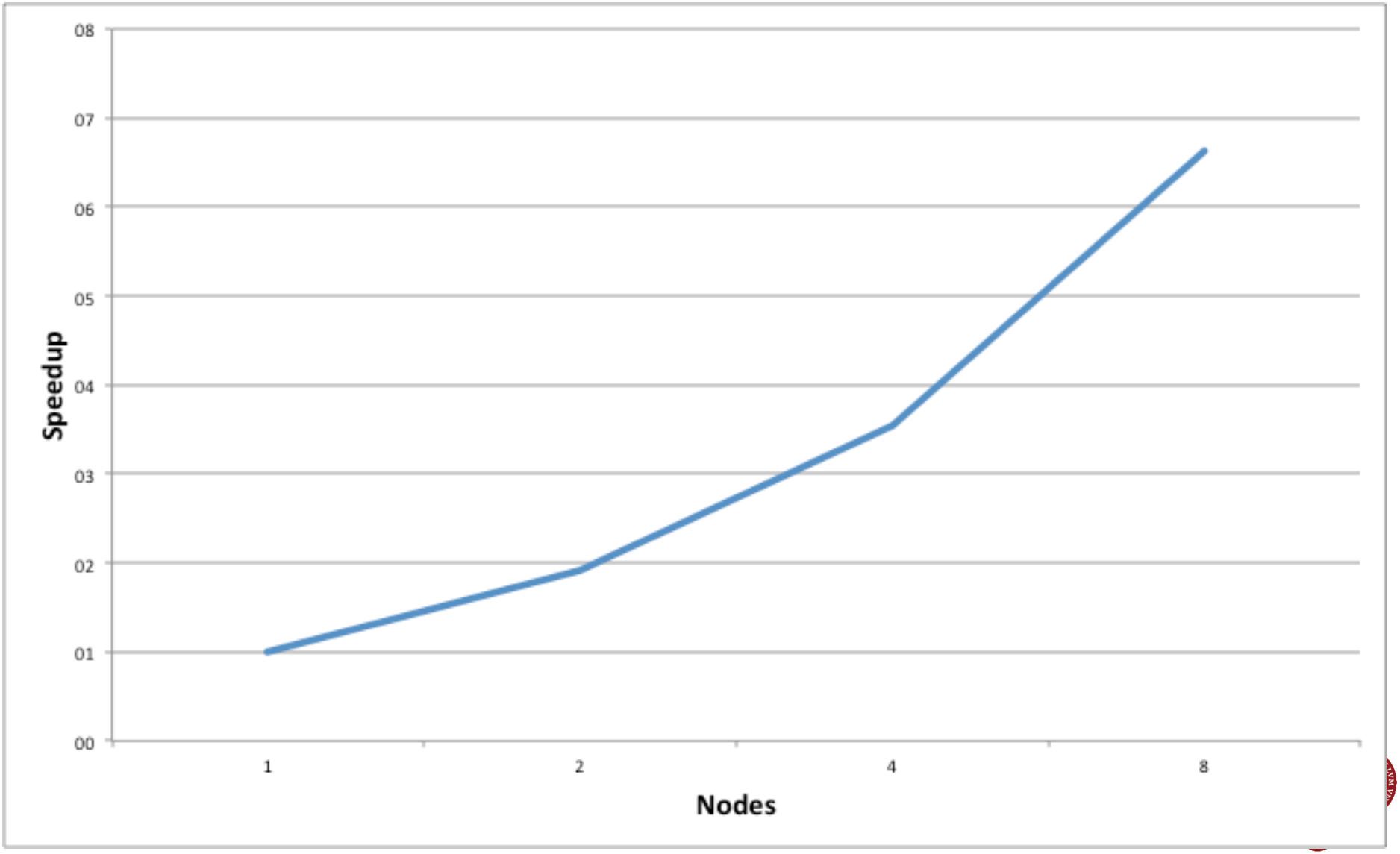
Multicore


30
25
20
15
10
5
0
Sec
1
2
4
8
16
32
Cores
Python
Bohrium
Speedup 15,5
Scalability: 16,4

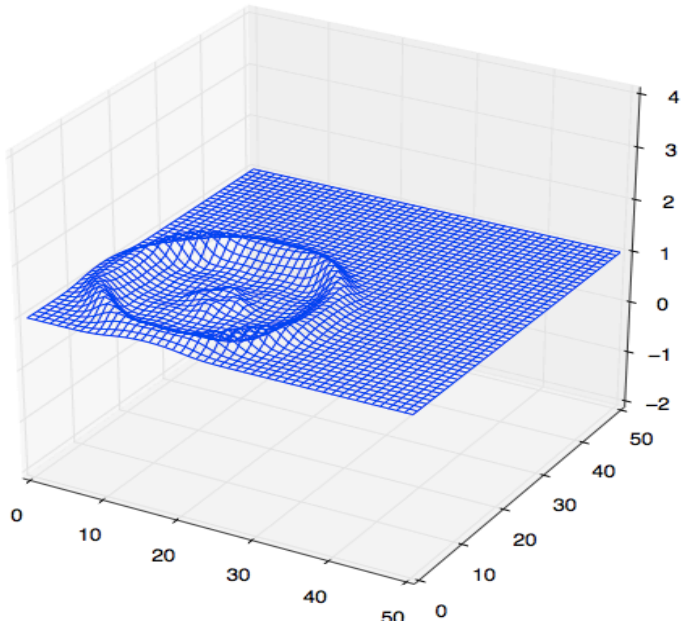
GPU


70
60
50
40
30
20
10
0
Speedup
100
200
400
800
1600
3200
Problem-size
GTX 680
ATI 7970

Cluster



Shallow Water

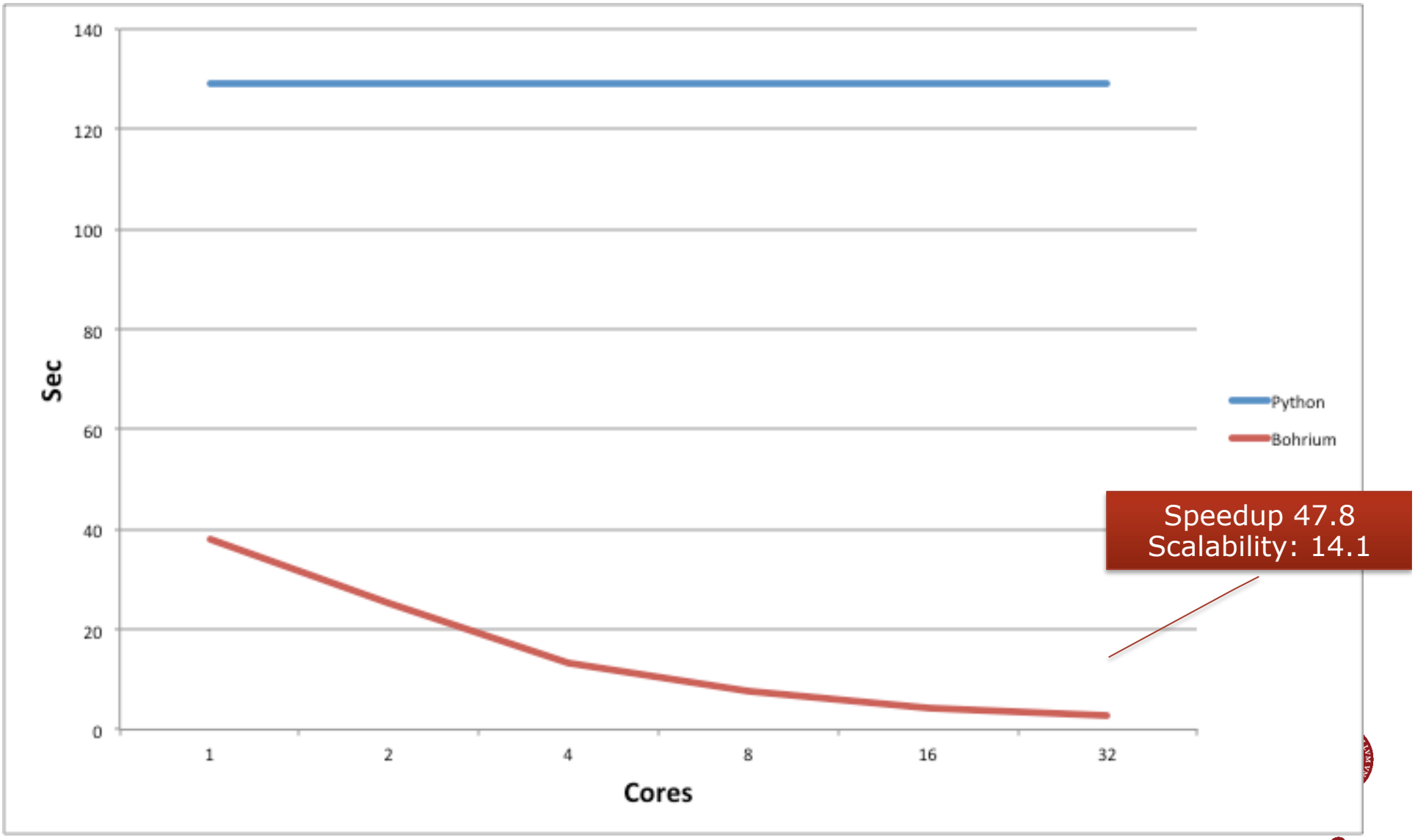


swater.py

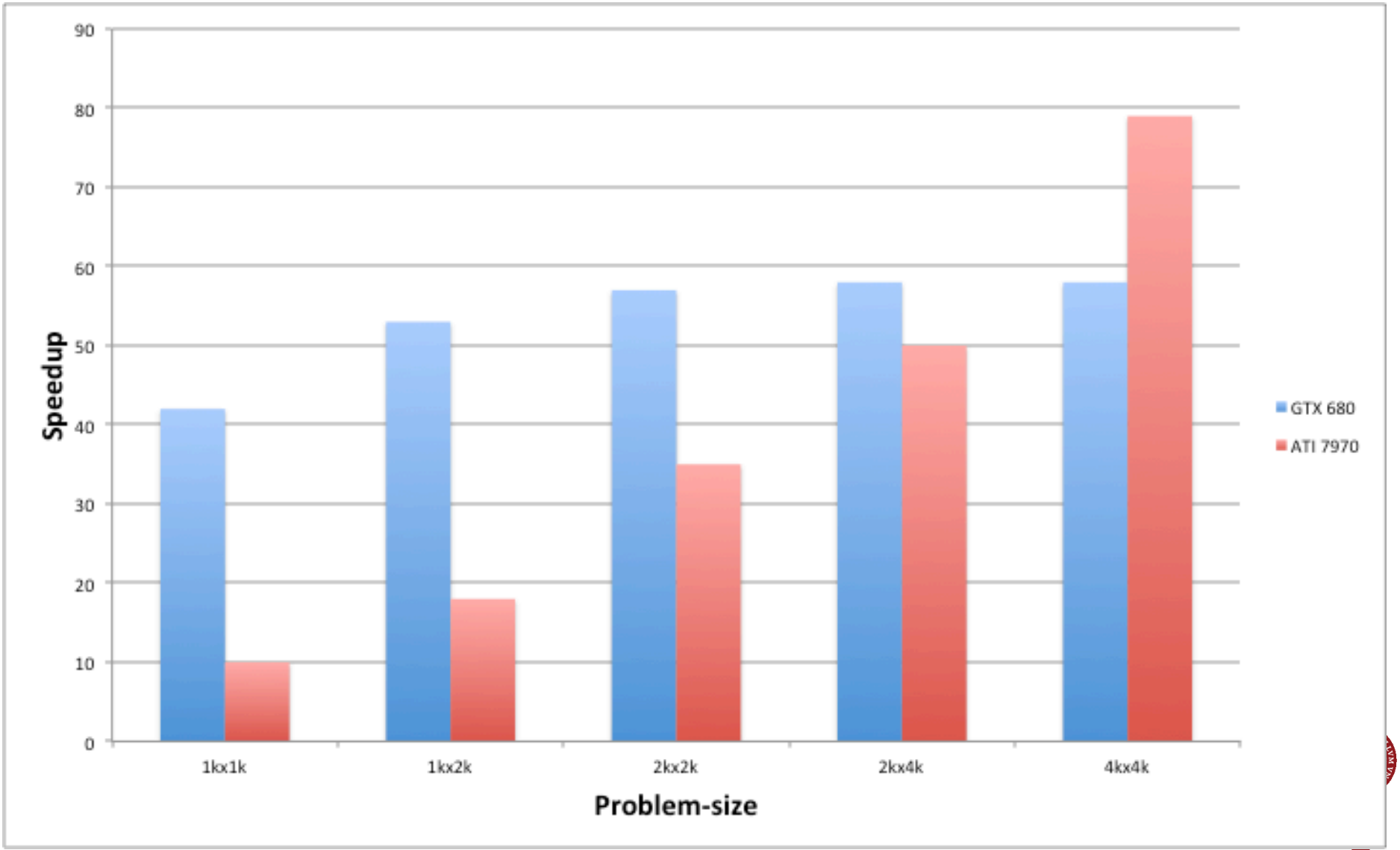
```
43  def step(H, U, V, dt=0.02, dx=1.0, dy=1.0):
44      # Reflecting boundary conditions
45      H[:,0] = H[:,1]   ; U[:,0] = U[:,1]     ; V[:,0] = -V[:,1]
46      H[:,-1] = H[:,-2] ; U[:,-1] = U[:,-2]   ; V[:,-1] = -V[:,-2]
47      H[0,:] = H[1,:]   ; U[0,:] = -U[1,:]    ; V[0,:] = V[1,:]
48      H[-1,:] = H[-2,:] ; U[-1,:] = -U[-2,:]  ; V[-1,:] = V[-2,:]
49
50      #First half step
51
52      # height
53      Hx = (H[1:,1:-1]+H[:-1,1:-1])/2 - dt/(2*dx)*(U[1:,1:-1]-U[:-1,1:-1])
54
55      # x momentum
56      Ux = (U[1:,1:-1]+U[:-1,1:-1])/2 - \
57           dt/(2*dx) * ((U[1:,1:-1]**2/H[1:,1:-1] + g/2*H[1:,1:-1]**2) -
58                       (U[:-1,1:-1]**2/H[:-1,1:-1] + g/2*H[:-1,1:-1]**2))
59
60      # y momentum
61      Vx = (V[1:,1:-1]+V[:-1,1:-1])/2 - \
62           dt/(2*dx) * ((U[1:,1:-1]*V[1:,1:-1]/H[1:,1:-1]) -
63                       (U[:-1,1:-1]*V[:-1,1:-1]/H[:-1,1:-1]))
64
65      # height
66      Hy = (H[1:-1,1:]+H[1:-1,:-1])/2 - dt/(2*dy)*(V[1:-1,1:]-V[1:-1,:-1])
67
68      #x momentum
69      Uy = (U[1:-1,1:]+U[1:-1,:-1])/2 - \
70           dt/(2*dy)*((V[1:-1,1:]*U[1:-1,1:]/H[1:-1,1:]) -
71                      (V[1:-1,:-1]*U[1:-1,:-1]/H[1:-1,:-1]))
72      #y momentum
73      Vy = (V[1:-1,1:]+V[1:-1,:-1])/2 - \
74           dt/(2*dy)*((V[1:-1,1:]**2/H[1:-1,1:] + g/2*H[1:-1,1:]**2) -
75                      (V[1:-1,:-1]**2/H[1:-1,:-1] + g/2*H[1:-1,:-1]**2))
76
77      #Second half step
78
79      # height
80      H[1:-1,1:-1] -= (dt/dx)*(Ux[1:,:]-Ux[:-1,:]) + (dt/dy)*(Vy[:,1:]-Vy[:,:-1])
81
82      # x momentum
83      U[1:-1,1:-1] -= (dt/dx)*((Ux[1:,:]**2/Hx[1:,:] + g/2*Hx[1:,:]**2) -
84                               (Ux[:-1,:]**2/Hx[:-1,:] + g/2*Hx[:-1,:]**2)) + \
85                      (dt/dy)*((Vy[:,1:] * Uy[:,1:]/Hy[:,1:]) -
86                               (Vy[:,:-1] * Uy[:,:-1]/Hy[:,:-1]))
87      # y momentum
88      V[1:-1,1:-1] -= (dt/dx)*((Ux[1:,:] * Vx[1:,:]/Hx[1:,:]) -
89                               (Ux[:-1,:]*Vx[:-1,:]/Hx[:-1,:])) + \
90                      (dt/dy)*((Vy[:,1:]**2/Hy[:,1:] + g/2*Hy[:,1:]**2) -
91                               (Vy[:,:-1]**2/Hy[:,:-1] + g/2*Hy[:,:-1]**2))
92      return (H, U, V)
```

Line 43, Column 1 Spaces: 4 Python

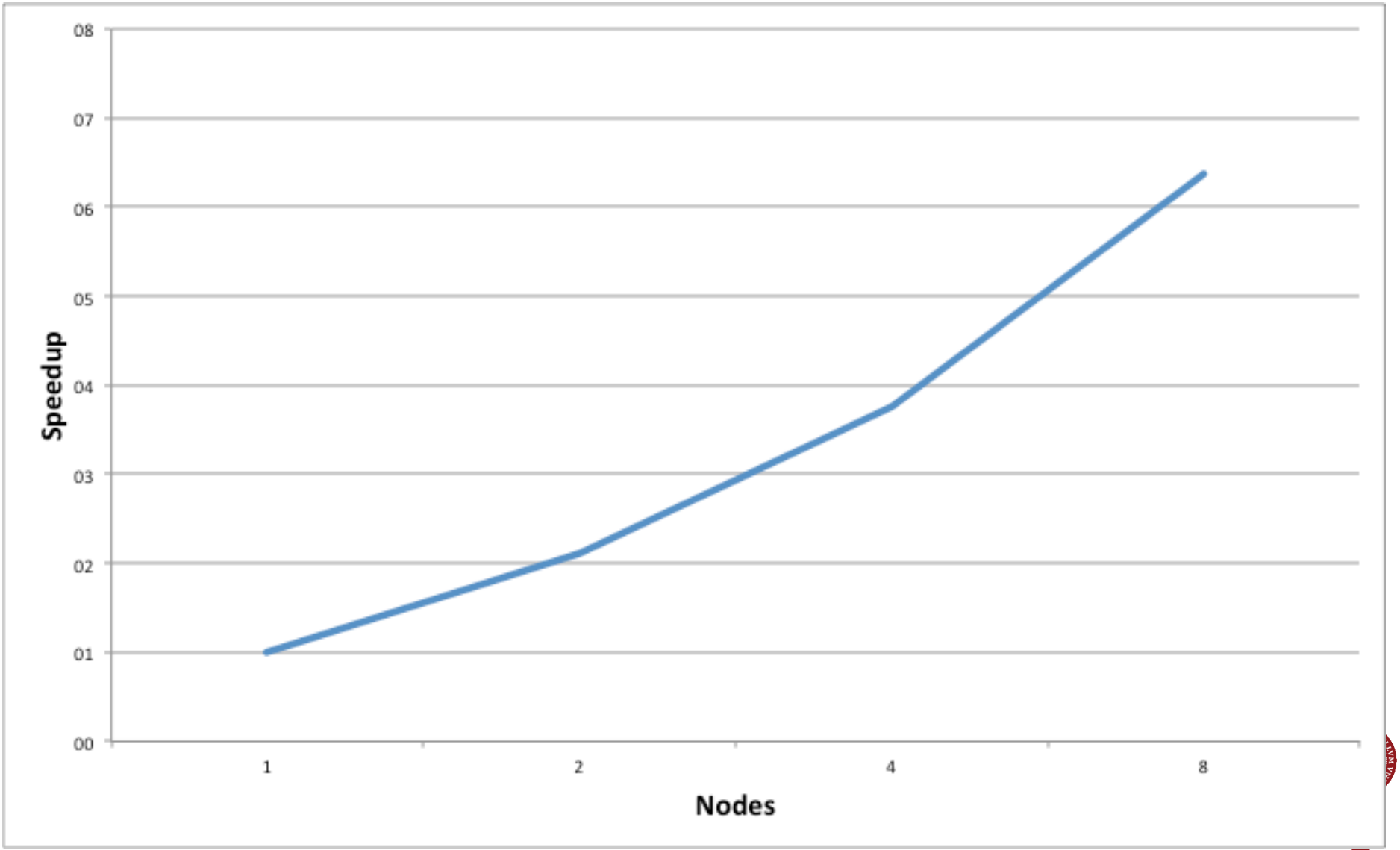
Multicore


140
120
100
80
60
40
20
0
Sec
1
2
4
8
16
32
Cores
Python
Bohrium
Speedup 47.8
Scalability: 14.1

GPU


90
80
70
60
50
40
30
20
10
0
Speedup
1kx1k
1kx2k
2kx2k
2kx4k
4kx4k
Problem-size
GTX 680
ATI 7970

Cluster



Conclusions

We believe that we need a new tool for prototyping in the Exascale age

Bohrium is one possible solution to that challenge

Using highly descriptive code we can extract parallelism and memory access patterns from the user code

Performance is actually very close to that of naïve C++

www.bh107.org