Numerical Python for Scalable Architectures

Mads Ruben Burgdorff Kristensen
Brian Vinter

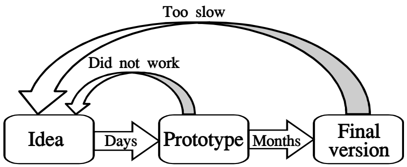
eScience Centre
University of Copenhagen
Denmark
madsbk@diku.dk/vinter@diku.dk

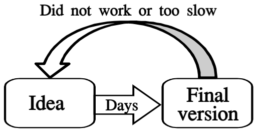
October 14, 2010

Outline

- Motivation
- Python
- Numerical Python (NumPy)
- DistNumPy
- Benchmarks

Motivation – Workflow





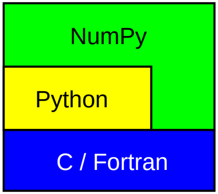
- High Productivity
  - High-level language
  - No compilation
  - Interactive
- High Performance
  - Low-level language such as C, Fortran, etc.
  - Compiling to machine code
  - Parallel Programming

Python

- General-purpose high-level programming language
- The design philosophy emphasizes code readability.
- Sacrificing performance over productivity
- Gluing libraries together

Numerical Python (NumPy)

- Framework for numerical computation similar to Matlab
- Introduces efficient arrays and a lot of useful array operations
  - Linear algebra functions
  - Fourier transforms
  - Random number arrays



Universal Functions

Vectorized operations called Universal Functions

```
#C
for(i = 0; i < rows; i++)
    c[i] = a[i] + b[i];

#NumPy
c = a + b
```

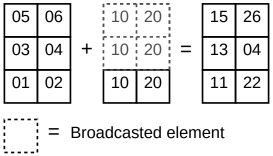
```
#Python
c = []
  for i in range(len(a)):
    c.append(a[i] + b[i])
```

Broadcast

Matching arrays that have different dimensions

```
#No broadcast
for i in range(len(A)):
    c[i] = a[i] + b

#Broadcast
c = a + b
```



DistNumPy

A distributed version of NumPy

Ideal workflow – High Productivity and High Performance

Transparent parallelism

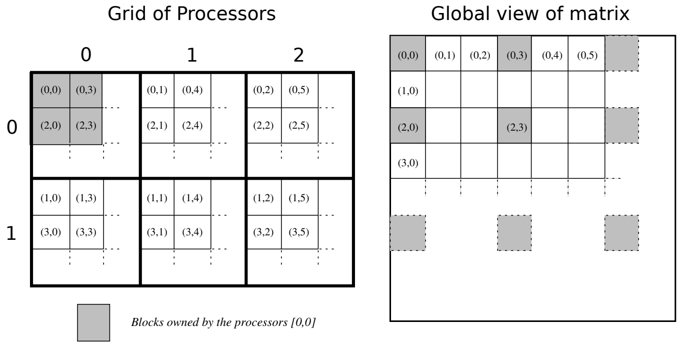
- DistNumPy introduces a distributed parallel array-backend
- Still sequential programming
- Parallel execution of universal functions

Monte Carlo π simulation

```
from numpy import *
S = 1000 #Number of samples
(x, y) = (empty([S], dist=True), empty([S], dist=True))
(x, y) = (random(x), random(y))
(x, y) = (square(x), square(y))
z = (x + y) < 1
print add.reduce(z) * 4.0 / S #The result
```

Data layout

N-Dimensional Block Cyclic Distribution



- Used in High Performance Fortran
- Diagonal workflow e.g. Gaussian elimination

Communication

- DistNumPy make use of MPI version 2.1
- One-sided, two-sided and collective communication
- Latency hiding by double buffering

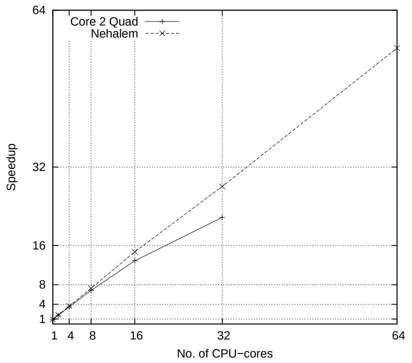
Benchmarks – Hardware

Table: Hardware specifications

CPU	Core 2 Quad	Nehalem
CPU Frequency	2.26 GHz	2.66 GHz
CPU per node	1	2
Cores per CPU	4	4
Memory per node	8 GB @ 6.5 GB/s	24 GB @ 25.6 GB/s
Number of nodes	8	8
Network	Gigabit Ethernet	Gigabit Ethernet

Scalability

Monte Carlo π simulation



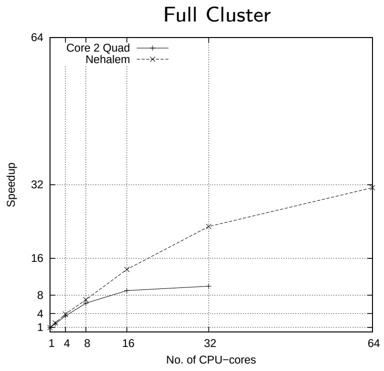
Nehalem – CPU utilization of 88% on 64 CPU-cores

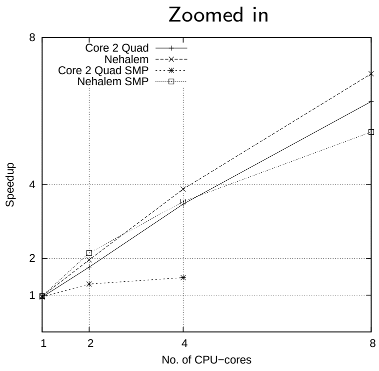
Jacobi solver

Jacobi.py

```
h = zeros(shape(B), float, dist=True)
dmax = 1.0
AD = A.diagonal()
while(dmax > tol):
    hnew = h + (B - add.reduce(A * h, 1)) / AD
    tmp = absolute((h - hnew) / h)
    dmax = maximum.reduce(tmp)
    h = hnew
print h #The result
```

Scalability – Jacobi Solver



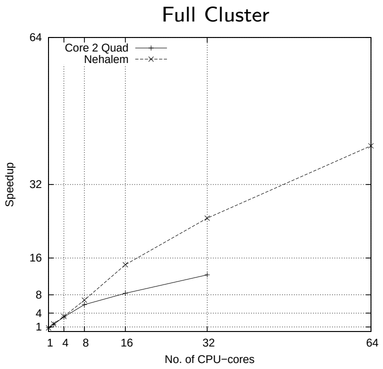


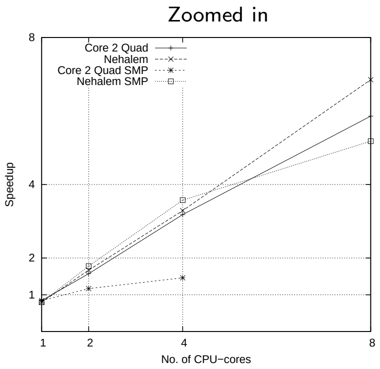
Nehalem – CPU utilization of 85% on 16 CPU-cores and 50% on 64 CPU-cores

N-body simulation

```
for i in range(k):
    Fx = dot(OnesCol, PxT) - dot(Px, OnesRow)
    Dsq = Fx * Fx + Fy * Fy + Fx * Fz + Identity
    D = sqrt(Dsq)
    #mutual forces between all pairs of objects
    F = G * dot(M, MT) / Dsq
    F = F - diag(diag(F))#set 'self attraction' to 0
    Fx = (Fx / D) * F
    #net force on each body
    Fnet_x = add.reduce(Fx,1)
    Fnet_x = Fnet_x[:,newaxis]
    Fnet_x *= dT
    #change in velocity:
    Vx += Fnet_x / M
    #change in position
    Px += Vx * dT
```

Scalability – N-body simulation





Nehalem – CPU utilization of 91% on 16 CPU-cores and 63% on 64 CPU-cores

Summary

- Fully transparent data distribution
- Fully transparent parallel execution
- However, the use of Universal Functions is required
- DistNumPy running Jacobi is roughly 50% slower than the C
    - 21 seconds for C
    - 31 seconds for NumPy
    - 32 seconds for DistNumPy (17 seconds on two CPU-cores)