

# **High-Performance and Distributed Computing**

# Why?

# How?

- In approximate decreasing order of reward/effort:
  - Algorithms
  - Implementation
  - Parallel computing
    - Shared memory
    - Distributed memory

# Algorithms

- Sorting ( $n \cdot \log(n)$  vs.  $n^2$ )
- Searching ( $\log(n)$  vs.  $n$ )
- Shortest paths
- Don't reinvent the wheel

Won't discuss more today.

# Implementation

- Make the code efficient before considering parallel computing
- 10x to 100x speedups not uncommon
- We'll see some general guidelines

# Why is code slow?

- Computing more than is necessary
- Accessing memory more than is necessary
- Compiled to inefficient low-level instructions

# Example

$$x_1 = A \backslash b_1 \quad \# \text{ solve } Ax_1 = b_1$$

$$x_2 = A \backslash b_2 \quad \# \text{ solve } Ax_2 = b_2$$

# How are linear systems solved?

- **Idea**: triangular systems are easy to solve
- Compute  $A = LU$ , where  $L$  is lower triangular and  $U$  is upper triangular (Gaussian elimination)
- Solve  $x = U^{-1}L^{-1}b$



# Example

$x_1 = A \backslash b_1$  # solve  $Ax_1 = b_1$

$x_2 = A \backslash b_2$  # solve  $Ax_2 = b_2$

# About twice as fast:

$LU = \text{lufact}(A)$

$x_1 = LU \backslash b_1$

$x_2 = LU \backslash b_2$

# Example

- If matrix is symmetric positive definite, then Cholesky factorization is twice as fast as LU
  - `cholfact` in Julia
- **lufact.jl**

# More

- Symmetric but not positive definite?
  - bkfact (Bunch-Kaufman)
- Sparsity
- **Large speedups possible by using better linear algebra** (4x speedup in our example)

# In optimization

- When solving a sequence of LPs, can “hot-start” the simplex algorithm using previous optimal basic solution
  - Available in JuMP

# Memory access

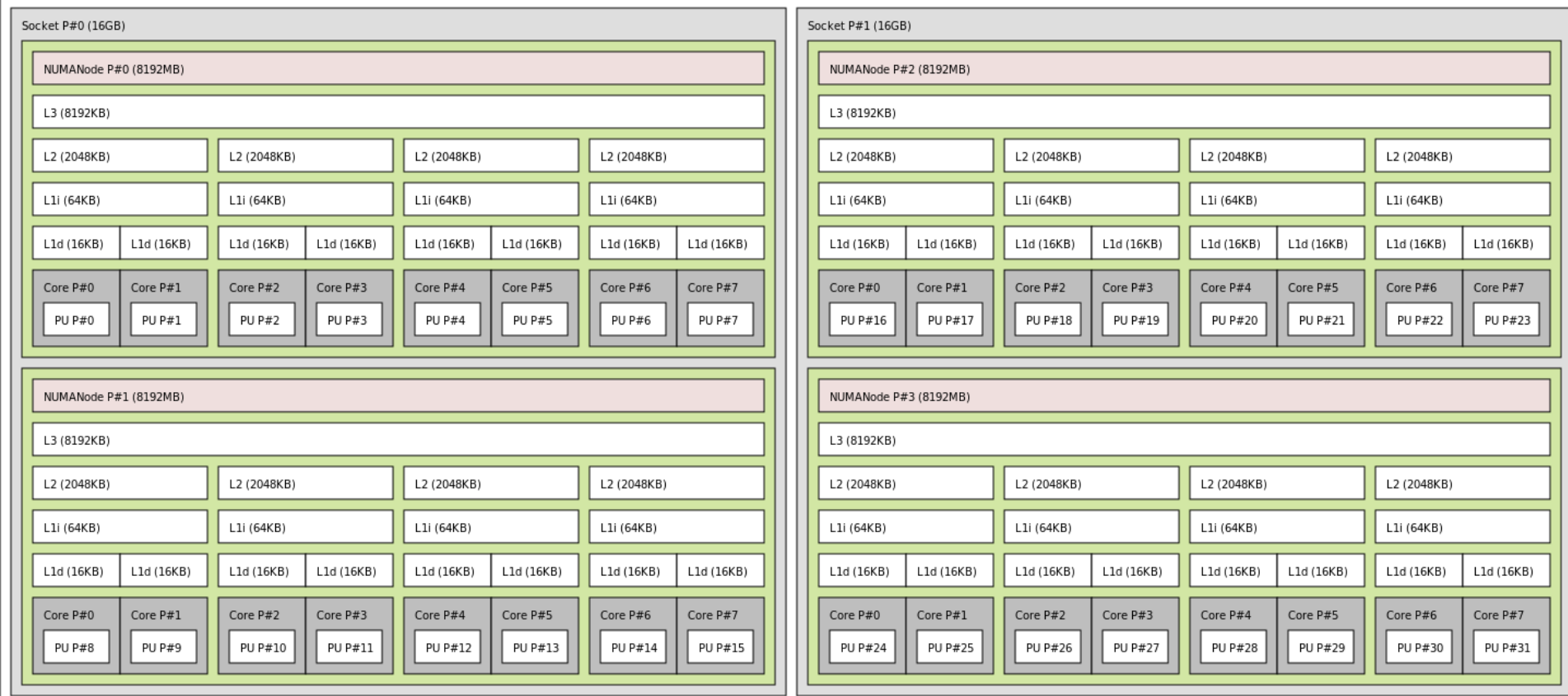
- RAM = Random-access memory
- Random memory access is about 100x slower than a floating-point operation

# Memory access

- **Random memory access is about 100x slower than a floating-point operation**

# Hardware solution

- Processor **predicts** which memory will be accessed soon and fetches it into a **cache** that's faster to access
  - How?



Memory hierarchy of AMD Bulldozer (<http://en.wikipedia.org/wiki/File:Hwloc.png>)



# Example

- Compute properties of a vector
  - L2, L1 norms and min/max element
- Single pass vs. built-in functions
- **locality.jl**

# Example: reusing memory

```
function normalize!(x)
    n = norm(x)
    for i in 1:length(x)
        x[i] /= n
    end
end
```

# Example: reusing memory

```
x = rand(100_000_000); # 800Mb
```

```
@time x/norm(x);
```

“elapsed time: 0.65 seconds”

```
@time normalize!(x); # includes compilation
```

“elapsed time: 0.49 seconds”

```
@time normalize!(x);
```

“elapsed time: 0.30 seconds”

# Efficient low-level operations

- BLAS/LAPACK
  - <http://www.netlib.org/lapack/>
  - Efficient implementation of linear algebra operations
    - 20+ years of tuning
    - Commercial implementations
  - Used internally by MATLAB, Scipy, R, Julia
  - Good to use when possible!

# Example: **axpy**

- BLAS function axpy:  $y \leftarrow \alpha x + y$
- **axpy.jl**

# Gradient descent example

- Let's make this faster
- $\mathbf{x} = \mathbf{x} - (\text{stepsize}/2^k) * \text{grad}$ 
  - Creates a new vector at each iteration
  - Use AXPY instead
  - Also "...fval - f( $\mathbf{x} - (\text{stepsize}/2^k) * \text{grad}$ ) < ..."
- **gradient2.jl**

$f(x) = (1/2) \cdot \text{dot}(x, Q \cdot x)$ ;  $\text{fgrad}(x) = Q \cdot x$

$Q \cdot x$  is computed twice. Restructure the code to use a single function that returns the value and gradient:

```
function genF(Q)
    function f_and_fgrad(x)
        grad = Q*x
        return ((1/2)*dot(x,grad), grad)
    end
end
```

- **gradient3.jl**

We're still allocating a new vector for the gradient for each call to `f`. Instead use:

```
function genF(Q)
    function f_and_fgrad(x, grad_out)
        A_mul_B!(grad_out, Q, x)
        return (1/2)*dot(x, grad_out)
    end
end
```

- **gradient4.jl**



# One more thing

$Q$  is symmetric. Let's use a specialized function for symmetric matrix-vector products:

```
function f_and_fgrad(x,grad_out)
    # grad_out <- 1.0*Q*x + 0.0*grad_out
    BLAS.symv!('u',1.0,Q,x,0.0,grad_out)
    return (1/2)*dot(x,grad_out)
end
```

- **gradient5.jl**

# Thoughts

- Reusing calculations, reusing memory, advanced linear algebra
- Applies to all languages
  - In Julia, don't need to code critical parts in C

# Thoughts

- “Premature optimization is the root of all evil”  
– Knuth
- Identify and fix bottlenecks **after** running
- Balance between readability and performance

# Break

# Parallel Computing

# Amdahl's Law

- What can we expect?
- If 75% of an algorithm (by time) can be parallelized, what's the best speedup we can obtain?

# Amdahl's Law

More generally, if proportion  $x$  of algorithm can be parallelized, then with  $n$  parallel threads, the best execution time we can expect is:

$$(1-x) + x/n$$

# **What is easy to parallelize?**



# Common, less trivial, uses:

- Linear algebra
- Physical simulations (chemistry, fluid dynamics, particles)

# Architectures

- Shared vs. distributed memory

# Shared memory

- All cores have access to the same RAM
- Example: your laptop
- Up to 100s of cores

# Shared memory

What's difficult?

- Multiple cores can write to the same block of memory at the same time
- Memory bandwidth is shared

# Distributed memory

- “Network of computers”
- Communication? **Interconnect**
  - Bandwidth
  - Latency
- “distributed” vs. “parallel” computing

# “Distributed” computing

- Slow interconnect, e.g. ethernet (ms latency), carrier pigeons
- Examples:
  - The cloud
  - Network of machines at a university
- Uses:
  - Mostly independent calculations with some synchronization

# “Parallel” computing

- Fast interconnect, e.g. InfiniBand ( $\mu$ s latency)
- Examples:
  - “Supercomputers” (<http://www.top500.org/lists/2013/11/>)
  - High-performance clusters
- Uses:
  - Solving 100,000 x 100,000 dense linear system
  - Solving LPs with  $10^9$  variables
  - Numerical PDE simulations

# Programming models

- Message passing (MPI)
  - Explicitly send and receive data
- Master-worker
  - Distribute tasks to workers
- MapReduce (saw this on Tuesday)



# Parallel computing in Optimization

Math Meth Oper Res (2012) 76:67–93  
DOI 10.1007/s00186-012-0390-9

ORIGINAL ARTICLE

**Could we use a million cores to solve an integer program?**

Thorsten Koch · Ted Ralphs · Yuji Shinano

A. Grothey, J. Gondzio

How to solve QPs with  $10^9$  variables



School of Mathematics



**How to solve QPs with  
 $10^9$  variables**

Andreas Grothey, Jacek Gondzio

Numerical Analysis 2005, Dundee

1

# Parallel computing in Optimization

On a smaller scale (shared memory):

- Parallel barrier method
  - Using parallel sparse linear algebra
- No general parallel simplex method
- Parallel IP
  - Parallel branch & bound
- In JuMP: `Model(solver=GurobiSolver(Threads=10))`

# Parallel linear algebra

- Via BLAS/LAPACK
- **threads.jl**

# In Julia

- Distributed memory (except for BLAS)
  - Multiple processes on a single machine or across machines via ssh
- <http://docs.julialang.org/en/release-0.2/manual/parallel-computing/>
- **parfor.jl**
- **pmap.jl**

# Case study

- Parallel computing for data-driven modeling optimization

# Cutting-plane method

$$\min \sum f_i(x)$$

- $f_i$  convex, non-smooth
- Can obtain subgradients:
  - $f_i(x') \geq f_i(x) + g^T(x'-x)$
- Idea:
  - Given a set of subgradients, we can form a piecewise linear model of each  $f_i$

# Cutting-plane method

$$\min \sum f_i(x)$$

- Idea:
  - Given a set of subgradients, we can form a piecewise linear model  $m_i$  of each  $f_i$
- Algorithm:
  - Solve  $\min \sum m_i(x)$
  - Evaluate each  $f_i(x^*)$ . If model  $m_i(x^*)$  doesn't match  $f_i(x^*)$ , then update model with new subgradient and repeat

# Cutting-plane method

- Algorithm:
  - Solve  $\min \sum m_i(x)$
  - **Evaluate each**  $f_i(x^*)$ . If model  $m_i(x^*)$  doesn't match  $f_i(x^*)$ , then update model with new subgradient and repeat.
- Evaluation is trivially parallel.
- Classical parallel decomposition example in optimization



# The problem

Decide the optimal capacity of each Hubway station using historical data

# Our formulation

- $\mathbf{x}$  - vector of station capacities
- $f(\mathbf{x})$  - cost of reshuffling bikes between stations (once per hour) so that all demand is met
  - Assume demands (trips) for each day are **known in advance**, optimal reshuffling can be formulated as a network flow problem
- Objective: design station capacities to minimize reshuffling costs

# The code

- Proof of concept, not polished
- Let's run it on 50 cores...
- **master.jl** and **subproblem.jl**

# Extensions

- Visualize solution
- Refine the model
- Time the master and subproblems, apply Amdahl's law

# Computing resources

- Amazon EC2
- University clusters

**Thanks!**