

Short course on High Performance Computing

Victor Eijkhout

2018

Justification

High Performance Computing is a field that brings together algorithms, software, and hardware. This course conveys the basics of computer architecture, scientific algorithms, and how to code algorithms to make them efficient on current hardware.

Processor Architecture

Justification

The performance of a parallel code has as one component the behaviour of the single processor or single-threaded code. In this section we discuss the basics of how a processor executes instructions, and how it handles the data these instructions operate on.

Table of Contents

Von Neumann machine

The ideal processor:

- (Stored program)
- An instruction contains the operation and two operand locations
- Processor decodes instruction, gets operands, computes and writes back the result
- Repeat

The actual state of affairs

- Single instruction stream versus multiple cores / floating point units
- Single instruction stream versus Instruction Level Parallelism
- Unit-time-addressable memory versus large latencies

Modern processors contain lots of magic to make them seem like Von Neumann machines.

Complexity measures

Traditional: processor speed was paramount. Operation counting.

Nowadays: memory is slower than processors (peak performance only out of register).

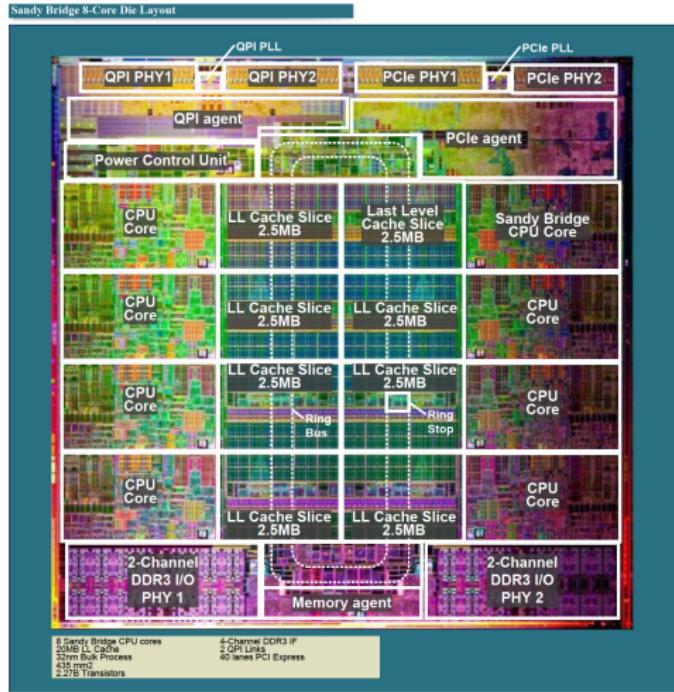
This course

Study data movement aspects

Dealing with latency

Algorithm design for processor reality

A first look at a processor



Copyright (c) 2011 Intel Corporation. All rights reserved.

Structure of a core

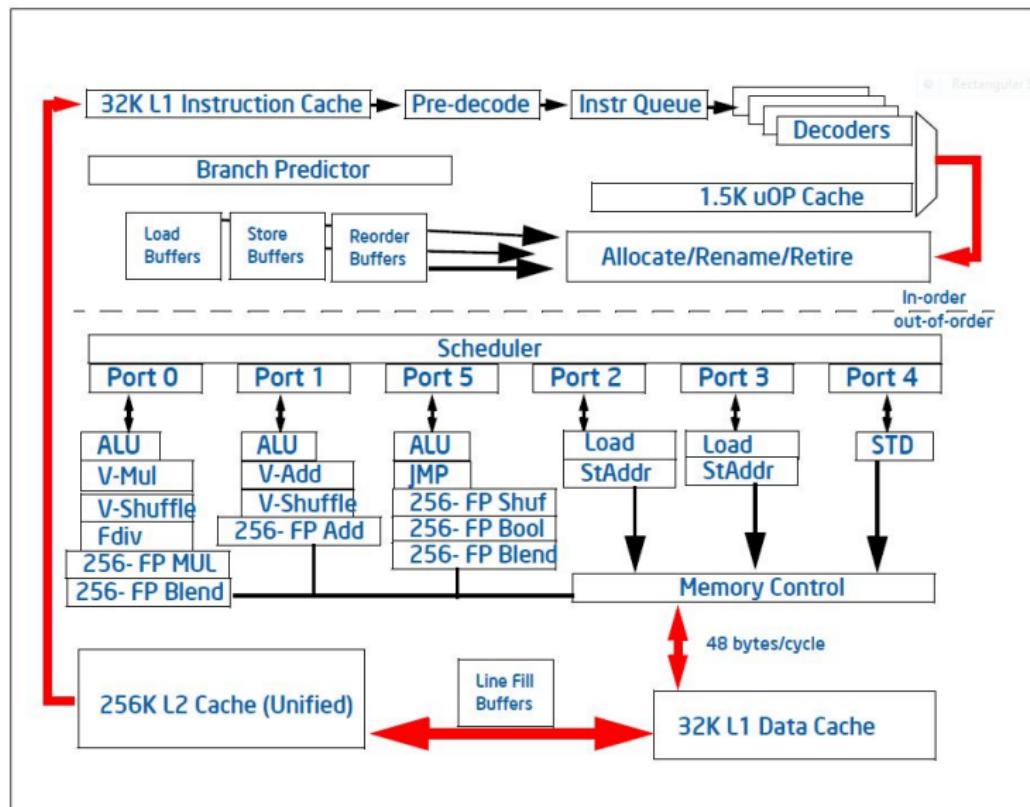
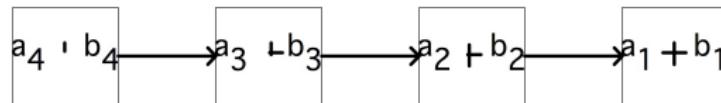
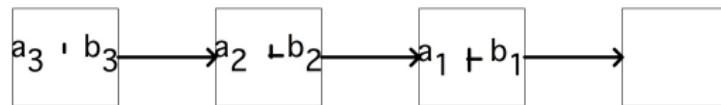
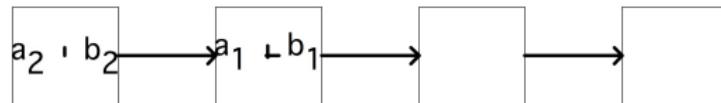
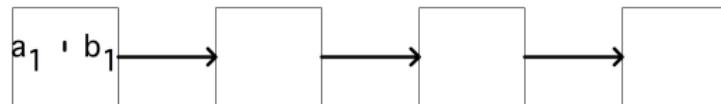


Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

Pipelining, pictorially

$$c_i \leftarrow a_i + b_i$$



1

Pipelining, formally

- Decoding the instruction operands.
- Data fetch into register
- Aligning the exponents:

$$\begin{aligned} .35 \times 10^{-1} + .6 \times 10^{-2} &\quad \text{becomes} \\ .35 \times 10^{-1} + .06 \times 10^{-1}. \end{aligned}$$

- Adding mantissas, giving .41.
- Normalizing the result, giving $.41 \times 10^{-1}$.
- Storing the result.

pipeline stages

Analysis

Operation timing:

$$\begin{cases} n & \text{operations} \\ \ell & \text{number of stages} \Rightarrow t(n) = n\ell\tau \\ \tau & \text{clock cycle} \end{cases}$$

With pipelining:

$$t(n) = [s + \ell + n - 1]\tau$$

where s is a setup cost

Asymptotic speedup is ℓ

$n_{1/2}$: value for which speedup is $\ell/2$

Applicability of pipelining

Pipelining works for:

vector addition

Pipelining does not work for:

recurrences

```
for (i) {  
    x[i+1] = a[i]*x[i] + b[i];  
}
```

Transform:

$$\begin{aligned}x_{n+2} &= a_{n+1}x_{n+1} + b_{n+1} \\&= a_{n+1}(a_nx_n + b_n) + b_{n+1} \\&= a_{n+1}a_nx_n + a_{n+1}b_n + b_{n+1}\end{aligned}$$

Instruction pipeline

- Instruction-Level Parallelism: more general notion of independent instructions
- Requires independent instructions
- As frequency goes up, pipeline gets longer: more demands on compiler

Instruction-Level Parallelism

- multiple-issue of independent instructions
- branch prediction and speculative execution
- out-of-order execution
- prefetching

Problems: complicated circuitry, hard to maintain performance

Implications

- Long pipeline needs many independent instructions:
demands on compiler
- Conditionals break the stream of independent instructions
 - Processor tries to predict branches
 - *branch misprediction penalty*:
pipeline needs to be flushed and refilled
 - avoid conditionals in inner loops!

Peak performance

Performance is a function of

- Clock frequency,
- SIMD width
- Load/store unit behaviour

Behaviour (out of L1):

Processor	year	add/mult/fma units (count \times width)	daxpy cycles (arith vs load/store)
MIPS R10000	1996	$1 \times 1 + 1 \times 1 + 0$	8/24
Alpha EV5	1996	$1 \times 1 + 1 \times 1 + 0$	8/12
IBM Power5	2004	$0 + 0 + 2 \times 1$	4/12
AMD Bulldozer	2011	$2 \times 2 + 2 \times 2 + 0$	2/4
Intel Sandy Bridge	2012	$1 \times 4 + 1 \times 4 + 0$	2/4
Intel Haswell	2014	$0 + 0 + 2 \times 4$	1/2

Floating point capabilities of several processor architectures, and DAXPY cycle number for 8 operands

Dirty secret

Processor design is sometimes optimized for certain algorithms

In particular: DGEMM/Linpack

Favourable property: one load per operation, no stores

Table of Contents

The Big Story

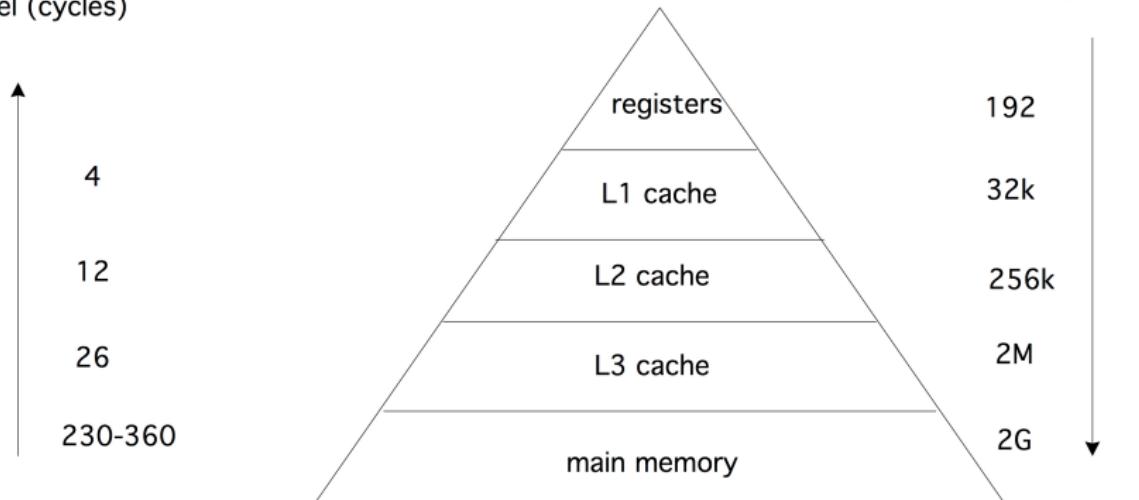
- DRAM memory is slow, so let's put small SRAM close to the processor
- This helps if data is reused
- Does the algorithm have reuse?
- Does the implementation reuse data?

Bandwidth and latency

Important theoretical concept:

- *latency* is delay between request for data and availability
- *bandwidth* is rate at which data arrives thereafter

Latency from next
level (cycles)



Registers

Computing out of registers

a := b + c

- load the value of *b* from memory into a *register*,
- load the value of *c* from memory into another register,
- compute the sum and write that into yet another register, and
- write the sum value back to the memory location of *a*.

Register usage

Assembly code

```
addl %eax, %edx
```

- Registers are named
- Can be explicitly addressed by the programmer
- ... as opposed to caches.
- Assembly coding or inline assembly (compiler dependent)
- ... but typically generated by compiler
- Very high bandwidth / low latency: peak performance only possible with data in register.

Examples of register usage

```
a := b + c  
d := a + e
```

a stays resident in register, avoid store and load

```
t1 = sin(alpha) * x + cos(alpha) * y;  
t2 = -cos(alpha) * x + sin(alpha) * y;
```

subexpression elimination:

```
s = sin(alpha); c = cos(alpha);  
t1 = s * x + c * y;  
t2 = -c * x + s * y
```

often done by compiler

Register variables

Hint to the compiler: declare *register variable*

```
register double t;
```

Declaring too many leads to register spill.

Caches

Cache basics

Fast SRAM in between memory and registers: mostly serves data reuse

```
... = ... x ..... // instruction using x  
.....           // several instructions not involving x  
... = ... x ..... // instruction using x
```

- load x from memory into cache, and from cache into register; operate on it;
- do the intervening instructions;
- request x from memory, but since it is still in the cache, load it from the cache into register; operate on it.
- essential concept: *data reuse*

Caches are associative

Cache levels

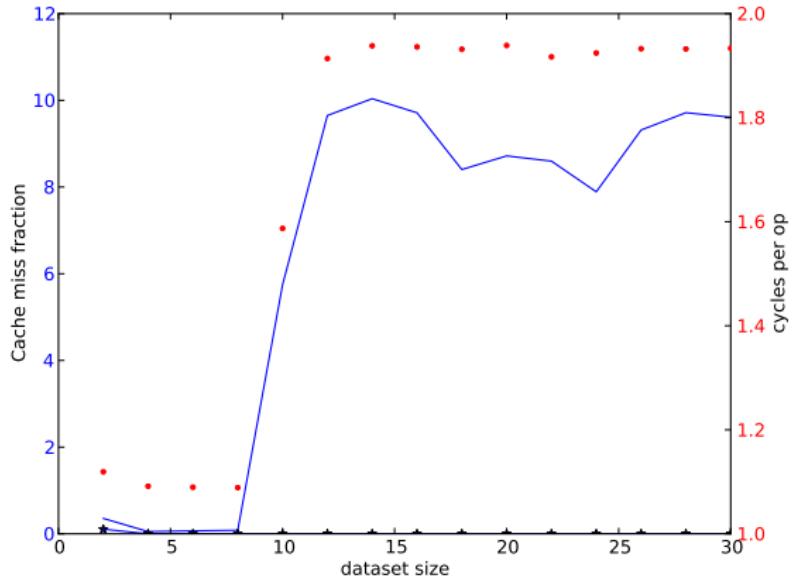
- Levels 1,2,3(,4): L1, L2, etc.
- Increasing size, increasing latency, increasing bandwidth
- (Note: L3/L4 can be fairly big; beware benchmarking)
- Cache hit / cache miss: one level is consulted, then the next
- L1 has separate data / instruction cache, other levels mixed
- Caches do not have enough bandwidth to serve the processor: coding for reuse on all levels.

Cache misses

- Compulsory miss: first time data is referenced
- Capacity miss: data was in cache, but has been flushed (overwritten) by LRU policy
- Conflict miss: two items get mapped to the same cache location, even if there are no capacity problems
- Invalidation miss: data becomes invalid because of activity of another core

Illustration of capacity

```
for (i=0; i<NRUNS; i++)  
    for (j=0; j<size; j++)  
        array[j] = 2.3*array[j]+1.2;
```

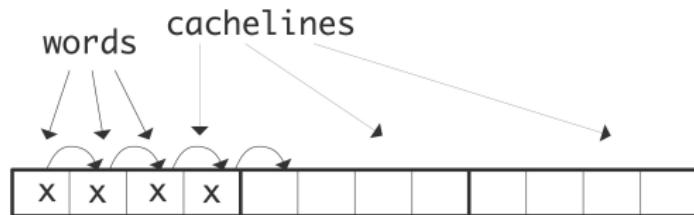


Cache lines

- Memory requests go by byte or word
- Memory transfers go by *cache line*:
4 or 8 words
- Cache line transfer costs bandwidth
- ⇒ important to use all elements

Cache line use

```
for (i=0; i<N; i++)  
    ... = ... x[i] ...
```

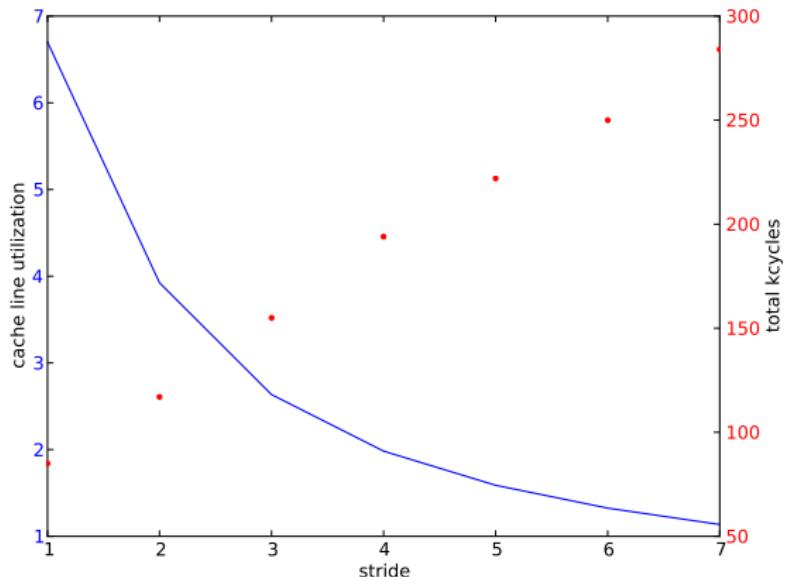


```
for (i=0; i<N; i+=stride)  
    ... = ... x[i] ...
```



Stride effects

```
for (i=0,n=0; i<L1WORDS; i++,n+=stride)  
    array[n] = 2.3*array[n]+1.2;
```



Spatial and temporal locality

Temporal locality: use an item, use it again but from cache
efficient because second transfer cheaper.

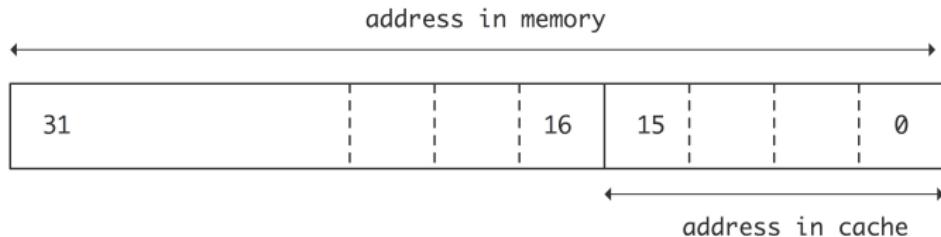
Spatial locality: use an item, then use one 'close to it'
(for instance from same cacheline)
efficient because item is already reachable even though not used before.

Cache mapping

Cache is smaller than memory, so we need a mapping scheme

- Ideal: any address can go anywhere; LRU policy for replacement
- pro: optimal; con: slow, expensive to manufacture
- Simple: direct mapping by truncating addresses
- pro: fast and cheap; con: I'll show you in a minute
- Practical: limited associativity; golden mean

Direct mapping



Direct mapping of 32-bit addresses into a 64K cache

- Use last number of bits to find cache address
- If (memory) addresses are cache size apart, they get mapped to the same cache location
- If you traverse an array, a contiguous chunk will be mapped to cache without conflict.

The problem with direct mapping

```
real*8 A(8192,3);
do i=1,512
  a(i,3) = ( a(i,1)+a(i,2) )/2
end do
```

In each iteration 3 elements map to the same cache location:
constant overwriting ('eviction', *cache thrasing*):
low performance

Associate cache mapping

- Allow each memory address to go to multiple (but not all) cache addresses; typically 2,4,8
- Prevents problems with multiple arrays
- Reasonable fast, still:
- Often lower associativity for L1 than L2, L3

Associativity	L1	L2
Intel (Woodcrest)	8	8
AMD (Bulldozer)	2	8

Illustration of associativity

{0, 12, 24, ... }
{1, 13, 25, ... }
{2, 14, 26, ... }
{3, 15, 27, ... }
{4, 16, 28, ... }
{5, 17, 29, ... }
{6, 18, 30, ... }
{7, 19, 31, ... }
{8, 20, 32, ... }
{9, 21, 33, ... }
{10, 22, 34, ... }
{11, 23, 35, ... }

{0, 12, 24, ... } {4, 16, 28, ... }
{8, 20, 32, ... }
{1, 13, 25, ... } {5, 17, 29, ... }
{9, 21, 33, ... }
{2, 14, 26, ... } {6, 18, 30, ... }
{10, 22, 34, ... }
{3, 15, 27, ... } {7, 19, 31, ... }
{11, 23, 35, ... }

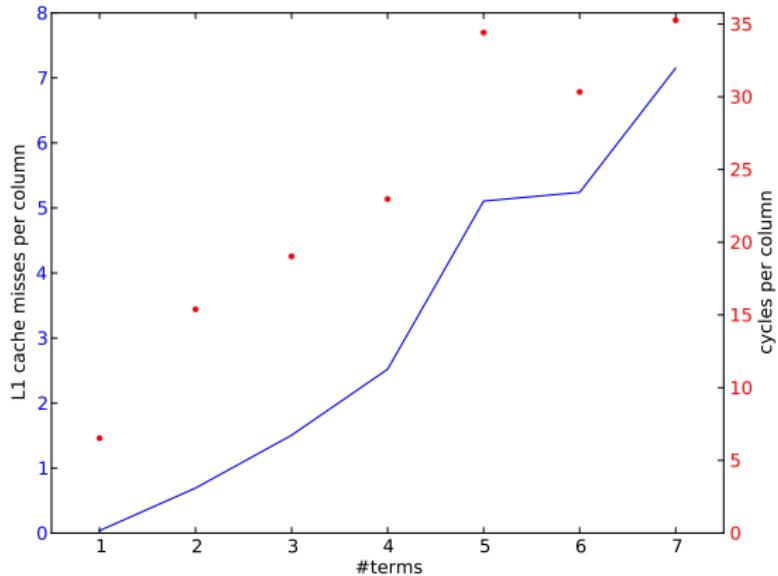
Two caches of 12 elements: direct mapped (left) and 3-way associative (right)

Direct map: 0–12 is conflict

Associative: no conflict

Associativity in practice

$$\forall j: y_j = y_j + \sum_{i=1}^m x_{i,j}$$

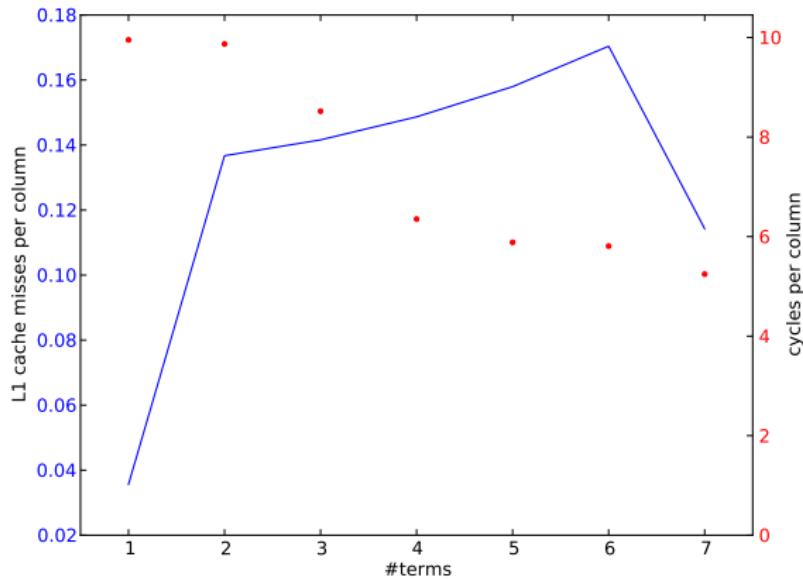


Eijkhout: IPCL 2016

The number of L1 cache misses and the number of cycles for each j column

One remedy

Do not user powers of 2.



The number of L1 cache misses and the number of cycles for each j column accumulation, vector length $4096 + 8$

More memory system topics

Bandwidth / latency

Simple model for sending n words:

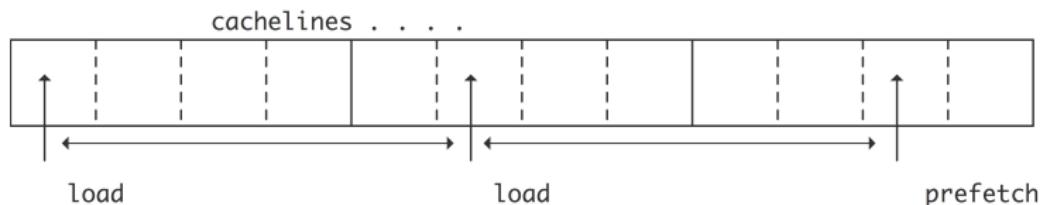
$$t = \alpha + \beta n$$

Quoted bandwidth figures are always optimistic:

- bandwidth shared between cores
- bandwidth wasted on coherence
- assumes optimal scheduling of DRAM banks

Prefetch

- Do you have to wait for every item from memory?
- Memory controller can infer streams: prefetch
- Sometimes controllable through assembly, directives, libraries (AltiVec)
- One form of latency hiding



Memory pages

Memory is organized in pages:

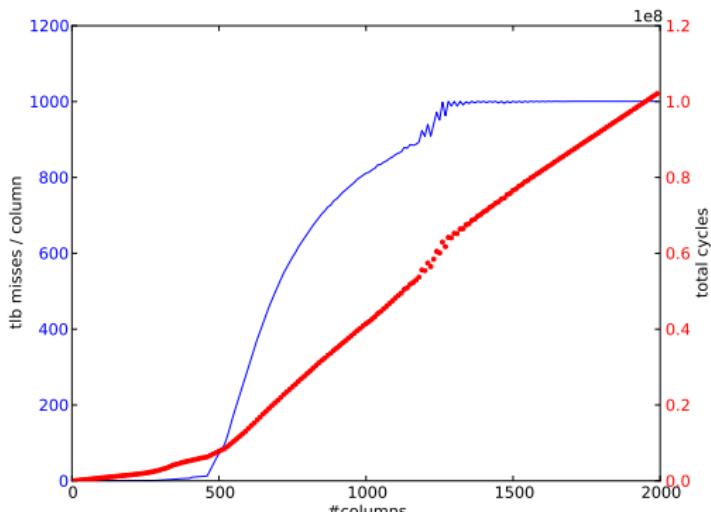
- Translation between logical address, as used by program, and physical in memory
- This serves virtual memory and relocatable code
- so we need another translation stage.

Page translation: TLB

- General page translation: slowish but extensive
- **TLB!** (**TLB!**) is a small list of frequently used pages
- Example of spatial locality: items on an already referenced page are found faster

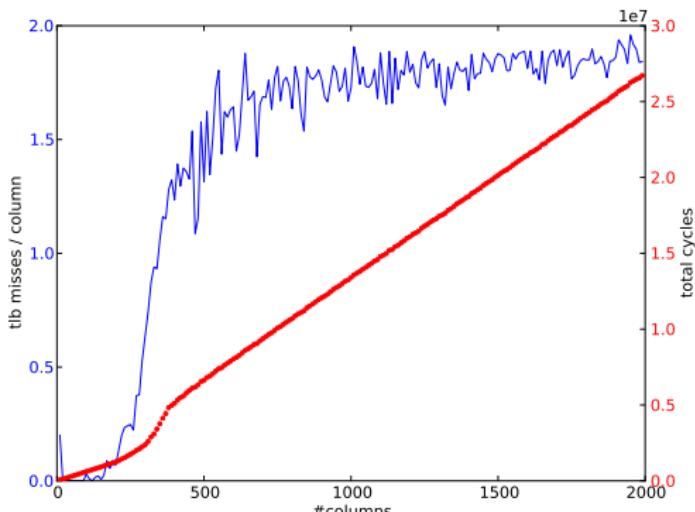
TLB misses

```
#define INDEX(i, j, m, n) i+j*m  
array = (double*) malloc(m*n*sizeof(double));  
/* traversal #2 */  
for (i=0; i<m; i++)  
    for (j=0; j<n; j++)  
        array[INDEX(i, j, m, n)] = array[INDEX(i, j, m, n)]+1;
```



TLB hits

```
#define INDEX(i, j, m, n) i+j*m  
array = (double*) malloc(m*n*sizeof(double));  
/* traversal #1 */  
for (j=0; j<n; j++)  
    for (i=0; i<m; i++)  
        array[INDEX(i, j, m, n)] = array[INDEX(i, j, m, n)]+1;
```



Little's Law

- Item loaded from memory, processed, new item loaded in response
- But this can only happen after latency wait
- Items during latency are independent, therefore

$$\text{Concurrency} = \text{Bandwidth} \times \text{Latency}.$$

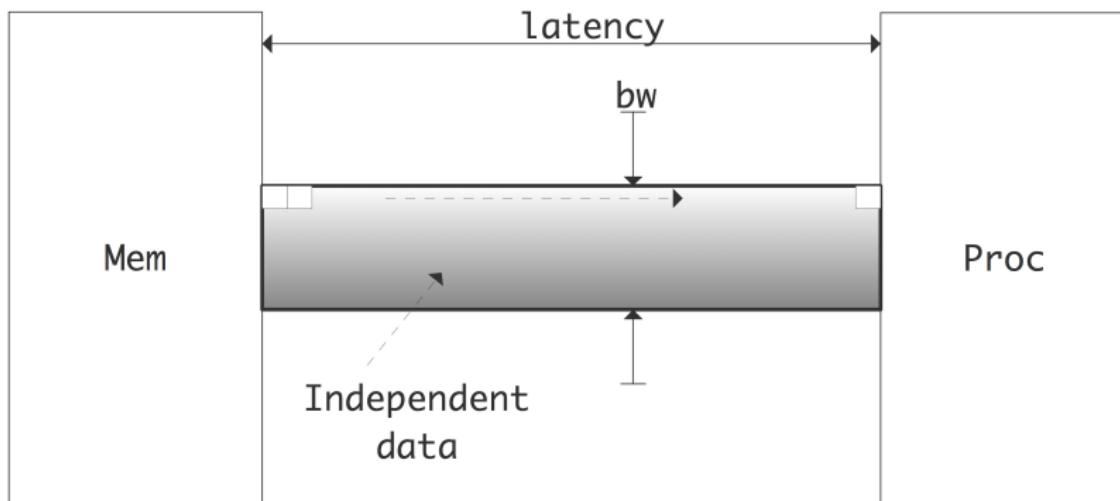


Table of Contents

Why multicore

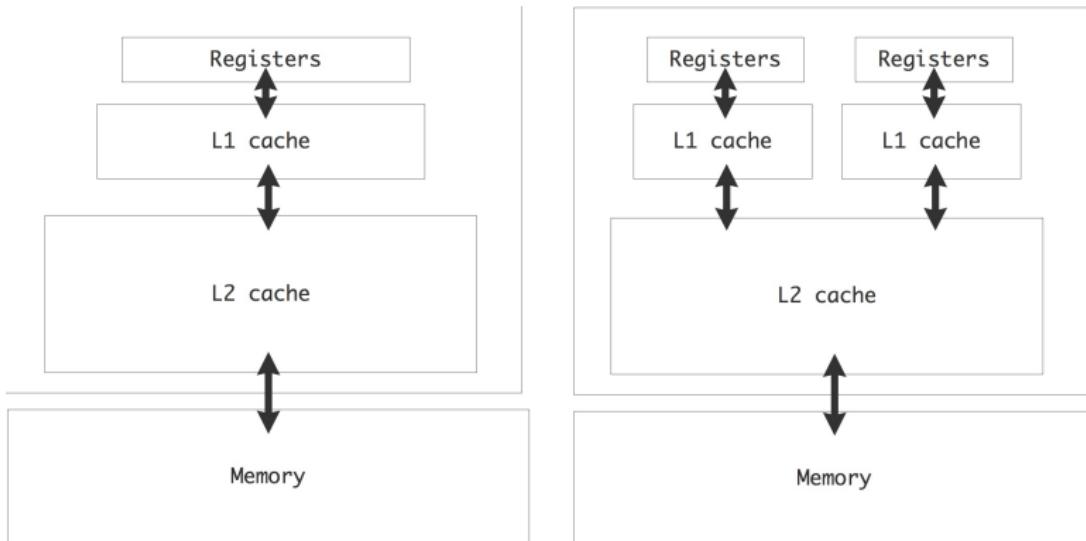
Quest for higher performance:

- Two cores at half speed more energy-efficient than one at full speed.
- Not enough instruction parallelism for long pipelines

Multicore solution:

- More theoretical performance
- Burden for parallelism is now on the programmer

Multicore caches



Cache coherence

MSI! (MSI!) coherence protocol:

Modified: the cacheline has been modified

Shared: the line is present in at least one cache and is unmodified.

Invalid: the line is not present, or it is present but a copy in another cache has been modified.

Coherence issues

- Coherence is automatic, so you don't have to worry about it. . .
- . . . except when it saps performance
- Beware false sharing
 - writes to different elements of a cache line

Balance analysis

- Sandy Bridge core can absorb 300 GB/s
- 4 DDR3/1600 channels provide 51 GB/s, difference has to come from reuse
- It gets worse: latency 80ns, bandwidth 51 GB/s,
Little's law: parallelism 64 cache lines
- However, each core only has 10 line fill buffers,
so we need 6–7 cores to provide the data for one core
- Power: cores are 72%, uncore 17, DRAM 11.
- Core power goes 40% to instruction handling, not arithmetic
- Time for a redesign of processors and programming; see my research presentation

Table of Contents

How much performance is possible?

Performance limited by

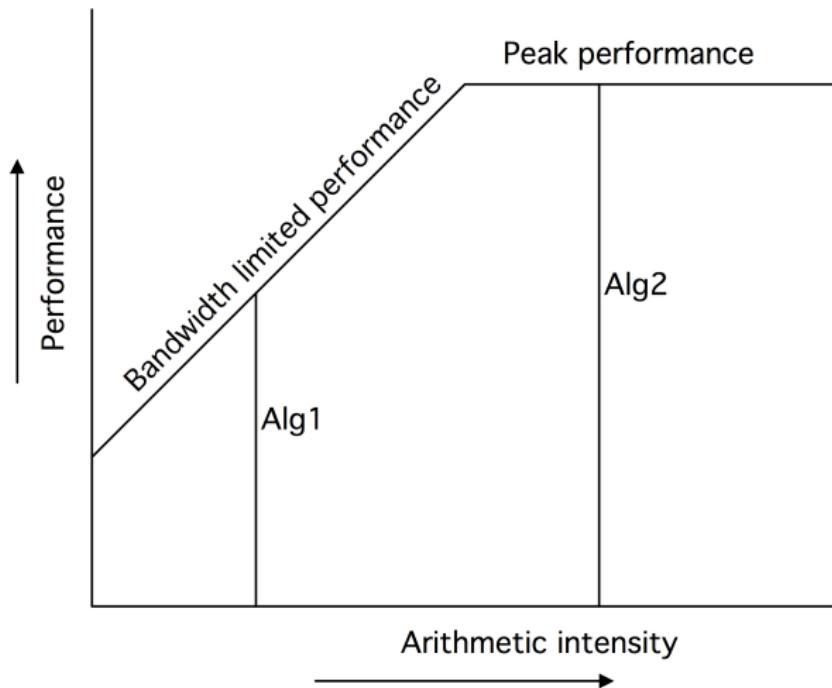
- Processor peak performance: absolute limit
- Bandwidth: linear correlation with performance

Arithmetic intensity: ratio of operations per transfer

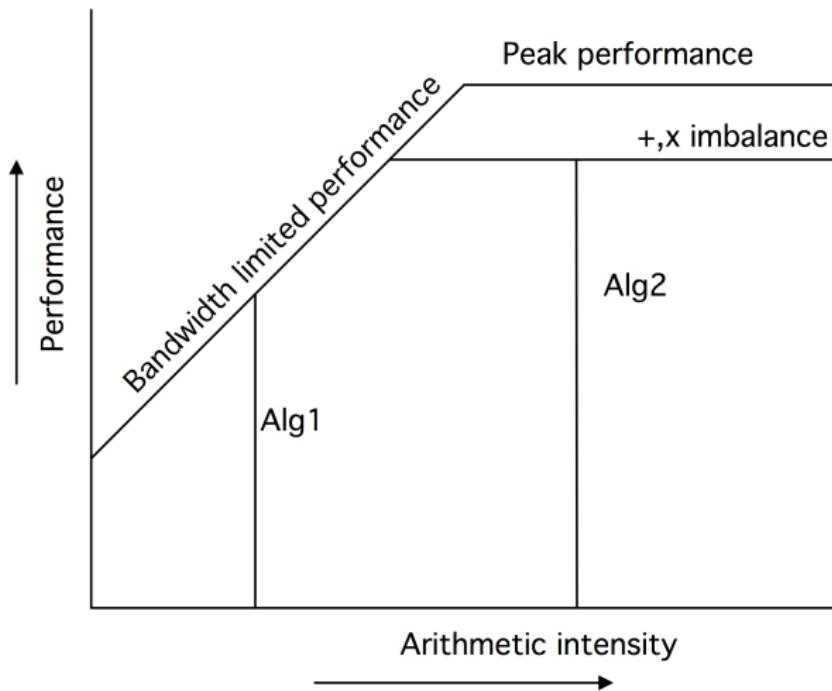
If AI high enough: processor-limited

otherwise: bandwidth-limited

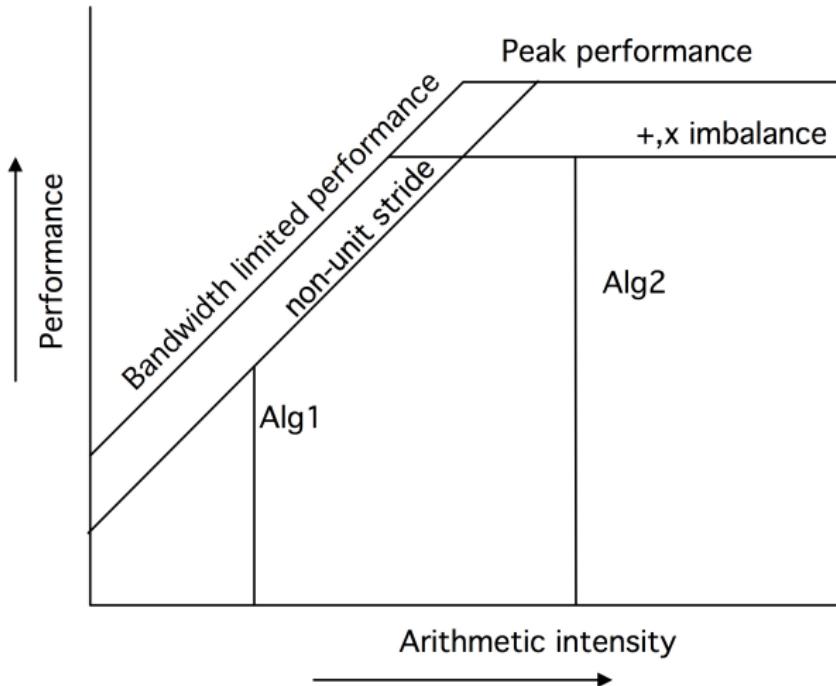
Performance depends on algorithm:



Insufficient utilization of functional units:



Imperfect data transfer:



Architecture aware programming

- Cache size: block loops
- pipelining and vector instructions: expose streams of instructions
- reuse: restructure code (both loop merge and splitting, unroll)
- TLB: don't jump all over memory
- associativity: watch out for powers of 2

Loop blocking

Multiple passes over data

```
for ( k < small bound )
    for ( i < N )
        x[i] = f( x[i], k, .... )
```

Block to be cache contained

```
for ( ii < N; ii += blocksize )
    for ( k < small bound )
        for ( i = ii; i < ii + blocksize; i++ )
            x[i] = f( x[i], k, .... )
```

This requires independence of operations

The ultimate in performance programming: DGEMM

Matrix-matrix product $C = A \cdot B$

$$\forall_i \forall_j \forall_k : c_{ij} += a_{ik} b_{kj}$$

- Three independent loop i, j, k
- all three blocked i', j', k'
- Many loop permutations, blocking factors to choose

DGEMM variant

Inner products

```
for ( i )
    for ( j )
        for ( k )
            c[i,j] += a[i,k] * b[k,j]
```

DGEMM variant

Outer product: updates with low-rank columns-times-vector

```
for ( k )
    for ( i )
        for ( j )
            c[i,j] += a[i,k] * b[k,j]
```

DGEMM variant

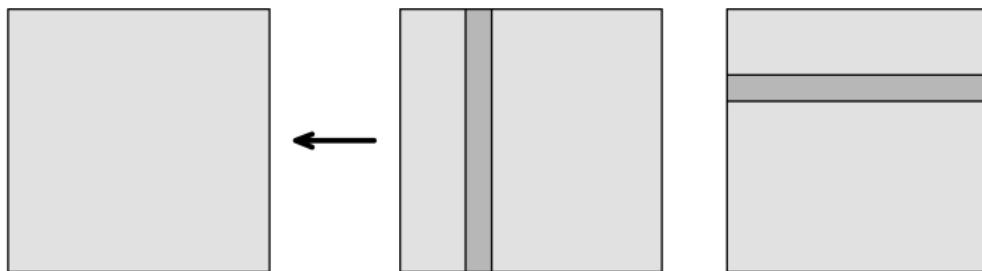
Building up rows by linear combinations

```
for ( i )
    for ( k )
        for ( j )
            c[i,j] += a[i,k] * b[k,j]
```

Exchanging i, j : building up columns

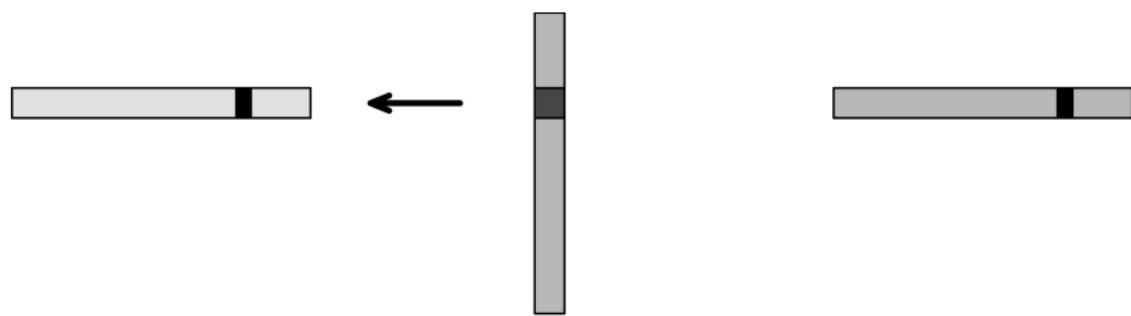
Rank 1 updates

$$C_{**} = \sum_k A_{*k} B_{k*}$$



Matrix-panel multiply

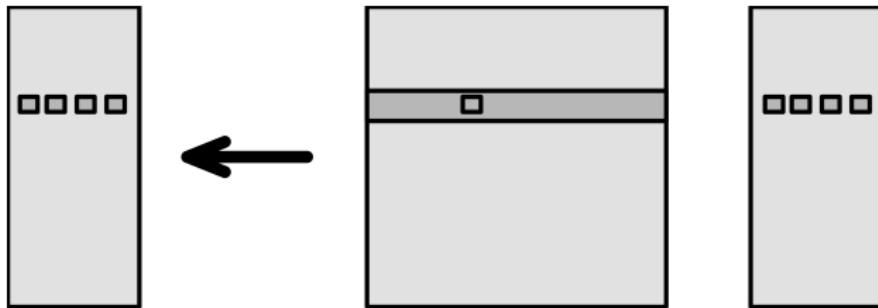
Block of A times ‘sliver’ of B



Inner algorithm

For inner i :

```
// compute C[i,*] :  
for k:  
    C[i,*] = A[i,k] * B[k,*]
```



Tuning

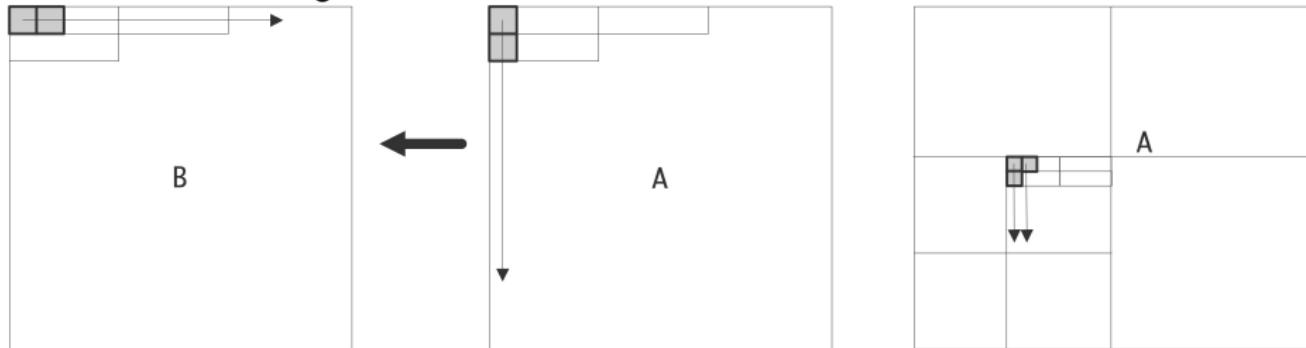
For inner i :

```
// compute C[i,*] :  
for k:  
    C[i,*] += A[i,k] * B[k,*]
```

- $C[i,*]$ stays in register
- $A[i,k]$ and $B[k,*]$ stream from L1
- blocksize of A for L2 size
- A stored by rows to prevent TLB problems

Cache-oblivious programming

Observation: recursive subdivision will ultimately make a problem small / well-behaved enough



Cache-oblivious matrix-matrix multiply

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

with $C_{11} = A_{11}B_{11} + A_{12}B_{21}$

Recursive approach will be cache contained.

Not as high performance as being cache-aware...

Table of Contents

Dennard scaling

Scale down feature size by s :

Feature size	$\sim s$
Voltage	$\sim s$
Current	$\sim s$
Frequency	$\sim s^{-1}$

Miracle conclusion:

$$\text{Power} = V \cdot I \sim s^2; \text{Power density} \sim 1$$

Everything gets better, cooling problem stays the same

Opportunity for more components, higher frequency

Dynamic power

Charge	$q = CV$
Work	$W = qV = CV^2$
Power	$W/\text{time} = WF = CV^2F$

(1)

Two cores at half frequency:

$$\left. \begin{array}{l} C_{\text{multi}} = 2C \\ F_{\text{multi}} = F/2 \\ V_{\text{multi}} = V/2 \end{array} \right\} \Rightarrow P_{\text{multi}} = P/4.$$

Same computation, less power

Parallelism

Justification

Parallelism can be approached in several different ways. This session will discuss data parallelism versus instruction parallelism, issues in shared memory parallelism, parallel programming systems, the interconnects of distributed memory parallelism, scaling measures.

Table of Contents

Why

- Parallelism has many aspects
- Let's look at a simple and structured problem
- Illustrate various aspects

Conway's Game of Life



→



→

Nothing



→



→



→

Repeat



→



→



→

Stable



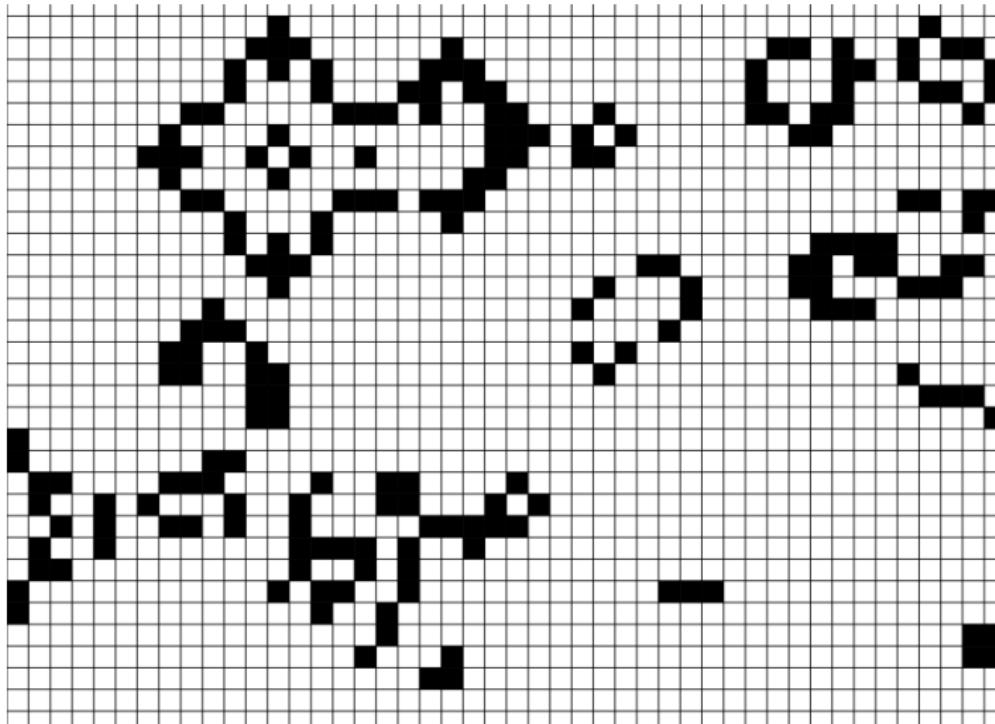
→



→

Nothing

Conway's Game of Life



<http://youtu.be/C2vgICfQawE?t=1m12s>

Code

Evaluate 3×3 box:

```
def life_evaluation( cells ):  
    # cells is a 3x3 array  
    count = 0  
    for i in [0,1,2]:  
        for j in [0,1,2]:  
            if i!=1 and j!=1:  
                count += cells[i,j]  
    return life_count_evaluation( cells[1,1],count )
```

Code

Naive implementation: save all boards

```
life_board.create(final_time,N,N)

for t in [0:final_time-1]:
    for i in [0:N-1]:
        for j in [0:N-1]:
            life_board[t+1,i,j] :=  
                life_evaluation( life_board[t,i-1:i+1,j-1:j+1] )
```

Code

More efficient: just two boards

```
life_board.create(N,N)
temp_board.create(N,N)

for t in [0:final_time-1]:
    life_generation( life_board,temp_board )

def life_generation( board,tmp ):
    for i in [0:N-1]:
        for j in [0:N-1]:
            tmp[i,j] = board[i,j]
    for i in [0:N-1]:
        for j in [0:N-1]:
            board[i,j] = life_evaluation( tmp[i-1:i+1,j-1:j+1] )
```

Can we do even better?

Parallelism: more than one processing element

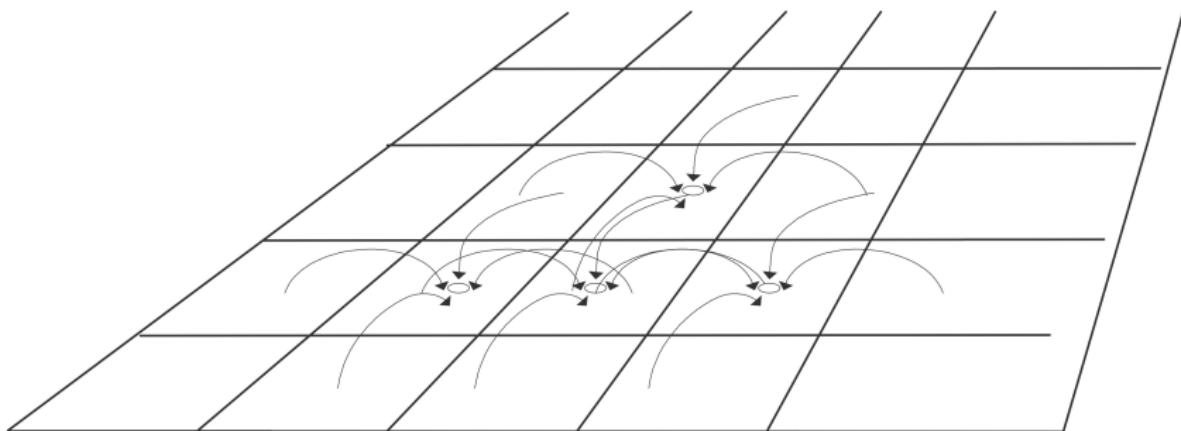
Parallel thinking: thinking about what can be done simultaneously, and what is really sequentially dependent.

Example: Game of Life is three loops, inner two have conceptually independent iterations.

(Strategy influences code design)

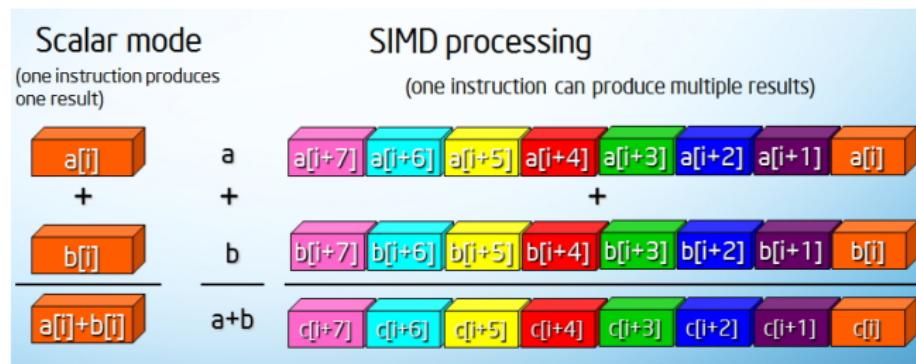
Data parallelism

All cells are updated identically



Data parallelism

Instruction level:



Code transformation for data parallelism

```
for i in [0:N]:  
    for j in [0:N]:  
        count = 0  
        for ii in {-1,0,+1}:  
            for jj in {-1,0,+1}:  
                if ii!=0 and jj!=0:  
                    count += board[i+ii,j+jj]  
  
for i in [0:N]:  
    for j in [0:N]:  
        count[j] = 0  
        for ii in {-1,0,+1}:  
            for jj in {-1,0,+1}:  
                if ii!=0 and jj!=0:  
                    for j in [0:N]:  
                        count[j] +=  
                            board[i+ii,j+jj]
```

GPU data parallelism

```
kerneldef life_step( board ):  
    i = my_i_number()  
    j = my_j_number()  
    board[i,j] = life_evaluation( board[i-1:i+1, j-1:j+1] )  
  
for t in [0:final_time]:  
    <<N,N>>life_step( board )
```

Loop-based parallelism

OpenMP: instruction parallelism instead of data

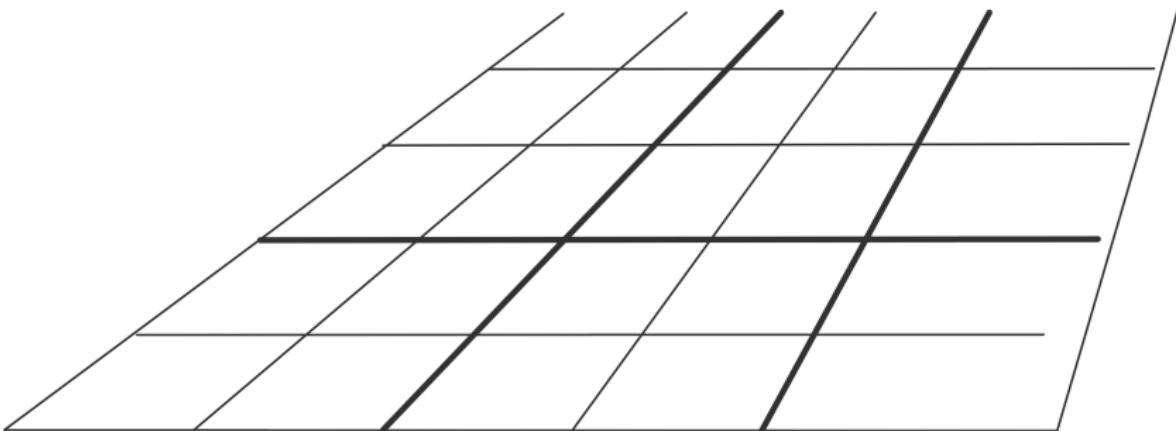
```
def life_generation( board,tmp ):  
    # OMP parallel for  
    for i in [0:N-1]:  
        for j in [0:N-1]:  
            tmp[i,j] = board[i,j]  
    # OMP parallel for  
    for i in [0:N-1]:  
        for j in [0:N-1]:  
            board[i,j] = life_evaluation( tmp[i-1:i+1,j-1:j+1] )
```

(collapse nested loops)

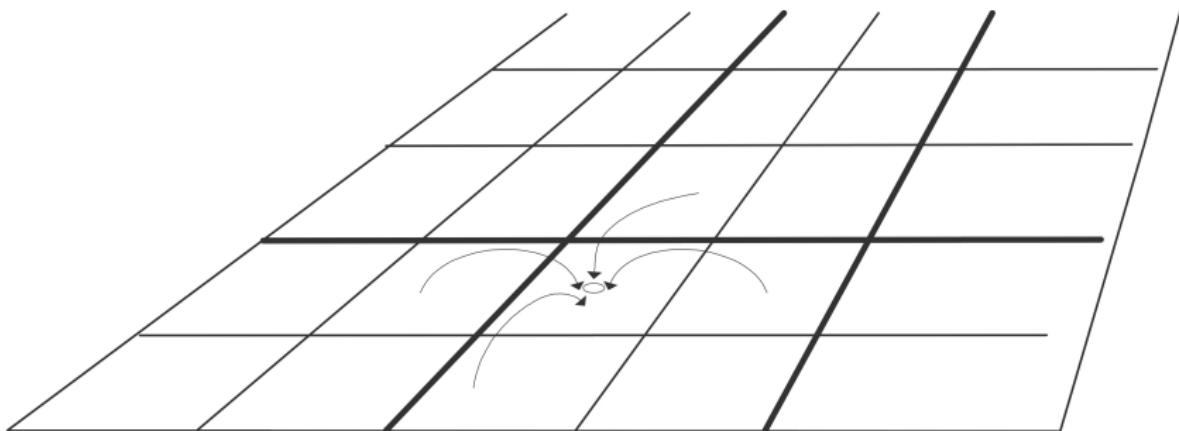
Coarse-grained parallelism

- Data parallelism requires synchronization on instruction level
- Not possible for multicore and parallel processors
- ⇒ coarser grained parallelism, for instance give each core/processor a subdomain of the Life board.

Work division

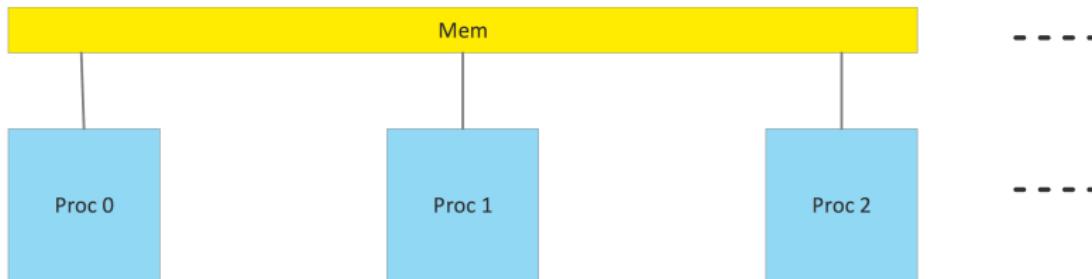


... has some data implications

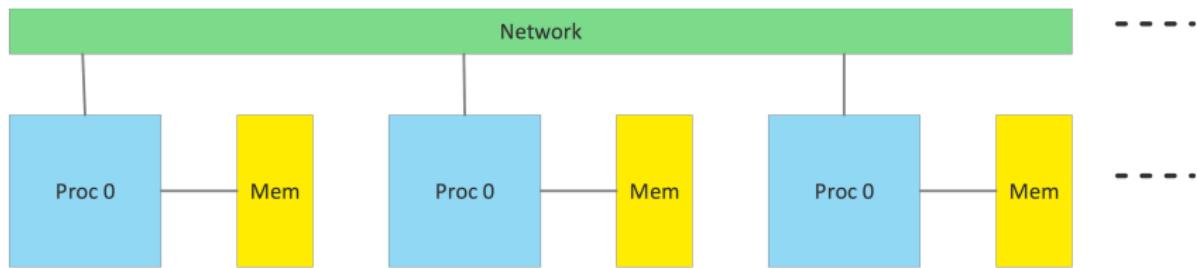


Reads from neighbours, no conflicting writes

Shared memory



Distributed memory

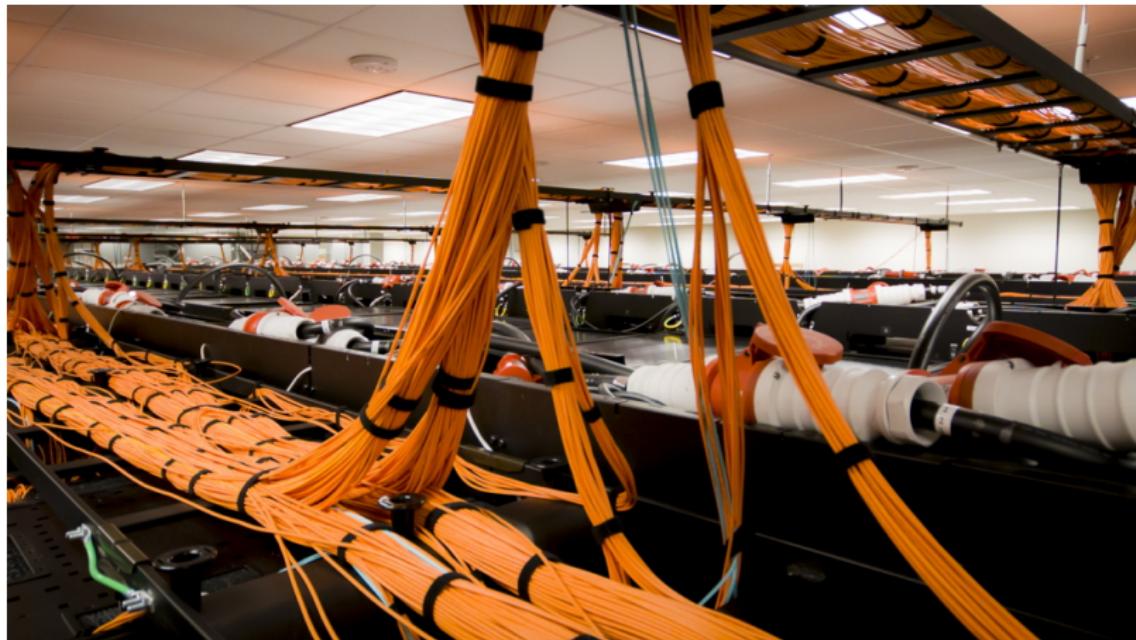


Tricky: any remote data access has to be managed.

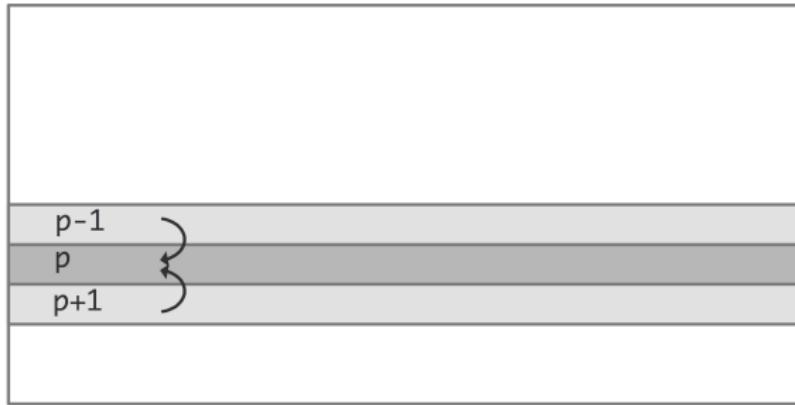
Distributed memory in real life



... takes a lot of cabling



Message passing

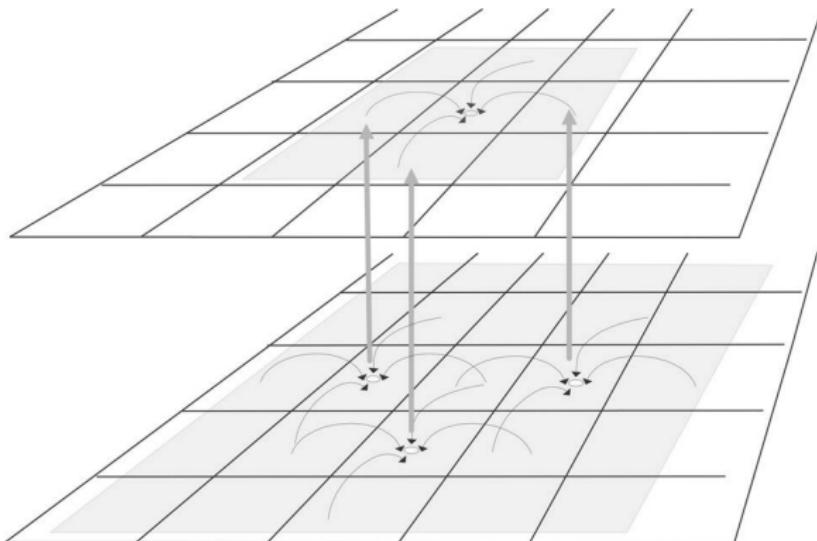


```
high_line = MPI_Receive(from=p-1,cells=N)
low_line = MPI_Receive(from=p+1,cells=N)

tmp_line = my_line.copy()
my_line = life_line_update(high_line,tmp_line,low_line,N)
```

Task scheduling

- So far: sequence of boards, parallelism in board
- Essential dependencies: (i, j) depends on $(i \pm 1, j \pm 1)$,
- ... depends on $(i \pm 2, j \pm 2)$, and so on



Declare tasks

```
for t in [0:T]:  
    for i in [0:N]:  
        for j in [0:N]:  
            task( id=[t+1,i,j],  
                  prereqs=[ [t,i,j], [t,i-1,j], [t,i+1,j] # et cetera  
                            ] )
```

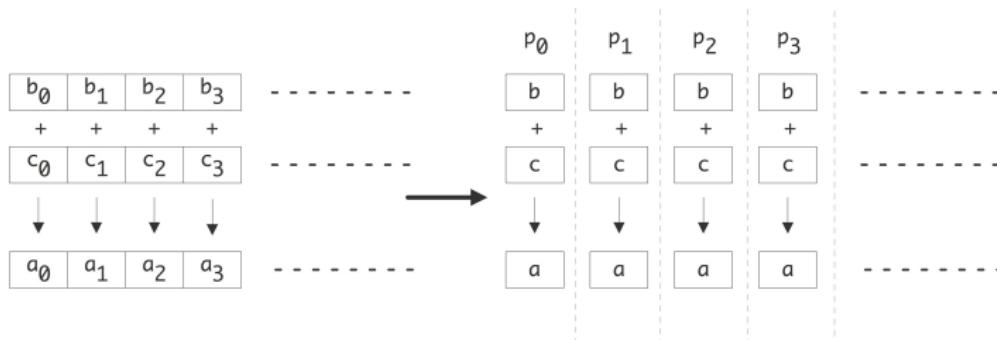
Schedule tasks

```
while there_are_tasks_left():
    for r in running_tasks:
        if r.finished():
            for t in scheduled_tasks:
                t.mark_available_input(r)
t = find_available_task()
p = find_available_processor()
schedule(t,p)
```

Table of Contents

The basic idea

```
for (i=0; i<n; i++)  
    a[i] = b[i] + c[i];
```



```
for (i=my_low; i<my_high; i++)  
    a[i] = b[i] + c[i];
```

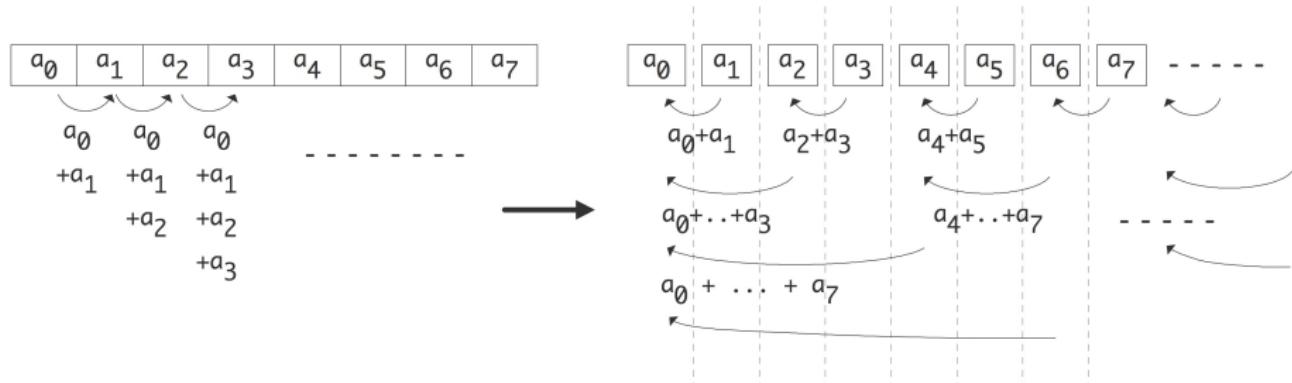
Time goes down linearly with processors

If it was always that easy...

```
s = 0;  
for (i=0; i<n; i++)  
    s += x[i]
```

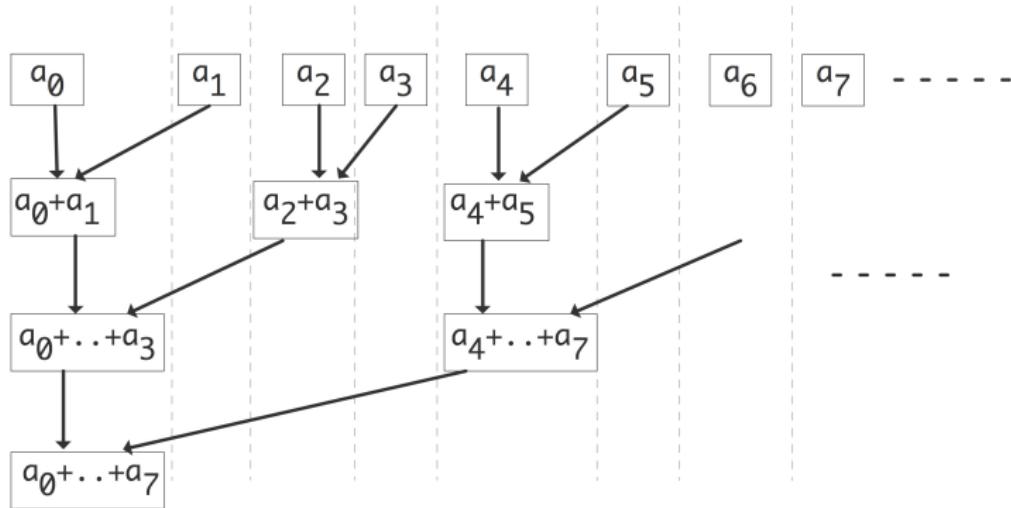
Recoding

```
for (s=2; s<n; s*=2)  
    for (i=0; i<n; i+=s)  
        x[i] += x[i+s/2]
```



And then there is hardware

Topology of the processors:



increasing distance: limit on parallel speedup

Table of Contents

Granularity

Definition

Definition: granularity is the measure for how many operations can be performed between synchronizations

Instruction level parallelism

$$\begin{aligned} a &\leftarrow b + c \\ d &\leftarrow e * f \end{aligned}$$

For the compiler / processor to worry about

Data parallelism

```
for (i=0; i<1000000; i++)  
    a[i] = 2*b[i];
```

- Array processors, vector instructions, pipelining, GPUs
- Sometimes harder to discover
- Often used mixed with other forms of parallelism

Task-level parallelism

```
if optimal (root) then
    exit
else
    parallel: SearchInTree (leftchild),SearchInTree (rightchild)
        Procedure SearchInTree(root)
```

Unsynchronized tasks: fork-join
general scheduler

```
while there are tasks left do
    wait until a processor becomes inactive;
    spawn a new task on it
```

Conveniently parallel

Example: Mandelbrot set

Parameter sweep,
often best handled by external tools

Medium-grain parallelism

Mix of data parallel and task parallel

```
my_lower_bound = // some processor-dependent number  
my_upper_bound = // some processor-dependent number  
for (i=my_lower_bound; i<my_upper_bound; i++)  
    // the loop body goes here
```

Efficiency and scaling

Speedup

- Single processor time T_1 , on p processors T_p
- speedup is $S_p = T_1 / T_p$, $S_P \leq p$
- efficiency is $E_p = S_p / p$, $0 < E_p \leq 1$

Many caveats

- Is T_1 based on the same algorithm? The parallel code?
- Sometimes superlinear speedup.
- Can the problem be run on a single processor?
- Can the problem be evenly divided?

Limits on speedup/efficiency

- F_s sequential fraction, F_p parallelizable fraction
- $F_s + F_p = 1$
- $T_1 = (F_s + F_p)T_1 = F_s T_1 + F_p T_1$
- Amdahl's law: $T_p = F_s T_1 + F_p T_1 / p$
- $P \rightarrow \infty: T_p \downarrow T_1 F_s$
- Speedup is limited by $S_P \leq 1/F_s$, efficiency is a decreasing function $E \sim 1/P$.
- loglog plot: straight line with slope -1

Amdahl's law with communication overhead

- Communication independent of p : $T_p = T_1(F_s + F_p/P) + T_c$
- assume fully parallelizable: $F_p = 1$
- then $S_p = \frac{T_1}{T_1/p + T_c}$
- For reasonable speedup: $T_c \ll T_1/p$ or $p \ll T_1/T_c$:
number of processors limited by ratio of scalar execution time and communication overhead

Gustafson's law

- Let $T_p = F_s + F_p \equiv 1$
- then $T_1 = F_s + p \cdot F_p$
- Speedup:

$$S_p = \frac{T_1}{T_p} = \frac{F_s + p \cdot F_p}{F_s + F_p} = F_s + p \cdot F_p = p - (p-1) \cdot F_s.$$

slowly decreasing function of p

Scaling

- Amdahl's law: strong scaling
same problem over increasing processors
- Often more realistic: weak scaling
increase problem size with number of processors,
for instance keeping memory constant
- Weak scaling: $E_p > c$
- example (below): dense linear algebra

Simulation scaling

- Assumption: simulated time S , running time T constant, now increase precision
- m memory per processor, and P the number of processors

$$M = Pm \quad \text{total memory.}$$

d the number of space dimensions of the problem, typically 2 or 3,

$$\Delta x = 1/M^{1/d} \quad \text{grid spacing.}$$

- stability:

$$\Delta t = \begin{cases} \Delta x = 1 / M^{1/d} & \text{hyperbolic case} \\ \Delta x^2 = 1 / M^{2/d} & \text{parabolic case} \end{cases}$$

With a simulated time S :

$$k = S/\Delta t \quad \text{time steps.}$$

- Assume time steps parallelizable

$$T = kM/P = \frac{S}{\Delta t} m.$$

Setting $T/S = C$, we find

$$m = C\Delta t,$$

memory per processor goes down.

$$m = C\Delta t = c \begin{cases} 1 / M^{1/d} & \text{hyperbolic case} \\ 1 / M^{2/d} & \text{parabolic case} \end{cases}$$

- Substituting $M = Pm$, we find ultimately

$$m = C \begin{cases} 1 / P^{1/(d+1)} & \text{hyperbolic} \\ 1 / P^{2/(d+2)} & \text{parabolic} \end{cases}$$

Table of Contents

Flynn Taxonomy

Consider instruction stream and data stream:

- SISD: single instruction single data
used to be single processor, now single core
- MISD: multiple instruction single data
redundant computing for fault tolerance?
- SIMD: single instruction multiple data
data parallelism, pipelining, array processing, vector instructions
- MIMD: multiple instruction multiple data
independent processors, clusters, MPPs

SIMD

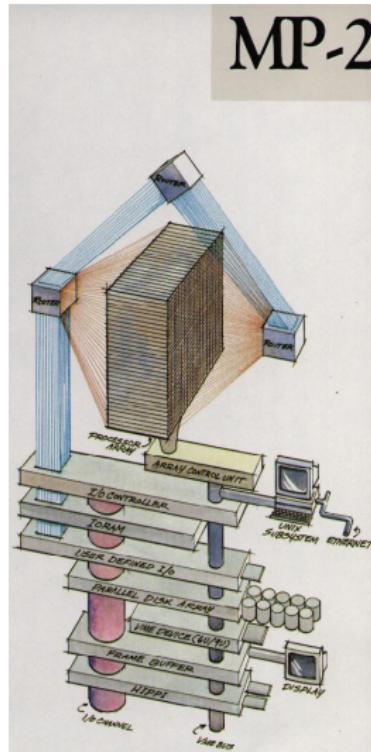
- Relies on streams of identical operations
- See pipelining
- Recurrences hard to accomodate

SIMD: array processors

Technology going back to the 1980s:

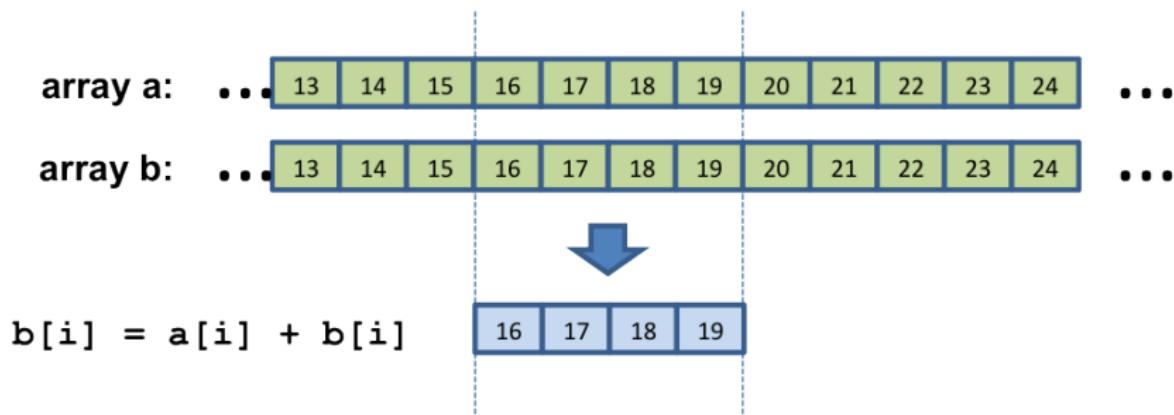
FPS, MasPar, CM, GoodYear

Major advantage: simplification of processor



SIMD as vector instructions

- Register width multiple of 8 bytes:
- simultaneous processing of more than one operand pair
- SSE: 2 operands,
- AVX: 4 or 8 operands



Controlling vector instructions

```
void func(float *restrict c, float *restrict a,
          float *restrict b, int n)
{
#pragma vector always
    for (int i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```

This needs aligned data (`posix_memalign`)

New branches in the taxonomy

- SPMD: single program multiple data
the way clusters are actually used
- SIMD: single instruction multiple threads
the GPU model

MIMD becomes SPMD

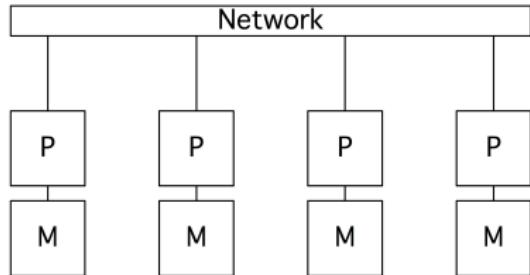
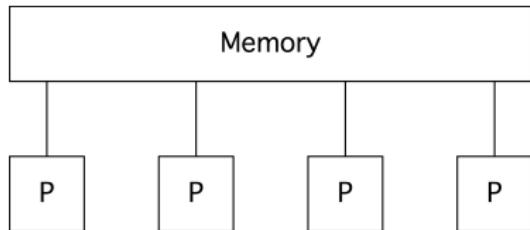
- MIMD: independent processors, independent instruction streams, independent data
- In practice very little true independence: usually the same executable Single Program Multiple Data
- Exceptional example: climate codes
- Old-style SPMD: cluster of single-processor nodes
- New-style: cluster of multicore nodes, ignore shared caches / memory
- (We'll get to hybrid computing in a minute)

GPUs and data parallelism

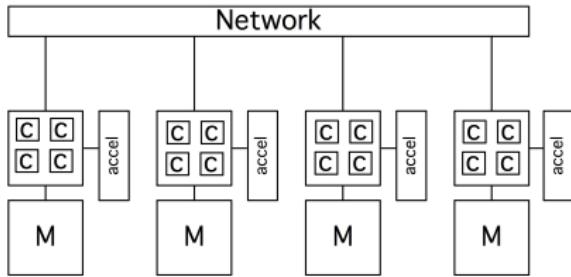
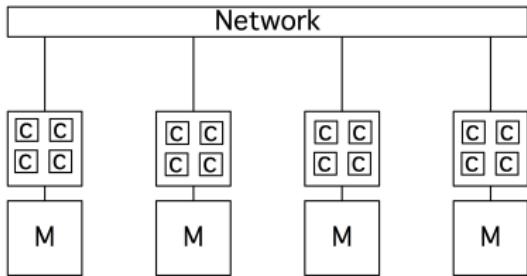
Lockstep in thread block,
single instruction model between streaming processors
(more about GPU threads later)

Table of Contents

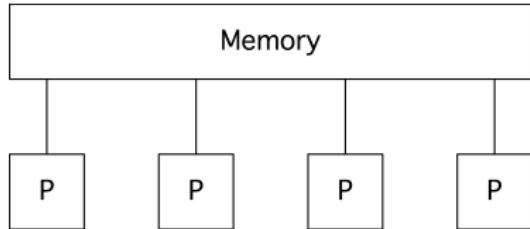
Major types of memory organization, classic



Major types of memory organization, contemporary



Symmetric multi-processing



- The ideal case of shared memory:
every address equally accessible
- This hasn't existed in a while
(Tim Mattson claims Cray-2)
- Danger signs: shared memory programming pretends that memory access is symmetric
in fact: hides reality from you

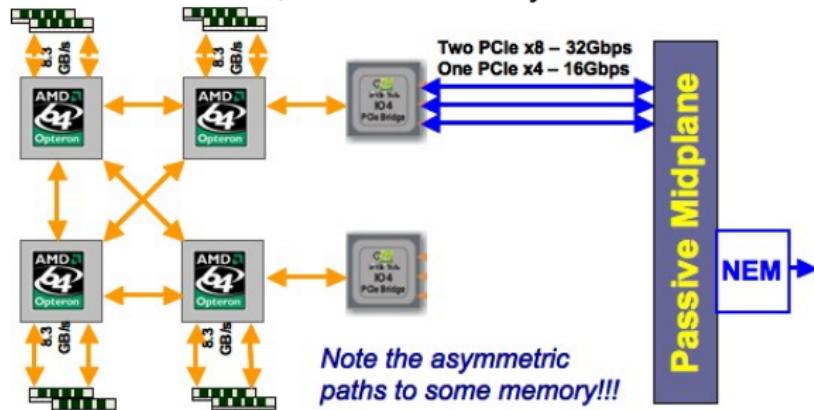
SMP, bus design

- Bus: all processors on the same wires to memory
- Not very scalable: requires slow processors or cache memory
- Cache coherence easy by ‘snooping’

Non-uniform Memory Access

Memory is equally programmable, but not equally accessible

- Different caches, different affinity



- Distributed shared memory: network latency
ScaleMP and other products watch me not believe it

Picture of NUMA

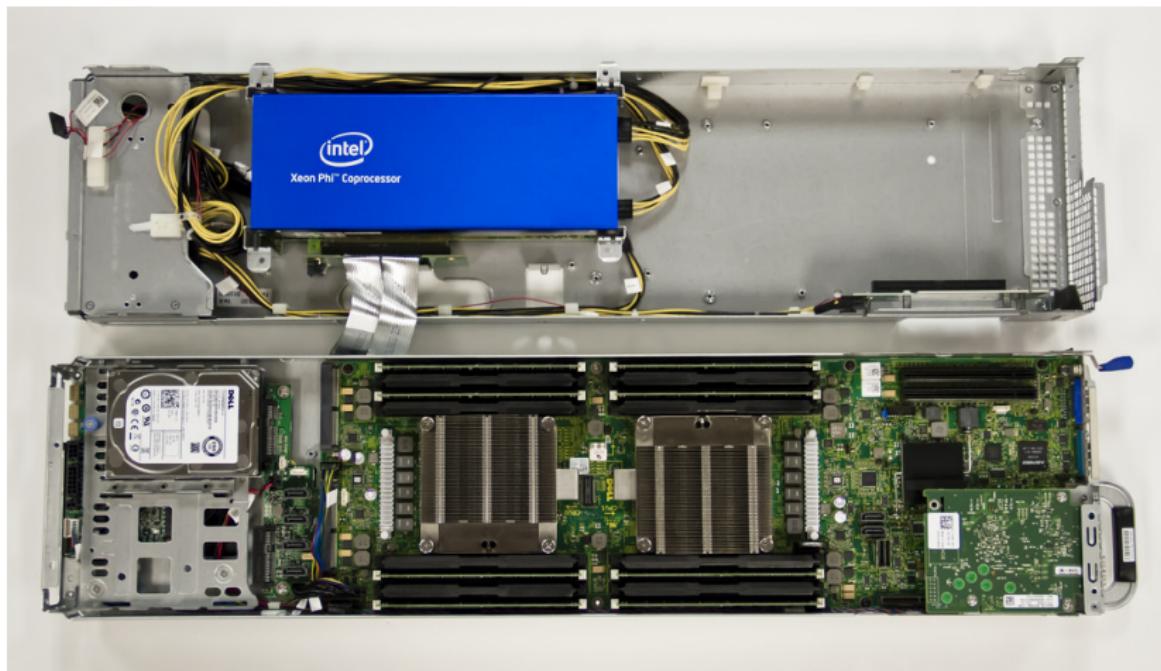


Table of Contents

Topology concepts

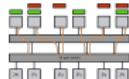
- Hardware characteristics
- Software requirement
- Design: how ‘close’ are processors?

Graph theory

- Degree: number of connections from one processor to others
- Diameter: maximum minimum distance (measured in hops)

Bandwidth

- Bandwidth per wire is nice, adding over all wires is nice, but. . .



- Bisection width: minimum number of wires through a cut
- Bisection bandwidth: bandwidth through a bisection

Design 1: bus

Already discussed; simple design, does not scale very far

Design 2: linear arrays

- Degree 2, diameter P , bisection width 1
- Scales nicely!
- but low bisection width

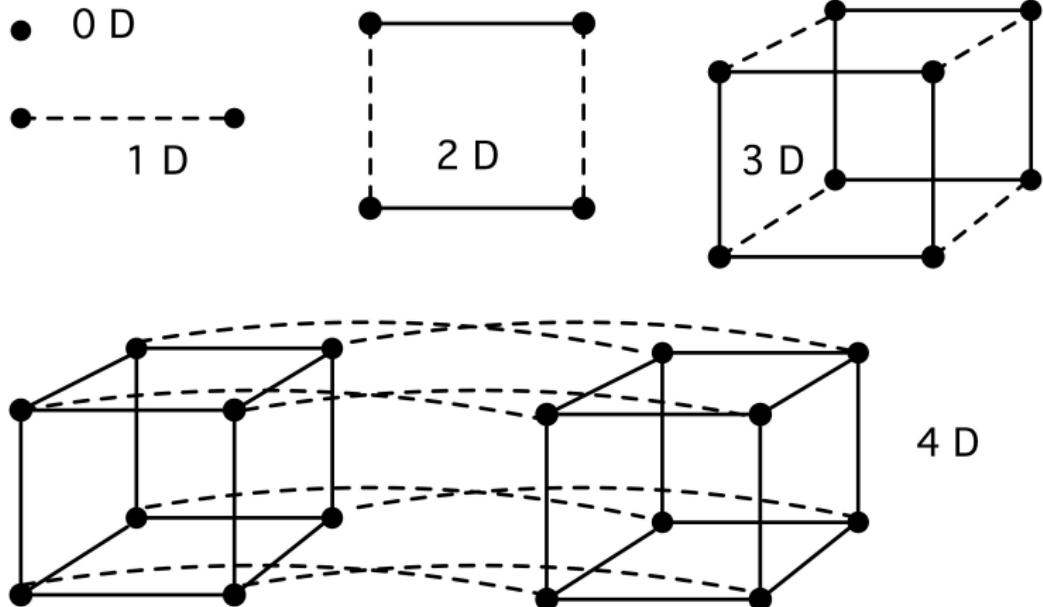
Exercise

Flip last bit, flip one before, ...

Design 3: 2/3-D arrays

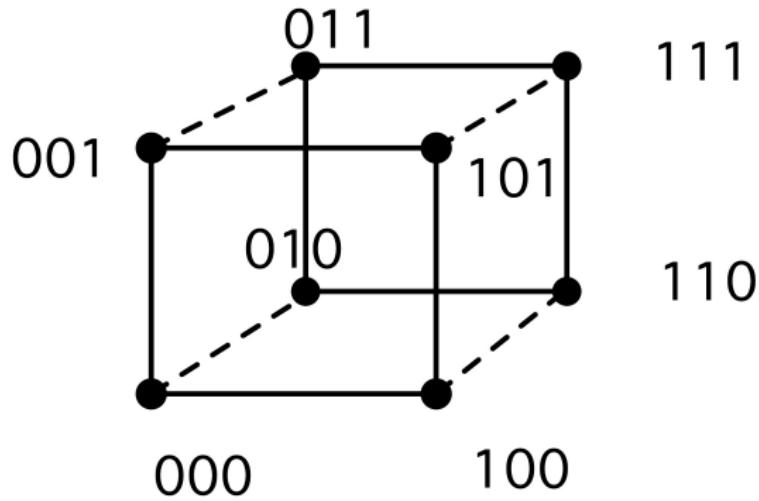
- Degree $2d$, diameter $P^{1/d}$
- Natural design: nature is three-dimensional
- More dimensions: less contention.
K-machine is 6-dimensional

Design 3: Hypercubes



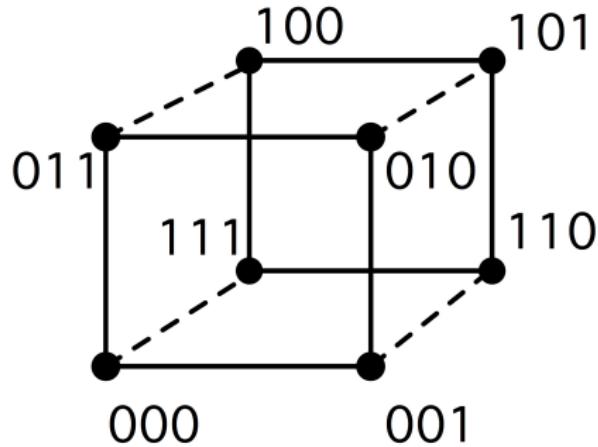
Hypercube numbering

Naive numbering:



Gray codes

Embedding linear numbering in hypercube:



Binary reflected Gray code

1D Gray code :

0 1

2D Gray code :

1D code and reflection: 0 1 : 1 0

append 0 and 1 bit: 0 0 : 1 1

2D code and reflection: 0 1 1 0 : 0 1 1 0

3D Gray code :

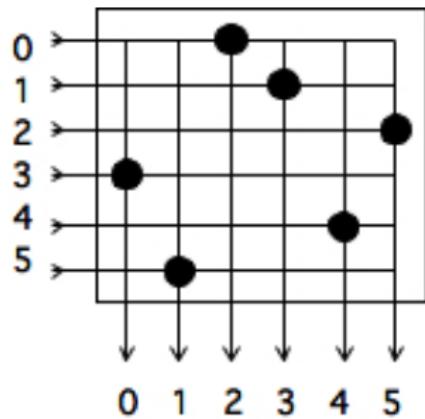
0 0 1 1 : 1 1 0 0

append 0 and 1 bit: 0 0 0 0 : 1 1 1 1

Switching networks

- Solution to all-to-all connection
- (Real all-to-all too expensive)
- Typically layered
- Switching elements: easy to extend

Cross bar

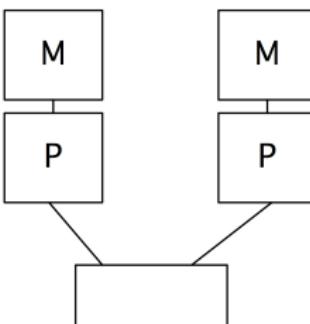
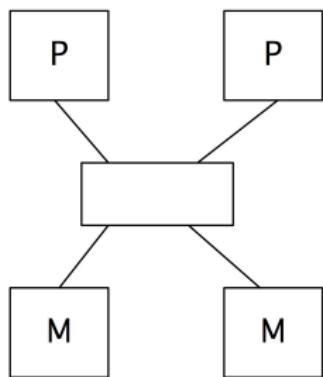


Advantage: non-blocking

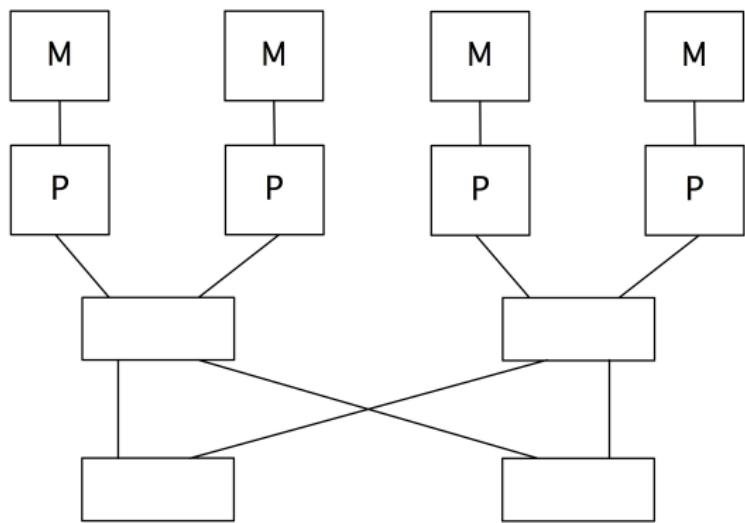
Disadvantage: cost

Butterfly exchange

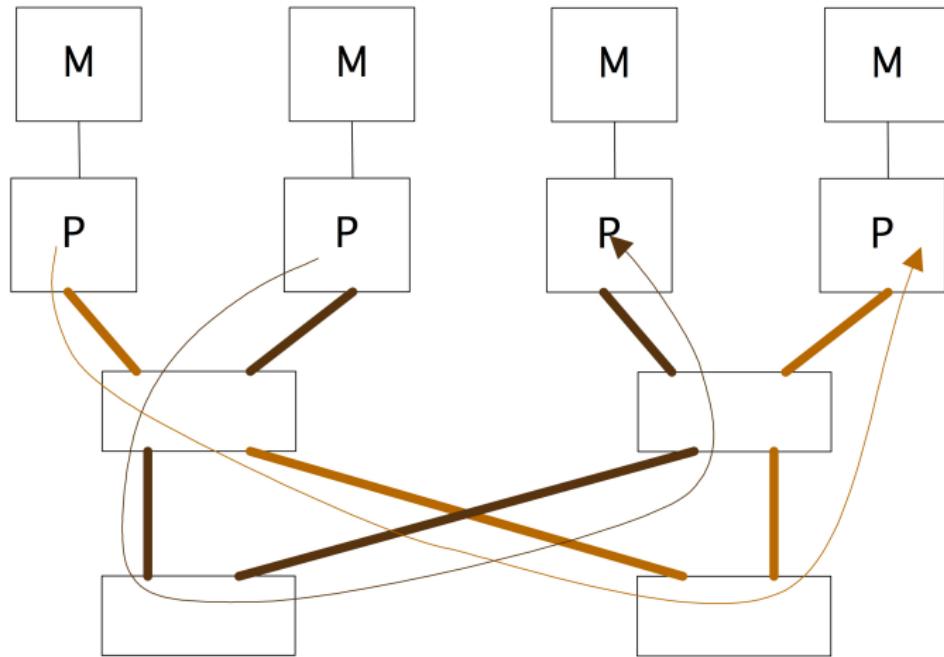
Process to segmented pool of memory, or between processors with private memory:



Building up butterflies

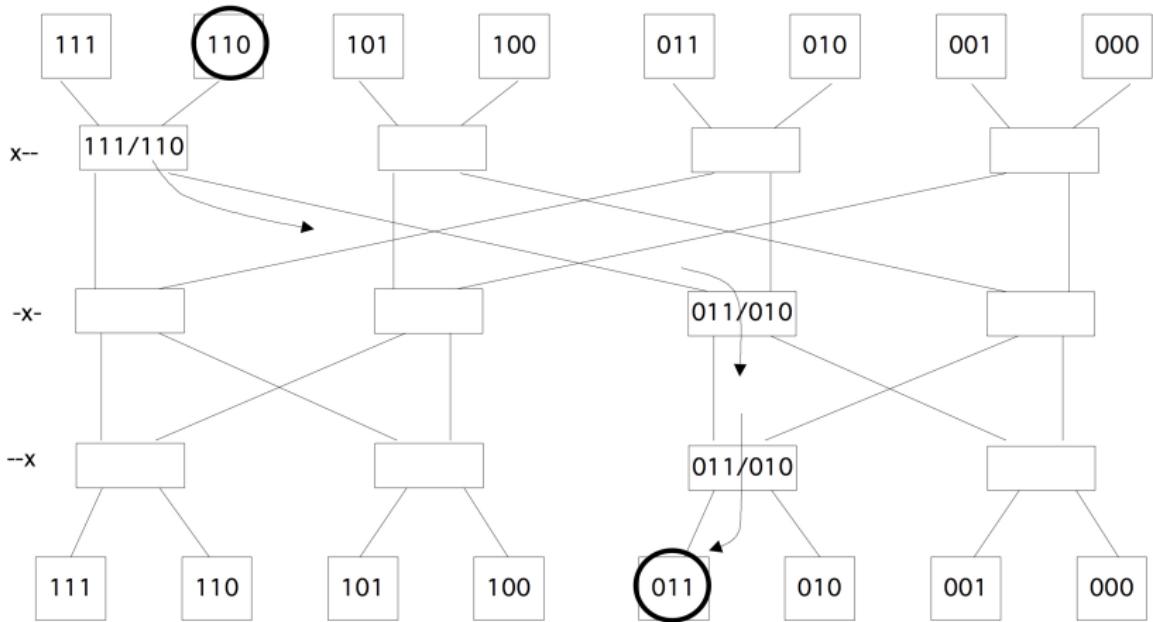


Uniform memory access

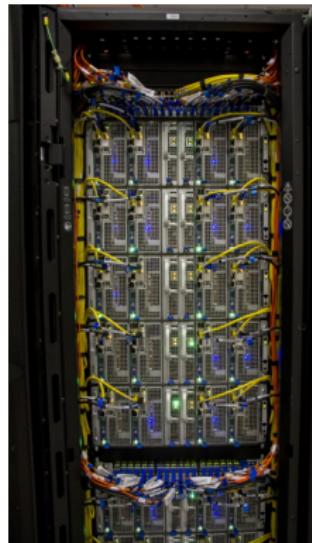
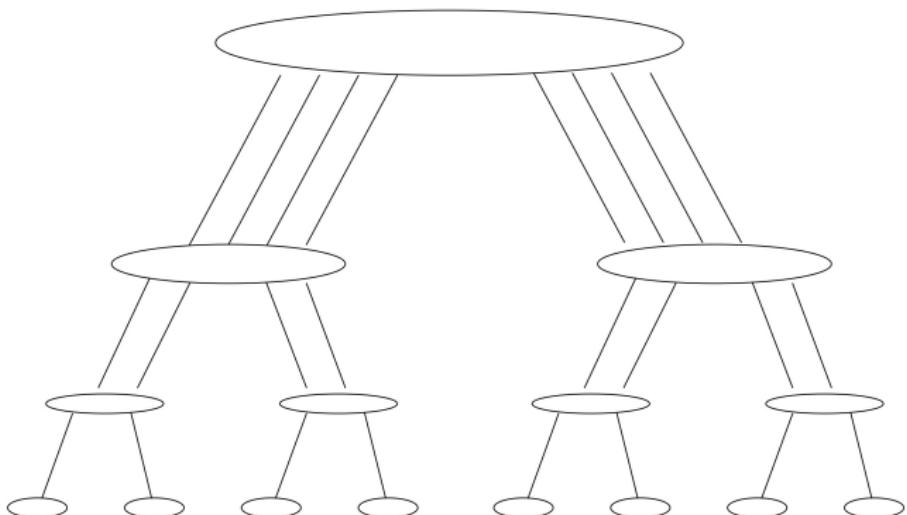


Contention possible

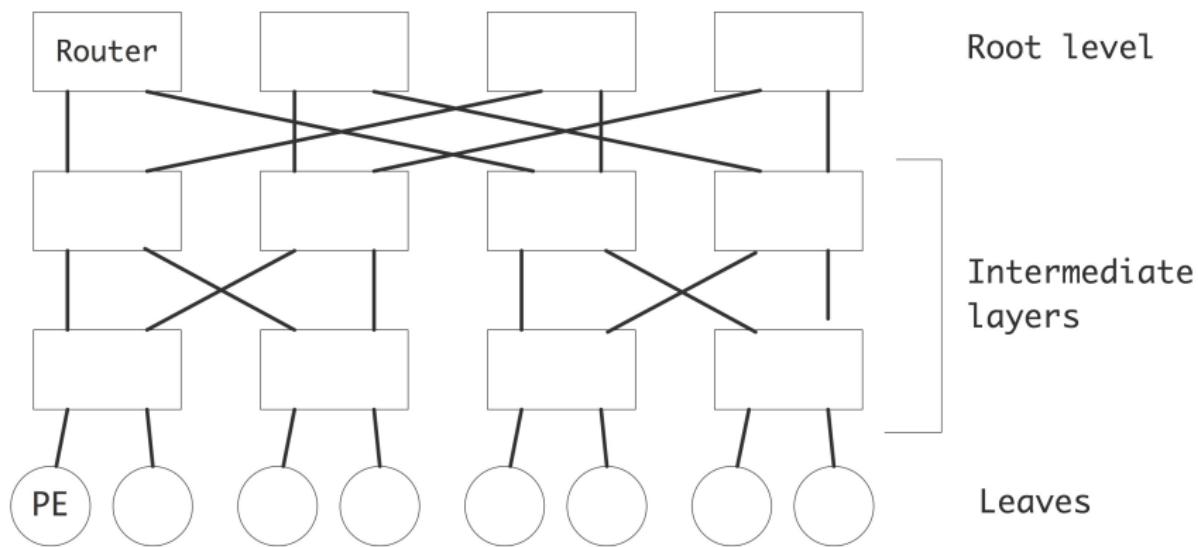
Route calculation



Fat Tree



Fat trees from switching elements



(Clos network)

Fat tree clusters



Mesh clusters



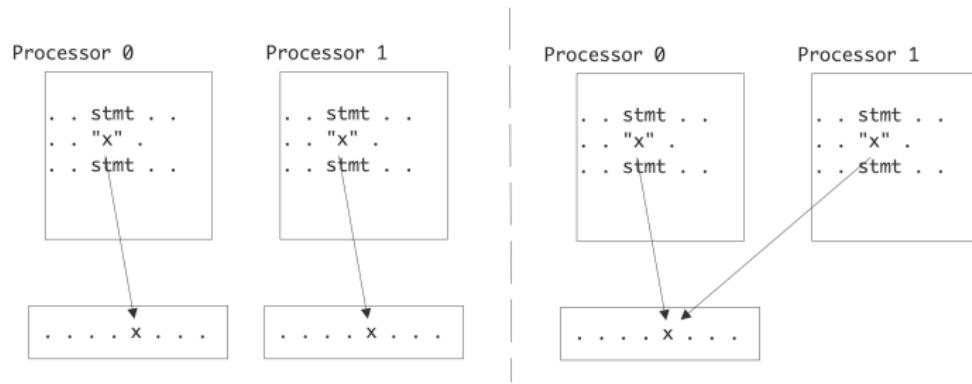
Levels of locality

- Core level: private cache, shared cache
- Node level: numa
- Network: levels in the switch

Table of Contents

Shared vs distributed memory programming

Different memory models:



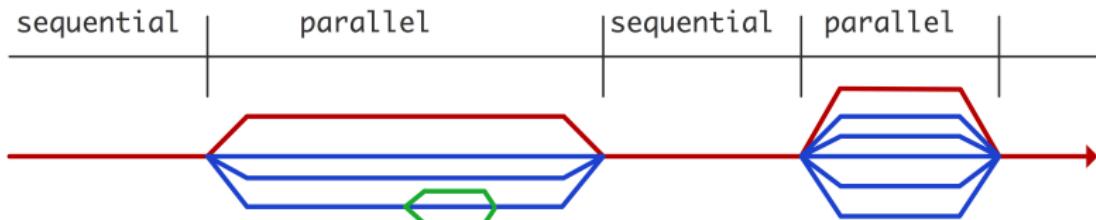
Different questions:

- Shared memory: synchronization problems such as critical sections
- Distributed memory: data motion

Thread parallelism

What is a thread

- Process: code, heap, stack
- Thread: same code but private program counter, stack, local variables
- dynamically (even recursively) created: fork-join



Incremental parallelization!

Thread context

- Private data (stack, local variables) is called ‘thread context’
- Context switch: switch from one thread execution to another
- context switches are expensive; alternative hyperthreading
- Intel Xeon Phi: hardware support for 4 threads per core
- GPUs: fast context switching between many threads

Thread programming 1

Pthreads

```
pthread_t threads[NTHREADS];
printf("forking\n");
for (i=0; i<NTHREADS; i++)
    if (pthread_create(threads+i, NULL, &adder, NULL) !=0)
        return i+1;
printf("joining\n");
for (i=0; i<NTHREADS; i++)
    if (pthread_join(threads[i], NULL) !=0)
        return NTHREADS+i+1;
```

Atomic operations

process 1: $I = I + 2$

process 2: $I = I + 3$

scenario 1.	scenario 2.	scenario 3.
$I = 0$		
read $I = 0$ do $I = 2$ write $I = 2$ write $I = 3$	read $I = 0$ do $I = 2$ write $I = 2$	read $I = 0$ do $I = 3$ write $I = 3$ read $I = 2$ do $I = 5$ write $I = 5$
$I = 3$	$I = 2$	$I = 5$

Dealing with atomic operations

Semaphores, locks, mutexes, critical sections, transactional memory

Software / hardware

Cilk

Sequential code:

```
int fib(int n){  
    if (n<2) return 1;  
    else {  
        int rst=0;  
        rst += fib(n-1);  
        rst += fib(n-2);  
        return rst;  
    }  
}
```

Cilk code:

```
cilk int fib(int n){  
    if (n<2) return 1;  
    else {  
        int rst=0;  
        rst += spawn fib(n-1);  
        rst += spawn fib(n-2);  
        sync;  
        return rst;  
    }  
}
```

Sequential consistency: program output identical to sequential

OpenMP

- Directive based
- Parallel sections, parallel loops, tasks

Distributed memory parallelism

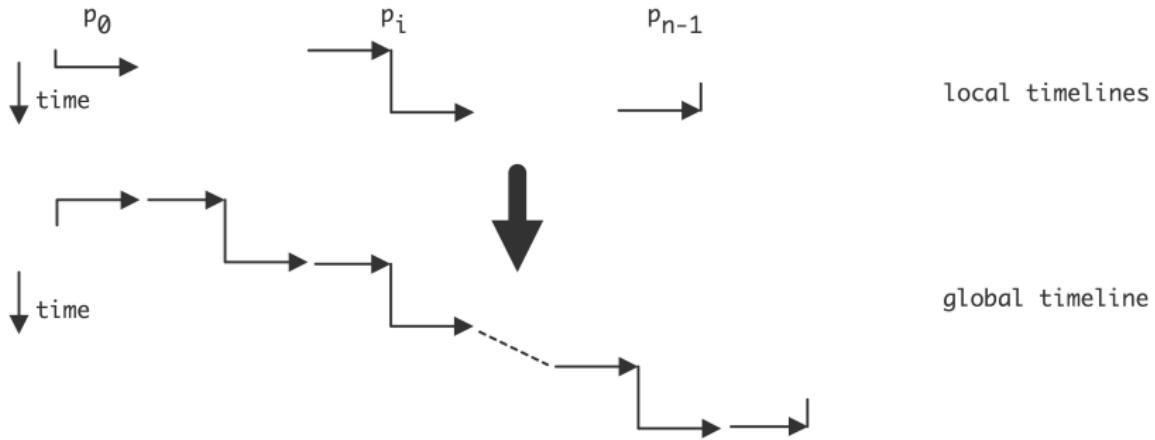
Global vs local view

$$\begin{cases} y_i \leftarrow y_i + x_{i-1} & i > 0 \\ y_i \text{ unchanged} & i = 0 \end{cases}$$

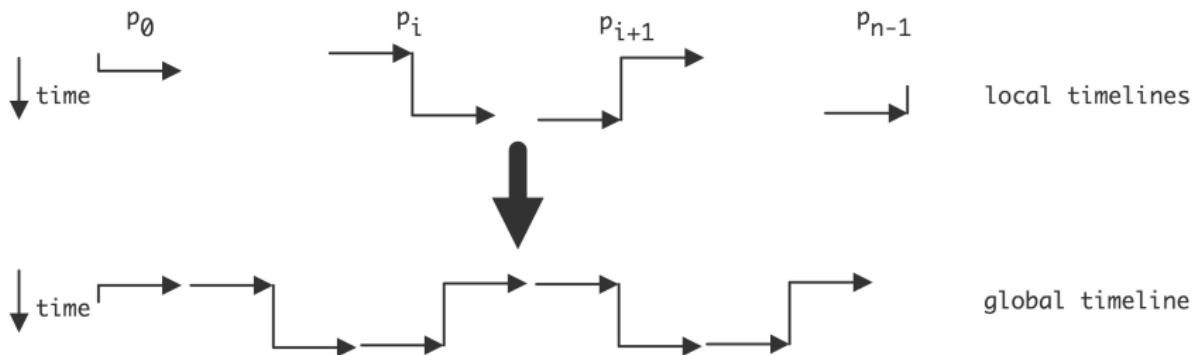
- If I am processor 0 do nothing, otherwise receive a y element from the left, add it to my x element.
- If I am the last processor do nothing, otherwise send my y element to the right.

(Let's think this through. . .)

Global picture



Careful coding



Better approaches

- Non-blocking send/receive
- One-sided

Hybrid/heterogeneous parallelism

Hybrid computing

- Use MPI between nodes, OpenMP inside nodes
- alternative: ignore shared memory and MPI throughout
- you save: buffers and copying
- bundling communication, load spread

Using threads for load balancing

Dynamic scheduling gives load balancing

Thread



Amdahl's law for hybrid programming

- p nodes with c cores each
- F_p core-parallel fraction, assume full MPI parallel
- ideal speedup pc , running time $T_1/(pc)$, actually:

$$T_{p,c} = T_1 \left(\frac{F_s}{p} + \frac{F_p}{pc} \right) = \frac{T_1}{pc} (F_s c + F_p) = \frac{T_1}{pc} (1 + F_s(c-1)).$$

- $T_1/T_{p,c} \approx p/F_s$
- Original Amdahl: $S_p < 1/F_s$, hybrid programming $S_p < p/F_s$

Design patterns

Array of Structures

```
struct { int number; double xcoord,ycoord; } _Node;
struct { double xtrans,ytrans} _Vector;
typedef struct _Node* Node;
typedef struct _Vector* Vector;

Node *nodes = (node) malloc( n_nodes*sizeof(struct _Node) );
```

Operations

Operate

```
void shift(node the_point, vector by) {  
    the_point->xcoord += by->xtrans;  
    the_point->ycoord += by->ytrans;  
}
```

in a loop

```
for (i=0; i<n_nodes; i++) {  
    shift(nodes[i], shift_vector);  
}
```

Along come the 80s

Vector operations

```
node_numbers = (int*) malloc( n_nodes*sizeof(int) );
node_xcoords = // et cetera
node_ycoords = // et cetera
```

and you would iterate

```
for (i=0; i<n_nodes; i++) {
    node_xcoords[i] += shift_vector->xtrans;
    node_ycoords[i] += shift_vector->ytrans;
}
```

and the wheel of reinvention turns further

The original design was better for MPI in the 1990s

except when vector instructions (and GPUs) came along in the 2000s

Latency hiding

- Memory and network are slow, prevent having to wait for it
- Hardware magic: out-of-order execution, caches, prefetching

Explicit latency hiding

Matrix vector product

$$\forall_{i \in I_p} : y_i = \sum_j a_{ij} x_j.$$

x needs to be gathered:

$$\forall_{i \in I_p} : y_i = \left(\sum_{j \text{ local}} + \sum_{j \text{ not local}} \right) a_{ij} x_j.$$

Overlap loads and local operations

Possible in MPI and Xeon Phi offloading,
very hard to do with caches

What's left

Parallel languages

- Co-array Fortran: extensions to the Fortran standard
- X10
- Chapel
- UPC
- BSP
- MapReduce
- Pregel, ...

UPC example

```
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main()
{
    int i;
    upc_forall(i=0; i<N; i++)
        v1plusv2[i]=v1[i]+v2[i];
}
```

Co-array Fortran example

Explicit dimension for ‘images’:

```
Real,dimension(100),codimension[*] :: X  
Real :: X(100)[*]  
Real :: X(100,200)[10,0:9,*]
```

determined by runtime environment

Grab bag of other approaches

- OS-based: data movement induced by cache misses
- Active messages: application level Remote Procedure Call
(see: Charm++)

Table of Contents

The load balancing problem

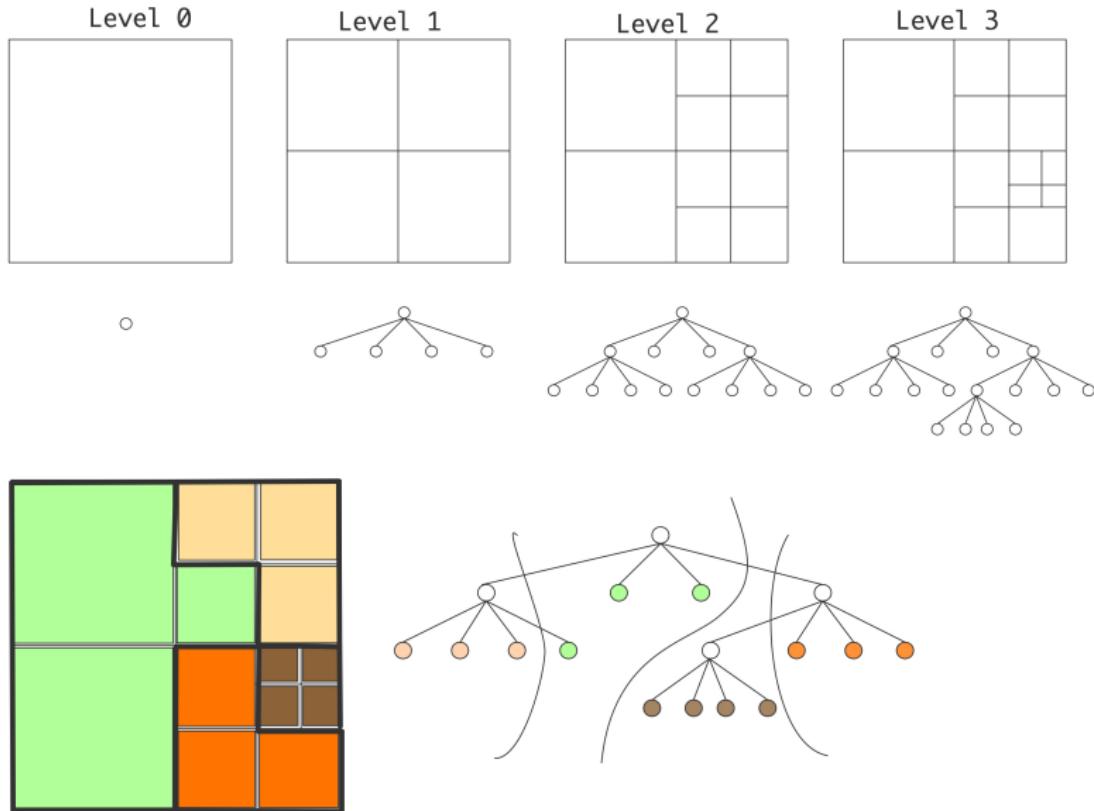
- Application load can change dynamically
 - e.g., mesh refinement, time-dependent problems
- Splitting off and merging loads
- No real software support: write application anticipating load management
- Initial balancing: graph partitioners

Load balancing and performance

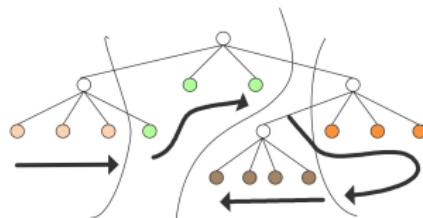
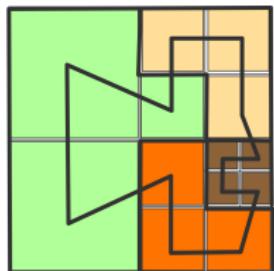
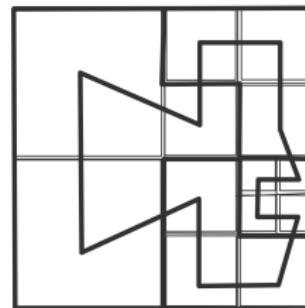
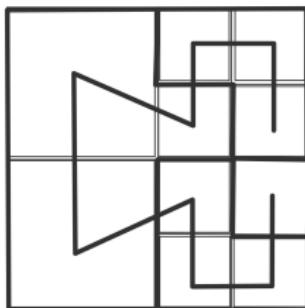
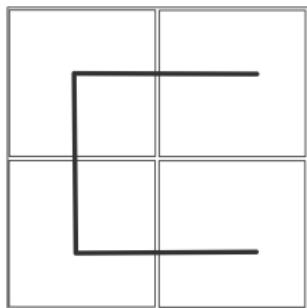
- Assignment to arbitrary processor violates locality
- Need a dynamic load assignment scheme that preserves locality under load migration
- Fairly easy for regular problems, for irregular?

Space-filling curves

Adaptive refinement and load assignment



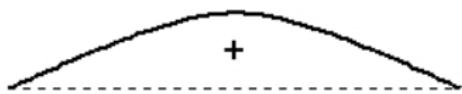
Assignment through Space-Filling Curve



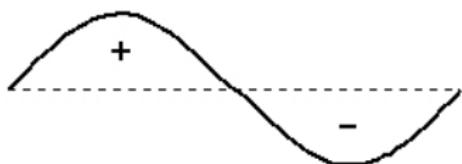
Domain partitioning by Fiedler vectors

Inspiration from physics

Modes of a Vibrating String



Lowest Frequency $\lambda(1)$



Second Frequency $\lambda(2)$

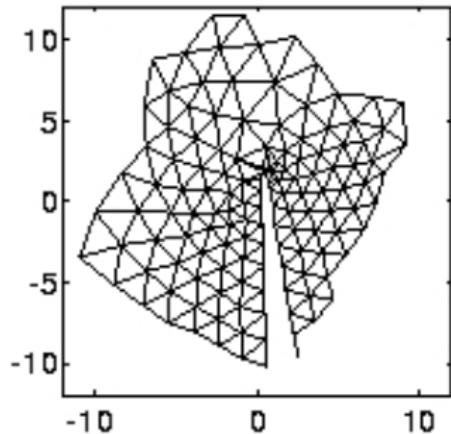


Third Frequency $\lambda(3)$

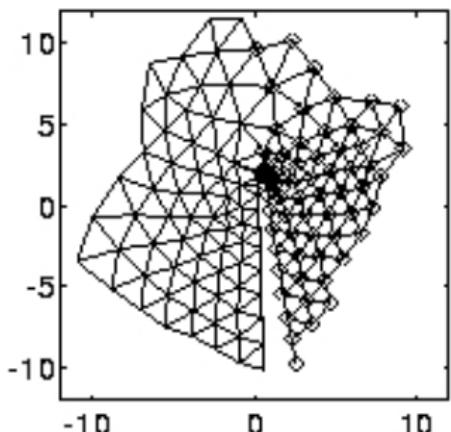
Graph laplacian

- Set $G_{ij} = -1$ if edge (i,j)
- Set G_{ii} positive to give zero rowsums
- First eigenvector is zero, positive eigenvector
- Second eigenvector has pos/neg, divides in two
- n -th eigenvector divides in n parts

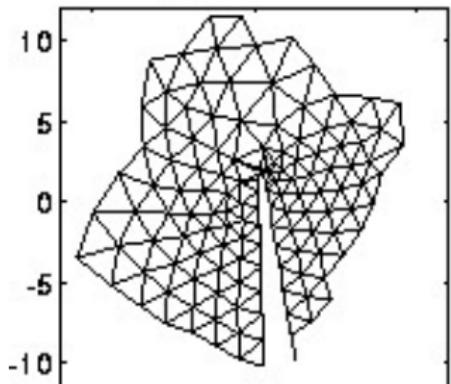
Original FE mesh



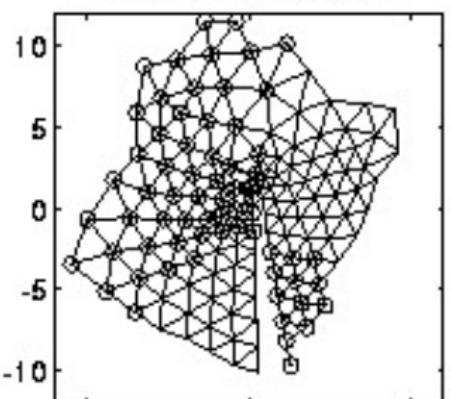
Circle node i if $v_2(i) > 0$



Original FE mesh



Circle node i if $v_4(i) > 0$



Computer arithmetic

Justification

This short session will explain the basics of floating point arithmetic, mostly focusing on round-off and its influence on computations.

Numbers in scientific computing

- Integers: $\dots, -2, -1, 0, 1, 2, \dots$
- Rational numbers: $1/3, 22/7$: not often encountered
- Real numbers $0, 1, -1.5, 2/3, \sqrt{2}, \log 10, \dots$
- Complex numbers $1 + 2i, \sqrt{3} - \sqrt{5}i, \dots$

Computers use a finite number of bits to represent numbers,
so only a finite number of numbers can be represented, and no irrational
numbers (even some rational numbers).

Table of Contents

Integers

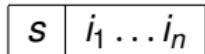
Scientific computation mostly uses real numbers. Integers are mostly used for array indexing.

16/32/64 bit: short, int, long, long long in C, size not standardized, use sizeof(long) et cetera. (Also unsigned int et cetera)

INTEGER*2/4/8 Fortran, also KIND

Negative integers

Use of sign bit: typically first bit



Simplest solution: $n > 0$, $\text{rep}(n) = 0, i_1, \dots, i_{31}$, then $\text{rep}(-n) = 1, i_1, \dots, i_{31}$

Problem: $+0$ and -0 ; also impractical in other ways.

Sign bit

bitstring	00 ··· 0	...	01 ··· 1	10 ··· 0	...	11 ··· 1
as unsigned int	0	...	$2^{31} - 1$	2^{31}	...	$2^{32} - 1$
as naive signed	0	...	$2^{31} - 1$	-0	...	$-2^{31} + 1$

Shifting

Interpret unsigned number n as $n - B$

bitstring	00 ··· 0	...	01 ··· 1	10 ··· 0	...	11 ··· 1
as unsigned int	0	...	$2^{31} - 1$	2^{31}	...	$2^{32} - 1$
as shifted int	-2^{31}	...	-1	0	...	$2^{31} - 1$

2's complement

Better solution: if $0 \leq n \leq 2^{31} - 1$,
then $\text{rep}(n) = 0, i_1, \dots, i_{31}$;
 $1 \leq n \leq 2^{31}$ then $\text{rep}(-n) = 2^{32} - n$.

bitstring	00 ··· 0	...	01 ··· 1	10 ··· 0	...	11 ··· 1
as unsigned int	0	...	$2^{31} - 1$	2^{31}	...	$2^{32} - 1$
as 2's comp. integer	0	...	$2^{31} - 1$	-2^{31}	...	-1

Subtraction in 2's complement

Subtraction $m - n$ is easy.

- Case: $m < n$. Observe that $-n$ has the bit pattern of $2^{32} - n$. Also,
 $m + (2^{32} - n) = 2^{32} - (n - m)$ where $0 < n - m < 2^{31} - 1$, so
 $2^{32} - (n - m)$ is the 2's complement bit pattern of $m - n$.
- Case: $m > n$. The bit pattern for $-n$ is $2^{32} - n$, so $m + (-n)$ as unsigned
is $m + 2^{32} - n = 2^{32} + (m - n)$. Here $m - n > 0$. The 2^{32} is an overflow
bit; ignore.

Overflow

There is a limited number of bits, so numbers that are too large in absolute value can not be represented.

Overflow.

This is not a fatal error: your program continues with the wrong result.

Exercise

Investigate what happens when you perform an integer calculation that leads to overflow. What does your compiler say if you try to write down a nonrepresentable number explicitly, for instance in a declaration or assignment statement?

Table of Contents

Floating point numbers

Analogous to scientific notation $x = 6.022 \cdot 10^{23}$:

$$x = \pm \sum_{i=0}^{t-1} d_i \beta^{-i} \beta^e$$

- sign bit
- β is the base of the number system
- $0 \leq d_i \leq \beta - 1$ the digits of the *mantissa*:
one digit before the *radix point*, so mantissa $< \beta$
- $e \in [L, U]$ exponent, stored with bias: unsigned int where $\text{fl}(L) = 0$

Examples of floating point systems

	β	t	L	U
IEEE single (32 bit)	2	24	-126	127
IEEE double (64 bit)	2	53	-1022	1023
Old Cray 64bit	2	48	-16383	16384
IBM mainframe 32 bit	16	6	-64	63
packed decimal	10	50	-999	999

BCD is tricky: 3 decimal digits in 10 bits

(we will often use $\beta = 10$ in the examples, because it's easier to read for humans, but all practical computers use $\beta = 2$)

Internal processing in 80 bit

Limitations

Overflow: more than $\beta(1 - \beta^{-t+1})\beta^U$ or less than $\beta(1 - \beta^{-t+1})\beta^L$

Underflow: numbers less than $\beta^{-t+1} \cdot \beta^L$

Exercise

For real numbers x, y , the quantity $g = \sqrt{(x^2 + y^2)/2}$ satisfies

$$g \leq \max\{|x|, |y|\}$$

so it is representable if x and y are. What can go wrong if you compute g using the above formula? Can you think of a better way?

Normalized numbers

Require first digit in the mantissa to be nonzero.

Equivalent: mantissa part $1 \leq x_m < \beta$

Unique representation for each number,
(do you see a problem?)

also: in binary this makes the first digit 1, so we don't need to store that.

With normalized numbers, underflow threshold is $1 \cdot \beta^L$;
'gradual underflow' possible, but usually not efficient.

IEEE 754

sign	exponent	mantissa
s	$e_1 \dots e_8$	$s_1 \dots s_{23}$
31	30 \dots 23	22 \dots 0

$(e_1 \dots e_8)$	numerical value
$(0 \dots 0) = 0$	$\pm 0.s_1 \dots s_{23} \times 2^{-126}$
$(0 \dots 01) = 1$	$\pm 1.s_1 \dots s_{23} \times 2^{-126}$
$(0 \dots 010) = 2$	$\pm 1.s_1 \dots s_{23} \times 2^{-125}$
...	
$(01111111) = 127$	$\pm 1.s_1 \dots s_{23} \times 2^0$
$(10000000) = 128$	$\pm 1.s_1 \dots s_{23} \times 2^1$
...	
$(11111110) = 254$	$\pm 1.s_1 \dots s_{23} \times 2^{127}$
$(11111111) = 255$	$\pm\infty$ if $s_1 \dots s_{23} = 0$, otherwise NaN

Table of Contents

Representation error

Error between number x and representation \tilde{x} :

absolute $x - \tilde{x}$ or $|x - \tilde{x}|$

relative $\frac{x - \tilde{x}}{x}$ or $\left| \frac{x - \tilde{x}}{x} \right|$

Equivalent: $\tilde{x} = x \pm \varepsilon \Leftrightarrow |x - \tilde{x}| \leq \varepsilon \Leftrightarrow \tilde{x} \in [x - \varepsilon, x + \varepsilon]$.

Also: $\tilde{x} = x(1 + \varepsilon)$ often shorthand for $\left| \frac{\tilde{x} - x}{x} \right| \leq \varepsilon$

Example

Decimal, $t = 3$ digit mantissa: let $x = 1.256$, $\tilde{x}_{\text{round}} = 1.26$, $\tilde{x}_{\text{truncate}} = 1.25$

Error in the 4th digit: $|\varepsilon| < \beta^{t-1}$ (this example had no exponent, how about if it does?)

Exercise

The number $e \approx 2.72$, the base for the natural logarithm, has various definitions. One of them is

$$e = \lim_{n \rightarrow \infty} (1 + 1/n)^n.$$

Write a single precision program that tries to compute e in this manner. Evaluate the expression for an upper bound $n = 10^k$ with $k = 1, \dots, 10$. Explain the output for large n . Comment on the behaviour of the error.

Machine precision

Any real number can be represented to a certain precision: $\tilde{x} = x(1 + \varepsilon)$ where
truncation: $\varepsilon = \beta^{-t+1}$
rounding: $\varepsilon = \frac{1}{2}\beta^{-t+1}$

This is called *machine precision*: maximum relative error.

32-bit single precision: $mp \approx 10^{-7}$

64-bit double precision: $mp \approx 10^{-16}$

Maximum attainable accuracy.

Another definition of machine precision: smallest number ε such that $1 + \varepsilon > 1$.

Exercise

Write a small program that computes the machine epsilon for both single and double precision. Does it make any difference if you set the *compiler optimization levels* low or high?

(For C++ programmers: can you write a templated program that works for single and double precision?)

Addition

- ➊ align exponents
- ➋ add mantissas
- ➌ adjust exponent to normalize

Example: $1.00 + 2.00 \times 10^{-2} = 1.00 + .02 = 1.02$. This is exact, but what happens with $1.00 + 2.55 \times 10^{-2}$?

Example: $5.00 \times 10^1 + 5.04 = (5.00 + 0.504) \times 10^1 \rightarrow 5.50 \times 10^1$

Any error comes from limiting the mantissa: if x is the true sum and \tilde{x} the computed sum, then $\tilde{x} = x(1 + \varepsilon)$ with $|\varepsilon| < 10^{-2}$

The ‘correctly rounded arithmetic’ model

Assumption (enforced by IEEE 754):

The numerical result of an operation is the rounding of the exactly computed result.

$$\text{fl}(x_1 \odot x_2) = (x_1 \odot x_2)(1 + \varepsilon)$$

where $\odot = +, -, *, /$

Note: this holds only for a single operation!

Guard digits

Correctly rounding is not trivial, especially for subtraction.

Example: $t = 2, \beta = 10$: $1.0 - 9.5 \times 10^{-1}$, exact result $0.05 = 5.0 \times 10^{-2}$.

- Simple approach: $1.0 - 9.5 \times 10^{-1} = 1.0 - 0.9 = 0.1 = 1.0 \times 10^{-1}$
- Using 'guard digit': $1.0 - 9.5 \times 10^{-1} = 1.0 - 0.95 = 0.05 = 5.0 \times 10^{-2}$,
exact.

In general 3 extra bits needed.

Fused Mul-Add instructions

$$a \leftarrow a * b + c \quad \text{or} \quad c \leftarrow a * b + c$$

- Addition plus multiplication, but not independent
- Processors can have dedicated hardware for FMA (also IEEE 754-2008)
- Internally evaluated in higher precision: 80-bit.
- Very useful for certain linear algebra (which?) Not for other operations (examples?)

Associativity

Compute $4 + 6 + 7$ in one significant digit.

Evaluation left-to-right gives:

$$\begin{aligned}(4 \cdot 10^0 + 6 \cdot 10^0) + 7 \cdot 10^0 &\Rightarrow 10 \cdot 10^0 + 7 \cdot 10^0 && \text{addition} \\ &\Rightarrow 1 \cdot 10^1 + 7 \cdot 10^0 && \text{rounding} \\ &\Rightarrow 1.0 \cdot 10^1 + 0.7 \cdot 10^1 && \text{using guard digit} \\ &\Rightarrow 1.7 \cdot 10^1 \\ &\Rightarrow 2 \cdot 10^1 && \text{rounding}\end{aligned}$$

On the other hand, evaluation right-to-left gives:

$$\begin{aligned}4 \cdot 10^0 + (6 \cdot 10^0 + 7 \cdot 10^0) &\Rightarrow 4 \cdot 10^0 + 13 \cdot 10^0 && \text{addition} \\ &\Rightarrow 4 \cdot 10^0 + 1 \cdot 10^1 && \text{rounding} \\ &\Rightarrow 0.4 \cdot 10^1 + 1.0 \cdot 10^1 && \text{using guard digit} \\ &\Rightarrow 1.4 \cdot 10^1 \\ &\Rightarrow 1 \cdot 10^1 && \text{rounding}\end{aligned}$$

Error propagation under addition

Let $s = x_1 + x_2$, and $x = \tilde{s} = \tilde{x}_1 + \tilde{x}_2$ with $\tilde{x}_i = x_i(1 + \varepsilon_i)$

$$\begin{aligned}\tilde{x} &= \tilde{s}(1 + \varepsilon_3) \\ &= x_1(1 + \varepsilon_1)(1 + \varepsilon_3) + x_2(1 + \varepsilon_2)(1 + \varepsilon_3) \\ &= x_1 + x_2 + x_1(\varepsilon_1 + \varepsilon_3) + x_2(\varepsilon_2 + \varepsilon_3) \\ \Rightarrow \tilde{x} &= s(1 + 2\varepsilon)\end{aligned}$$

⇒ errors are added

Assumptions: all ε_i approximately equal size and small;

$$x_i > 0$$

Multiplication

- ① add exponents
- ② multiply mantissas
- ③ adjust exponent

$$\text{Example: } .123 \times .567 \times 10^1 = .069741 \times 10^1 \rightarrow .69741 \times 10^0 \rightarrow .697 \times 10^0.$$

What happens with relative errors?

Table of Contents

Subtraction

Correct rounding only applies to a single operation.

Example: $1.24 - 1.23 = 0.01 \rightarrow 1. \times 10^{-2}$:

result is exact, but only one significant digit.

What if $1.24 = \text{fl}(1.244)$ and $1.23 = \text{fl}(1.225)$? Correct result 1.9×10^{-2} ; almost 100% error.

- *Cancellation* leads to loss of precision
- subsequent operations with this result are inaccurate
- this can not be fixed with guard digits and such
- ⇒ avoid subtracting numbers that are likely close.

ABC-formula

Example: $ax^2 + bx + c = 0 \rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

suppose $b > 0$ and $b^2 \gg 4ac$ then the '+' solution will be inaccurate

Better: compute $x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$ and use $x_+ \cdot x_- = -c/a$.

Serious example

Evaluate $\sum_{n=1}^{10000} \frac{1}{n^2} = 1.644834$

in 6 digits: machine precision is 10^{-6} in single precision

First term is 1, so partial sums are ≥ 1 , so $1/n^2 < 10^{-6}$ gets ignored, \Rightarrow last 7000 terms (or more) are ignored, \Rightarrow sum is 1.644725: 4 correct digits

Solution: sum in reverse order; exact result in single precision

Why? Consider ratio of two terms:

$$\frac{n^2}{(n-1)^2} = \frac{n^2}{n^2 - 2n + 1} = \frac{1}{1 - 2/n + 1/n^2} \approx 1 + \frac{2}{n}$$

with aligned exponents:

$n-1: .00\cdots 0$	$10\cdots 00$
$n: .00\cdots 0$	$10\cdots 01 \quad 0\cdots 0$

$k = \log(n/2)$ positions

The last digit in the smaller number is not lost if $n < 2/\varepsilon$

Another serious example

Previous example was due to finite representation; this example is more due to algorithm itself.

Consider $y_n = \int_0^1 \frac{x^n}{x-5} dx = \frac{1}{n} - 5y_{n-1}$ (monotonically decreasing)
 $y_0 = \ln 6 - \ln 5$.

In 3 decimal digits:

computation	correct result
$y_0 = \ln 6 - \ln 5 = .182 322 \times 10^1 \dots$	1.82
$y_1 = .900 \times 10^{-1}$.884
$y_2 = .500 \times 10^{-1}$.0580
$y_3 = .830 \times 10^{-1}$	going up? .0431
$y_4 = -.165$	negative? .0343

Reason? Define error as $\tilde{y}_n = y_n + \varepsilon_n$, then

$$\tilde{y}_n = 1/n - 5\tilde{y}_{n-1} = 1/n + 5n_{n-1} + 5\varepsilon_{n-1} = y_n + 5\varepsilon_{n-1}$$

so $\varepsilon_n \geq 5\varepsilon_{n-1}$: exponential growth.

Stability of linear system solving

Problem: solve $Ax = b$, where b inexact.

$$A(x + \Delta x) = b + \Delta b.$$

Since $Ax = b$, we get $A\Delta x = \Delta b$. From this,

$$\begin{aligned} \left\{ \begin{array}{l} Ax = b \\ \Delta x = A^{-1}\Delta b \end{array} \right\} &\Rightarrow \left\{ \begin{array}{l} \|A\|\|x\| \geq \|b\| \\ \|\Delta x\| \leq \|A^{-1}\|\|\Delta b\| \end{array} \right. \\ &\Rightarrow \frac{\|\Delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\| \frac{\|\Delta b\|}{\|b\|} \end{aligned}$$

'Condition number'. Attainable accuracy depends on matrix properties

Consequences of roundoff

Multiplication and addition are not associative:
problems for parallel computations.

Operations with “same” outcomes are not equally stable:
matrix inversion is unstable, elimination is stable

Exercise

Consider the iteration

$$x_{n+1} = f(x_n) = \begin{cases} 2x_n & \text{if } 2x_n < 1 \\ 2x_n - 1 & \text{if } 2x_n \geq 1 \end{cases}$$

Does this function have a fixed point, $x_0 \equiv f(x_0)$, or is there a cycle
 $x_1 = f(x_0)$, $x_0 \equiv x_2 = f(x_1)$ et cetera?

Now code this function and see what happens with various starting points x_0 .
Can you explain this?

Table of Contents

Complex numbers

Two real numbers: real and imaginary part.

Storage:

- Store real/imaginary adjacent: easy to pass address of one number
- Store array of real, then array of imaginary. Better for stride 1 access if only real parts are needed. Other considerations.

Other arithmetic systems

Some compilers support higher precisions.

Arbitrary precision: GMPlib

Interval arithmetic

Partial Differential Equations

Justification

Partial differential equations are an important source of large-scale engineering problems. Here we take a look at their computational aspects.

Boundary value problems

Consider in 1D

$$\begin{cases} -u''(x) = f(x, u, u') & x \in [a, b] \\ u(a) = u_a, u(b) = u_b \end{cases}$$

in 2D:

$$\begin{cases} -u_{xx}(\bar{x}) - u_{yy}(\bar{x}) = f(\bar{x}) & \bar{x} \in \Omega = [0, 1]^2 \\ u(\bar{x}) = u_0 & \bar{x} \in \delta\Omega \end{cases}$$

Approximation of 2nd order derivatives

Taylor series (write h for δx):

$$u(x+h) = u(x) + u'(x)h + u''(x)\frac{h^2}{2!} + u'''(x)\frac{h^3}{3!} + u^{(4)}(x)\frac{h^4}{4!} + u^{(5)}(x)\frac{h^5}{5!} + \dots$$

and

$$u(x-h) = u(x) - u'(x)h + u''(x)\frac{h^2}{2!} - u'''(x)\frac{h^3}{3!} + u^{(4)}(x)\frac{h^4}{4!} - u^{(5)}(x)\frac{h^5}{5!} + \dots$$

Subtract:

$$u(x+h) + u(x-h) = 2u(x) + u''(x)h^2 + u^{(4)}(x)\frac{h^4}{12} + \dots$$

so

$$u''(x) = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} - u^{(4)}(x)\frac{h^4}{12} + \dots$$

Numerical scheme:

$$-\frac{u(x+h) - 2u(x) + u(x-h)}{h^2} = f(x, u(x), u'(x))$$

(2nd order PDEs are very common!)

This leads to linear algebra

$$-u_{xx} = f \rightarrow \frac{2u(x) - u(x+h) - u(x-h)}{h^2} = f(x, u(x), u'(x))$$

Equally spaced points on $[0, 1]$: $x_k = kh$ where $h = 1/(n+1)$, then

$$-u_{k+1} + 2u_k - u_{k-1} = -1/h^2 f(x_k, u_k, u'_k) \quad \text{for } k = 1, \dots, n$$

Written as matrix equation:

$$\begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} f_1 + u_0 \\ f_2 \\ \vdots \end{pmatrix}$$

Matrix properties

- Very sparse, banded
- Symmetric (only because 2nd order problem)
- Sign pattern: positive diagonal, nonpositive off-diagonal
(true for many second order methods)
- Positive definite (just like the continuous problem)
- Constant diagonals (from constant coefficients in the DE)

Sparse matrix in 2D case

Sparse matrices so far were tridiagonal: only in 1D case.

Two-dimensional: $-u_{xx} - u_{yy} = f$ on unit square $[0, 1]^2$

Difference equation:

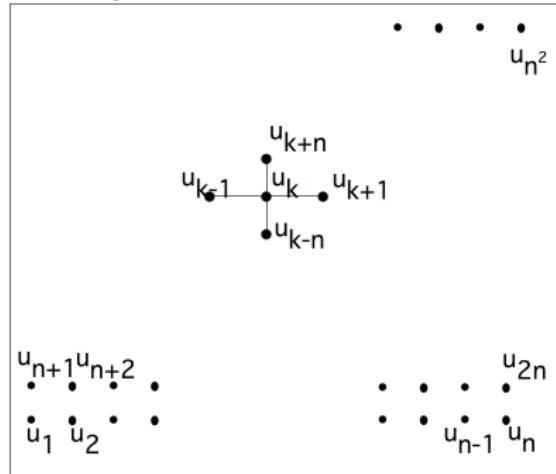
$$4u(x, y) - u(x + h, y) - u(x - h, y) - u(x, y + h) - u(x, y - h) = h^2 f(x, y)$$

$$4u_k - u_{k-1} - u_{k+1} - u_{k-n} - u_{k+n} = f_k$$

Consider a graph where $\{u_k\}_k$ are the edges
and (u_i, u_j) is an edge iff $a_{ij} \neq 0$.

The graph view of things

Poisson eq:



This is a graph!

This is the (adjacency) graph of a sparse matrix.

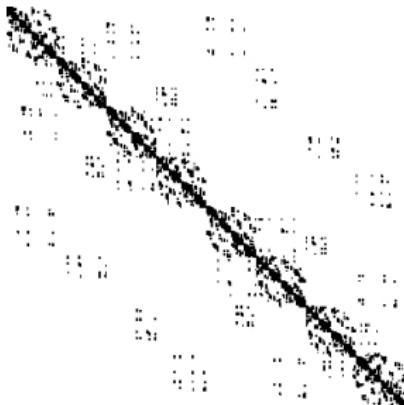
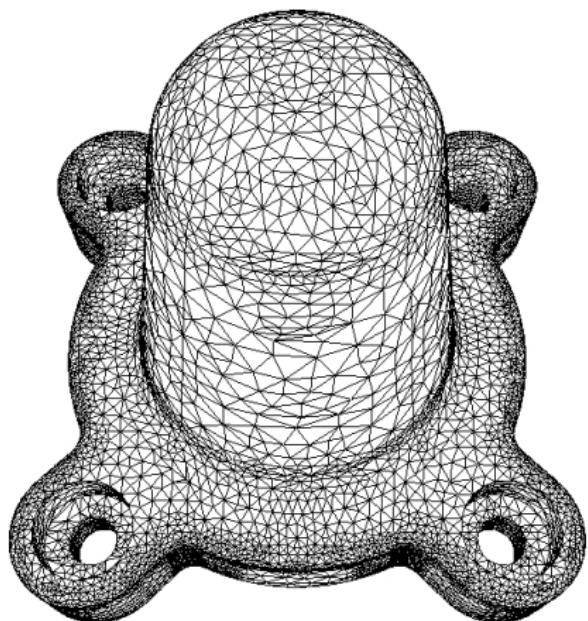
Sparse matrix from 2D equation

$$\left(\begin{array}{cccc|ccc|c} 4 & -1 & & & 0 & -1 & & & 0 \\ -1 & 4 & 1 & & & -1 & & & \\ \ddots & \ddots & \ddots & & & \ddots & & & \\ & \ddots & \ddots & -1 & & & \ddots & & \\ 0 & & -1 & 4 & 0 & & & -1 & \\ \hline -1 & & & 0 & 4 & -1 & & -1 & -1 \\ & -1 & & & -1 & 4 & -1 & & -1 \\ & & \uparrow & \ddots & & \uparrow & \uparrow & \uparrow & \uparrow \\ & k-n & & & k-1 & k & k+1 & & k+n \end{array} \right)$$

Matrix properties

- Very sparse, banded
- Factorization takes less than n^2 space, n^3 work
- Symmetric (only because 2nd order problem)
- Sign pattern: positive diagonal, nonpositive off-diagonal
(true for many second order methods)
- Positive definite (just like the continuous problem)
- Constant diagonals: only because of the constant coefficient differential equation
- Factorization: lower complexity than dense, recursion length less than N .

Realistic meshes



Linear algebra

Justification

Many algorithms are based in linear algebra, including some non-obvious ones such as graph algorithms. This session will mostly discuss aspects of solving linear systems, focusing on those that have computational ramifications.

Table of Contents

Linear algebra

- Mathematical aspects: mostly linear system solving
- Practical aspects: even simple operations are hard
 - Dense matrix-vector product: scalability aspects
 - Sparse matrix-vector: implementation

Let's start with the math...

Two approaches to linear system solving

Solve $Ax = b$

Direct methods:

- Deterministic
- Exact up to machine precision
- Expensive (in time and space)

Iterative methods:

- Only approximate
- Cheaper in space and (possibly) time
- Convergence not guaranteed

Really bad example of direct method

Cramer's rule

write $|A|$ for determinant, then

$$x_i = \frac{\left| \begin{array}{ccccccccc} a_{11} & a_{12} & \dots & a_{1i-1} & b_1 & a_{1i+1} & \dots & a_{1n} \\ a_{21} & & \dots & & b_2 & & \dots & a_{2n} \\ \vdots & & & & \vdots & & & \vdots \\ a_{n1} & & \dots & & b_n & & \dots & a_{nn} \end{array} \right|}{|A|}$$

Time complexity $O(n!)$

Gaussian elimination

Example

$$\begin{pmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 3 \end{pmatrix} x = \begin{pmatrix} 16 \\ 26 \\ -19 \end{pmatrix}$$

$$\left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 12 & -8 & 6 & 26 \\ 3 & -13 & 3 & -19 \end{array} \right] \rightarrow \left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 0 & -4 & 2 & -6 \\ 0 & -12 & 2 & -27 \end{array} \right] \rightarrow \left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 0 & -4 & 2 & -6 \\ 0 & 0 & -4 & -9 \end{array} \right]$$

Solve x_3 , then x_2 , then x_1

6, -4, -4 are the 'pivots'

Pivoting

If a pivot is zero, exchange that row and another.

(there is always a row with a nonzero pivot if the matrix is nonsingular)

best choice is the largest possible pivot

in fact, that's a good choice even if the pivot is not zero

Roundoff control

Consider

$$\begin{pmatrix} \varepsilon & 1 \\ 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1 + \varepsilon \\ 2 \end{pmatrix}$$

with solution $x = (1, 1)^t$

Ordinary elimination:

$$\begin{pmatrix} \varepsilon & 1 \\ 0 & (1 - \frac{1}{\varepsilon}) \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 - \frac{1}{\varepsilon} \end{pmatrix}$$

$$\Rightarrow x_2 = \frac{2 - \frac{1}{\varepsilon}}{1 - \frac{1}{\varepsilon}} \Rightarrow x_1 = \frac{1 - x_2}{\varepsilon}$$

Roundoff 2

If $\varepsilon < \varepsilon_{\text{mach}}$, then $2 - 1/\varepsilon = -1/\varepsilon$ and $1 - 1/\varepsilon = -1/\varepsilon$, so

$$x_2 = \frac{2 - \frac{1}{\varepsilon}}{1 - \frac{1}{\varepsilon}} = 1, \Rightarrow x_1 = \frac{1 - x_2}{\varepsilon} = 0$$

Roundoff 3

Pivot first:

$$\begin{pmatrix} 1 & 1 \\ \varepsilon & 1 \end{pmatrix} x = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 1 \\ 0 & 1 - \varepsilon \end{pmatrix} x = \begin{pmatrix} 2 \\ 1 - 2\varepsilon \end{pmatrix}$$

If ε very small:

$$x_2 = \frac{1 - 2\varepsilon}{1 - \varepsilon} = 1, \quad x_1 = 2 - x_2 = 1$$

LU factorization

Same example again:

$$A = \begin{pmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 3 \end{pmatrix}$$

2nd row minus $2 \times$ first; 3rd row minus $1/2 \times$ first;
equivalent to

$$L_1 Ax = L_1 b, \quad L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix}$$

(elementary reflector)

LU 2

Next step: $L_2 L_1 A x = L_2 L_1 b$ with

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix}$$

Define $U = L_2 L_1 A$, then $A = L_1^{-1} L_2^{-1} U$

'LU factorization'

LU 3

Observe:

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix} \quad L_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 0 & 1 \end{pmatrix}$$

Likewise

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix} \quad L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{pmatrix}$$

Even more remarkable:

$$L_1^{-1} L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 3 & 1 \end{pmatrix}$$

Can be computed in place! (pivoting?)

Solve LU system

$Ax = b \rightarrow LUx = b$ solve in two steps:

$Ly = b$, and $Ux = y$

Forward sweep:

$$\begin{pmatrix} 1 & & & \emptyset \\ \ell_{21} & 1 & & \\ \ell_{31} & \ell_{32} & 1 & \\ \vdots & & \ddots & \\ \ell_{n1} & \ell_{n2} & \cdots & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

$$y_1 = b_1, \quad y_2 = b_2 - \ell_{21}y_1, \dots$$

Solve LU 2

Backward sweep:

$$\begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ u_{22} & \dots & u_{2n} \\ \ddots & & \vdots \\ \emptyset & & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

$$x_n = u_{nn}^{-1} y_n, \quad x_{n-1} = u_{n-1,n-1}^{-1} (y_{n-1} - u_{n-1,n} x_n), \dots$$

Computational aspects

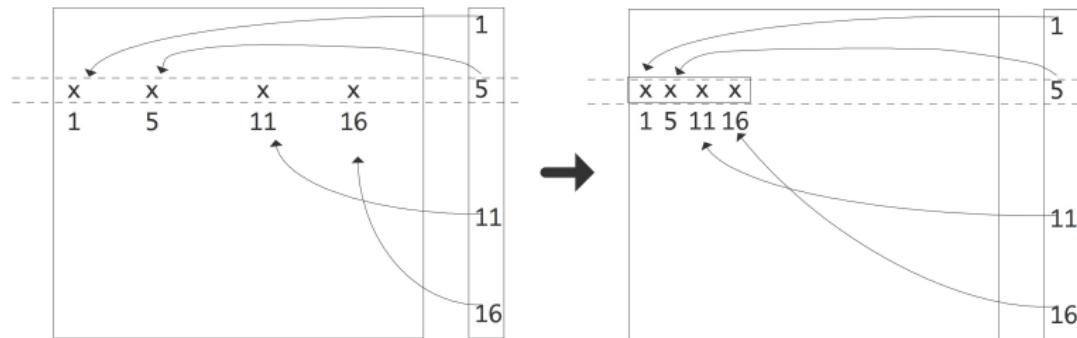
- Factoring and solving are recursive: parallelism is not trivial
- (compare matrix-matrix and matrix-vector product)
- Complexity: $O(n^3)$ operations for factorization,
 $O(n^2)$ for solution
- Much more stable than inversion, not quite as stable as QR

Table of Contents

Sparse matrix storage

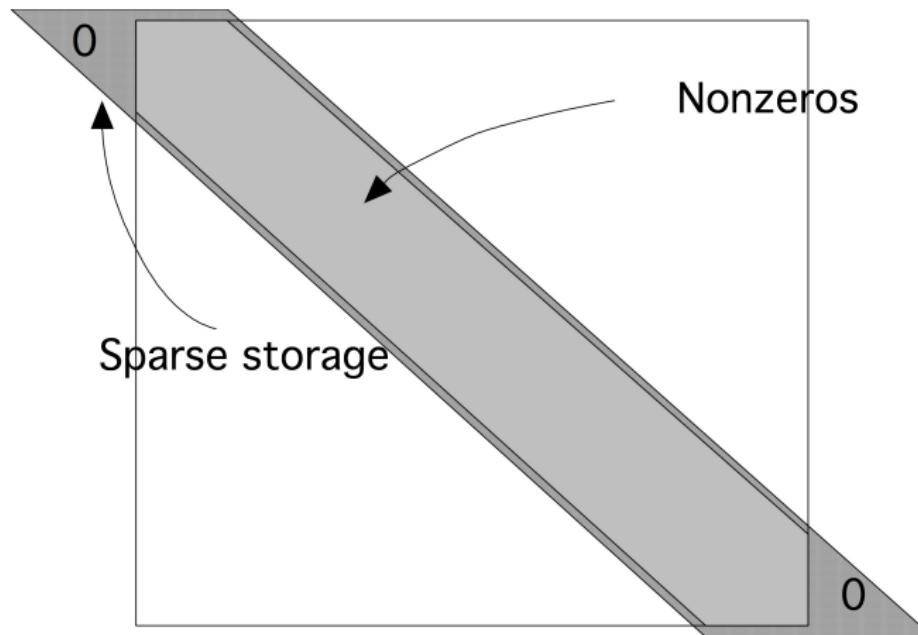
Matrix above has many zeros: n^2 elements but only $O(n)$ nonzeros. Big waste of space to store this as square array.

Matrix is called ‘sparse’ if there are enough zeros to make specialized storage feasible.



Storage by diagonals

Use the banded format:



Diagonal matrix-vector product

$$y_i \leftarrow y_i + A_{ii}x_i,$$

$$y_i \leftarrow y_i + A_{ii+1}x_{i+1} \quad \text{for } i < n,$$

$$y_i \leftarrow y_i + A_{ii-1}x_{i-1} \quad \text{for } i > 1.$$

```
for diag = -diag_left, diag_right
    for loc = max(1,1-diag), min(n,n-diag)
        y(loc) = y(loc) + val(loc,diag) * x(loc+diag)
    end
end
```

Pro/con

Pro: long vectors (D'Azevedo: $20\times$ speedup on Cray X-1)

Con: limited, little cache reuse

Variants: jagged diagonal

Compressed Row Storage

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}. \quad (2)$$

Compressed Row Storage (CRS): store all nonzeros by row, their column indices, pointers to where the columns start (1-based indexing):

val	10	-2	3	9	3	7	8	7	3	9	13	4	2	-1
col_ind	1	5	1	2	6	2	3	4	1	5	6	2	5	6
row_ptr	1	3	6	9	13	17	20	.						

Sparse matrix operations

Most common operation: matrix-vector product

```
for (row=0; row<nrows; row++) {  
    s = 0;  
    for (icol=ptr[row]; icol<ptr[row+1]; icol++) {  
        int col = ind[icol];  
        s += a[aptr] * x[col];  
        aptr++;  
    }  
    y[row] = s;  
}
```

Operations with changes to the nonzero structure are much harder!

Indirect addressing of x gives low spatial and temporal locality.

Exercise: sparse coding

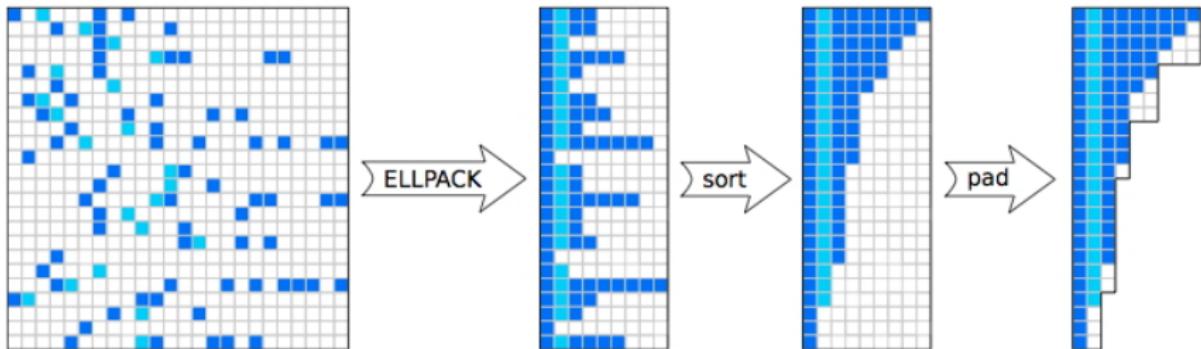
What if you need access to both rows and columns at the same time?
Implement an algorithm that tests whether a matrix stored in CRS format is
symmetric. Hint: keep an array of pointers, one for each row, that keeps track
of how far you have progressed in that row.

Jagged diagonal storage

Align irregular sparse matrices along ‘jagged’ diagonals

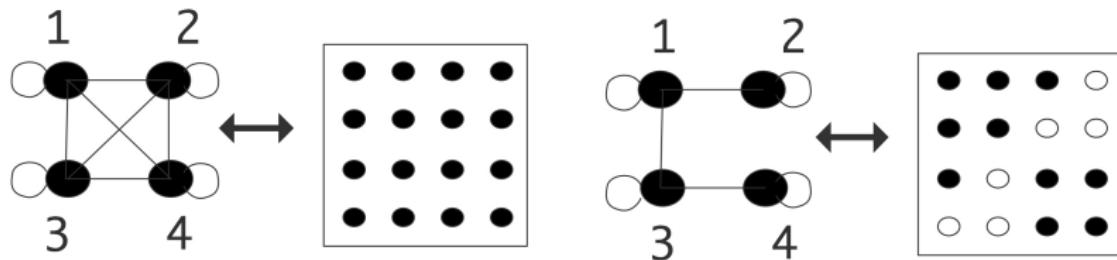
$$\begin{pmatrix} 10 & -3 & 0 & 1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{pmatrix} \rightarrow \begin{pmatrix} 10 & -3 & 1 & & & \\ 9 & 6 & -2 & & & \\ 3 & 8 & 7 & & & \\ 6 & 7 & 5 & 4 & & \\ 9 & 13 & & & & \\ 5 & -1 & & & & \end{pmatrix}$$

Long vectors make it suitable for GPU

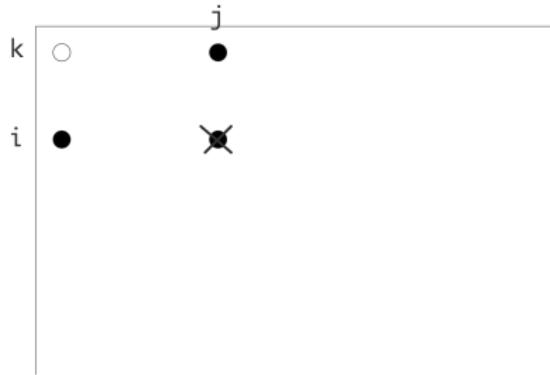


Graph theory of sparse matrices

Some things (reducibility) are easiest seen in a graph



Graph theory of sparse elimination



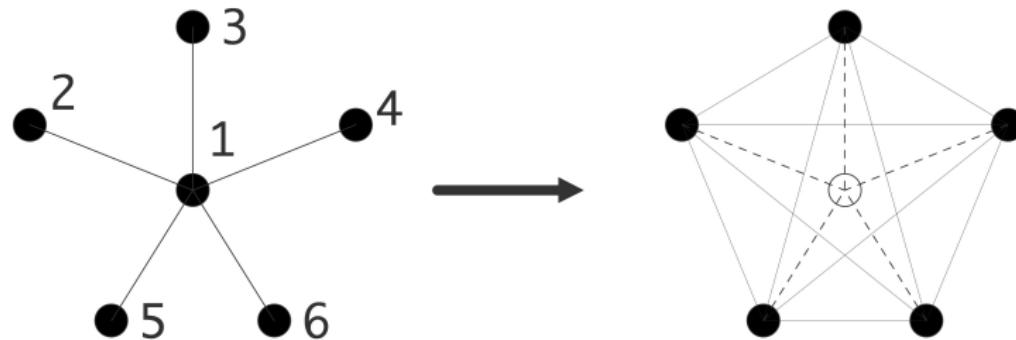
$$a_{ij} \leftarrow a_{ij} - a_{ik} a_{kk}^{-1} a_{kj}$$



Fill-in

Fill-in: index (i, j) where $a_{ij} = 0$ but $\ell_{ij} \neq 0$ or $u_{ij} \neq 0$.

Fill-in is a function of ordering



$$\begin{pmatrix} * & * & \cdots & * \\ * & * & & 0 \\ \vdots & & \ddots & \\ * & 0 & & * \end{pmatrix}$$

After factorization the matrix is dense.
Can this be permuted?

LU of a sparse matrix

$$\begin{array}{c} \left(\begin{array}{cc|cc|cc} 4 & -1 & 0 & \dots & | & -1 \\ -1 & 4 & -1 & 0 & \dots & | & 0 & -1 \\ \ddots & \ddots & \ddots & & | & & \ddots \\ \hline -1 & 0 & \dots & & | & 4 & -1 \\ 0 & -1 & 0 & \dots & | & -1 & 4 & -1 \end{array} \right) \\ \Rightarrow \left(\begin{array}{cc|cc|cc} 4 & -1 & 0 & \dots & | & -1 \\ 4 - \frac{1}{4} & -1 & 0 & \dots & | & -1/4 & -1 \\ \ddots & \ddots & \ddots & & | & & \ddots \\ \hline -1/4 & \dots & & & | & 4 - \frac{1}{4} & -1 \\ -1 & 0 & \dots & & | & -1 & 4 & -1 \end{array} \right) \end{array}$$

Exercise: LU of a band matrix

Suppose a matrix A is banded with *halfbandwidth* p :

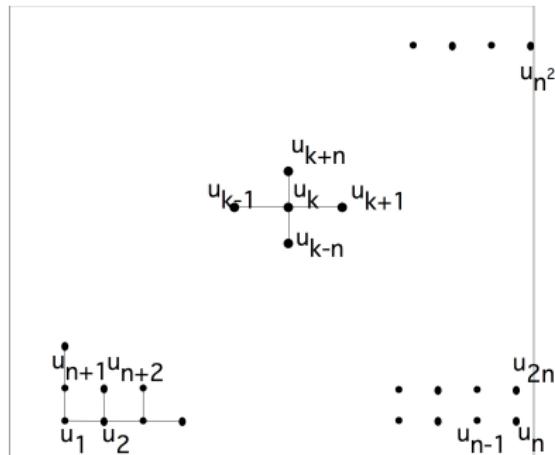
$$a_{ij} = 0 \quad \text{if } |i - j| > p$$

Derive how much space an LU factorization of A will take if no pivoting is used.
(For bonus points: consider partial pivoting.)

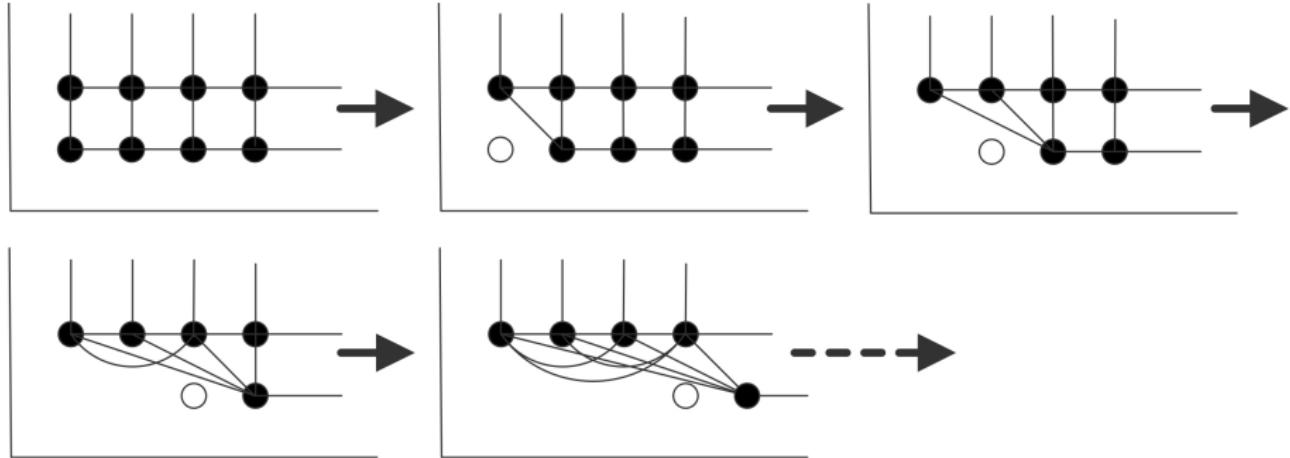
Can you also derive how much space the inverse will take? (Hint: if $A = LU$, does that give you an easy formula for the inverse?)

Domain view

Graph of connectivity of variables:



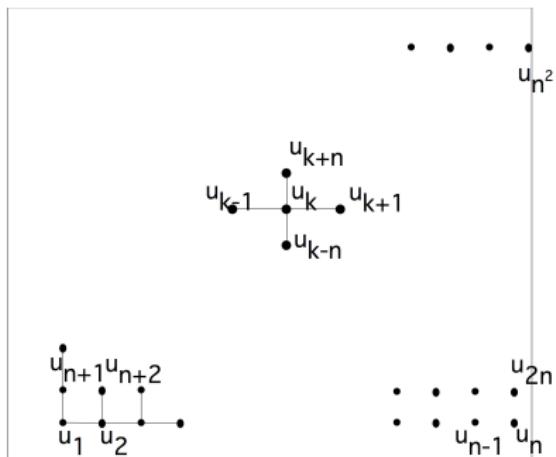
now start eliminating in sequence



Remaining matrix has a dense leading block

Fill-in during LU

Recall:



Fill-in during LU

2D BVP: Ω is $n \times n$, gives matrix of size $N = n^2$, with bandwidth n .

Matrix storage $O(N)$

LU storage $O(N^{3/2})$ (limited to band)

LU factorization work $O(N^2)$

Cute fact: storage can be computed linear in #nonzeros

Table of Contents

Two different approaches

Solve $Ax = b$

Direct methods:

- Deterministic
- Exact up to machine precision
- Expensive (in time and space)

Iterative methods:

- Only approximate
- Cheaper in space and (possibly) time
- Convergence not guaranteed

Stationary iteration

Iterative methods

Choose any x_0 and repeat

$$x^{k+1} = Bx^k + c$$

until $\|x^{k+1} - x^k\|_2 < \varepsilon$ or until $\frac{\|x^{k+1} - x^k\|_2}{\|x^k\|} < \varepsilon$

Example of iterative solution

Example system

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution $(2, 1, 1)$.

Suppose you know (physics) that solution components are roughly the same size, and observe the dominant size of the diagonal, then

$$\begin{pmatrix} 10 & & \\ & 7 & \\ & & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

might be a good approximation: solution $(2.1, 9/7, 8/6)$.

Iterative example'

Example system

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution $(2, 1, 1)$.

Also easy to solve:

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution $(2.1, 7.95/7, 5.9/6)$.

Abstract presentation

- To solve $Ax = b$; too expensive; suppose $K \approx A$ and solving $Kx = b$ is possible
- Define $Kx_0 = b$, then error correction $x_0 = x + e_0$, and $A(x_0 - e_0) = b$
- so $Ae_0 = Ax_0 - b = r_0$; this is again unsolvable, so
- $K\tilde{e}_0 = r_0$ and $x_1 = x_0 - \tilde{e}_0$.
- In one formula:

$$x_1 = x_0 - K^{-1}r_0$$

- now iterate: $e_1 = x_1 - x$, $Ae_1 = Ax_1 - b = r_1$ et cetera

Iterative scheme:

$$x_{i+1} = x_i - K^{-1}r_i \quad \text{where } r_i = Ax_i - b$$

Error analysis

- One step

$$r_1 = Ax_1 - b = A(x_0 - \tilde{e}_0) - b \quad (3)$$

$$= r_0 - AK^{-1}r_0 \quad (4)$$

$$= (I - AK^{-1})r_0 \quad (5)$$

- Inductively: $r_n = (I - AK^{-1})^n r_0$ so $r_n \downarrow 0$ if $|\lambda(I - AK^{-1})| < 1$
Geometric reduction (or amplification!)
- This is ‘stationary iteration’: no dependence on the iteration number.
Simple analysis, limited applicability.

Complexity analysis

- Direct solution is $O(N^3)$
except sparse, then $O(N^{5/2})$ or so
- Iterative per iteration cost $O(N)$ assuming sparsity.
- Number of iterations is complicated function of spectral properties:
 - Stationary iteration #it = $O(N^2)$
 - Other methods #it = $O(N)$
(2nd order only, more for higher order)
 - Multigrid and fast solvers: #it = $O(\log N)$ or even $O(1)$

Choice of K

- The closer K is to A , the faster convergence.
- Diagonal and lower triangular choice mentioned above: let

$$A = D_A + L_A + U_A$$

be a splitting into diagonal, lower triangular, upper triangular part, then

- Jacobi method: $K = D_A$ (diagonal part),
- Gauss-Seidel method: $K = D_A + L_A$ (lower triangle, including diagonal)
- SOR method: $K = \omega D_A + L_A$

Computationally

If

$$A = K - N$$

then

$$Ax = b \Rightarrow Kx = Nx + b \Rightarrow Kx_{i+1} = Nx_i + b$$

Equivalent to the above, and you don't actually need to form the residual.

Jacobi

$$K = D_A$$

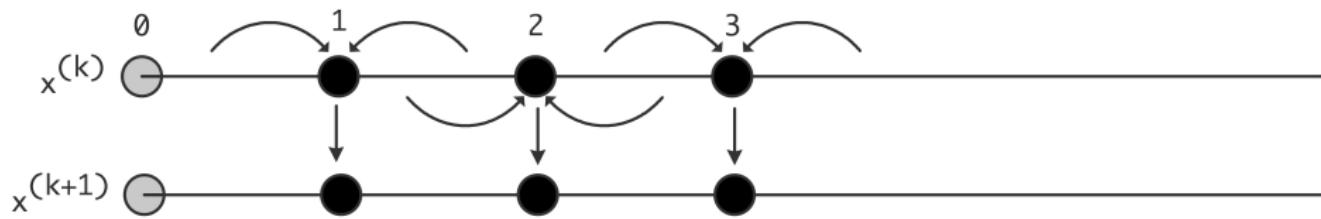
Algorithm:

```
for k = 1, ... until convergence, do:  
    for i = 1 ... n:  
        //  $a_{ii}x_i^{(k+1)} = \sum_{j \neq i} a_{ij}x_j^{(k)} + b_i \Rightarrow$   
         $x_i^{(k+1)} = a_{ii}^{-1}(\sum_{j \neq i} a_{ij}x_j^{(k)} + b_i)$ 
```

Implementation:

```
for k = 1, ... until convergence, do:  
    for i = 1 ... n:  
         $t_i = a_{ii}^{-1}(-\sum_{j \neq i} a_{ij}x_j + b_i)$   
        copy  $x \leftarrow t$ 
```

Jacobi in pictures:



Gauss-Seidel

$$K = D_A + L_A$$

Algorithm:

for $k = 1, \dots$ until convergence, do:

for $i = 1 \dots n$:

$$\begin{aligned} // a_{ii}x_i^{(k+1)} + \sum_{j < i} a_{ij}x_j^{(k+1)}) &= \sum_{j > i} a_{ij}x_j^{(k)} + b_i \Rightarrow \\ x_i^{(k+1)} &= a_{ii}^{-1}(-\sum_{j < i} a_{ij}x_j^{(k+1)}) - \sum_{j > i} a_{ij}x_j^{(k)} + b_i \end{aligned}$$

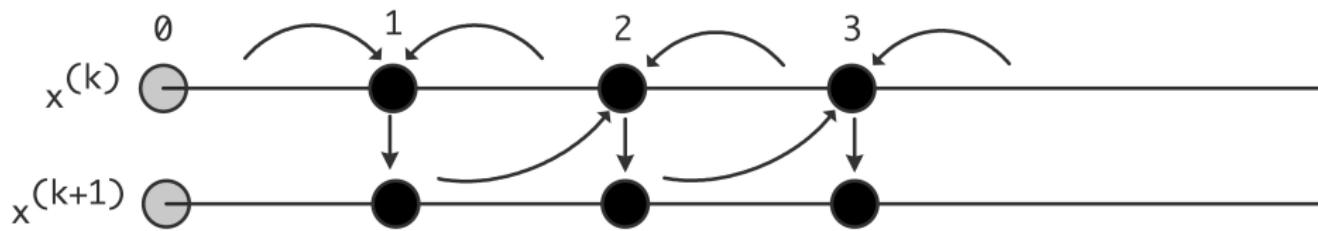
Implementation:

for $k = 1, \dots$ until convergence, do:

for $i = 1 \dots n$:

$$x_i = a_{ii}^{-1}(-\sum_{j \neq i} a_{ij}x_j + b_i)$$

GS in pictures:



Choice of K through incomplete LU

- Inspiration from direct methods: let $K = LU \approx A$

Gauss elimination:

```
for k,i,j:  
    a[i,j] = a[i,j] - a[i,k] * a[k,j] / a[k,k]
```

Incomplete variant:

```
for k,i,j:  
    if a[i,j] not zero:  
        a[i,j] = a[i,j] - a[i,k] * a[k,j] / a[k,k]
```

\Rightarrow sparsity of $L + U$ the same as of A

Applicability

Incomplete factorizations mostly work for M-matrices:
2nd order FDM and FEM

Can be severe headache for higher order

Stopping tests

When to stop converging? Can size of the error be guaranteed?

- Direct tests on error $e_n = x - x_n$ impossible; two choices
- Relative change in the computed solution small:

$$\|x_{n+1} - x_n\| / \|x_n\| < \varepsilon$$

- Residual small enough:

$$\|r_n\| = \|Ax_n - b\| < \varepsilon$$

Without proof: both imply that the error is less than some other ε' .

Polynomial iterative methods

General form of iterative methods 1.

System $Ax = b$ has the same solution as $K^{-1}Ax = K^{-1}b$.

Let \tilde{x} be a guess and define

$$r = Ax - b, \quad \tilde{r} = K^{-1}r = K^{-1}A\tilde{x} - K^{-1}b$$

then

$$b = A\tilde{x} - K\tilde{r}$$

and

$$x = A^{-1}b = \tilde{x} - A^{-1}K\tilde{r} = \tilde{x} - (K^{-1}A)^{-1}\tilde{r} = \tilde{x} - K^{-1}(AK^{-1})^{-1}r.$$

A little linear algebra

Cayley-Hamilton theorem:

$$A \text{ nonsingular} \Rightarrow \exists_{\phi}: \phi(A) = 0.$$

Write

$$\phi(x) = 1 + x\pi(x),$$

Apply this to $K^{-1}A$:

$$0 = \phi(K^{-1}A) = I + K^{-1}A\pi(K^{-1}A) \Rightarrow (K^{-1}A)^{-1} = -\pi(K^{-1}A)$$

so (with previous slide):

$$x_{\text{true}} = x_{\text{initial}} + K^{-1}\pi(AK^{-1})r_{\text{initial}}$$

Polynomial iteration

The above result

$$x_{\text{true}} = x_{\text{initial}} + K^{-1}\pi(AK^{-1})r_{\text{initial}}$$

inspires us to:

$$x_{i+1} = x_0 + K^{-1}\pi^{(i)}(AK^{-1})r_0$$

Sequence of polynomials of increasing degree

Residuals

$$x_{i+1} = x_0 + K^{-1}\pi^{(i)}(AK^{-1})r_0$$

Multiply by A and subtract b :

$$r_{i+1} = r_0 + \tilde{\pi}^{(i)}(AK^{-1})r_0$$

So:

$$r_i = \hat{\pi}^{(i)}(AK^{-1})r_0$$

where $\hat{\pi}^{(i)}$ is a polynomial of degree i with $\hat{\pi}^{(i)}(0) = 1$.

⇒ convergence theory

General form of iterative methods 3.

$$x_{i+1} = x_0 + \sum_{j \leq i} K^{-1} r_j \alpha_{ji}.$$

or equivalently:

$$x_{i+1} = x_i + \sum_{j \leq i} K^{-1} r_j \alpha_{ji}.$$

General form of iterative methods 4.

$$r_{i+1}\gamma_{i+1,i} = AK^{-1}r_i + \sum_{j \leq i} r_j\gamma_{ji}$$

and $\gamma_{i+1,i} = \sum_{j \leq i} \gamma_{ji}$.

Write this as $AK^{-1}R = RH$ where

$$H = \begin{pmatrix} -\gamma_{11} & -\gamma_{12} & \dots & & \\ \gamma_{21} & -\gamma_{22} & -\gamma_{23} & \dots & \\ 0 & \gamma_{32} & -\gamma_{33} & -\gamma_{34} & \\ \emptyset & \ddots & \ddots & \ddots & \ddots \end{pmatrix}$$

H is a Hessenberg matrix, and note zero column sums.

Divide A out:

$$x_{i+1}\gamma_{i+1,i} = K^{-1}r_i + \sum_{j \leq i} x_j\gamma_{ji}$$

General form of iterative methods 5.

$$\begin{cases} r_i = Ax_i - b \\ x_{i+1}\gamma_{i+1,i} = K^{-1}r_i + \sum_{j \leq i} x_j\gamma_{ji} \\ r_{i+1}\gamma_{i+1,i} = AK^{-1}r_i + \sum_{j \leq i} r_j\gamma_{ji} \end{cases} \quad \text{where } \gamma_{i+1,i} = \sum_{j \leq i} \gamma_{ji}.$$

Orthogonality

Idea one:

If you can make all your residuals orthogonal to each other, and the matrix is of dimension n , then after n iterations you have to have converged: it is not possible to have an $n+1$ -st residual that is orthogonal and nonzero.

Idea two:

The sequence of residuals spans a series of subspaces of increasing dimension, and by orthogonalizing the initial residual is projected on these spaces. This means that the errors will have decreasing sizes.

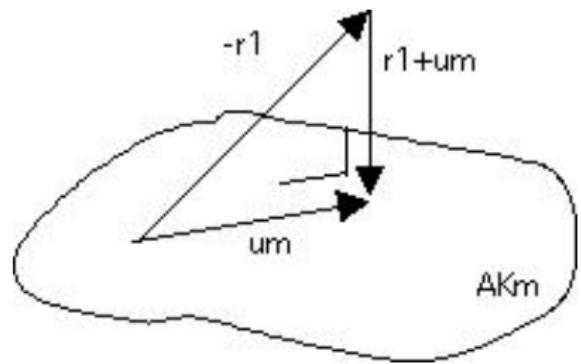
Minimization

Related concepts:

- Positive definite operator

$$\forall_x: x^t A x > 0$$

- Inner product
- Projection
- Minimization



Full Orthogonalization Method

Let r_0 be given

For $i \geq 0$:

let $s \leftarrow K^{-1}r_i$

let $t \leftarrow AK^{-1}r_i$

for $j \leq i$:

let γ_j be the coefficient so that $t - \gamma_j r_j \perp r_j$

for $j \leq i$:

form $s \leftarrow s - \gamma_j x_j$

and $t \leftarrow t - \gamma_j r_j$

let $x_{i+1} = (\sum_j \gamma_j)^{-1}s$, $r_{i+1} = (\sum_j \gamma_j)^{-1}t$.

Coupled recurrences form

$$x_{i+1} = x_i - \sum_{j \leq i} \alpha_{ji} K^{-1} r_j \quad (6)$$

This equation is often split as

- Update iterate with search direction: direction:

$$x_{i+1} = x_i - \delta_i p_i,$$

- Construct search direction from residuals:

$$p_i = K^{-1} r_i + \sum_{j < i} \beta_{ij} K^{-1} r_j.$$

Inductively:

$$p_i = K^{-1} r_i + \sum_{j < i} \gamma_{ij} p_j,$$

Conjugate Gradients

Basic idea:

$$r_i^t K^{-1} r_j = 0 \quad \text{if } i \neq j.$$

Split recurrences:

$$\begin{cases} x_{i+1} = x_i - \delta_i p_i \\ r_{i+1} = r_i - \delta_i A p_i \\ p_i = K^{-1} r_i + \sum_{j < i} \gamma_{ij} p_j, \end{cases}$$

Residuals and search directions

Symmetric Positive Definite case

Three term recurrence is enough:

$$\begin{cases} x_{i+1} = x_i - \delta_i p_i \\ r_{i+1} = r_i - \delta_i A p_i \\ p_{i+1} = K^{-1} r_{i+1} + \gamma_i p_i \end{cases}$$

Preconditioned Conjugate Gradients

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$

for $i = 1, 2, \dots$

solve $Mz^{(i-1)} = r^{(i-1)}$

$\rho_{i-1} = r^{(i-1)T} z^{(i-1)}$

if $i = 1$

$p^{(1)} = z^{(0)}$

else

$\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$

$p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$

endif

$q^{(i)} = Ap^{(i)}$

$\alpha_i = \rho_{i-1}/p^{(i)T} q^{(i)}$

$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$

$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$

 check convergence; continue if necessary

end

Three approaches to iterative methods

- Conjugate gradients: constant storage and inner products; works only for symmetric systems
- GMRES (like FOM): growing storage and inner products: restarting and numerical cleverness
- BiCGstab and QMR: relax the orthogonality

CG derived from minimization

Special case of SPD:

For which vector x with $\|x\| = 1$ is $f(x) = 1/2x^t Ax - b^t x$ minimal? (7)

Taking derivative:

$$f'(x) = Ax - b.$$

Update

$$x_{i+1} = x_i + p_i \delta_i$$

optimal value:

$$\delta_i = \underset{\delta}{\operatorname{argmin}} \|f(x_i + p_i \delta)\| = \frac{r_i^t p_i}{p_i^t A p_i}$$

Other constants follow from orthogonality.

High performance linear algebra

Justification

Bringing architecture-awareness to linear algebra, we discuss how high performance results from using the right formulation and implementation of algorithms.

Table of Contents

Simple model of parallel computation

- α : message latency
- β : time per word (inverse of bandwidth)
- γ : time per floating point operation

Send n items and do m operations:

$$cost = \alpha + \beta \cdot n + \gamma \cdot m$$

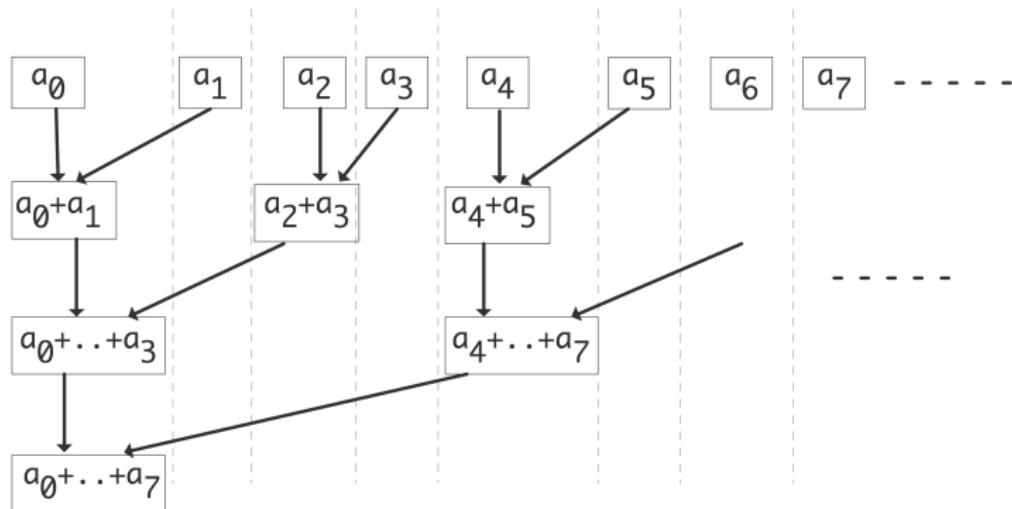
Pure sends: no γ term,

pure computation: no α, β terms,

sometimes mixed: reduction

Model for collectives

- One simultaneous send and receive:
- doubling of active processors
- collectives have a $\alpha \log_2 p$ cost component



Broadcast

	$t = 0$	$t = 1$	$t = 2$
p_0	$x_0 \downarrow, x_1 \downarrow, x_2 \downarrow, x_3 \downarrow$	$x_0 \downarrow, x_1 \downarrow, x_2 \downarrow, x_3 \downarrow$	x_0, x_1, x_2, x_3
p_1		$x_0 \downarrow, x_1 \downarrow, x_2 \downarrow, x_3 \downarrow$	x_0, x_1, x_2, x_3
p_2			x_0, x_1, x_2, x_3
p_3			x_0, x_1, x_2, x_3

On $t = 0$, p_0 sends to p_1 ; on $t = 1$ p_0, p_1 send to p_2, p_3 .

Optimal complexity:

$$\lceil \log_2 p \rceil \alpha + n\beta.$$

Actual complexity:

$$\lceil \log_2 p \rceil (\alpha + n\beta).$$

Good enough for short vectors.

Reduce

Optimal complexity:

$$\lceil \log_2 p \rceil \alpha + n\beta + \frac{p-1}{p} \gamma n.$$

Spanning tree algorithm:

	$t = 1$	$t = 2$	$t = 3$
p_0	$x_0^{(0)}, x_1^{(0)}, x_2^{(0)}, x_3^{(0)}$	$x_0^{(0:1)}, x_1^{(0:1)}, x_2^{(0:1)}, x_3^{(0:1)}$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_1	$x_0^{(1)} \uparrow, x_1^{(1)} \uparrow, x_2^{(1)} \uparrow, x_3^{(1)} \uparrow$		
p_2	$x_0^{(2)}, x_1^{(2)}, x_2^{(2)}, x_3^{(2)}$	$x_0^{(2:3)} \uparrow, x_1^{(2:3)} \uparrow, x_2^{(2:3)} \uparrow, x_3^{(2:3)} \uparrow$	
p_3	$x_0^{(3)} \uparrow, x_1^{(3)} \uparrow, x_2^{(3)} \uparrow, x_3^{(3)} \uparrow$		

Running time

$$\lceil \log_2 p \rceil (\alpha + n\beta + \frac{p-1}{p} \gamma n).$$

Good enough for short vectors.

Long vector broadcast

Combine scatter and bucket-allgather:

	$t = 0$	$t = 1$	<i>etcetera</i>
p_0	$x_0 \downarrow$	x_0	$x_3 \downarrow$ x_0, x_2, x_3
p_1	$x_1 \downarrow$	$x_0 \downarrow, x_1$	x_0, x_1, x_3
p_2	$x_2 \downarrow$	$x_1 \downarrow, x_2$	x_0, x_1, x_2
p_3	$x_3 \downarrow$	$x_2 \downarrow, x_3$	x_1, x_2, x_3

Complexity becomes

$$p\alpha + \beta n(p - 1)/p$$

better if n large

Allgather

Gather n elements: each processor owns n/p ;
optimal running time

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n \beta.$$

	$t = 1$	$t = 2$	$t = 3$
p_0	$x_0 \downarrow$	$x_0 x_1 \downarrow$	$x_0 x_1 x_2 x_3$
p_1	$x_1 \uparrow$	$x_0 x_1 \downarrow$	$x_0 x_1 x_2 x_3$
p_2	$x_2 \downarrow$	$x_2 x_3 \uparrow$	$x_0 x_1 x_2 x_3$
p_3	$x_3 \uparrow$	$x_2 x_3 \uparrow$	$x_0 x_1 x_2 x_3$

Same time as gather, half of gather-and-broadcast.

Reduce-scatter

	$t = 1$	$t = 2$	$t = 3$
p_0	$x_0^{(0)}, x_1^{(0)}, x_2^{(0)} \downarrow, x_3^{(0)} \downarrow$	$x_0^{(0:2:2)}, x_1^{(0:2:2)} \downarrow$	$x_0^{(0:3)}$
p_1	$x_0^{(1)}, x_1^{(1)}, x_2^{(1)} \downarrow, x_3^{(1)} \downarrow$	$x_0^{(1:3:2)} \uparrow, x_1^{(1:3:2)}$	$x_1^{(0:3)}$
p_2	$x_0^{(2)} \uparrow, x_1^{(2)} \uparrow, x_2^{(2)}, x_3^{(2)}$	$x_2^{(0:2:2)}, x_3^{(0:2:2)} \downarrow$	$x_2^{(0:3)}$
p_3	$x_0^{(3)} \uparrow, x_1^{(3)} \uparrow, x_2^{(3)}, x_3^{(3)}$	$x_0^{(1:3:2)} \uparrow, x_1^{(1:3:2)}$	$x_3^{(0:3)}$

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n(\beta + \gamma).$$

Table of Contents

Parallel matrix-vector product; general

- Assume a division by block rows
- Every processor p has a set of row indices I_p

Mvp on processor p :

$$\forall_i: y_i = \sum_j a_{ij}x_j$$

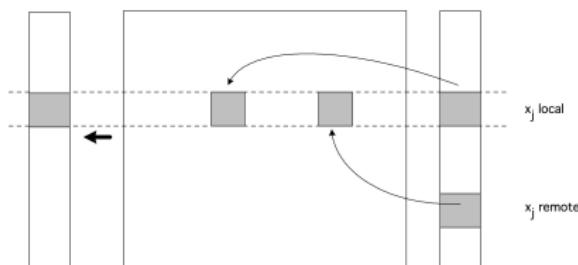
$$\forall_i: y_i = \sum_q \sum_{j \in I_q} a_{ij}x_j$$

Local and remote operations

Local and remote parts:

$$\forall i: y_i = \sum_{j \in I_p} a_{ij}x_j + \sum_{q \neq p} \sum_{j \in I_q} a_{ij}x_j$$

Local part I_p can be executed right away, I_q requires communication.



Combine:
communication and computation;
only used in the sparse case

Note possible overlap

Exercise

How much can overlap help you?

Dense MVP

- Separate communication and computation:
- first allgather
- then matrix-vector product

Cost computation 1.

Algorithm:

Step	Cost (lower bound)
Allgather x_i so that x is available on all nodes	
Locally compute $y_i = A_i x$	$\approx 2 \frac{n^2}{P} \gamma$

Allgather

Assume that data arrives over a binary tree:

- latency $\alpha \log_2 P$
- transmission time, receiving n/P elements from $P - 1$ processors

Algorithm with cost:

Step	Cost (lower bound)
Allgather x_i so that x is available on all nodes	$\lceil \log_2(P) \rceil \alpha + \frac{P-1}{P} n\beta \approx \log_2(P)\alpha + n\beta$
Locally compute $y_i = A_i x$	$\approx 2 \frac{n^2}{P} \gamma$

Parallel efficiency

$$E_p^{\text{1D-row}}(n) = \frac{s_p^{\text{1D-row}}(n)}{p} = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}}.$$

Strong scaling, weak scaling?

Optimistic scaling

Processors fixed, problem grows:

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}}.$$

Roughly $E_p \sim 1 - n^{-1}$

Strong scaling

Problem fixed, $p \rightarrow \infty$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}}.$$

Strong scaling

Problem fixed, $p \rightarrow \infty$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}}.$$

Roughly $E_p \sim p^{-1}$

Weak scaling

Memory fixed:

$$M = n^2/p$$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2\sqrt{M}} \frac{\beta}{\gamma}}$$

Weak scaling

Memory fixed:

$$M = n^2/p$$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2\sqrt{M}} \frac{\beta}{\gamma}}$$

Does not scale: $E_p \sim 1/\sqrt{p}$

problem in β term: too much communication

Two-dimensional partitioning

x_0		x_3		x_6		x_9	
a_{00}	a_{01}	a_{02}	y_0	a_{03}	a_{04}	a_{05}	
a_{10}	a_{11}	a_{12}		a_{13}	a_{14}	a_{15}	y_1
a_{20}	a_{21}	a_{22}		a_{23}	a_{24}	a_{25}	
a_{30}	a_{31}	a_{32}		a_{33}	a_{34}	a_{35}	
x_1		x_4		x_7		x_{10}	
a_{40}	a_{41}	a_{42}	y_4	a_{43}	a_{44}	a_{45}	
a_{50}	a_{51}	a_{52}		a_{53}	a_{54}	a_{55}	y_5
a_{60}	a_{61}	a_{62}		a_{63}	a_{64}	a_{65}	
a_{70}	a_{71}	a_{72}		a_{73}	a_{74}	a_{75}	
x_2		x_5		x_8		x_{11}	
a_{80}	a_{81}	a_{82}	y_8	a_{83}	a_{84}	a_{85}	
a_{90}	a_{91}	a_{92}		a_{93}	a_{94}	a_{95}	y_9
$a_{10,0}$	$a_{10,1}$	$a_{10,2}$		$a_{10,3}$	$a_{10,4}$	$a_{10,5}$	
$a_{11,0}$	$a_{11,1}$	$a_{11,2}$		$a_{11,3}$	$a_{11,4}$	$a_{11,5}$	
$a_{10,6}$	$a_{10,7}$	$a_{10,8}$		$a_{11,7}$	$a_{11,7}$	$a_{11,8}$	y_{10}
$a_{11,9}$							y_{11}

Two-dimensional partitioning

x_0 a_{00} a_{01} a_{02} a_{10} a_{11} a_{12} a_{20} a_{21} a_{22} a_{30} a_{31} a_{32}	y_0 $x_1 \uparrow$ y_4	x_3 x_4 y_5	y_1 x_7 y_6	x_6 x_8 y_{10}	y_2 x_{10} y_7	x_9 x_{11} y_{11}	y_3

Key to the algorithm

- Consider block (i, j)
- it needs to multiple by the xs in column j
- it produces part of the result of row i

Algorithm

- Collecting x_j on each processor p_{ij} by an *allgather* inside the processor columns.
- Each processor p_{ij} then computes $y_{ij} = A_{ij}x_j$.
- Gathering together the pieces y_{ij} in each processor row to form y_i , distribute this over the processor row: combine to form a *reduce-scatter*.
- Setup for the next A or A^t product

Analysis 1.

Step	Cost (lower bound)
Allgather x_i 's within columns	$\lceil \log_2(r) \rceil \alpha + \frac{r-1}{p} n\beta$ $\approx \log_2(r)\alpha + \frac{n}{c}\beta$ $\approx 2\frac{n^2}{p}\gamma$
Perform local matrix-vector multiply	
Reduce-scatter y_i 's within rows	

Reduce-scatter

Time:

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n(\beta + \gamma).$$

Step	Cost (lower bound)
Allgather x_i 's within columns	$\lceil \log_2(r) \rceil \alpha + \frac{r-1}{p} n\beta$ $\approx \log_2(r)\alpha + \frac{n}{c}\beta$ $\approx 2\frac{n^2}{p}\gamma$
Perform local matrix-vector multiply	
Reduce-scatter y_i 's within rows	$\lceil \log_2(c) \rceil \alpha + \frac{c-1}{p} n\beta + \frac{c-1}{p} m\gamma$ $\approx \log_2(r)\alpha + \frac{n}{c}\beta + \frac{n}{c}\gamma$

Efficiency

Let $r = c = \sqrt{p}$, then

$$E_p^{\sqrt{p} \times \sqrt{p}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n} \frac{(2\beta + \gamma)}{\gamma}}$$

Strong scaling

Same story as before for $p \rightarrow \infty$:

$$E_p^{\sqrt{p} \times \sqrt{p}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n} \frac{(2\beta+\gamma)}{\gamma}} \sim p^{-1}$$

No strong scaling

Weak scaling

Constant memory $M = n^2/p$:

$$E_p^{\sqrt{p} \times \sqrt{p}}(n) = \frac{1}{1 + \frac{p \log_2(p) \alpha}{2n^2} \gamma + \frac{\sqrt{p} (2\beta + \gamma)}{2n} \gamma}$$

Weak scaling

Constant memory $M = n^2/p$:

$$E_p^{\sqrt{p} \times \sqrt{p}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n} \frac{(2\beta+\gamma)}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M} \frac{\alpha}{\gamma} + \frac{1}{2\sqrt{M}} \frac{(2\beta+\gamma)}{\gamma}}$$

Weak scaling

Constant memory $M = n^2/p$:

$$E_p^{\sqrt{p} \times \sqrt{p}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n} \frac{(2\beta+\gamma)}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M} \frac{\alpha}{\gamma} + \frac{1}{2\sqrt{M}} \frac{(2\beta+\gamma)}{\gamma}}$$

Weak scaling:

for $p \rightarrow \infty$ this is $\approx 1/\log_2 P$:

only slowly decreasing.

LU factorizations

- Needs a cyclic distribution
- This is very hard to program, so:
- Scalapack, 1990s product, not extendible, impossible interface
- Elemental: 2010s product, extendible, nice user interface (and it is way faster)

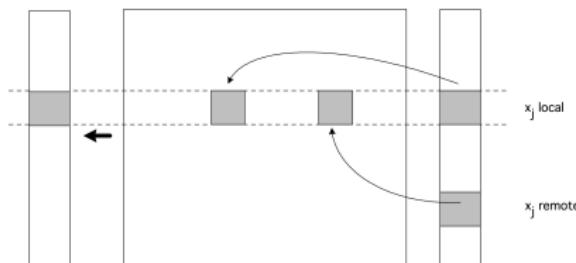
Table of Contents

Local and remote operations

Local and remote parts:

$$\forall_i: y_i = \sum_{j \in \text{local}} a_{ij} x_j + \sum_{j \in \text{remote}} a_{ij} x_j$$

Local part I_p can be executed right away, I_q requires communication.



Combine:

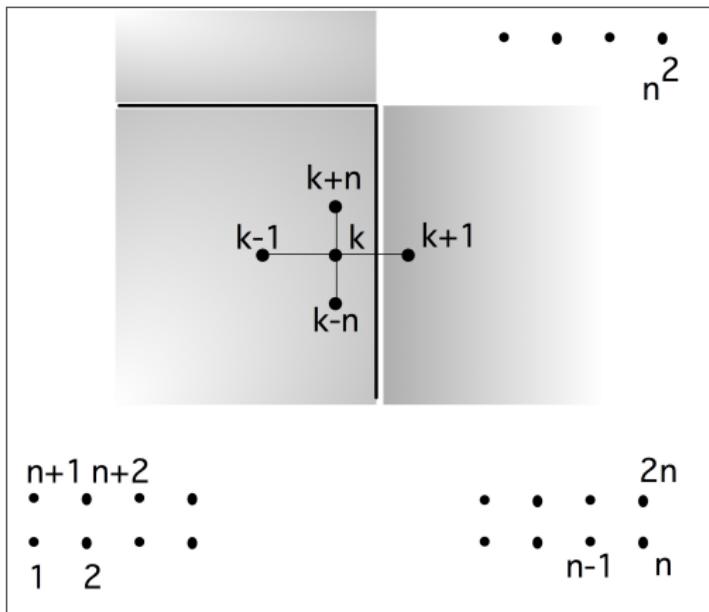
Note possible overlap communication and computation;
only used in the sparse case

Sparse matrix operations

- Traditional: PDE, discussed next
- New: graph algorithms and big data, discussed later

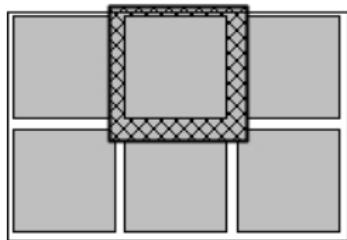
Operator view of spmv

Difference stencil



Parallel operator view

induces ghost region:



Limited number of neighbours, limited buffer space

Matrix vs operator view

- Domain partitioning: processor ‘owns’ variable i
- owns all connections from i to other j s
- \Rightarrow processor owns whole matrix row
- \Rightarrow 1D partitioning of the matrix, always

$$A = \left(\begin{array}{cccc|ccc|c} 4 & -1 & & \emptyset & -1 & & \emptyset & \\ -1 & 4 & -1 & & & -1 & & \\ \ddots & \ddots & \ddots & & & \ddots & & \\ \ddots & \ddots & \ddots & -1 & & \ddots & & \\ \emptyset & & & -1 & 4 & \emptyset & & -1 \\ \hline -1 & & & \emptyset & 4 & -1 & & -1 \\ & -1 & & & -1 & 4 & -1 & -1 \\ & \uparrow & \ddots & & \uparrow & \uparrow & \uparrow & \uparrow \\ k-n & & & & k-1 & k & k+1 & k+n \\ & & & & -1 & & -1 & \\ \hline & & & & \ddots & & \ddots & \end{array} \right) \quad (8)$$

Scaling

- Same phenomenon as with dense matrix:
- n^2 variables, memory needed is cn^2/p
- 1D partitioning *of domain* does not weakly scale
 - Message size is one line: n
 - is $\sqrt{p}\sqrt{M}$, goes up with processors
- 2D partitioning *of domain* scales weakly.
 - message size $n/\sqrt{p} = \sqrt{M}$
 - constant in M

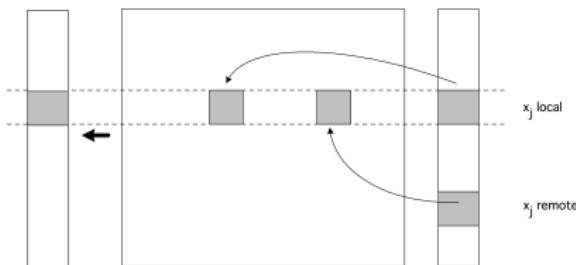
MPI implementation

- Assume general communication structure:
neighbour processors can not statically be determined
- Assume no structural symmetry
- For matrix-vector product:
each processor issues send and receive requests
- Problem: receives are easy, sends are hard
- *Inspector-executor*: one-time discovery of structure,
followed by many executions

Asymmetry in reasoning

Say

- Processor owns row i , $a_{ij} \neq 0$, processor does not own j



- Needed: message from j to i
- Processor i can discover this
- Processor j not in general

Reduce-scatter

Make $p \times p$ matrix C :

$$C_{ij} = \begin{cases} 1 & i \text{ receives from } j \\ 0 & \text{otherwise} \end{cases}$$

Then

$$s_j = \sum_i C_{ij}$$

number of messages sent by j

Reduce-scatter, proc i has C_{i*}

Reason for more cleverness

- The above is collective, implies synchronization
- temp space $O(P)$
- can we get this down to $O(\#\text{neighbours})$?
- *can we detect that we have received all requests without knowing how many to expect?*

MPI 3 non-blocking barrier

- Barrier: test that every process has reached this point blocking
- Ibarrier: non-blocking test
- Ibarrier calls does not block, yields MPI_Request pointer
- Use Wait or Test on the request

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
```

DTD algorithms

'Distributed Termination Detection'

used to be (extremely) tricky, now easy with MPI 3

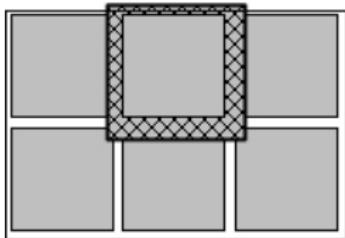
Establish sparse neighbours:

- Send all your own requests (Isend)
- Loop:
 - Test on send requests; if all done, enter non-blocking barrier
 - Probe for request messages, receive if there is something
 - If you're in the barrier, also test for the barrier to complete
- ⇒ if the barrier completes, you have received all your requests

(For safety, use MPI_Issend)

Table of Contents

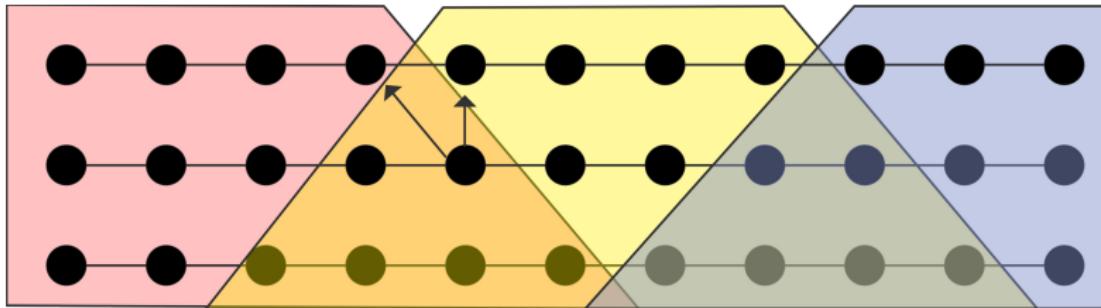
Sparse matrix vector product induces ghost region;



Is this important?

- No: surface/volume argument
- Yes: communication is much slower than computation
- Case of multiple products is considerably more interesting

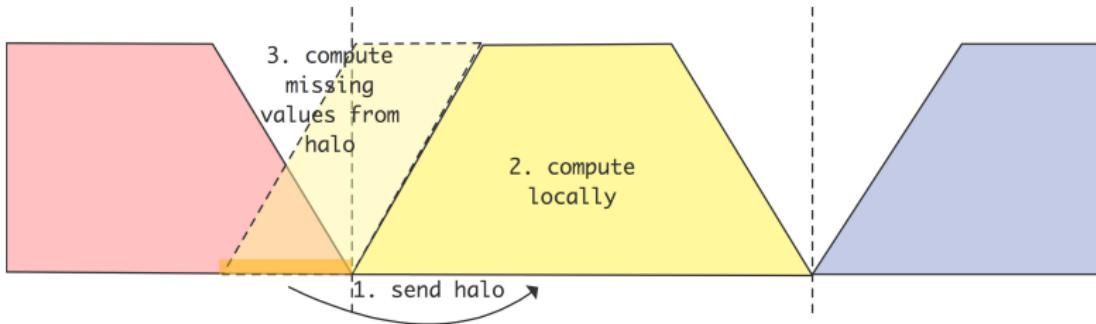
Optimization of multiple products



Recursive closure of the halo:

- Only one latency
- On-node code can be optimized (caching or cache-oblivious)

Latency hiding



Overlap halo transfer with local computation:
programming complication

Communication minimizing

Optimal solution ('communication avoiding'):

- Half the communication
- half the redundant work
- Complication: local work done in two stages

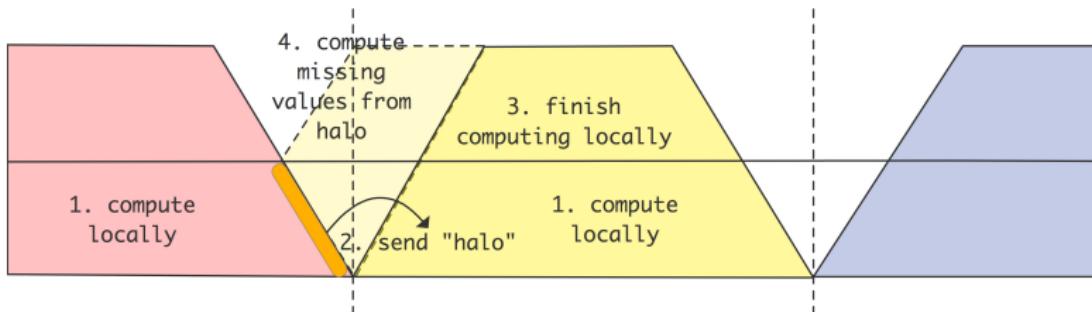


Table of Contents

What's in an iterative method?

From easy to hard

- Vector updates
These are trivial
- Inner product
- Matrix-vector product
- Preconditioner solve

Inner products

Collective operations

Collective operation: data from all processes is combined.
(Is a matrix-vector product a collective?)

Examples: sum-reduction, broadcast
These are each other's mirror image, computationally.

Naive realization of collectives

Broadcast:



Single message:

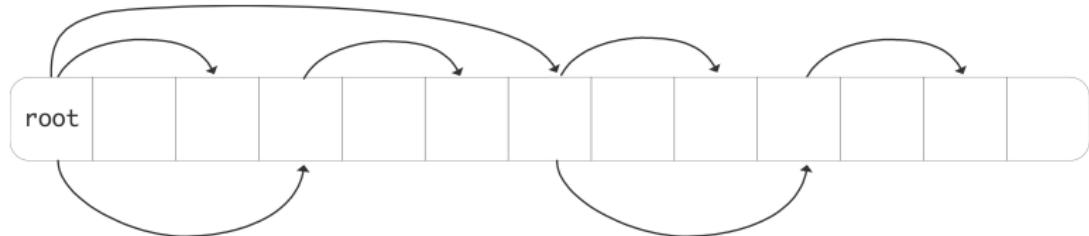
$$\alpha = \text{message startup} \approx 10^{-6} \text{ s}, \quad \beta = \text{time per word} \approx 10^{-9} \text{ s}$$

- Time for message of n words:

$$\alpha + \beta n$$

- Time for collective? Can you improve on that?

Better implementation of collective



- What is the running time now?
- Can you come up with lower bounds on the α, β terms? Are these achieved here?
- How about the case of really long buffers?

Inner products

- Only operation that intrinsically has a p dependence
- Collective, so induces synchronization
- \Rightarrow exposes load unbalance, can take lots of time
- Research in approaches to hiding: overlapping with other operations

What do those inner products serve?

- Orthogonality of residuals
- Basic algorithm: Gram-Schmidt
- one step: given u, v

$$v' \leftarrow v - \frac{u^t v}{u^t u} u.$$

then $v' \perp u$

- bunch of steps: given U, v

$$v' \leftarrow v - \frac{U^t v}{U^t U} U.$$

then $v' \perp U$.

Gram-Schmidt algorithm

Modified Gram-Schmidt

For $i = 1, \dots, n$:

$$\text{let } c_i = u_i^t v / u_i^t u_i$$

$$\text{update } v \leftarrow v - c_i u_i$$

More numerical stable

Full Orthogonalization Method

Let r_0 be given

For $i \geq 0$:

let $s \leftarrow K^{-1}r_i$

let $t \leftarrow AK^{-1}r_i$

for $j \leq i$:

let γ_j be the coefficient so that $t - \gamma_j r_j \perp r_j$

for $j \leq i$:

form $s \leftarrow s - \gamma_j x_j$

and $t \leftarrow t - \gamma_j r_j$

let $x_{i+1} = (\sum_j \gamma_j)^{-1}s$, $r_{i+1} = (\sum_j \gamma_j)^{-1}t$.

Modified Gramm-Schmidt

Let r_0 be given

For $i \geq 0$:

let $s \leftarrow K^{-1}r_i$

let $t \leftarrow AK^{-1}r_i$

for $j \leq i$:

let γ_j be the coefficient so that $t - \gamma_j r_j \perp r_j$

form $s \leftarrow s - \gamma_j x_j$

and $t \leftarrow t - \gamma_j r_j$

let $x_{i+1} = (\sum_j \gamma_j)^{-1}s$, $r_{i+1} = (\sum_j \gamma_j)^{-1}t$.

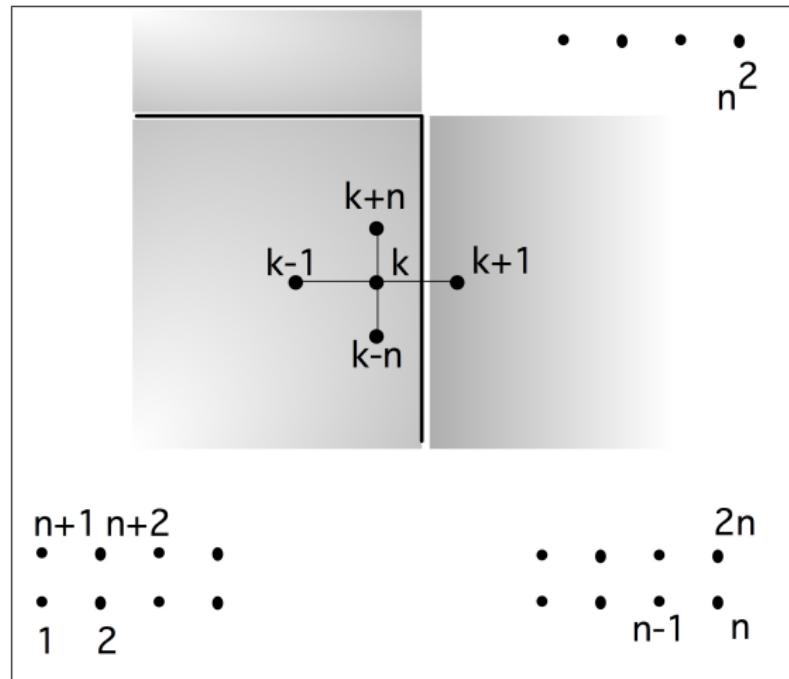
Practical differences

- Modified GS more stable
- Inner products are global operations: costly

Matrix-vector product

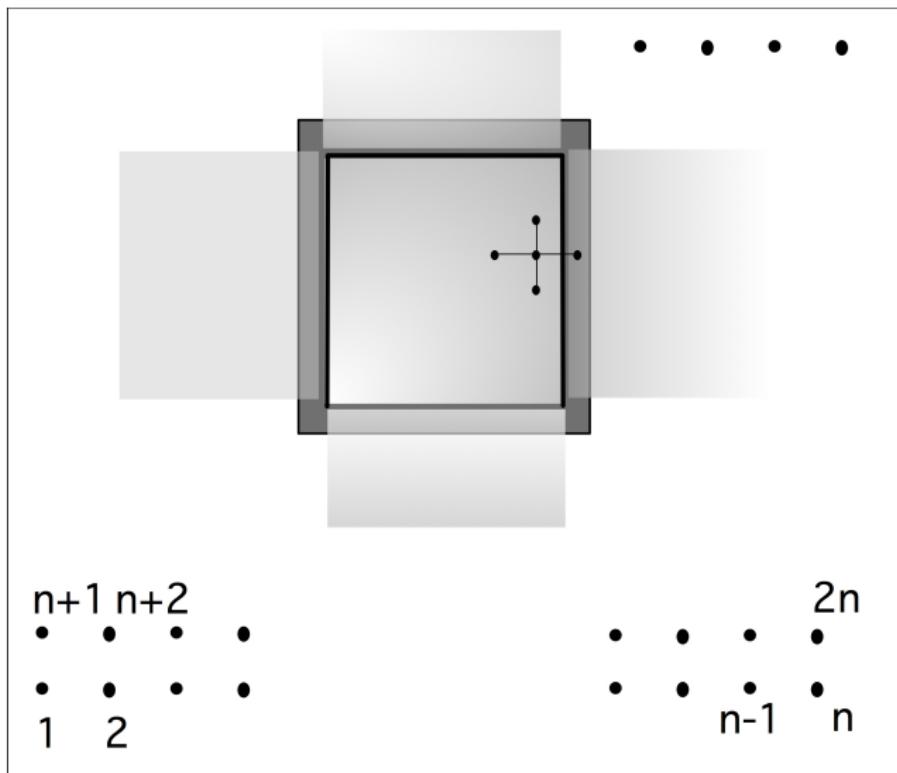
PDE, 2D case

A difference stencil applied to a two-dimensional square domain, distributed over processors. Each point connects to neighbours \Rightarrow each process connects to neighbours.



Halo region

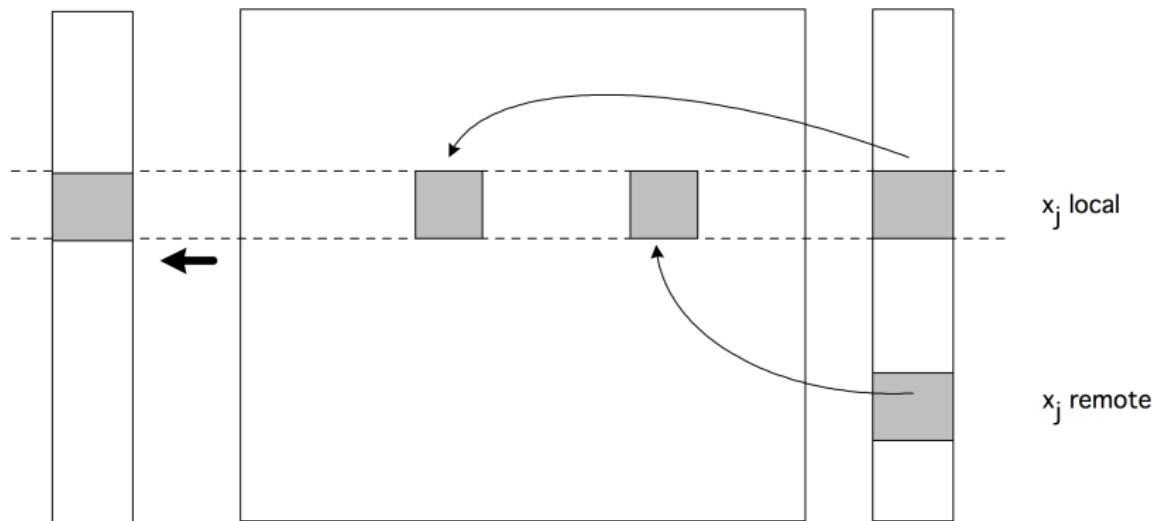
The ‘halo’ region of a process, induced by a stencil



Matrices in parallel

$$y \leftarrow Ax$$

and A, x, y all distributed:



Matrix-vector product performance

- Large scale:
 - partition for scalability
 - minimize communication (Metis, Zoltan: minimize edge cuts)
 - dynamic load balancing? requires careful design
- Processor scale:
 - Performance largely bounded by bandwidth
 - Some optimization possible

Beware of optimizations that change the math!

Preconditioners

Preconditioners

- There's much that can be said here.
- Some comments to follow
- There is intrinsic dependence in solvers, hence in preconditioners:
 - parallelism is very tricky.
 - approximate inverses

Table of Contents

Fill-in during LU

Fill-in: index (i,j) where $a_{ij} = 0$ but $\ell_{ij} \neq 0$ or $u_{ij} \neq 0$.

2D BVP: Ω is $n \times n$, gives matrix of size $N = n^2$, with bandwidth n .

Matrix storage $O(N)$

LU storage $O(N^{3/2})$

LU factorization work $O(N^2)$

Cute fact: storage can be computed linear in #nonzeros

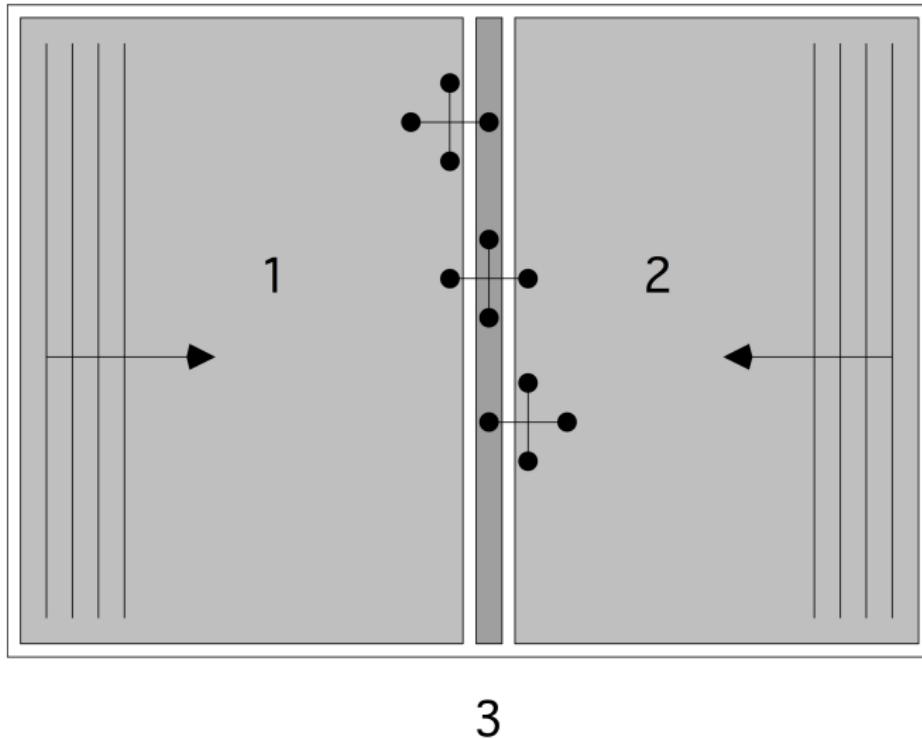
Fill-in is a function of ordering

$$\begin{pmatrix} * & * & \cdots & * \\ * & * & & \emptyset \\ \vdots & & \ddots & \\ * & \emptyset & & * \end{pmatrix}$$

After factorization the matrix is dense.

Can this be permuted?

Domain decomposition



$$\left(\begin{array}{c|ccccc|ccccc|c}
 \star & \star & & & & & & & & 0 \\
 \star & \star & \star & & & & & & & \vdots \\
 \cdot & \cdot & \cdot & \cdot & & & & & & 0 \\
 & \star & \star & \star & & & & & & \star \\
 & \star & \star & & & & & & & \\
 \hline
 & & & & \star & \star & & & 0 \\
 & & & & \star & \star & \star & & \vdots \\
 & & & \emptyset & \cdot & \cdot & \cdot & & \vdots \\
 & & & & \star & \star & \star & 0 \\
 & & & & \star & \star & & \star \\
 \hline
 0 & \dots & \dots & 0 & \star & 0 & \dots & \dots & 0 & \star & \star
 \end{array} \right) \quad \left. \begin{array}{l}
 (n^2 - n)/2 \\
 (n^2 - n)/2 \\
 n
 \end{array} \right\}$$

DD factorization

$$A^{\text{DD}} = \begin{pmatrix} A_{11} & \emptyset & A_{13} \\ \emptyset & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} =$$
$$\begin{pmatrix} I & & \\ \emptyset & I & \\ A_{31}A_{11}^{-1} & A_{32}A_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & \emptyset & A_{13} \\ & A_{22} & A_{23} \\ & & S \end{pmatrix}$$
$$S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}$$

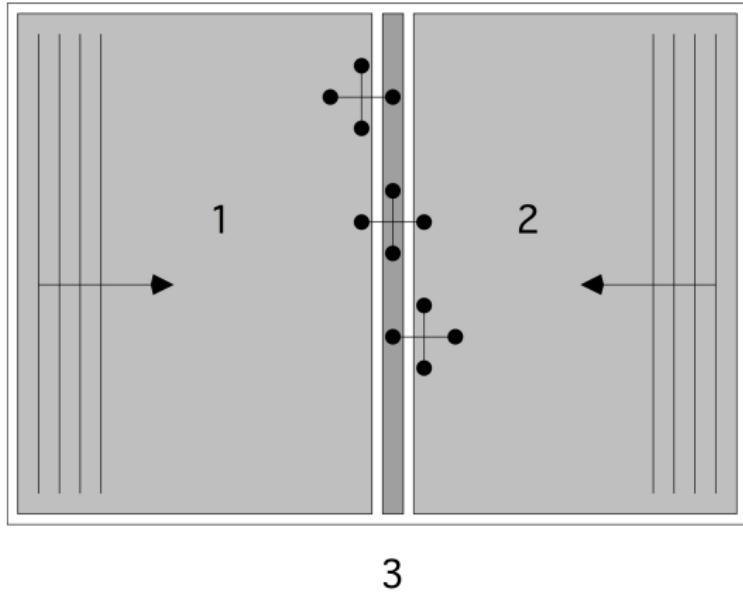
Parallelism...

Graph theory of sparse elimination

$$a_{ij} \leftarrow a_{ij} - a_{ik} a_{kk}^{-1} a_{kj}$$

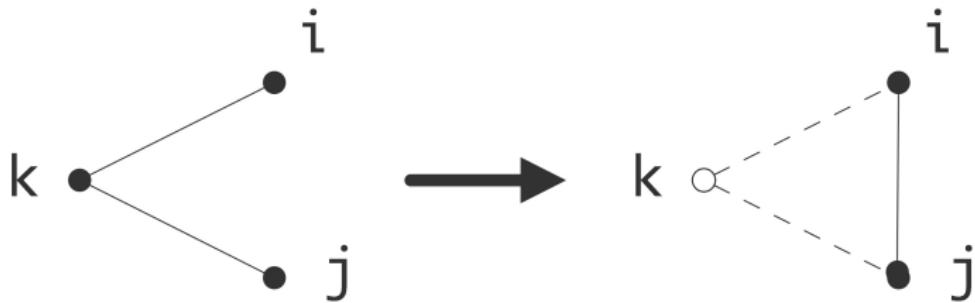


Graph theory of sparse elimination



Graph theory of sparse elimination

$$a_{ij} \leftarrow a_{ij} - a_{ik} a_{kk}^{-1} a_{kj}$$



So inductively S is dense

More about separators

- This is known as ‘domain decomposition’ or ‘substructuring’
- Separators have better spectral properties

Recursive bisection

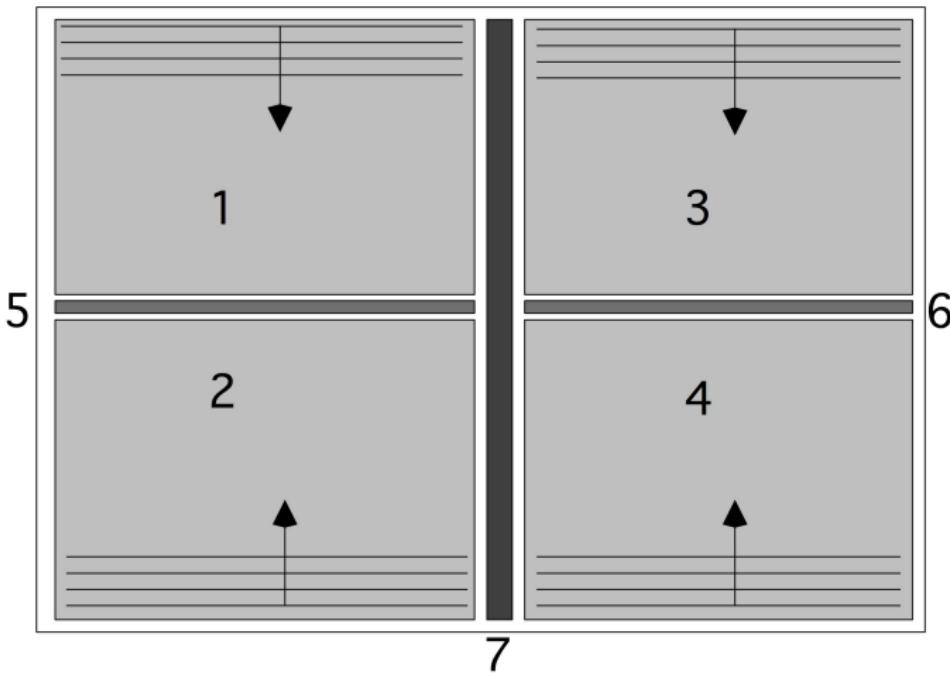
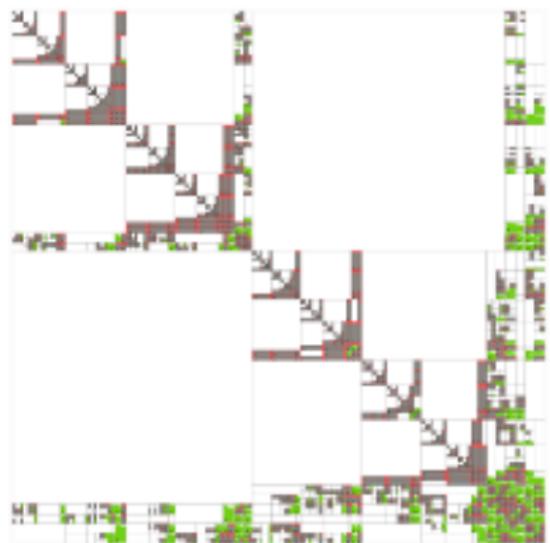


Figure: A four-way domain decomposition

$$A^{\text{DD}} = \begin{pmatrix} A_{11} & & & A_{15} & A_{17} \\ & A_{22} & & A_{25} & A_{27} \\ & & A_{33} & & A_{36} & A_{37} \\ & & & A_{44} & & A_{46} & A_{47} \\ A_{51} & A_{52} & & & A_{55} & & A_{57} \\ & & A_{63} & A_{64} & & A_{66} & A_{67} \\ A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix}$$

The domain/operator/graph view is more insightful, don't you think?

How does this look in reality?



Complexity

With $n = \sqrt{N}$:

- one dense matrix on a separator of size n , plus
- two dense matrices on separators of size $n/2$
- $\rightarrow 3/2 n^2$ space and $5/12 n^3$ time
- and then four times the above with $n \rightarrow n/2$

$$\begin{aligned}\text{space} &= 3/2n^2 + 4 \cdot 3/2(n/2)^2 + \dots \\ &= N(3/2 + 3/2 + \dots) \quad \log n \text{ terms} \\ &= O(N \log N)\end{aligned}$$

$$\begin{aligned}\text{time} &= 5/12n^3/3 + 4 \cdot 5/12(n/2)^3/3 + \dots \\ &= 5/12N^{3/2}(1 + 1/4 + 1/16 + \dots) \\ &= O(N^{3/2})\end{aligned}$$

Unfortunately only in 2D.

More direct factorizations

Minimum degree, multifrontal, . . .

Finding good separators and domain decompositions is tough in general.

Table of Contents

Sparse operations in parallel: mvp

$$\text{Mvp } y = Ax$$

```
for i=1..n  
y[i] = sum over j=1..n a[i,j]*x[j]
```

In parallel:

```
for i=myfirstrow..mylastrow  
y[i] = sum over j=1..n a[i,j]*x[j]
```

How about ILU solve?

Consider $Lx = y$

```
for i=1..n
    x[i] = (y[i] - sum over j=1..i-1 ell[i,j]*x[j])
            / a[i,i]
```

Parallel code:

```
for i=myfirstrow..mylastrow
    x[i] = (y[i] - sum over j=1..i-1 ell[i,j]*x[j])
            / a[i,i]
```

Problems?

Block method

```
for i=myfirstrow..mylastrow  
x[i] = (y[i] - sum over j=myfirstrow..i-1 ell[i,j]*x[j])  
/ a[i,i]
```

Block Jacobi with local GS solve

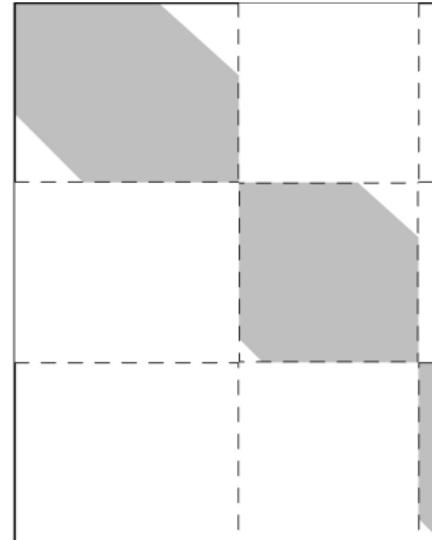
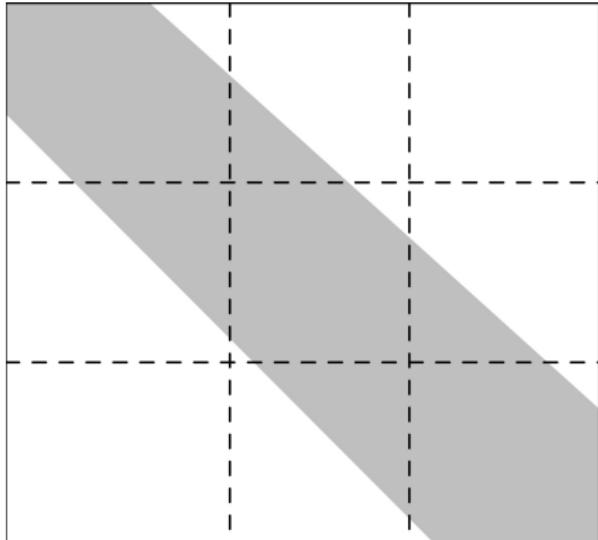


Figure: Sparsity pattern corresponding to a block Jacobi preconditioner

Variable reordering

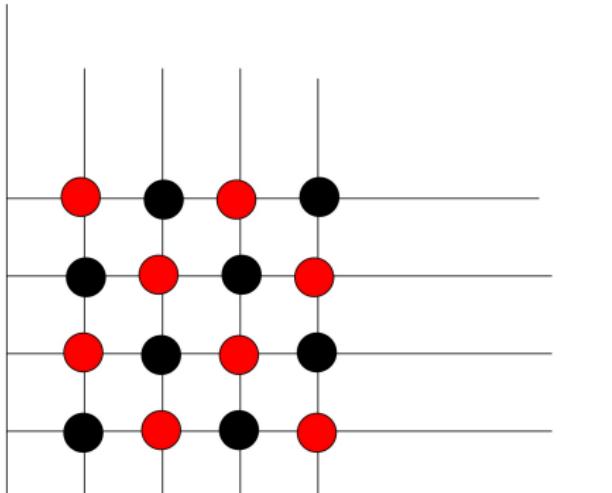
$$\begin{pmatrix} a_{11} & a_{12} & & & \emptyset \\ a_{21} & a_{22} & a_{23} & & \\ & a_{32} & a_{33} & a_{34} & \\ \emptyset & \ddots & \ddots & \ddots & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \end{pmatrix}$$

with redblack

$$\begin{pmatrix} a_{11} & & a_{12} & & \emptyset \\ & a_{33} & a_{32} & a_{34} & \\ & & \ddots & \ddots & \\ & a_{55} & & & \\ & \ddots & & & \\ & & & & \emptyset \\ a_{21} & a_{23} & & a_{22} & \\ & a_{43} & a_{45} & & \\ & \ddots & \ddots & & \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ \vdots \\ x_2 \\ x_4 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_3 \\ y_5 \\ \vdots \\ y_2 \\ y_4 \\ \vdots \end{pmatrix}$$

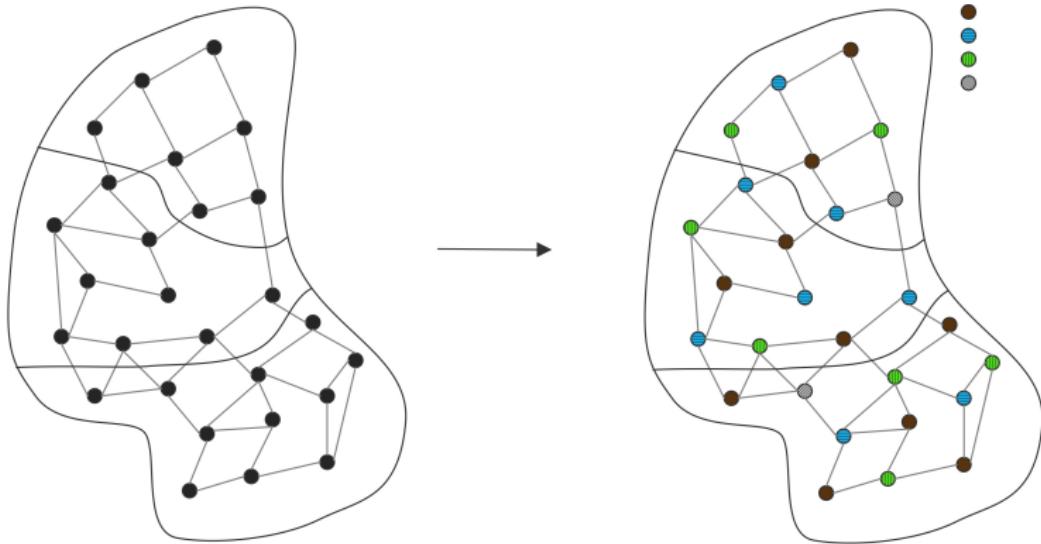
Two-processor parallel Gauss-Seidel or ILU

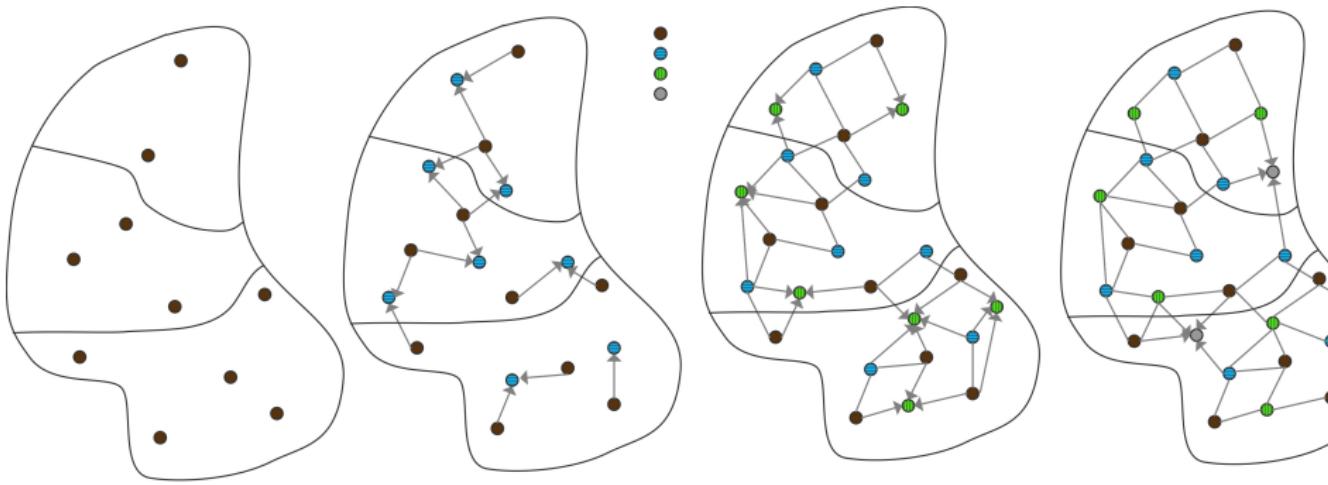
2D redblack



In general, colouring, colour number

Multicolour ILU





How do you get a multi-colouring?

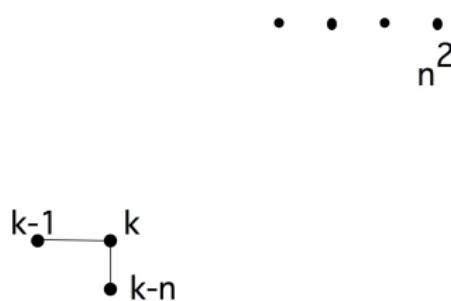
Exactly colour number is NP-completely: don't bother.

For preconditioner an approximation is good enough:
Luby / Jones-Plassman algorithm

- Give every node a random value
- First colour: all nodes with a higher value than all their neighbours
- Second colour: higher value than all neighbours except in first colour
- et cetera

Table of Contents

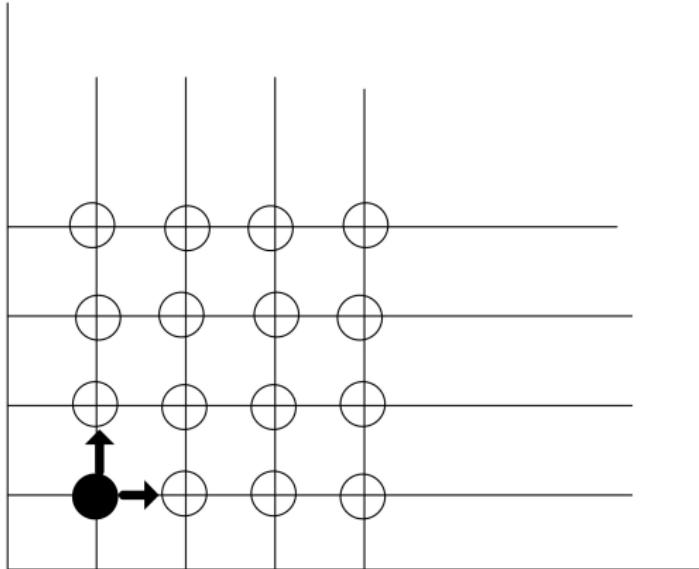
Recurrences

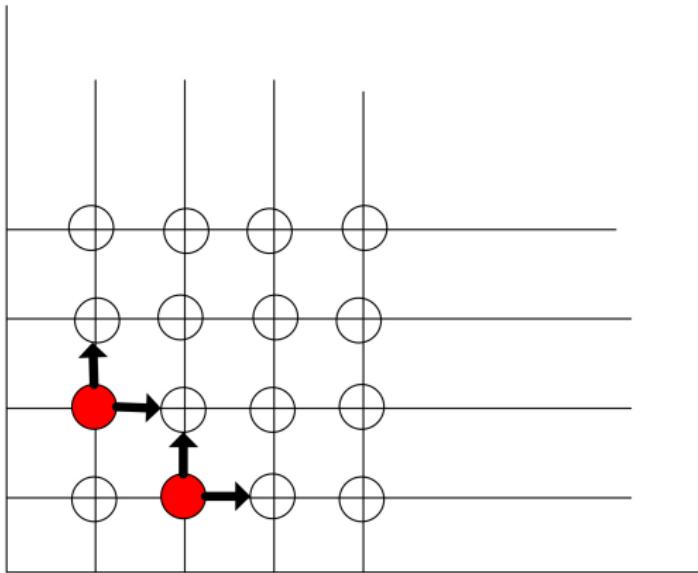


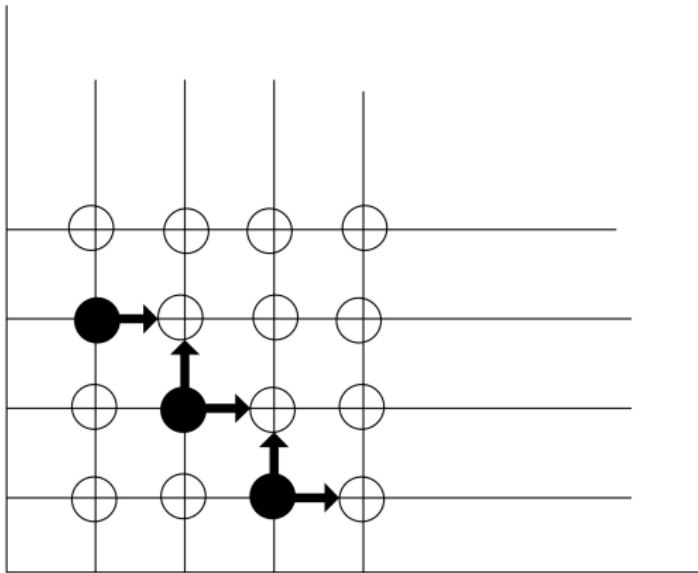
$$x_{i,j} = f(x_{i-1,j}, x_{i,j-1})$$

Intuitively: recursion length n^2

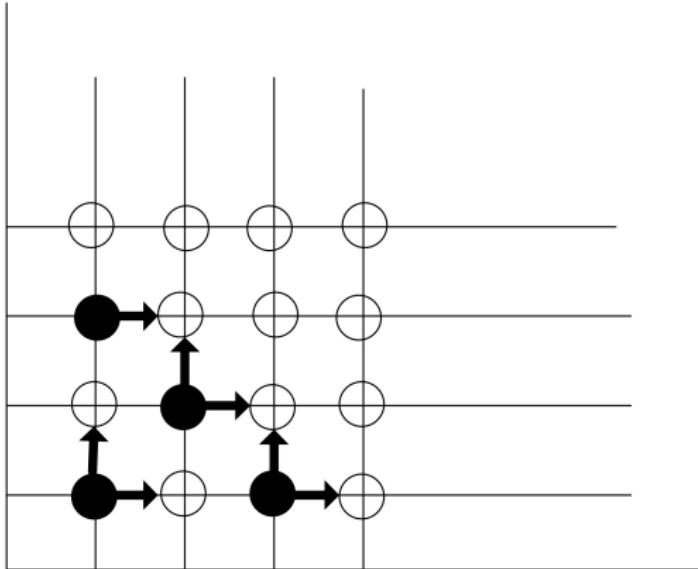
However...



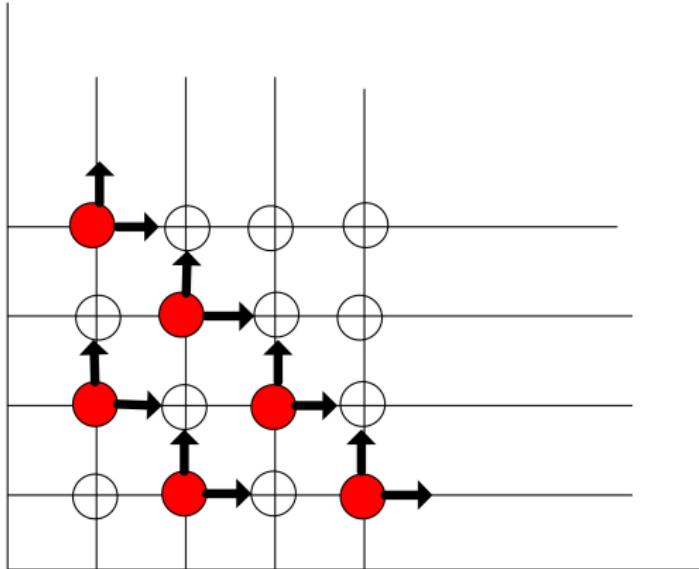




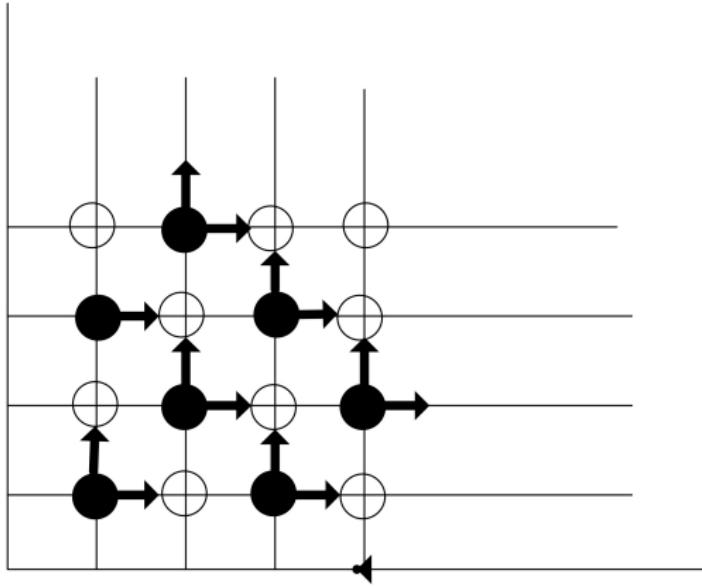
And in fact



But then too



And



Conclusion

- ① Wavefronts have sequential length $2n$,
average parallelism $n/2$
- ② Equivalency of wavefronts and multicolouring

Recursive doubling

Write recurrence $x_i = b_i - a_{i-1}x_{i-1}$ as

$$\begin{pmatrix} 1 & & & \emptyset \\ a_{21} & 1 & & \\ & \ddots & \ddots & \\ \emptyset & & a_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

for short: $A = I + B$

Transform

$$\begin{pmatrix} 1 & & & & & & & \\ 0 & 1 & & & & & & \\ & -a_{32} & 1 & & & & & \\ & 0 & & 1 & & & & \\ & & -a_{54} & 1 & & & & \\ & & 0 & & 1 & & & \\ & & & -a_{76} & & 1 & & \\ & & & & \ddots & & \ddots & \\ & & & & & \ddots & & \ddots \end{pmatrix} \times (I + B) =$$

$$\begin{pmatrix} 1 & & & & & & & \\ a_{21} & 1 & & & & & & \\ -a_{32}a_{21} & 0 & 1 & & & & & \\ & & a_{43} & 1 & & & & \\ & & -a_{54}a_{43} & 0 & 1 & & & \\ & & & a_{65} & & 1 & & \\ & & & -a_{76}a_{65} & 0 & & 1 & \\ & & & & \ddots & & \ddots & \\ & & & & & \ddots & & \ddots \end{pmatrix}$$

- Recurrence over half the elements
- Parallel calculation of other half
- Now recurse...

Turning implicit operations into explicit

Normalize ILU solve to $(I - L)$ and $(I - U)$

Approximate $(I - L)x = y$ by $x \approx (I + L + L^2)y$

Convergence guaranteed for diagonally dominant

Table of Contents

Cholesky algorithm

$$\text{Chol} \begin{pmatrix} A_{11} & A_{21}^t \\ A_{21} & A_{22} \end{pmatrix} = LL^t \quad \text{where} \quad L = \begin{pmatrix} L_{11} & 0 \\ \tilde{A}_{21} & \text{Chol}(A_{22} - \tilde{A}_{21}\tilde{A}_{21}^t) \end{pmatrix}$$

and where $\tilde{A}_{21} = A_{21}L_{11}^{-t}$, $A_{11} = L_{11}L_{11}^t$.

Implementation

for $k = 1, \text{nblocks}$:

Chol: factor $L_k L_k^t \leftarrow A_{kk}$

Trsm: solve $\tilde{A}_{>k,k} \leftarrow A_{>k,k} L_k^{-t}$

Gemm: form the product $\tilde{A}_{>k,k} \tilde{A}_{>k,k}^t$

Syrk: symmetric rank- k update $A_{>k,>k} \leftarrow A_{>k,>k} - \tilde{A}_{>k,k} \tilde{A}_{>k,k}^t$

Blocked implementation

Extra level of inner loops:

for $k = 1, \text{nblocks}$:

Chol: factor $L_k L_k^t \leftarrow A_{kk}$

for $\ell > k$:

Trsm: solve $\tilde{A}_{\ell,k} \leftarrow A_{\ell,k} L_k^{-t}$

for $\ell_1, \ell_2 > k$:

Gemm: form the product $\tilde{A}_{\ell_1,k} \tilde{A}_{\ell_2,k}^t$

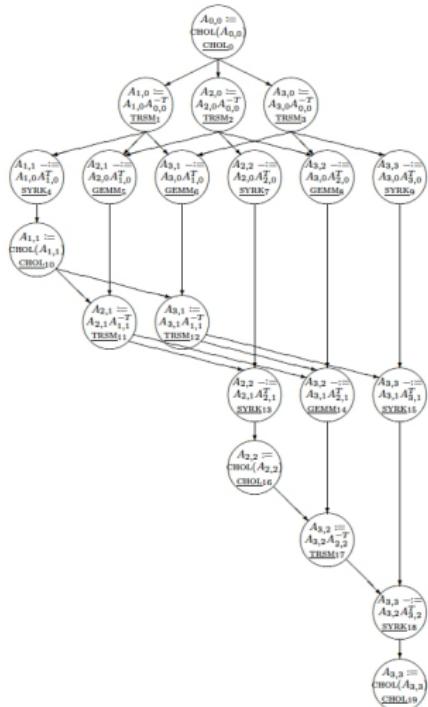
for $\ell_1, \ell_2 > k, \ell_1 \leq \ell_2$:

Syrk: symmetric rank- k update

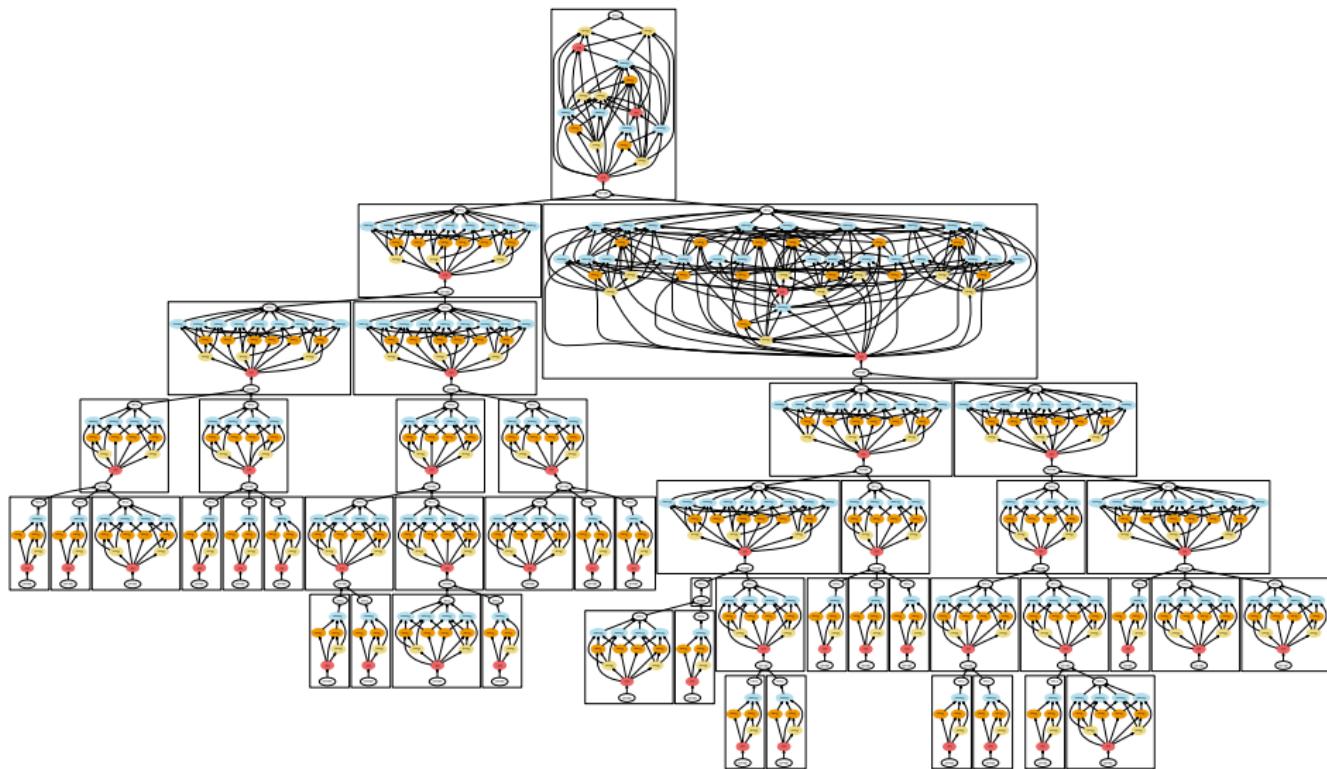
$$A_{\ell_1,\ell_2} \leftarrow A_{\ell_1,\ell_2} - \tilde{A}_{\ell_1,k} \tilde{A}_{\ell_2,k}^t$$

finished			
A_{kk}	$A_{k,k+1}$	$A_{k,k+2} \dots$	
$A_{k+1,k}$	$A_{k+1,k+1}$	$A_{k+1,k+2} \dots$	
$A_{k+2,k}$	$A_{k+2,k+2}$		
\vdots	\vdots		

You can graph this



Sometimes...



DAG schedulers

- Directed Acyclic Graph (dataflow)
- Each node has dependence on other nodes, can execute when dependencies available
- Quark/DaGue (TN): dependence on memory area written pretty much limited to dense linear algebra
- OpenMP has a pretty good scheduler
- Distributed memory scheduling is pretty hard

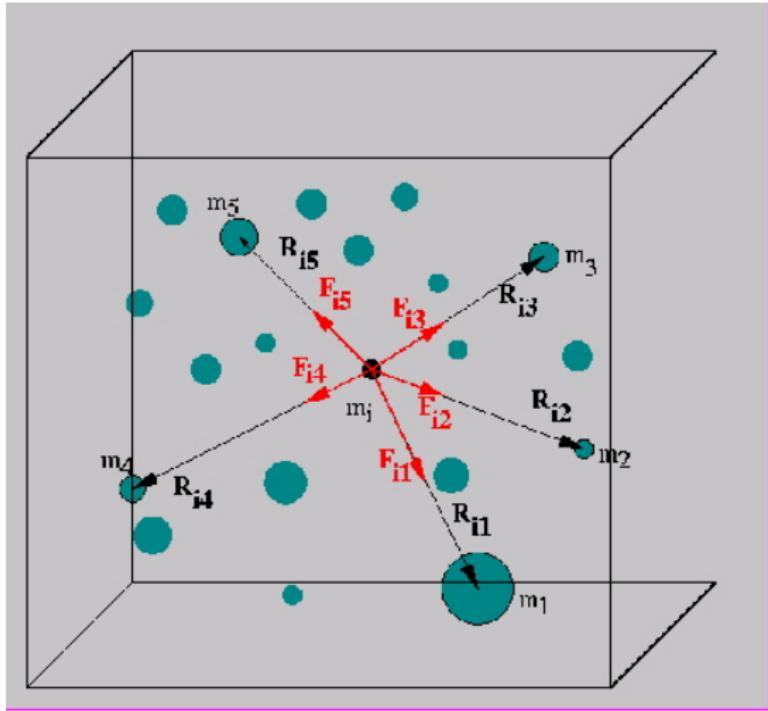
Applications

Justification

We briefly discuss two applications that, while at first glance not linear-algebra like, surprisingly can be covered by the foregoing concepts.

Table of Contents

Summing forces



Particle interactions

for each particle i

for each particle j

let \bar{r}_{ij} be the vector between i and j ;

then the force on i because of j is

$$f_{ij} = -\bar{r}_{ij} \frac{m_i m_j}{|\bar{r}_{ij}|}$$

(where m_i, m_j are the masses or charges) and

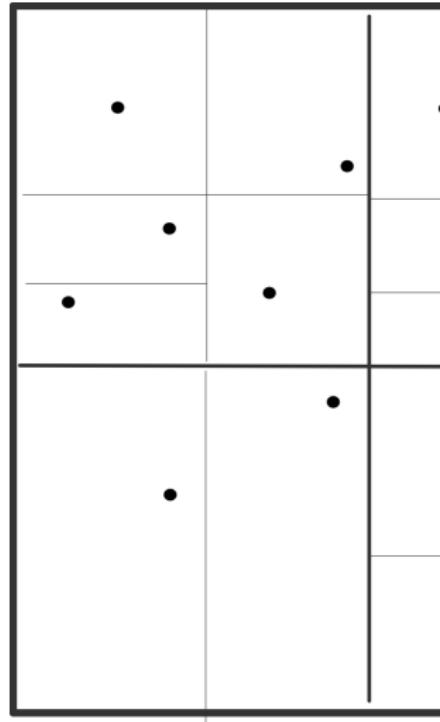
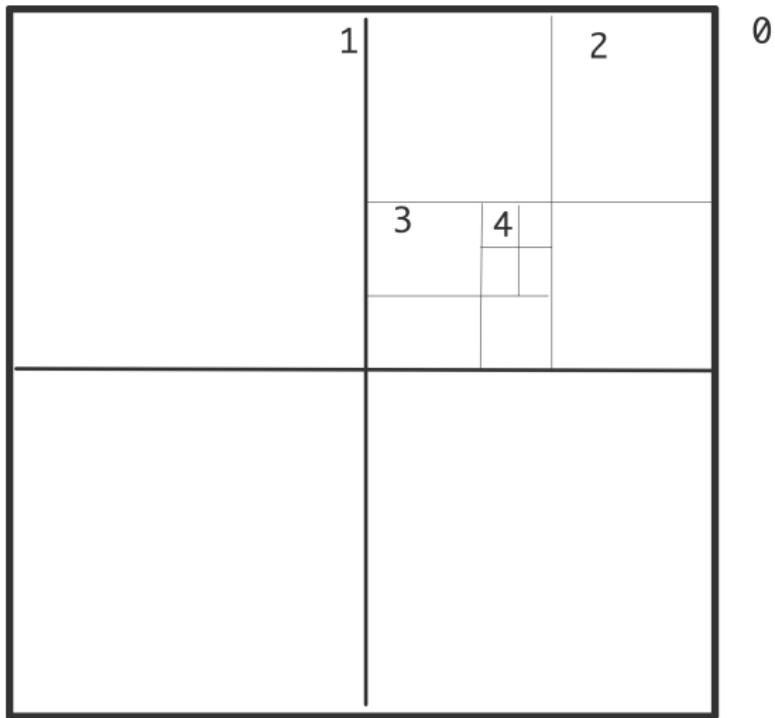
$$f_{ji} = -f_{ij}.$$

Sum forces and move particle over Δt

Complexity reduction

- Naive all-pairs algorithm: $O(N^2)$
- Clever algorithms: $O(N \log N)$, sometimes even $O(N)$
- Octtree algorithm: Barnes-Hut

Octree



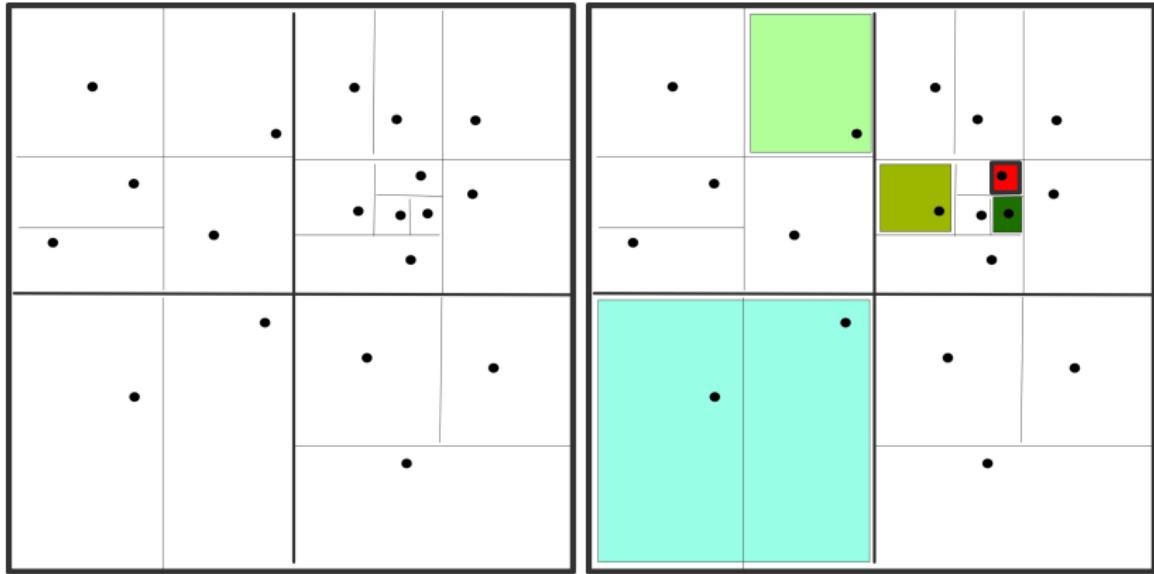
Dynamic octree creation

```
Procedure Quad_Tree_Build
    Quad_Tree = {empty}
    for j = 1 to N // loop over all N particles
        Quad_Tree_Insert(j, root) // insert particle j in QuadTree
    endfor
    Traverse the Quad_Tree eliminating empty leaves
```

```
Procedure Quad_Tree_Insert(j, n) // Try to insert particle j at node n
    if n an internal node // n has 4 children
        determine which child c of node n contains particle j
        Quad_Tree_Insert(j, c)
    else if n contains 1 particle // n is a leaf
        add n's 4 children to the Quad_Tree
        move the particle already in n into the child containing it
        let c be the child of n containing j
        Quad_Tree_Insert(j, c)
    else // n empty
        store particle j in node n
    end
```

Octree algorithm

- Consider cells on the top level
- if distance/diameter ratio small enough, take center of mass
- otherwise consider children cells



Masses calculation

```
// Compute the CM = Center of Mass and TM = Total Mass of all the particles
( TM, CM ) = Compute_Mass( root )

function ( TM, CM ) = Compute_Mass( n )
    if n contains 1 particle
        store (TM, CM) at n
        return (TM, CM)
    else          // post order traversal
        // process parent after all children
        for all children c(j) of n
            ( TM(j), CM(j) ) = Compute_Mass( c(j) )
        // total mass is the sum
        TM = sum over children j of n: TM(j)
        // center of mass is weighted sum
        CM = sum over children j of n: TM(j)*CM(j) / TM
        store ( TM, CM ) at n
    return ( TM, CM )
```

Force evaluation

```
// for each particle, compute the force on it by tree traversal
for k = 1 to N
    f(k) = TreeForce( k, root )
    // compute force on particle k due to all particles inside root

function f = TreeForce( k, n )
    // compute force on particle k due to all particles inside node n
    f = 0
    if n contains one particle // evaluate directly
        f = force computed using formula on last slide
    else
        r = distance from particle k to CM of particles in n
        D = size of n
        if D/r < theta // ok to approximate by CM and TM
            compute f
        else           // need to look inside node
            for all children c of n
                f = f + TreeForce ( k, c )
```

Complexity

- Each cell considers ‘rings’ of equi-distant cells
- but at doubling distance
- $c \log N$ cells to consider for each particle
- $N \log N$ overall

Computational aspects

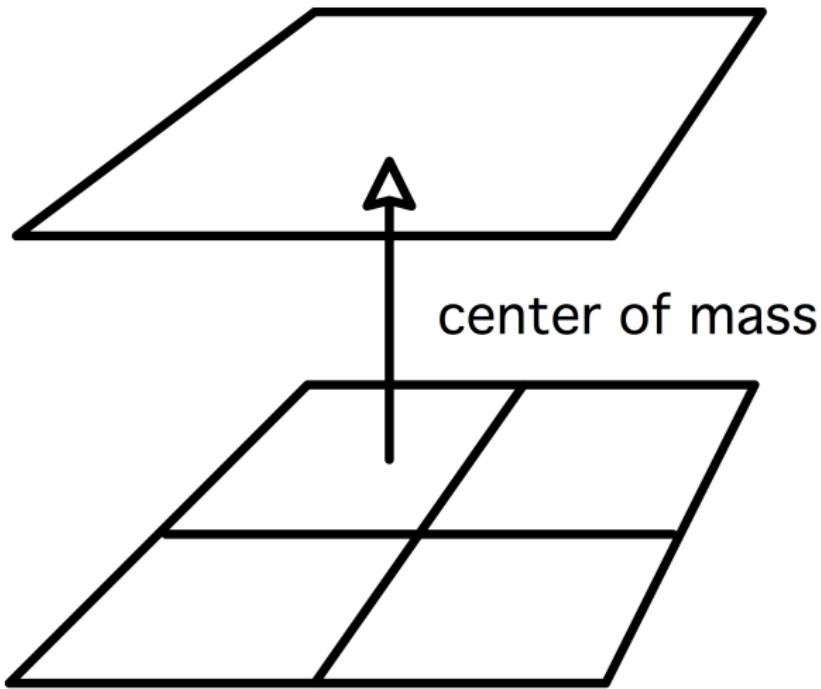
- After position update, particles can move to next box: load redistribution
- Naive octree algorithm is formulated for shared memory
- Distributed memory by using inspector-executor paradigm

Step 1: force by a particle

for level ℓ from one above the finest to the coarsest:

for each cell c on level ℓ

let $g_c^{(\ell)}$ be the combination of the $g_i^{(\ell+1)}$ for all children i of c



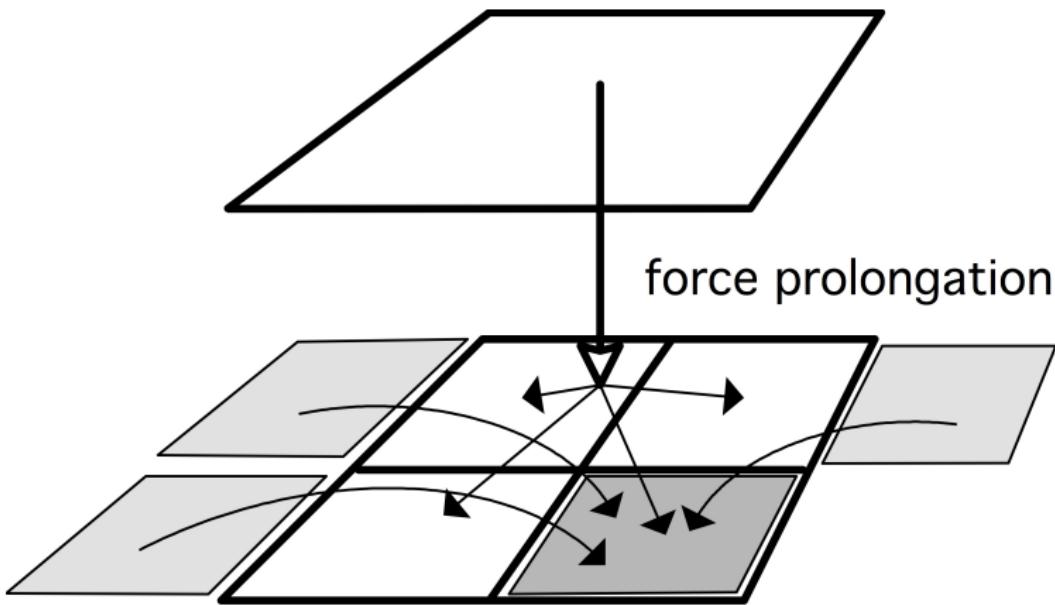
Step 2: force on a particle

for level ℓ from one below the coarsest to the finest:

for each cell c on level ℓ :

let $f_c^{(\ell)}$ be the sum of

1. the force $f_p^{(\ell-1)}$ on the parent p of c , and
 2. the sums $g_i^{(\ell)}$ for all i on level ℓ that
satisfy the cell opening criterium



- Center of mass calculation and force prolongation are local
- Force from neighbouring cells is a neighbour communication
- Neighbour communication can start before up/down tree calculation is finished: latency hiding

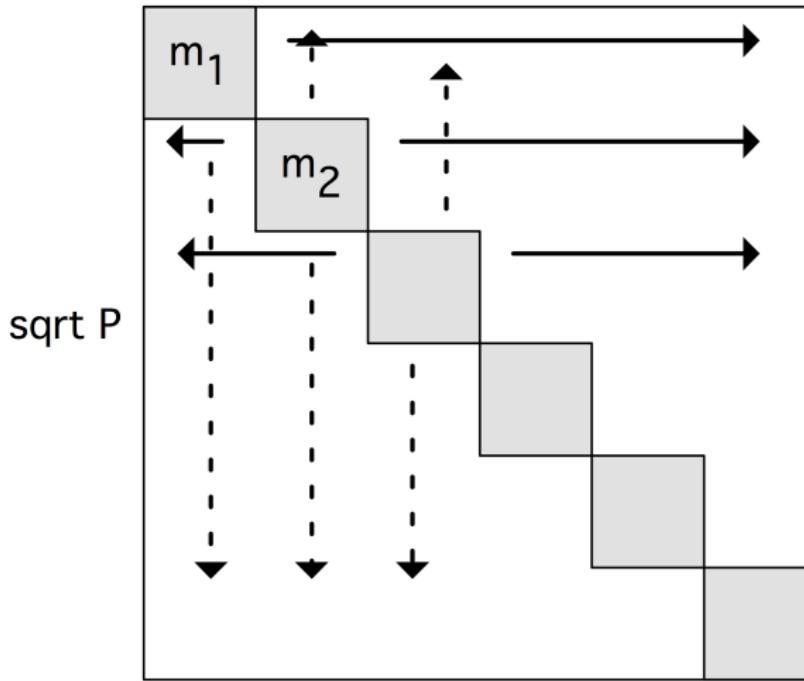
All-pairs methods

- Traditional algorithm: distribute particles, for each particle gather and update compute
- Problem: allgather has $O(N)\beta$ cost
- does not go down with P , so does not scale weakly

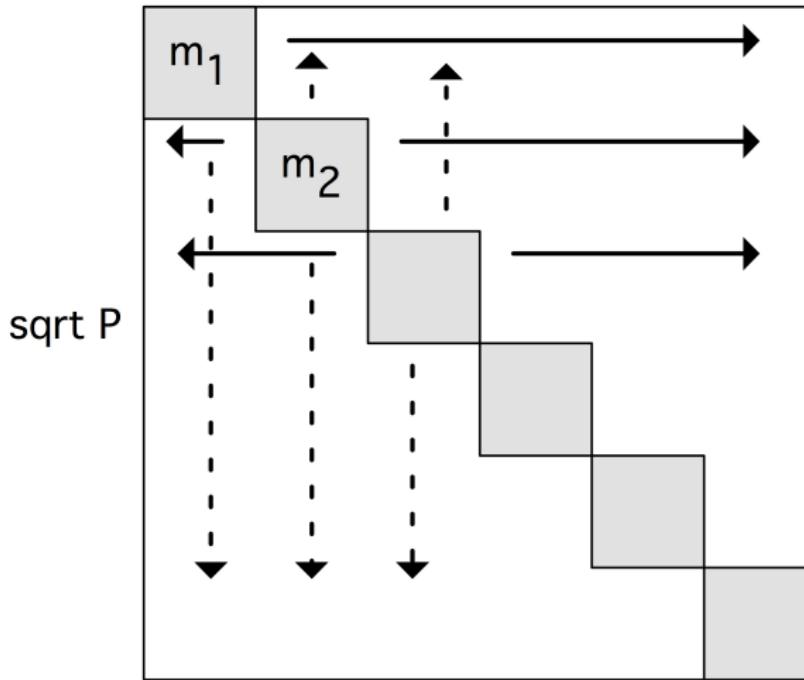
1.5D calculation

- Better algorithm: use $\sqrt{P} \times \sqrt{P}$ processor grid,
- Divide particles in bins of N/\sqrt{P}
- Processor (i, j) computes interaction of boxes i and j :

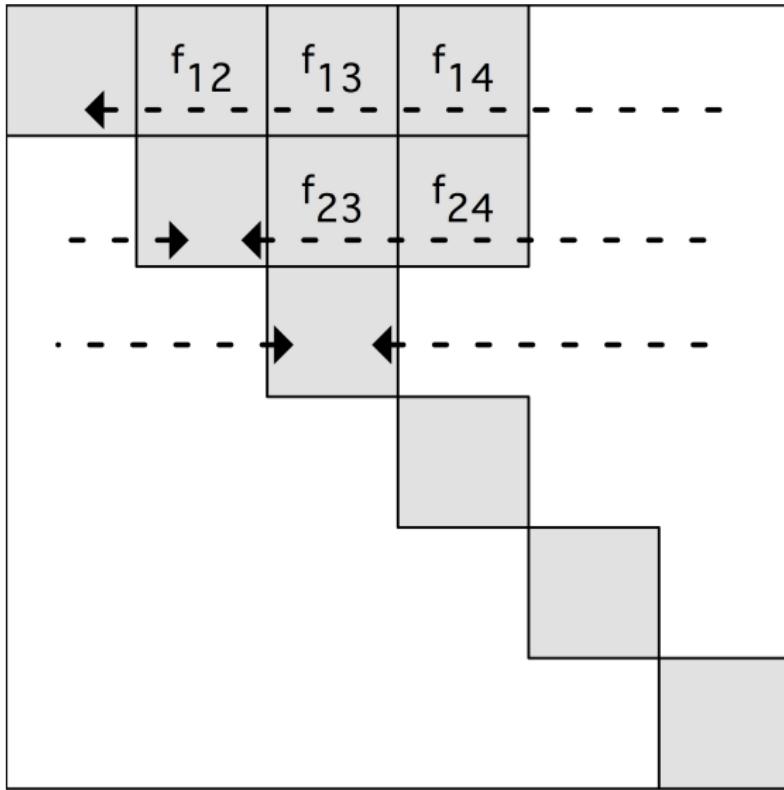
\sqrt{P}



\sqrt{P}



	$m_1 m_2$	$m_1 m_3$	$m_1 m_4$	- - - -
		$m_2 m_3$	$m_2 m_4$	- - - -
- - - -			- - - - -	
				- - - - - -



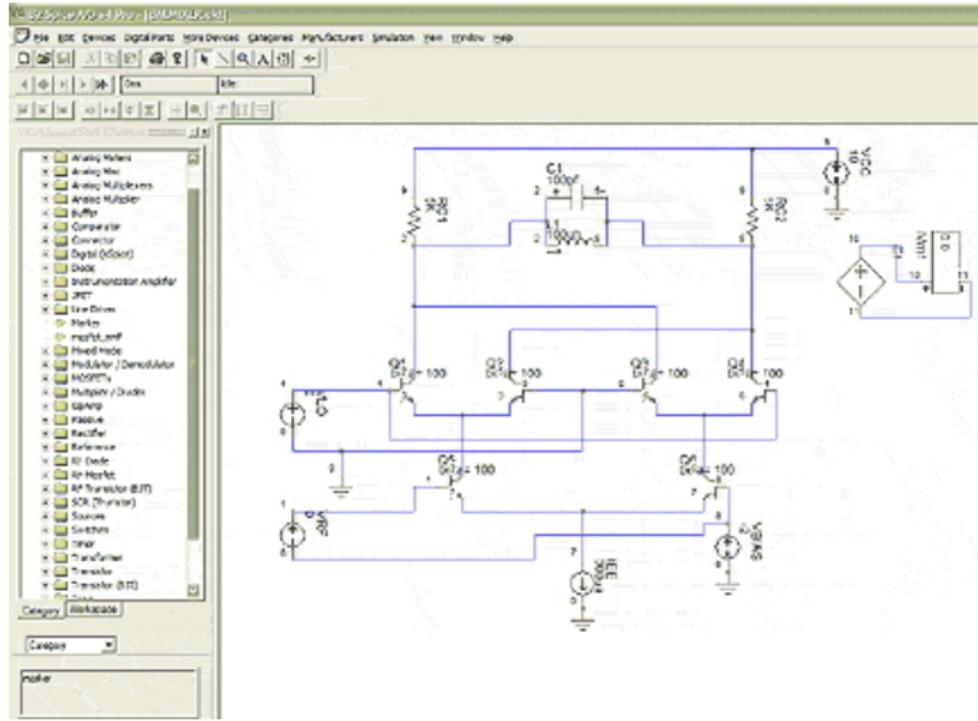
- Better algorithm: use $\sqrt{P} \times \sqrt{P}$ processor grid,
- Divide particles in boxes of $M = N/\sqrt{P}$
- Processor (i,j) computes interaction of boxes i and j :
- this requires broadcast (for duplication) and reduction (for summing) in processor rows and columns
- Bandwidth cost $\beta N/\sqrt{P}$ which is M : scalable.

Table of Contents

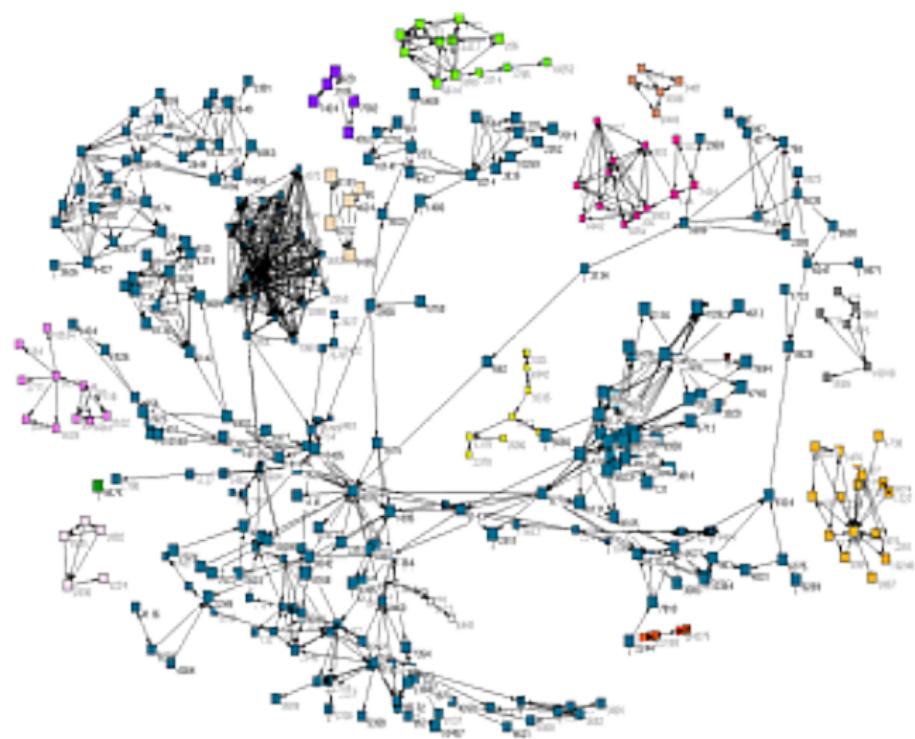
Graph algorithms

- Traditional: search, shortest path, connected components
- New: centrality

Traditional use of graph algorithms



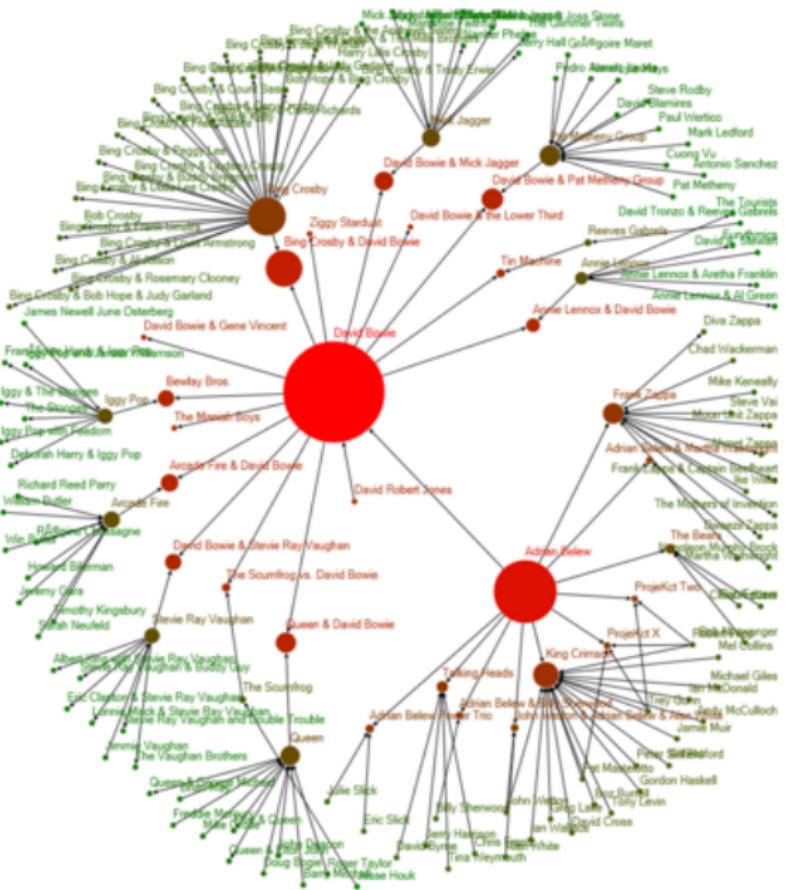
1990s use of graph algorithms



2010 use of graph algorithms



2010 use of graph algorithms



Traditional graph algorithm

Input : A graph, and a starting node s

Output: A function $d(v)$ that measures the distance from s to v

Let s be given, and set $d(s) = 0$

Initialize the finished set as $U = \{s\}$

Set $c = 1$

while *not finished* **do**

 Let V the neighbours of U that are not themselves in U

if $V = \emptyset$ **then**

 We're done

else

 Set $d(v) = c + 1$ for all $v \in V$.

$U \leftarrow U \cup V$

 Increase $c \leftarrow c + 1$

Computational characteristics

- Uses a queue: central storage
- Parallelism not self-evident
- Flexible assignment of work to processors, so no locality

Matrix formulation

Let

$$x_i = \begin{cases} 1 & i = s \\ \infty & \text{otherwise} \end{cases}$$

Let x zero except in i ,
then $x^t G$ nonzero in j if there is an edge (i,j)

Matrix algorithm

Define a product as

$$y^t = x^t G \equiv \forall_i: (y^t)_j = \min_{i: G_{ij} \neq 0} x_i + 1,$$

Iterate

$$x, x^t G, x^t G^2, \dots$$

After k (diameter) iterations $(x^t G^k)_i$ is the distance $d(s, i)$.

Single Source Shortest Path

Similar to previous, but non-unit edge weights

Let s be given, and set $d(s) = 0$

Set $d(v) = \infty$ for all other nodes v

for $|E| - 1$ times **do**

for all edges $e = (u, v)$ **do**

Relax: **if** $d(u) + w_{uv} < d(v)$ **then**

Set $d(v) \leftarrow d(u) + w_{uv}$

$$y^t = x^t G \equiv \forall_i: y_j = \min \left\{ x_j, \min_{i: G_{ij} \neq 0} \{x_i + g_{ij}\} \right\},$$

All-pairs shortest path

$$\Delta_{k+1}(u, v) = \min\{\Delta_k(u, v), \Delta_k(u, k) + \Delta_k(k, v)\}. \quad (9)$$

Algebraically:

```
for k from zero to |V| do
  D ← D · min[D(:, k) min · + D(k, :)]
```

Similarity to Gaussian elimination

Pagerank

T stochastic: all rowsums are 1.

Prove $x^t e = 1 \Rightarrow x^t T = 1$

Pagerank is essentially a power method: $x^t, x^t T, x^t T^2, \dots$ modeling page transitions.

Prevent getting stuck with random jump:

$$x^t \leftarrow s x^t T + (1 - s) e^t$$

Solution of linear system:

$$x^t(I - sT) = (1 - s)e^t$$

Observe

$$(I - sT)^{-1} = I + sT + s^2 T^2 + \dots$$

'Real world' graphs

- Graphs imply sparse matrix vector product
- ... but the graphs are unlike PDE graphs
- differences:
 - low diameter
 - high degree
 - power law
- treat as random sparse: use dense techniques
- 2D matrix partitioning: each block non-null, but sparse

Parallel treatment

- Intuitive approach: partitioning of nodes
- equivalent to 1D matrix distribution
- not scalable \Rightarrow 2D distribution
- equivalent to distribution of edges
- unlike with PDE graphs, random placement may actually be good

Parallel programming topics

Table of Contents

Elementary datatypes

C		F	
MPI_INT	integer	MPI_INTEGER	Integer
MPI_CHAR	signed char	MPI_CHARACTER	Character
MPI_LONG	signed long int		
MPI_UNSIGNED	unsigned int		
MPI_FLOAT	float	MPI_REAL	real
MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	Double Precision
MPI_BYTE		MPI_BYTE	(raw I/O byte)

Derived datatypes

- Identify structure in data for efficient transfer
- Recursive construction from elementary types
- Packing is done for you

Derived datatypes

- Contiguous: contiguous blocks (kinda pointless)
- Vector: strided blocks
- Indexed: irregular blocks
- Struct: Completely general placement and data types

Scheme

```
MPI_Type_<type>( oldtype, .... , &newtype );
MPI_Type_commit( newtype );
MPI_Send( .... newtype .... );
MPI_Type_free( newtype );
```

Type signature

Signature describes the structure of a datatype

Send and receive type do not have to be equal:
only be of equal signature

Example: send 4 doubles, receive one 4-double type

Contiguous

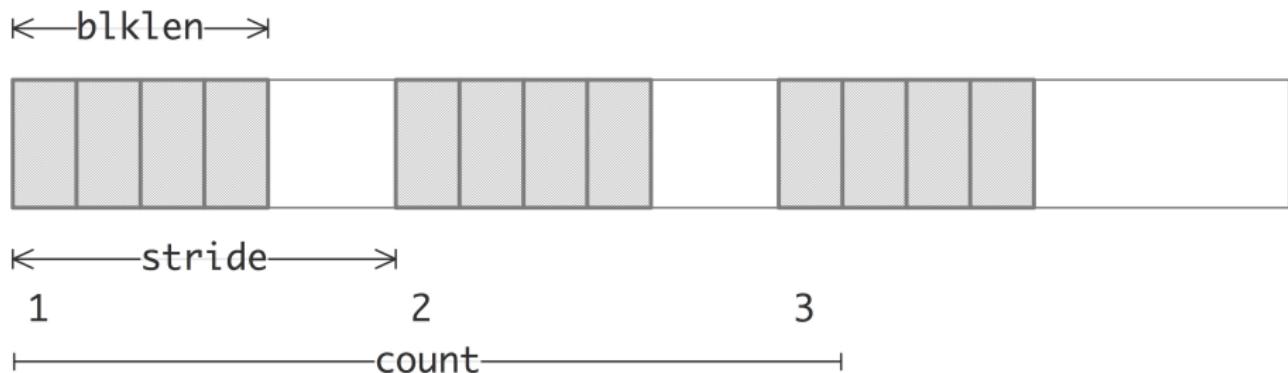


A contiguous datatype consists of a block of elements of a constituent type

Contiguous

```
MPI_Datatype newvectortype;
if (mytid==sender) {
    MPI_Type_contiguous(count,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,receiver,0,comm);
    MPI_Type_free(&newvectortype);
} else if (mytid==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE, sender,0,comm,
             &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(count==recv_count);
}
```

Vector

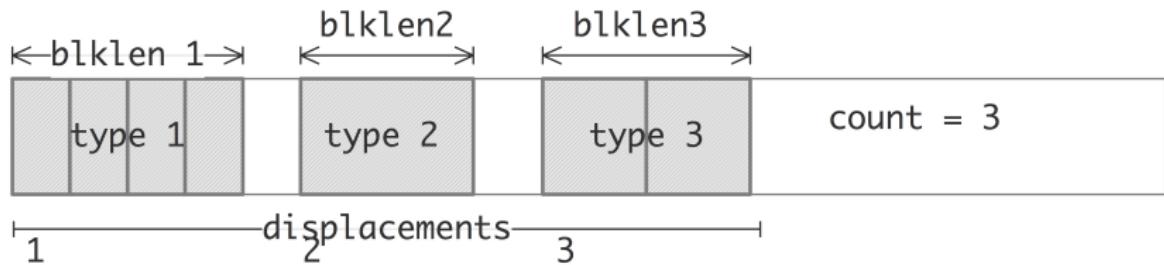


A vector datatype is built up out of strided blocks of elements of a constituent type

Vector

```
MPI_Datatype newvectortype;
if (mytid==sender) {
    MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,the_other,0,comm);
    MPI_Type_free(&newvectortype);
} else if (mytid==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
             &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(recv_count==count);
}
```

Indexed / Struct



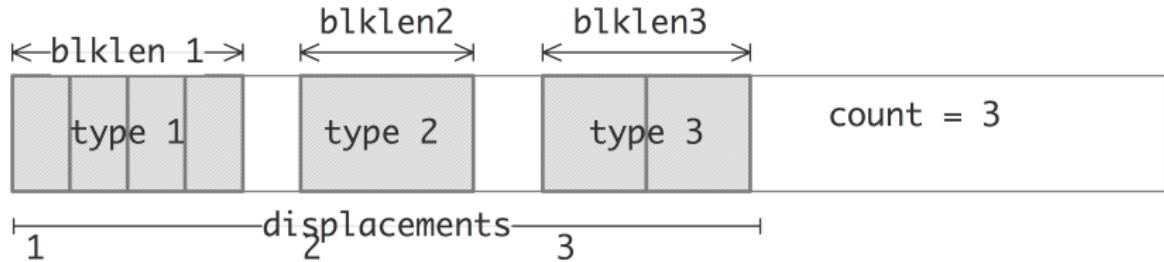
Indexed and Struct types have arbitrary blocks with arbitrary placements;
Struct can have multiple types

Indexed

```
indices = (int*) malloc(count*sizeof(int));
blocklengths = (int*) malloc(count*sizeof(int));
source = (double*) malloc(totalcount*sizeof(double));
target = (double*) malloc(count*sizeof(double));

MPI_Datatype newvectortype;
if (mytid==sender) {
    MPI_Type_indexed(count,blocklengths,indices,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,the_other,0,comm);
    MPI_Type_free(&newvectortype);
} else if (mytid==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
             &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(recv_count==count);
}
```

Struct



Components of a structure are not necessarily aligned:
tricky location calculations

Struct

```
struct object {
    char c;
    double x[2];
    int i;
};

MPI_Datatype newstructuretype;
int structlen = 3;
int blocklengths[structlen]; MPI_Datatype types[structlen];
MPI_Aint displacements[structlen];
// where are the components relative to the structure?
blocklengths[0] = 1; types[0] = MPI_CHAR;
displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;
blocklengths[1] = 2; types[1] = MPI_DOUBLE;
displacements[1] = (size_t)&(myobject.x[0]) - (size_t)&myobject;
blocklengths[2] = 1; types[2] = MPI_INT;
displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;
```

Struct (cont'd)

```
MPI_Type_create_struct(
    structlen,blocklengths,displacements,types,
    &newstructuretype);
MPI_Type_commit(&newstructuretype);
{
    MPI_Aint typesize;
    MPI_Type_extent(newstructuretype,&typesize);
    if (mytid==0) printf("Type extent: %d bytes\n",typesize);
}
if (mytid==sender) {
    MPI_Send(&myobject,1,newstructuretype,the_other,0,comm);
} else if (mytid==receiver) {
    MPI_Recv(&myobject,1,newstructuretype,the_other,0,comm,
        MPI_STATUS_IGNORE);
}
MPI_Type_free(&newstructuretype);
```

Packing: another approach to heterogeneous types

- The `MPI_Pack` command adds data to a send buffer;
- the `MPI_Unpack` command retrieves data from a receive buffer;
- the buffer is sent with a datatype of `MPI_PACKED`.

```
int MPI_Pack(  
    void *inbuf, int incount, MPI_Datatype datatype,  
    void *outbuf, int outcount, int *position,  
    MPI_Comm comm);
```

```
int MPI_Unpack(  
    void *inbuf, int insize, int *position,  
    void *outbuf, int outcount, MPI_Datatype datatype,  
    MPI_Comm comm);
```

Pack example

```
if (mytid==sender) {  
    MPI_Pack (&nstarts, 1, MPI_INT, buffer, buflen, &position, comm);  
    for (i=0; i<nstarts; i++) {  
        double value = rand() / (double) RAND_MAX;  
        MPI_Pack (&value, 1, MPI_DOUBLE, buffer, buflen, &position, comm);  
    }  
    MPI_Pack (&nstarts, 1, MPI_INT, buffer, buflen, &position, comm);  
    MPI_Send(buffer, position, MPI_PACKED, other, 0, comm);  
}
```

Pack example (cont'd)

```
} else if (mytid==receiver) {  
    int irecv_value;  
    double xrecv_value;  
    MPI_Recv(buffer,buflen,MPI_PACKED,other,0,comm,  
             MPI_STATUS_IGNORE);  
    MPI_Unpack(buffer,buflen,&position,&nstarts,1,MPI_INT,comm);  
    for (i=0; i<nstarts; i++) {  
        MPI_Unpack(buffer,buflen,&position,&xrecv_value,1,  
                   MPI_DOUBLE,comm);  
    }  
    MPI_Unpack(buffer,buflen,&position,&irecv_value,1,  
               MPI_INT,comm);  
    ASSERT(irecv_value==nstarts);  
}
```

Table of Contents

Communicator trickery

- Communicator duplication
- Disjoint subcommunicators
- Nondisjoint subcommunicators
- Topologies, inter-communicators, spawning (sorry, not in this lecture)

Communicator duplication

Simplest new communicator: identical copy

Useful for libraries
separate library traffic from application

Comm dup

```
class library {  
private:  
    MPI_Comm comm;  
    int mytid,ntids,other;  
    MPI_Request request[2];  
public:  
    library(MPI_Comm incomm) {  
        comm = incomm;  
        MPI_Comm_rank(comm,&mytid);  
        other = 1-mytid;  
    };  
    void communication_start() {  
        int sdata=6,rdata;  
        MPI_Isend(&sdata,1,MPI_INT,other,2,comm,&(request[0]));  
        MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,comm,&(request[1]));  
    }  
    void communication_end() {  
        MPI_Status status[2];  
        MPI_Waitall(2,request,status); }  
};
```

Comm dup (cont'd)

```
MPI_Isend(&sdata,1,MPI_INT,other,1,comm,&(request[0]));
my_library.communication_start();
MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,comm,
          &(request[1]));
MPI_Waitall(2,request,status);
my_library.communication_end();
```

Communicator splitting

Processor grid:

```
MPI_Comm_rank( &mytid );
proc_i = mytid % proc_column_length;
proc_j = mytid / proc_column_length;
```

Communicator per column:

```
MPI_Comm column_comm;
MPI_Comm_split( MPI_COMM_WORLD, proc_j, i, &column_comm );
```

Broadcast in that column:

```
MPI_Bcast( data, /* tag: */ 0, column_comm );
```

Comm split example

```
MPI_Comm_size(comm, &npes);  
MPI_Comm_rank(comm, &rank);  
int icolor = rank%3, key = npes-rank;  
  
MPI_Comm_split(comm, icolor, key, &newcomm);  
MPI_Comm_rank(newcomm, &newrank);
```

rank	icolor	key	newrank	color zero
0	0	9	2	⇐
1	1	8	2	
2	2	7	2	
3	0	6	1	⇐
4	1	5	1	
5	2	4	1	
6	0	3	0	⇐
7	1	2	0	
8	2	1	0	

Communicators and groups

Communicator to group to communicator:

```
MPI_Comm_group( comm, &group );
MPI_Comm_create( old_comm, group, &new_comm );
```

and groups are manipulated with MPI_Group_incl, MPI_Group_excl,
MPI_Group_difference and a few more.

Table of Contents

General idea

- Collectives are blocking
- Any process delay visible to every other process
- Some architectures have separate network for collectives
- Idea: let collective progress independently, test for completion

Example

Non-blocking collective gives MPI_Request pointer

```
int MPI_Iallreduce(const void *sendbuf, void *recvbuf,  
                   int count, MPI_Datatype datatype,  
                   MPI_Op op, MPI_Comm comm,  
                   MPI_Request *request)
```

Test for completion with MPI_Wait and MPI_Test and such.

Latency hiding

Overlapping collective communication with useful computation,

Example: in iterative methods

```
x ← ...
xtx  non-blocking start
y ← Ax
y ← y/xtx
```

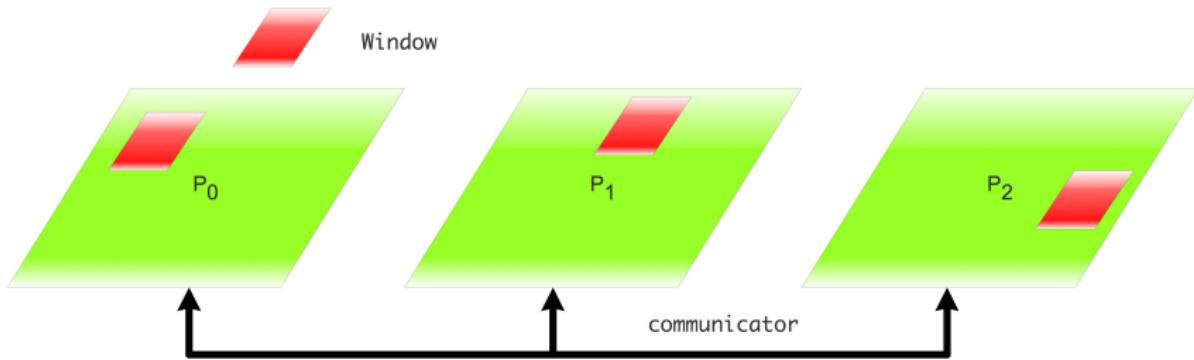
Table of Contents

Basic notions

- One-sided access: Put, Get, Accumulate
- Window: memory designated for one-sided communication
- Origin: process that makes the one-sided call
target: process that exposes memory window
- Different synchronization mechanism: active and passive target synchronization
- Memory model: shared memory explicitly synchronized with local memory

Windows

```
MPI_Win_create (void *base, MPI_Aint size,  
    int disp_unit, MPI_Info info,  
    MPI_Comm comm, MPI_Win *win)
```



Note: collective on communicator

Put/Get/Accumulate

```
MPI_Put (void *origin_addr, int origin_count,  
MPI_Datatype origin_datatype, int target_rank,  
MPI_Aint target_disp, int target_count,  
MPI_Datatype target_datatype,  
MPI_Win window)
```

```
MPI_Accumulate (void *origin_addr, int origin_count,  
MPI_Datatype origin_datatype, int target_rank,  
MPI_Aint target_disp, int target_count,  
MPI_Datatype target_datatype,  
MPI_Op op, MPI_Win window)
```

Globally defined epochs

```
MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);  
MPI_Get( /* operands */, win);  
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

Induces synchronization

Restrictions from memory model

Windows are not coherent with local memory:

- Data in window undefined until fence synchronization
- No put and get in same epoch

Fine-grained synchronization

Exposure:

```
MPI_Win_post( /* group of origin processes */ )
MPI_Win_wait()
```

Access:

```
MPI_Win_start( /* group of target processes */ )
// access operations
MPI_Win_complete()
```

Example

```
if (mytid==origin) {
    MPI_Group_incl(all_group,1,&target,&two_group);
    // access
    MPI_Win_start(two_group,0,the_window);
    MPI_Put( /* data on origin: */    &my_number, 1,MPI_INT,
             /* data on target: */   target,0,    1,MPI_INT,
             the_window);
    MPI_Win_complete(the_window);
}
if (mytid==target) {
    MPI_Group_incl(all_group,1,&origin,&two_group);
    // exposure
    MPI_Win_post(two_group,0,the_window);
    MPI_Win_wait(the_window);
}
```

Passive target synchronization

```
If (rank == 0) {  
    MPI_Win_lock (MPI_LOCK_EXCLUSIVE, 1, 0, win);  
    MPI_Put (outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);  
    MPI_Win_unlock (1, win);  
}
```

Lock types are:

- MPI_LOCK_SHARED for Get;
- MPI_LOCK_EXCLUSIVE for Put and Accumulate

Start/complete/post/wait example

```
if (mytid==origin) {  
    MPI_Group_incl(all_group,1,&target,&two_group);  
    // access  
    MPI_Win_start(two_group,0,the_window);  
    MPI_Put( /* data on origin: */    &my_number, 1,MPI_INT,  
            /* data on target: */    target,0,    1,MPI_INT,  
            the_window);  
    MPI_Win_complete(the_window);  
}  
  
if (mytid==target) {  
    MPI_Group_incl(all_group,1,&origin,&two_group);  
    // exposure  
    MPI_Win_post(two_group,0,the_window);  
    MPI_Win_wait(the_window);  
}
```

Passive target synchronization

- Emulates shared memory
- Origin process locks window on target
- MPI-2 was lacking atomic operations
- MPI-3 has the fix

Passive target example

```
for (int i=0; i<ninputs; i++)
    myjobs[i] = 0;
if (mytid!=repository) {
    float contribution=(float)mytid,table_element;
    int loc=0;
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE,repository,0,the_window);
    // read the table element by getting the result from adding zero
    err = MPI_Fetch_and_op(&contribution,&table_element,MPI_FLOAT,
                          repository,loc,MPI_SUM,the_window); CHK(err);
    MPI_Win_unlock(repository,the_window);
}
```

MPI_T tool interface

VarList: list internal variables

https://computation-rnd.llnl.gov/mpi_t/varList.php

Gyan: monitor internal variables

https://computation-rnd.llnl.gov/mpi_t/gyan.php

Table of Contents

Analysis basics

- Measurements: repeated and controlled
beware of transients, do you know where your data is?
- Document everything
- Script everything

Compiler options

- Defaults are a starting point
- use reporting options: `-opt-report`, `-vec-report`
useful to check if optimization happened / could not happen
- test numerical correctness before/after optimization change (there are options for numerical correctness)

Optimization basics

- Use libraries when possible: don't reinvent the wheel
- Premature optimization is the root of all evil (Knuth)

Code design for performance

- Keep inner loops simple: no conditionals, function calls, casts
- Avoid small functions: try macros or inlining
- Keep in mind all the cache,TLB, SIMD stuff from before
- SIMD: Fortran array syntax helps

Multicore / multithread

- Use numactl: prevent process migration
- ‘first touch’ policy: allocate data where it will be used
- Scaling behaviour mostly influenced by bandwidth

Multinode performance

- Influenced by load balancing
- Use HPCtoolkit, Scalasca, TAU for plotting
- Explore ‘eager’ limit (mvapich2: environment variables)

Classes of programming errors

Logic errors:

functions behave differently from how you thought,
or interact in ways you didn't envision

Hard to debug

Coding errors:

send without receive

forget to allocate buffer

Debuggers can help

Defensive programming

Defensive programming

- Keep It Simple ('restrict expressivity')
- Example: use collective instead of spelling it out
- easier to write / harder to get wrong
the library and runtime are likely to be better at optimizing than you

Memory management

Beware of memory leaks:
keep allocation and free in same lexical scope

Modular design

Design for debuggability, also easier to optimize

Separation of concerns: try to keep code aspects separate

Premature optimization is the root of all evil (Knuth)

MPI performance design

Be aware of latencies: bundle messages
(this may go again separation of concerns)

Consider ‘eager limit’

Process placement, reduction in number of processes

Debugging

Debugging

I assume you know about gdb and valgrind . . .

- Interactive use of gdb, starting up multiple xterms feasible on small scale
- Use gdb to inspect dump:
can be useful, often a program crashes hard and leaves no dump

Note: compile options -g -O0

Parallel debuggers

Allinea DDT 4.2-54404

File View Control Search Tools Window Help

Current Group: All Focus on current: Group Process Thread Step Threads Together

All 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Create Group

Project Files problem1.c

Search (Ctrl+K)

Application Code

- /
- Sources
- problem1.c

External Code

problem1.c

```
18     MPI_Finalize();
19     MPI_Barrier(comm);
20 }
21
22
23 int main(int argc,char **argv) {
24     MPI_Comm comm;
25
26     MPI_Init(&argc,&argv);
27     comm = MPI_COMM_WORLD;
28
29     loop_for_awhile(comm);
30
31     MPI_Finalize();
32     return 0;
33 }
```

Locals Current Line(s) Current Stack

Variable Name Value

argc	1
argv	0x7fffff7958

Type: none selected

Input/Output | Breakpoints | Watchpoints | Stacks | Tracepoints | Tracepoint Output | Logbook | Evaluate

Stacks

Processes Function /

16 main (problem1.c:26)

Eijkhout: HPC intro Ready

Buggy code

```
for (it=0; ; it++) {  
    double randomnumber = ntids * ( rand() / (double)RAND_MAX );  
    printf("[%d] iteration %d, random %e\n",mytid,it,randomnumber);  
    if (randomnumber>mytid && randomnumber<mytid+1./(ntids+1))  
        MPI_Finalize();  
    MPI_Barrier(comm);  
}
```

Parallel inspection

Allinea DDT 4.2-34404

File View Control Search Tools Window Help

Current Group: All Focus on current: Group Process Thread Step Threads Together

All 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Create Group

Project Files

Search (Ctrl+K)

Application Code

- /
- Sources
 - problem1.c
 - loop_for_awh
 - main(int argc)
- External Code

problem1.c ch3_smp_progress.c

```
10 // Initialize the random number generator
11 srand((int)(mytid*(double)RAND_MAX/ntids));
12
13
14 for (it=0; it++ ) {
15     double randomnumber = ntids * ( rand() / (double)RAND_MAX );
16     printf("%d iteration %d, random %e\n", mytid, it, randomnumber);
17     if (randomnumber>mytid && randomnumber<mytid+1./ntids)
18         MPI_Finalize();
19     MPI_BARRIER(comm);
20 }
21
22
23 int main(int argc,char **argv) {
24     MPI_Comm comm;
25
26     MPI_Init(&argc,&argv);
27     comm = MPI_COMM_WORLD;
```

Locals

Variable Name	Value
-comm	1140850688
-it	31
-mytid	1
-ntids	16
-randomnumber	1.056087621

Current Line(s)

Current Stack

Input/Output* Breakpoints Watchpoints Stacks Tracepoints Tracepoint Output Logbook

Stacks

Processes Function

Processes	Function
16	main (problem1.c:29)
1	loop_for_awh (problem1.c:18)
15	loop_for_awh (problem1.c:19)
15	PMPI_BARRIER (barrier.c:411)
15	MPIR_BARRIER_IMPL (barrier.c:266)
15	MPIR_BARRIER_MV2 (barrier_osu.c:198)
15	MPIR_BARRIER_INTRA_MV2 (barrier_osu.c:166)
1	MPIR_SHMEM_BARRIER_MV2 (barrier_osu.c:104)
1	MPIDI_CH3I_SHMEM_COLL_BARRIER_GATHER (ch3_shmem_coll.c:940)
1	MPIDI_CH3I_PROGRESS_TEST (ch3_progress.c:471)
1	MPIDI_CH3I_SMP_READ_PROGRESS (ch3_smp_progress.c:743)
1	MPIDI_CH3I_SMP_PULL_HEADER (ch3_smp_progress.c:4345)
14	MPIDI_SHMEM_BARRIER_MV2 (barrier_osu.c:113)

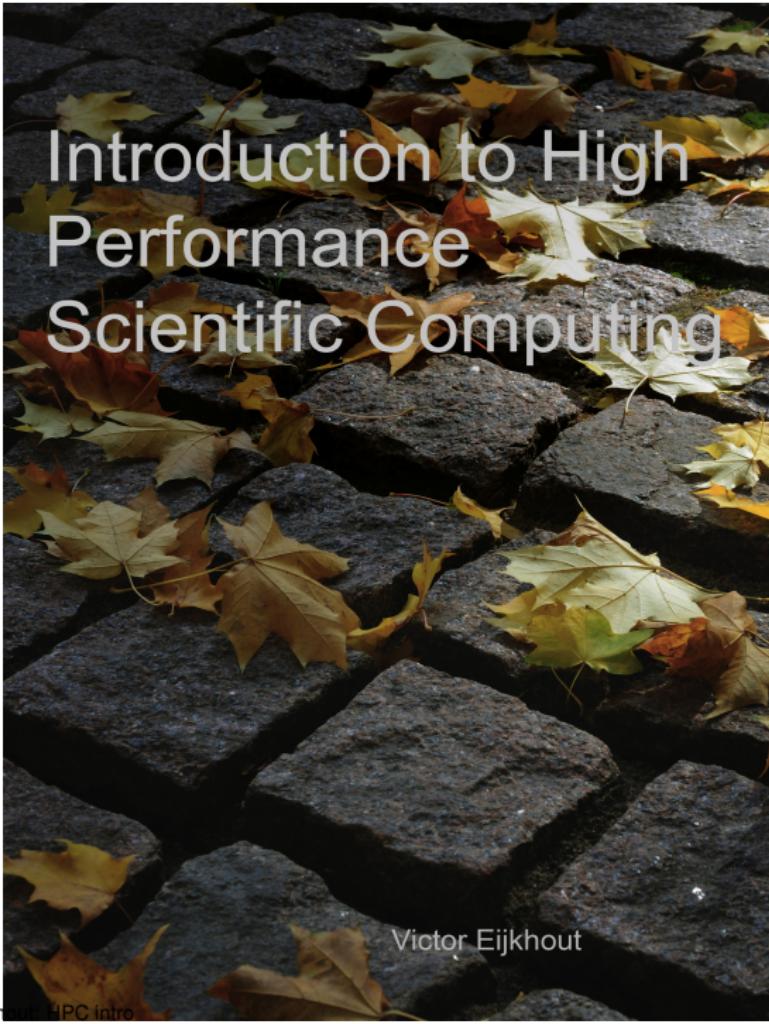
Eijkhout: HPC intro Ready

Stack trace

Processes	Function
16	main (problem1.c:29)
1	+ loop_for_awhile (problem1.c:18)
15	loop_for_awhile (problem1.c:19)
15	PMPI_Barrier (barrier.c:411)
15	MPIR_Barrier_Impl (barrier.c:266)
15	MPIR_Barrier_MV2 (barrier_osu.c:198)
15	MPIR_Barrier_intra_MV2 (barrier_osu.c:166)
1	MPIR_shmem_barrier_MV2 (barrier_osu.c:104)
1	MPIDI_CH3I_SHMEM_COLL_Barrier_gather (ch3_shmem_coll.c:940)
1	MPIDI_CH3I_Progress_test (ch3_progress.c:471)
1	MPIDI_CH3I_SMP_read_progress (ch3_smp_progress.c:743)
1	MPIDI_CH3I_SMP_pull_header (ch3_smp_progress.c:4345)
14	+ MPIR_shmem_barrier_MV2 (barrier_osu.c:113)

Variable inspection

Locals	Current Line(s)	Current Stack											
Locals													
<table><thead><tr><th>Variable Name</th><th>Value</th></tr></thead><tbody><tr><td>-comm</td><td>— 1140850688</td></tr><tr><td>-it</td><td>— 31</td></tr><tr><td>-mytid</td><td>— 1</td></tr><tr><td>-ntids</td><td>— 16</td></tr><tr><td>-randomnumber</td><td>— 1.056087621</td></tr></tbody></table>		Variable Name	Value	-comm	— 1140850688	-it	— 31	-mytid	— 1	-ntids	— 16	-randomnumber	— 1.056087621
Variable Name	Value												
-comm	— 1140850688												
-it	— 31												
-mytid	— 1												
-ntids	— 16												
-randomnumber	— 1.056087621												



Introduction to High Performance Scientific Computing

Victor Eijkhout

Index