

Lecture 2

Data parallelism

Announcements



Today's lecture

- Finish up last lecture
 - Address space organization
 - Control
- Data Parallelism
 - Parallel Random Access Machines
 - Vector architectures
 - Vectorization

Memory hierarchies

Address space organization

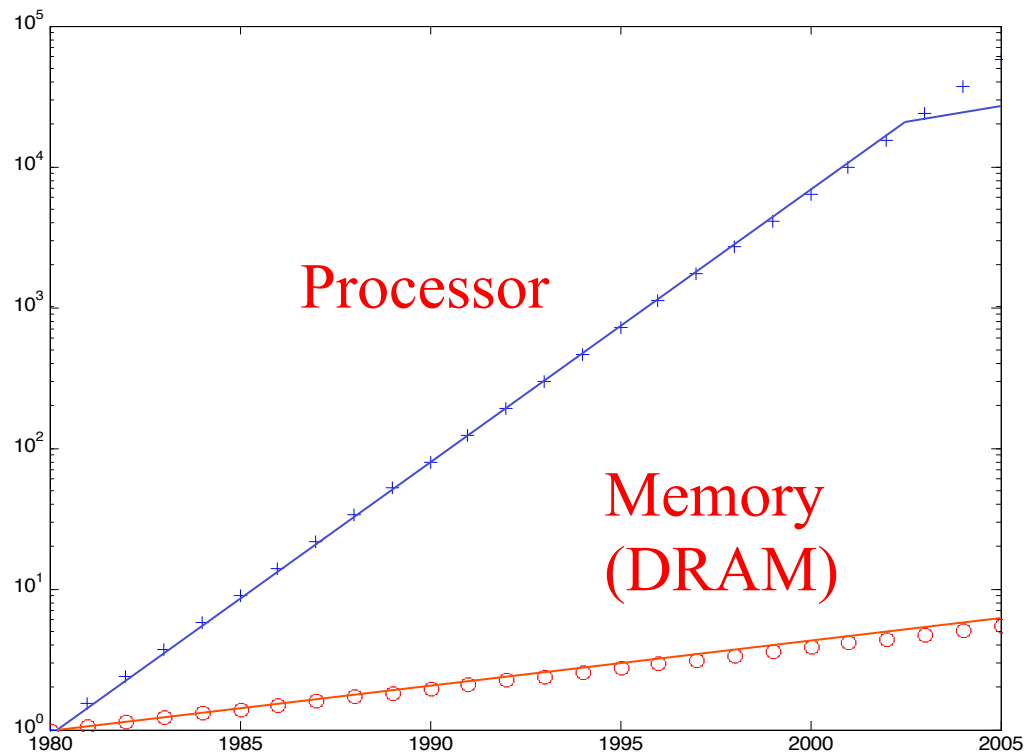
Control

The hardware

- Address space organization
 - Shared memory
 - Distributed memory
- Control mechanism

The processor-memory gap

- The result of technological trends
- Difference in processing and memory speeds growing exponentially over time



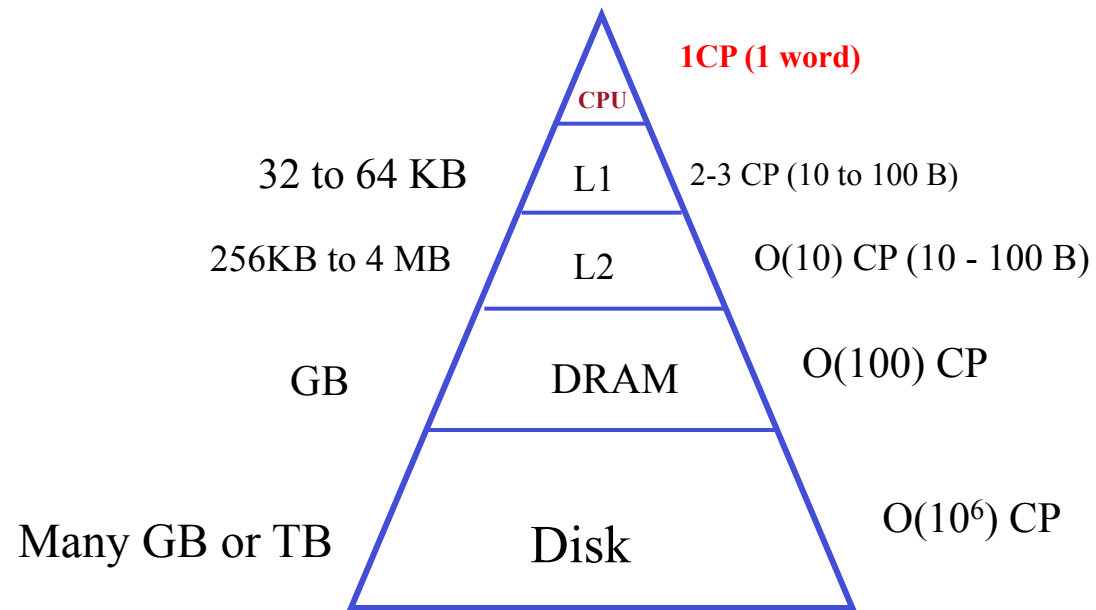
An important principle: locality

- Programs generally exhibit two forms of locality in accessing memory
 - Temporal locality (time)
 - Spatial locality (space)
- Often involves loops
- Opportunities for reuse

```
for t=0 to T-1
  for i = 1 to N-2
     $u[i] = (u[i-1] + u[i+1]) / 2$ 
```

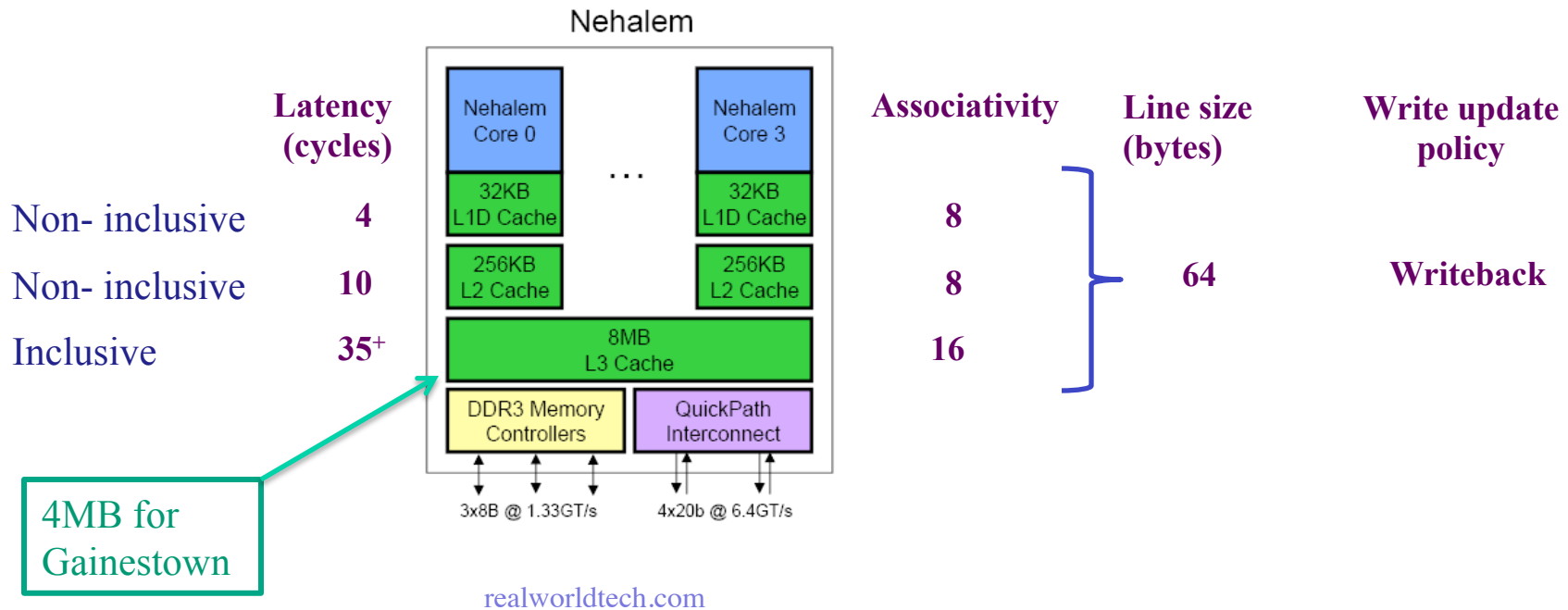
Memory hierarchies

- Exploit reuse through a hierarchy of smaller but faster memories
- Put things in faster memory if we reuse them frequently



Nehalem's Memory Hierarchy

- Source: *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Table 2.7

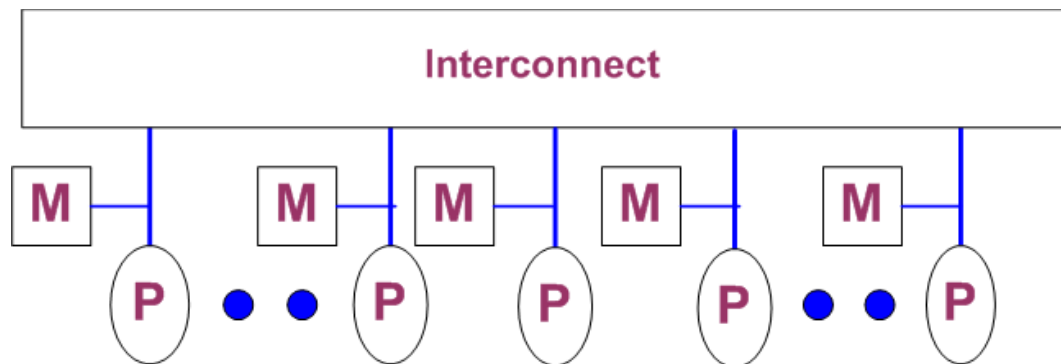
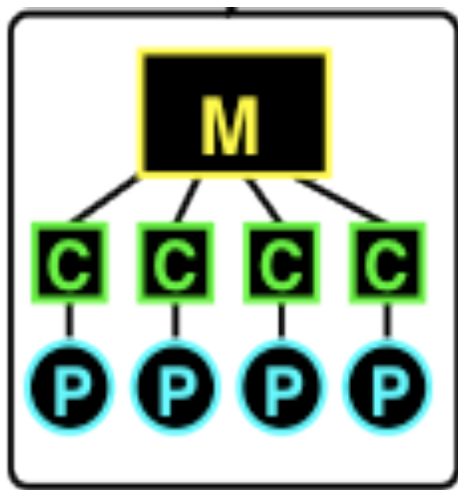


Address Space Organization

- We classify the address space organization of a parallel computer according to whether or not it provides global memory
- If there is global memory we have a “shared memory” or “shared address space” architecture
 - *multiprocessor* vs *partitioned global address space*
- When there is no global memory, we have a “shared nothing” architecture, also known as a *multicomputer*

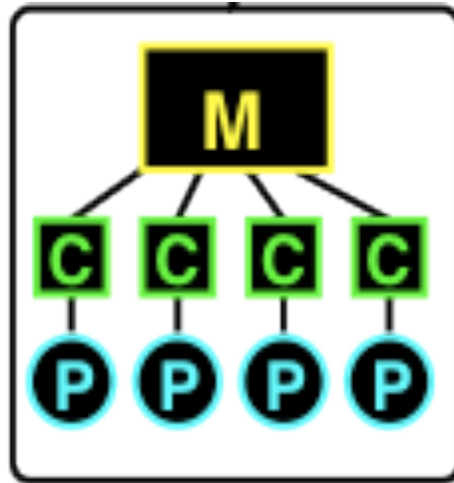
Multiprocessor organization

- Hardware automatically performs the global to local mapping using address translation mechanisms
- 2 types, according to uniformity of memory access times
 - **UMA**: Uniform Memory Access time
 - **NUMA**: Non-Uniform Memory Access time



UMA shared memory

- Uniform Memory Access time
- In the absence of contention, all processors observe the same memory access time
- Also called **Symmetric Multiprocessors**
- Usually bus based
- Not scalable



Intel Clovertown Memory Hierarchy

- Ieng-203
- Intel Xeon X5355 (Intro: 2006)
- Two “Woodcrest” dies on a multichip module



techreport.com/articles.x/10021/2

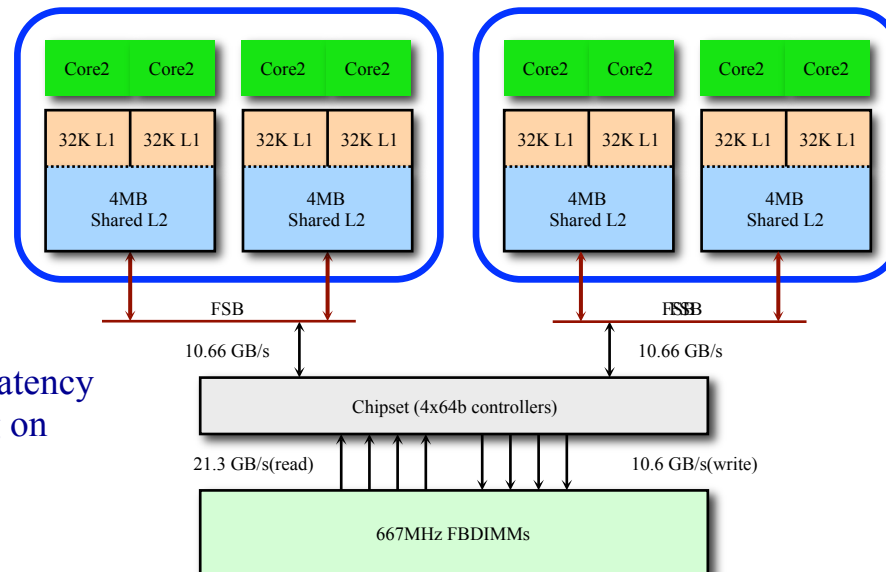
Line Size = 64B (L1 and L2)

Access latency (clocks)

3

14*

* Software-visible latency will vary depending on access patterns and other factors



Associativity

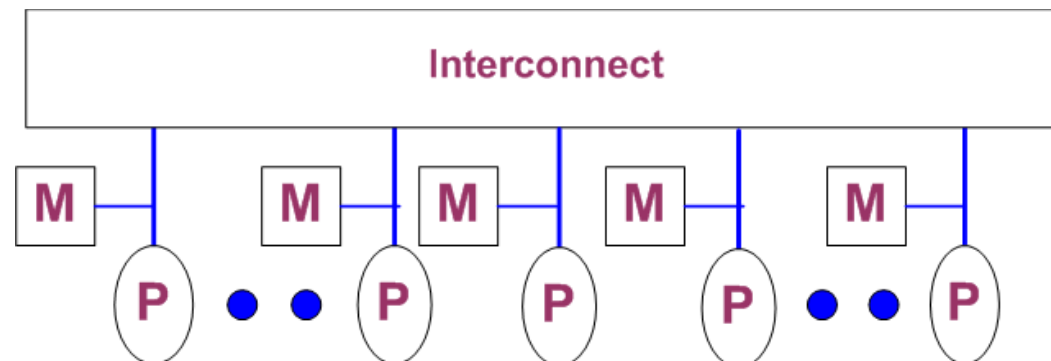
8

16

Sam Williams et al.

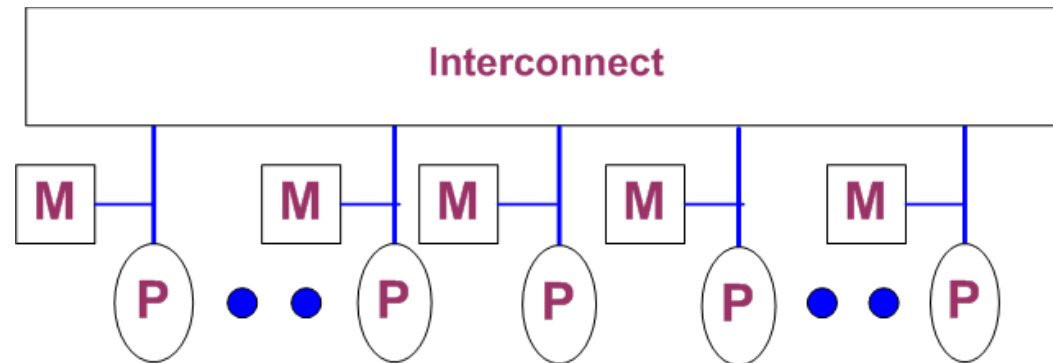
NUMA

- Non-Uniform Memory Access time
 - Processors see distant-dependent access times to memory
 - Implies physically distributed memory
- We often call these *distributed shared memory architectures*
 - Commercial example: SGI Altix UV, up to 1024 cores
 - Dash prototype at San Diego Supercomputer Center
 - Software/hardware support to monitor sharers



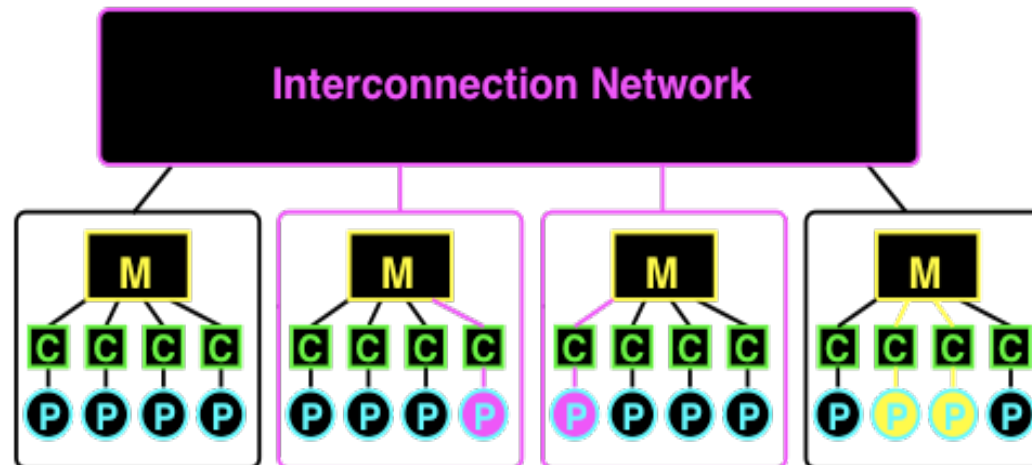
Architectures without shared memory

- A processor has direct access to local memory only
- Send and receive messages to obtain copies of data from other processors
- We call this a *shared nothing* architecture, or a *multicomputer*



Hybrid organizations

- Multi-tier organizations are hierarchically organized
- Each node is a multiprocessor, usually an SMP
- Nodes communicate by passing messages, processors within a node communicate via shared memory
- All clusters and high end systems today



Parallel processing this course

- Hardware
 - Mainframe
 - GPUs
- Primary programming models
 - MPI
 - CUDA
- Alternatives
 - Threads
 - Non-traditional (actors, dataflow)

The hardware

- Address space organization
 - Shared memory
 - Distributed memory
- Control mechanism

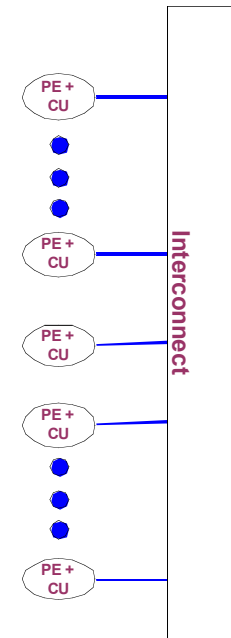
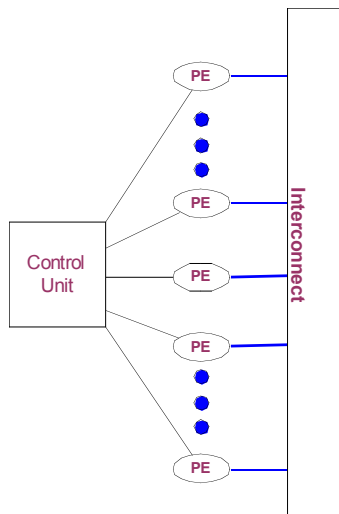
Control Mechanism

Flynn's classification (1966)

How do the processors issue instructions?

SIMD: Single Instruction, Multiple Data

Execute a global instruction stream in lock-step



MIMD: Multiple Instruction, Multiple Data
Clusters and servers processors execute
instruction streams independently

SIMD (Single Instruction Multiple Data)

- Operate on regular arrays of data
- Two landmark SIMD designs
 - ILIAC IV (1960s)
 - Connection Machine 1 and 2 (1980s)
- Vector computer: Cray-1 (1976)
- Intel and others support SIMD for multimedia and graphics
 - SSE
 - Streaming SIMD extensions, AltiVec
 - Operations defined on vectors
- GPUs, Cell Broadband Engine
- Reduced performance on data dependent or irregular computations

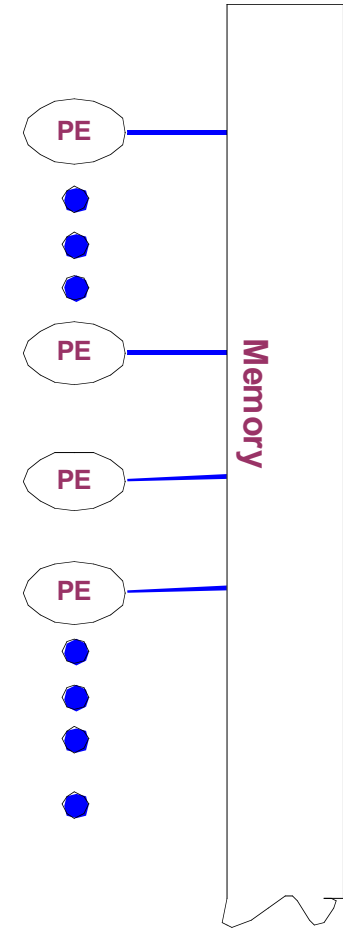
$$\begin{bmatrix} 2 \\ 4 \\ 8 \\ 7 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 5 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 5 \\ 2 \end{bmatrix}$$

```
forall i = 0 : n-1  
    x[i] = y[i] + z [ K[i] ]  
end forall
```

```
forall i = 0 : n-1  
    if ( x[i] < 0) then  
        y[i] = x[i]  
    else  
        y[i] = √x[i]  
    end if  
end forall
```

A theoretical basis: the PRAM

- Parallel Random Access Machine
- Idealized parallel computer
 - Unbounded number of processors
 - Shared memory of unbounded size
 - Constant access time
- Access time is comparable to that of a machine instruction
- All processors execute in lock step
- Processor can remain idle (SIMD)



Why is the PRAM interesting?

- Inspires real world systems and algorithms
 - E.g. GPUs process very long vectors
- Formal basis for fundamental limitations
 - If a PRAM algorithm is inefficient, then so is any parallel algorithm
 - If a PRAM algorithm is efficient, does it follow that any parallel algorithm is efficient?

How do we handle concurrent accesses?

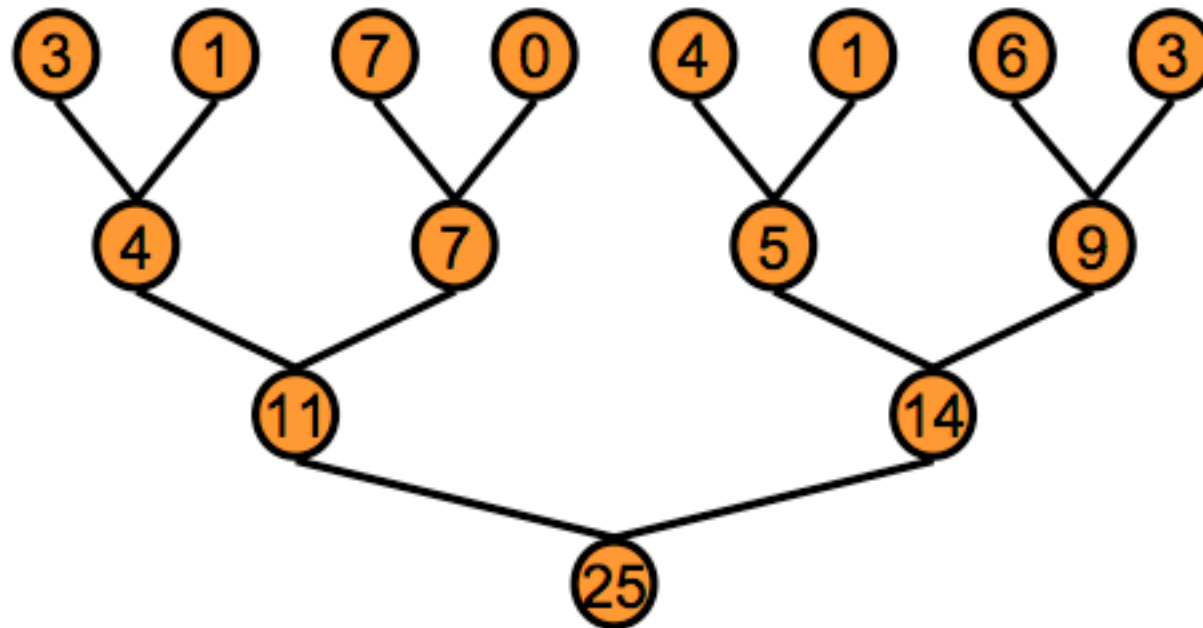
- Our options are to prohibit or permit concurrency in reads and writes
- There are therefore 4 flavors
- We'll focus on
CRCW = Concurrent Read Concurrent Write
- All processors may read or write

CRCW PRAM

- What if more than one processor attempts to write to the same location?
- Rules for combining multiple writes
 - **Common**: All processors must write the same value
 - **Arbitrary**: Only allow 1 arbitrarily chosen processor to write
 - **Priority**: Assign priorities to the processors, and allow the highest-priority processor's write
 - **Combine** the written values in some meaningful way, e.g. sum, max, using an associative operator.

Summation on PRAM

- Using n processors, we can sum a list of n numbers in $O(\lg n)$ time



A natural programming model for a PRAM: the data parallel model

- Apply an operation uniformly over all processors in a single step
- Assign each array element to a virtual processor
- Implicit barrier synchronization between each step

$$\begin{array}{|c|} \hline 2 \\ \hline 8 \\ \hline 18 \\ \hline 12 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hline -2 \\ \hline 7 \\ \hline 10 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline 10 \\ \hline 11 \\ \hline 2 \\ \hline \end{array}$$

Sorting on a PRAM

- A 2 step algorithm called *rank sort*
- Compute the rank (position in sorted order) for each element in parallel
 - ▶ Compare all possible pairings of input values in parallel, n^2 -fold parallelism
 - ▶ CRCW model with update on write using summation
- Move each value to its correctly sorted position according to the rank: n -fold parallelism
- $O(1)$ running time

Rank sort on a PRAM

1. Compute the rank of each key using n^2 -fold parallelism
2. Move each value in position according to the rank: n -fold parallelism

```
forall i=0:n-1, j=0:n-1
    if ( x[i] > x[j] ) then rank[i] = 1 end if
forall i=0:n-1
    y[rank[i]] = x[i]
```

Compute Ranks

forall $i=0:n-1$, $j=0:n-1$

if ($x[i] > x[j]$) then rank[i] = 1 end if

$O(N^2)$

parallelism

i

	1	7	3	-1	5	6
1				1		
7	1		1	1	1	1
3	1			1		
-1						
5	1		1	1		
6	1		1	1	1	

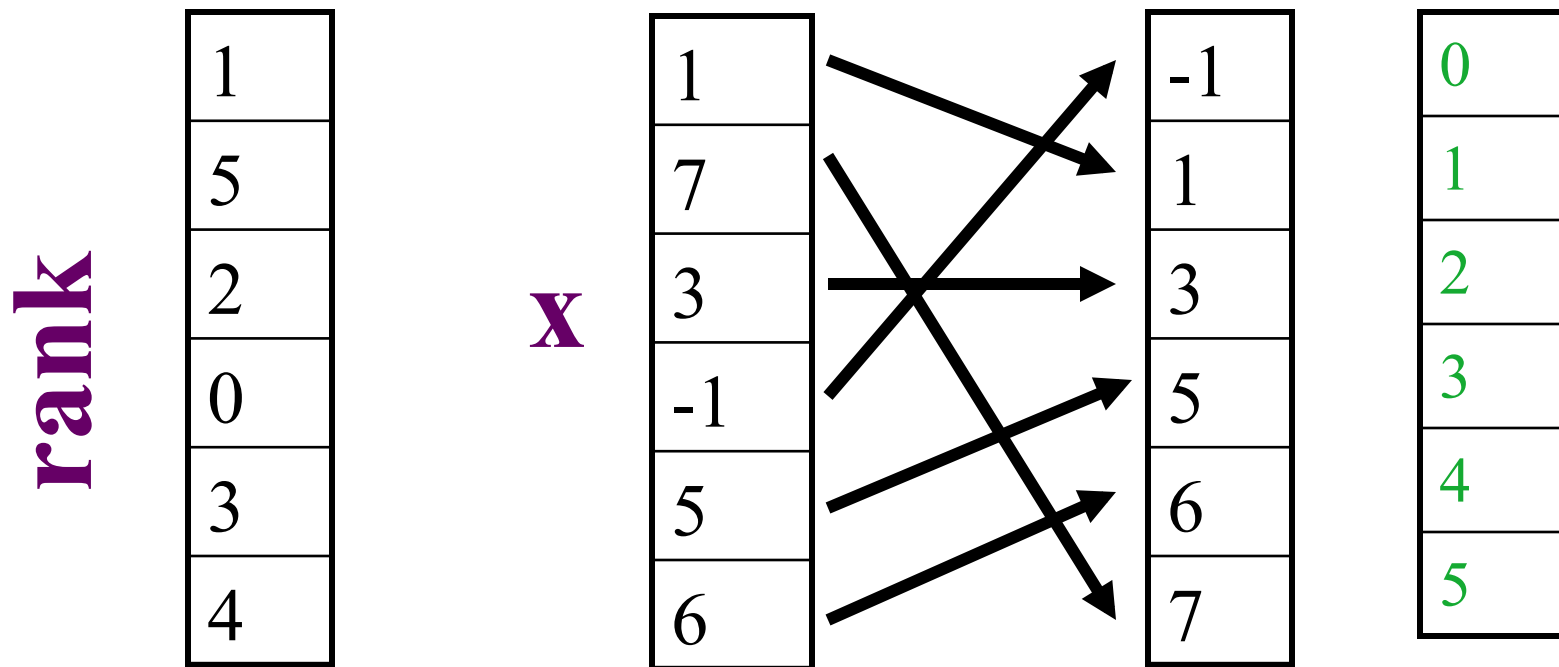
Update on write:
summation

rank

1
5
2
0
3
4

Route the data using the ranks

forall $i=0:n-1$ $y[\text{rank}[i]] = x[i]$



Parallel speedup and efficiency

- Definition *parallel speedup* on P processors

$$S_P = \frac{\text{Running time of the best serial program on 1 processor}}{\text{Running time of the parallel program on P processors}}$$

- The speedup is $(n \lg n) / O(1) = O(n \lg n)$
- No matter how many processors we have, the speedup for this workload is limited by the amount of available work
- Intrinsic limitation of the algorithm

Enter real world constraints

- The PRAM provides a necessary condition for an efficient algorithm on physical hardware
- But the condition is not sufficient; e.g. rank sort

```
forall ( i=0:n-1, j=0:n-1 )  
  if ( x[i] > x[j]) then rank[i] = 1 end if  
forall ( i=0:n-1 ) y[rank[i]] = x[i]
```

- Real world computers have finite resources including memory and network capacity
 - We cannot ignore communication network capacity, nor the cost of building a contention free network
 - Not all computations can execute efficiently in lock-step

Data parallelism in practice

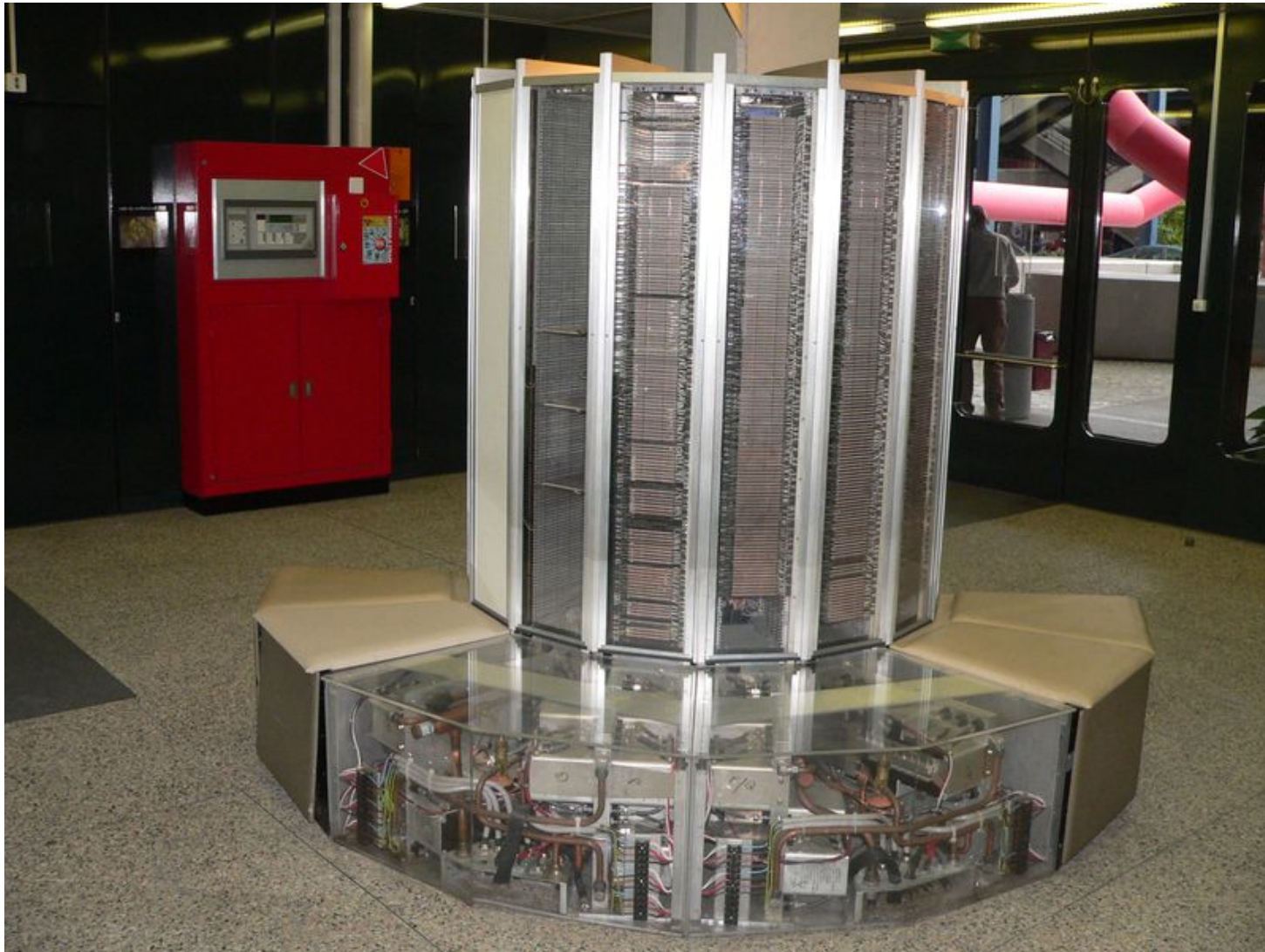
- Vectorizing compilers
- CUDA, OpenCL
- APL (1962)
- Connection Machine Fortran (early 80s)
- Fortran 90, 95, HPF (High Perf. Fortran) – 1994
- Matlab (late 1970s)
- Co-Array Fortran

Vector processing

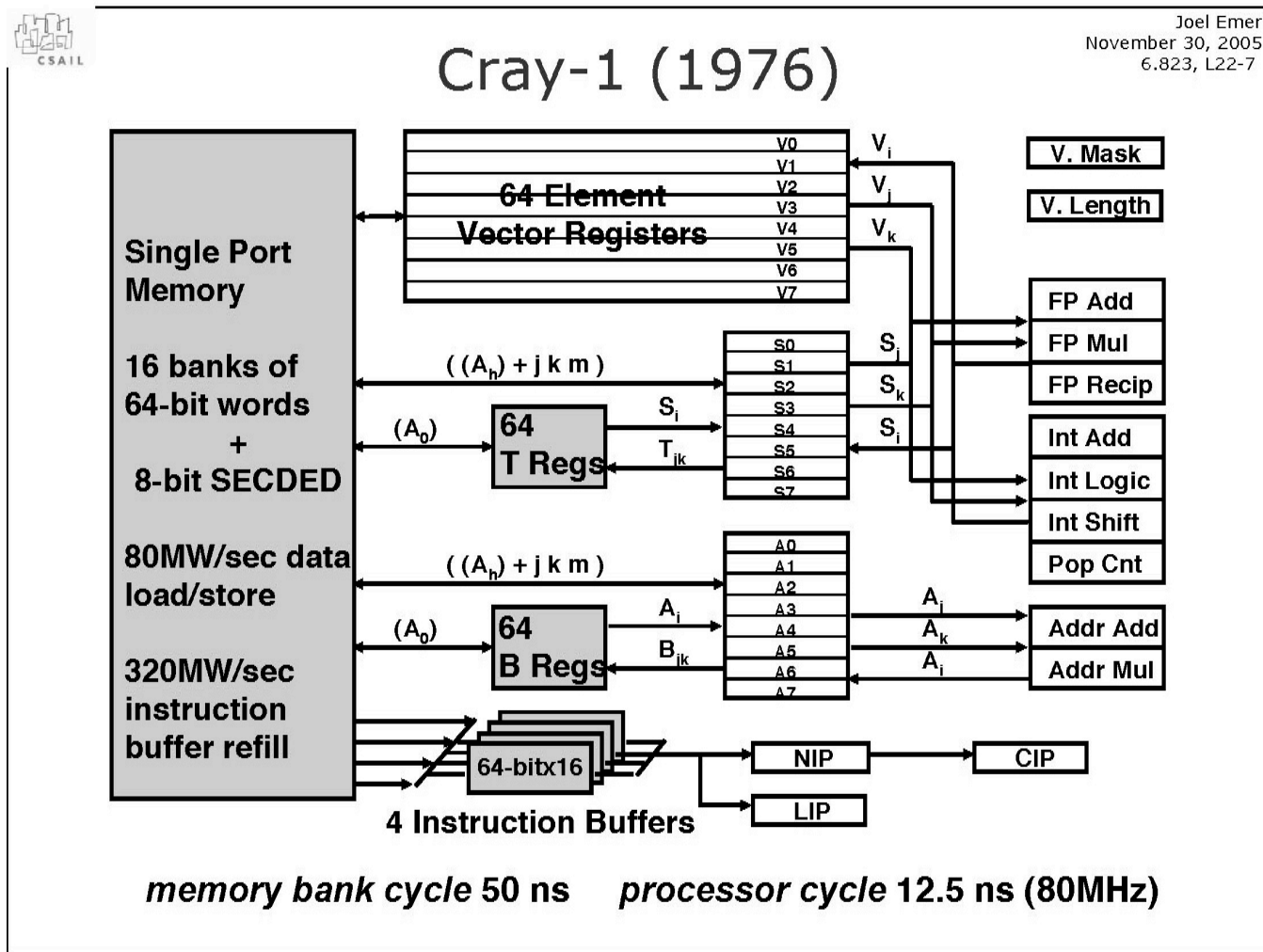
Road Map

- Vector processing (SIMD)
- Introduction to Stream Processing

The CRAY-1 ca, 1976



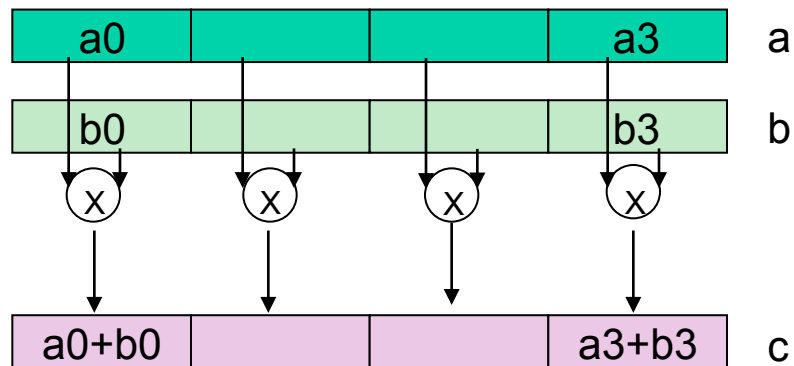
Cray-1 Block diagram



Streaming SIMD Extensions

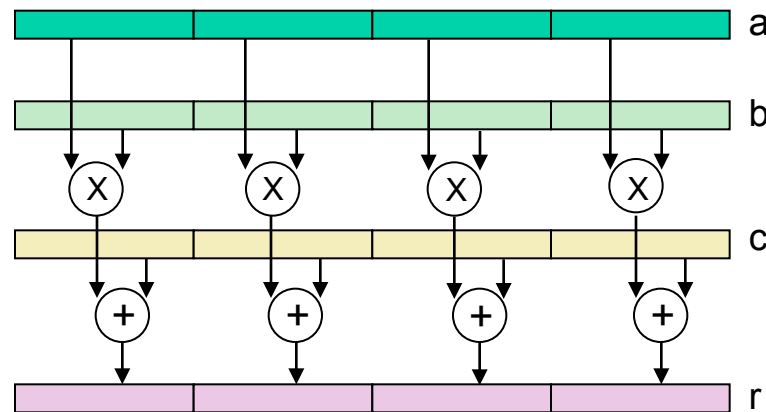
- en.wikipedia.org/wiki/Streaming_SIMD_Extensions
- SSE (SSE4 on Intel Nehalem), AltiVec
- Short vectors: up to 256 bits

for $i = 0:3$ { $c[i] = a[i] * b[i];$ }



More about streaming extensions

- Fused multiply-add
- Memory accesses must be contiguous and aligned
- How to sum the values in a vector?



$$r[0:3] = c[0:3] + a[0:3]*b[0:3]$$

Courtesy of Mercury Computer Systems, Inc.

How do we use the SSE instructions?

- Low level: assembly language or libraries
- Higher level: a vectorizing compiler

```
/opt/intel/bin/icpc -O2 -vec-report3 t2a.c
```

```
float *a, *b, *c = ...;
```

```
for (int i=0; i<n; i++)
```

```
    a[i] = b[i] + c[i];
```

```
t2a.c(19): (col. 3) remark: LOOP WAS VECTORIZED.
```

- Intel Xeon E5504 “Gainestown” @ 2.00GHz (lilliput)
- Double precision
 - With vectorization: 1.48 sec. [0.180 Gflops/s]
 - Without vectorization: 2.96 sec. [0.090 Gflops/s]
- Single precision
 - With vectorization: 0.574 Gflops/s
 - Without vectorization: 0.142 Gflops/s

How does the vectorizer work?

- Transformed code

```
for (i = 0; i < 1024; i+=4)
    a[i:i+3] = b[i:i+3] + c[i:i+3];
```

- Vector instructions

```
for (i = 0; i < 1024; i+=4){
    vB = vec_ld( &b[i] );
    vC = vec_ld( &c[i] );
    vA = vec_add( vB, vC );
    vec_st( vA, &a[i] );
}
```

Data dependencies prevent vectorization

- Data dependencies

```
for (int i = 1; i < N; i++)
```

```
    b[i] = b[i-1] + 2;
```

```
b[1] = b[0] + 2;
```

```
b[2] = b[1] + 2;
```

```
b[3] = b[2] + 2;
```

flow.c(6): warning #592: variable "b" is used before its value is set

```
b[i] = b[i-1] + 2; /* data dependence cycle */
```

^

flow.c(5): (col. 1) remark: loop was not vectorized: existence of vector dependence.

flow.c(6): (col. 2) remark: vector dependence: assumed FLOW dependence between b line 6 and b line 6.

But note different output from C++ compiler:

~~flow.c(6): warning #592: variable "b" is used before its value is set~~
~~b[i] = b[i-1] + 2;~~

Restrictions on vectorization

- Inner loops only

```
for(int j=0; j< reps; j++)  
    for (int i=0; i<N; i++)  
        a[i] = b[i] + c[i];
```

t2av.cpp(95): (col. 7) remark: loop was not vectorized: not inner loop.

What prevents vectorization

- Interrupted flow out of the loop

```
for (i=0; i<n; i++) {  
    a[i] = b[i] + c[i];  
    maxval = (a[i] > maxval ? a[i] : maxval);  
    if (maxval > 1000.0) break;  
}
```

t2mx.c(13): (col. 5) remark: loop was not vectorized:
nonstandard loop is not a vectorization candidate.

- This loop will vectorize

```
for (i=0; i<n; i++) {  
    a[i] = b[i] + c[i];  
    maxval = (a[i] > maxval ? a[i] : maxval);  
}
```

Compiler makes conservative decisions

- Interrupted flow out of the loop

```
void copy(char *p, char *q, int n) {  
    int i;  
    for (i = 0; i < n; i++) p[i] = q[i];  
}
```

copy.c(3): (col. 3) remark: loop was not vectorized: not inner loop.

copy.c(3): (col. 3) remark: loop was not vectorized: existence of vector dependence.

copy.c(3): (col. 27) remark: vector dependence: assumed ANTI dependence between q line 3 and p line 3.

copy.c(3): (col. 27) remark: vector dependence: assumed FLOW dependence between p line 3 and q line 3.

copy.c(3): (col. 27) remark: vector dependence: assumed FLOW dependence between p line 3 and q line 3.

copy.c(3): (col. 27) remark: vector dependence: assumed ANTI dependence between q line 3 and p line 3.

Run-time data dependence testing

- Restrict keyword needed to ensure correct semantics <http://www.devx.com/tips/Tip/13825>
“During the scope of the pointer declaration, all data accessed through it will be accessed .. through any other pointer... [thus] a given object cannot be changed through another pointer.”

```
/opt/intel/bin/icpc -O2 -vec-report3 -restrict t2a.c
```

```
void copy_conserve(char *restrict p, char *restrict q, int n) {  
    int i;  
    if (p+n < q || q+n < p)  
    #pragma ivdep  
        for (i = 0; i < n; i++) p[i] = q[i];      /* vector loop */  
    else  
        for (i = 0; i < n; i++) p[i] = q[i];      /* serial loop */  
}
```

```
copy.c(11): (col. 3) remark: LOOP WAS VECTORIZED.
```

Alignment

- Unaligned data movement is expensive
- Accesses aligned on 16 byte boundaries go faster
- Intel compiler can handle some alignments

<http://drdobbs.com/cpp/184401611>

```
double a[N], b[N];  
for (int i = 1; i < N-1; i++)  
    a[i+1] = b[i] * 3;
```

```
a[2] = b[1] - 1  
for (int i = 2; i < N-1; i++)  
    a[i+1] = b[i] * 3
```

cannot be vectorized:

```
void fill (char *x){  
    for (int i = 0; i < 1024; i++)  
        x[i] = 1;  
}
```

a[0]	a[1]	a[2]	a[3]
------	------	------	------

b[0]	b[1]	b[2]	b[3]
------	------	------	------

```
peel = x & 0x0f;  
if (peel != 0) {  
    peel = 16 - peel;  
    for (i = 0; i < peel; i++) x[i] = 1;  
}  
/* aligned access */  
for (i = peel; i < 1024; i++) x[i] = 1;
```

Fin